# Pro\*COBOL® Developer's Guide





Pro\*COBOL Developer's Guide, 23c

F47579-02

Copyright © 1999, 2023, Oracle and/or its affiliates.

Primary Author: Celin Cherian

Contributing Authors: Syed Mujeed Ahmed, Jack Melnick, Neelam Singh, Subhranshu Banerjee, Beethoven Cheng, Michael Chiocca, Nancy Ikeda, Alex Keh, Thomas Kurian, Shiao-Yen Lin, Diana Lorentz, Ajay Popat, Chris Racicot, Pamela Rothman, Simon Slack, Gael Stevens

Contributors: Valarie Moore

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

# Contents

	Preface		
	Intended Audience	xxvi	
	Documentation Accessibility	xxvi	
	Related Documents	xxvii	
	Conventions	xxvii	
	Changes in Pro*COBOL		
	New Features in Pro*COBOL Release 23c	xxviii	
	New Features in Pro*COBOL Release 21c	xxviii	
	Features in Pro*COBOL Release 19c	xxviii	
Par	t I Introduction and Concepts		
1	Introduction		
	1.1 What is Pro*COBOL?	1-1	
	1.2 The Pro*COBOL Precompiler	1-1	
	1.2.1 Language Alternatives	1-2	
	1.3 Advantages of the Pro*COBOL Precompiler	1-3	
	1.4 The SQL Language	1-3	
	1.5 The PL/SQL Language	1-3	
	1.6 Pro*COBOL Features and Benefits	1-4	
	1.7 Directory Structure	1-5	
	1.7.1 Header Files	1-6	
	1.7.2 Library File	1-6	
	1.7.3 Known Problems, Restrictions, and Workarounds	1-6	
	1.8 Compatibility, Upgrading, and Migration	1-7	
2	Precompiler Concepts		
	2.1 Key Concepts of Embedded SQL Programming	2-1	



	2.1.1	Steps in Develop	oing an Embedded SQL Application	2-1
	2.1.2	Embedded SQL	Statements	2-2
	2.1	2.1 Executable	e versus Declarative Statements	2-3
	2.1.3	Embedded SQL	Syntax	2-4
	2.1.4	Static Versus Dy	namic SQL Statements	2-5
	2.1.5	Embedded PL/S	QL Blocks	2-5
	2.1.6	Host Variables a	nd Indicator Variables	2-5
	2.1.7	Oracle Datatype:	s	2-6
	2.1.8	Tables		2-7
	2.1.9	Errors and Warn	ings	2-7
	2.1	9.1 SQLCODE	E/SQLSTATE Status Variables	2-7
	2.1.	9.2 SQLCA St	atus Variable	2-7
	2.1	9.3 WHENEVE	ER Statement	2-8
	2.1.	9.4 ORACA		2-8
	2.1	9.5 Precompile	er Options and Error Handling	2-9
	2.1.10	SQL99 Syntax	Support	2-9
2.2	Progr	amming Guidelin	es	2-9
	2.2.1	Abbreviations		2-9
	2.2.2	Case-Insensitivit	у	2-9
	2.2.3	COBOL Versions	s Supported	2-9
	2.2.4	Coding Areas		2-9
	2.2.5	Commas		2-10
	2.2.6	Comments		2-10
	2.2.7	Continuation Lin	es	2-11
	2.2.8	Copy Statements	s	2-11
	2.2.9	Decimal-Point is	Comma	2-11
	2.2.10	Delimiters		2-12
	2.2.11	Division Heade	rs that are Optional	2-12
	2.2.12	Embedded SQI	L Syntax	2-12
	2.2.13	Figurative Cons	stants	2-13
	2.2.14	File Length		2-13
	2.2.15	FILLER is Allov	ved	2-13
	2.2.16	Host Variable N	lames	2-13
	2.2.17	Hyphenated Na	ames	2-14
	2.2.18	Level Numbers		2-14
	2.2.19	MAXLITERAL [	Default	2-14
	2.2.20	Multibyte Datat	ypes	2-14
	2.2.21	NULLs in SQL		2-14
	2.2.22	Paragraph and	Section Names	2-14
	2.2.23	REDEFINES C	lause	2-15
	2.2.24	Relational Oper	rators	2-15



2.2.25 Sentence Terminator	2-16
2.3 The Declare Section	2-16
2.3.1 Contents of a Declare Section	2-16
2.3.1.1 An Example	2-16
2.3.2 Precompiler Option DECLARE_SECTION	2-17
2.3.3 Using the INCLUDE Statement	2-17
2.3.3.1 Filename Extensions	2-17
2.3.3.2 Search Paths	2-18
2.4 Nested Programs	2-18
2.4.1 Support for Nested Programs	2-19
2.4.1.1 Declaring the SQLCA	2-20
2.4.1.2 Nested Program Example	2-20
2.5 Conditional Precompilations	2-20
2.5.1 An Example	2-21
2.5.2 Defining Symbols	2-21
2.6 Separate Precompilations	2-22
2.6.1 Guidelines	2-22
2.6.1.1 Referencing Cursors	2-22
2.6.1.2 Specifying MAXOPENCURSORS	2-22
2.6.1.3 Using a Single SQLCA	2-22
2.6.1.4 Using a Single DATE_FORMAT	2-23
2.6.2 Restrictions	2-23
2.7 Compiling and Linking	2-23
2.8 Sample DEPT and EMP Tables	2-23
2.8.1 Sample DEPT and EMP Data	2-23
2.9 Sample EMP Program: SAMPLE1.PCO	2-24
Database Concepts	
3.1 Connecting to Oracle	3-2
3.2 Default Databases and Connections	3-3
	3-3
3.2.1 Concurrent Logons	
<ul><li>3.2.1 Concurrent Logons</li><li>3.2.2 Using Username/Password</li></ul>	3-4
· ·	
3.2.2 Using Username/Password	3-4
3.2.2 Using Username/Password 3.2.2.1 Named Database Connections	3-4 3-7
3.2.2 Using Username/Password 3.2.2.1 Named Database Connections 3.2.3 Automatic Logons	3-4 3-7 3-8
<ul> <li>3.2.2 Using Username/Password</li> <li>3.2.2.1 Named Database Connections</li> <li>3.2.3 Automatic Logons</li> <li>3.2.3.1 The AUTO_CONNECT Precompiler Option</li> </ul>	3-4 3-5 3-8 3-8
3.2.2 Using Username/Password 3.2.2.1 Named Database Connections 3.2.3 Automatic Logons 3.2.3.1 The AUTO_CONNECT Precompiler Option 3.2.4 Changing Passwords at Runtime	3-2 3-7 3-8 3-8
<ul> <li>3.2.2 Using Username/Password</li> <li>3.2.2.1 Named Database Connections</li> <li>3.2.3 Automatic Logons</li> <li>3.2.3.1 The AUTO_CONNECT Precompiler Option</li> <li>3.2.4 Changing Passwords at Runtime</li> <li>3.2.5 Connect Without Alter Authorization</li> </ul>	3-4 3-4 3-7 3-8 3-8 3-8 3-9



3.3 Key Terms	3-9
3.4 How Transactions Guard a Database	3-10
3.5 Beginning and Ending Transactions	3-11
3.6 Using the COMMIT Statement	3-11
3.6.1 WITH HOLD Clause in DECLARE CURSOR Statements	3-12
3.6.2 CLOSE_ON_COMMIT Precompiler Option	3-12
3.7 Using the ROLLBACK Statement	3-12
3.7.1 Statement-Level Rollbacks	3-13
3.8 Using the SAVEPOINT Statement	3-14
3.9 Using the RELEASE Option	3-15
3.10 Using the SET TRANSACTION Statement	3-16
3.11 Overriding Default Locking	3-16
3.11.1 Using the FOR UPDATE OF Clause	3-17
3.11.1.1 Restrictions	3-17
3.12 Fetching Across Commits	3-17
3.12.1 Using the LOCK TABLE Statement	3-18
3.13 Handling Distributed Transactions	3-18
3.14 Guidelines for Transaction Processing	3-19
3.14.1 Designing Applications	3-19
3.14.2 Obtaining Locks	3-19
3.14.3 Using PL/SQL	3-20
3.14.4 X/Open Applications	3-20
4 Datatypes and Host Variables	
4.1 The Oracle Database Datatypes	4-1
4.1.1 Internal Datatypes	4-1
4.1.2 External Data Types	4-3
4.1.2.1 CHAR	4-5
4.1.2.2 CHARF	4-5
4.1.2.3 CHARZ	4-5
4.1.2.4 DATE	4-5
4.1.2.5 DECIMAL	4-6
4.1.2.6 DISPLAY	4-6
4.1.2.7 FLOAT	4-6
4.1.2.8 INTEGER	4-6
4.1.2.9 LONG	4-6
4.1.2.10 LONG RAW	4-6
4.1.2.11 LONG VARCHAR	4-6
4.1.2.12 LONG VARRAW	4-7
4.1.2.13 NUMBER	4-7



	4.1	L.2.14	OVER-PUNCH	4-7
	4.1	L.2.15	RAW	4-7
	4.1	L.2.16	ROWID	4-7
	4.1	L.2.17	STRING	4-8
	4.1	L.2.18	UNSIGNED	4-8
	4.1	L.2.19	VARCHAR	4-8
	4.1	L.2.20	VARCHAR2	4-8
	4.1	L.2.21	VARNUM	4-9
	4.1	L.2.22	VARRAW	4-9
	4.1	L.2.23	SQL Pseudocolumns and Functions	4-9
	4.1	L.2.24	BOOLEAN	4-11
4.2	Date	time a	and Interval Datatype Descriptors	4-11
4.3	Host	Varia	bles	4-13
	4.3.1	Decl	aring Host Variables	4-13
	4.3	3.1.1	Example Declarations	4-17
	4.3	3.1.2	Initialization	4-17
	4.3	3.1.3	Restrictions	4-17
	4.3.2	Refe	erencing Host Variables	4-18
	4.3	3.2.1	Group Items as Host Variables	4-18
	4.3	3.2.2	Restrictions	4-19
4.4	Indic	ator V	'ariables	4-20
	4.4.1	Usin	g Indicator Variables	4-20
	4.4	1.1.1	On Input	4-20
	4.4	1.1.2	On Output	4-20
	4.4.2	Decl	aring Indicator Variables	4-20
	4.4.3	Refe	erencing Indicator Variables	4-21
	4.4	1.3.1	Use in Where Clauses	4-21
	4.4	1.3.2	Avoid Error Messages	4-21
	4.4	1.3.3	ANSI Requirements	4-22
	4.4	1.3.4	Indicator Variables for Multibyte NCHAR Variables	4-22
	4.4	1.3.5	Indicator Variables with Host Group Items	4-22
4.5	VAR	CHAR	R Variables	4-23
	4.5.1	Decl	aring VARCHAR Variables	4-23
	4.5.2	Impli	icit VARCHAR Group Items	4-24
	4.5.3	Refe	erencing VARCHAR Variables	4-24
4.6	Hand	dling E	Boolean Data	4-25
4.7	Hand	dling C	Character Data	4-25
	4.7.1	Defa	ult for PIC X	4-26
	4.7.2	Effec	cts of the PICX Option	4-26
	4.7.3	Fixe	d-Length Character Variables	4-26
	4.7	7.3.1	On Input	4-26



4.7.3.2 On Output	4-21
4.7.4 Varying-Length Variables	4-27
4.7.4.1 On Input	4-27
4.7.4.2 On Output	4-27
4.8 Universal ROWIDs	4-28
4.8.1 Subprogram SQLROWIDGET	4-29
4.9 Globalization Support	4-30
4.10 Unicode Support for Pro*COBOL	4-32
4.10.1 NLS_LOCAL=YES Restrictions	4-33
4.10.2 Character Strings in Embedded SQL	4-33
4.10.3 Embedded DDL	4-33
4.10.4 Blank Padding	4-33
4.10.5 Indicator Variables	4-34
4.10.6 Various Combinations of PIC X/PIC N Variables and NCHAR/CHAR Columns	4-34
4.10.6.1 PIC X and NCHAR Column	4-34
4.10.6.2 PIC N and CHAR column	4-34
4.10.7 Working With New Options	4-35
4.11 Datatype Conversion	4-35
4.12 Explicit Control Over DATE String Format	4-37
4.13 Datatype Equivalencing	4-38
4.13.1 Usefulness of Equivalencing	4-38
4.13.2 Host Variable Equivalencing	4-38
4.13.2.1 CONVBUFSZ Clause in VAR Statement	4-40
4.13.2.2 An Example	4-41
4.13.3 Using the CHARF Datatype Specifier	4-42
4.13.4 Guidelines	4-42
4.13.5 RAW and LONG RAW Values	4-43
4.14 Platform Endianness Support	4-43
4.15 Sample Program 4: Datatype Equivalencing	4-44
Embedded SQL	
5.1 Using Host Variables	5-1
5.1.1 Output Versus Input Host Variables	5-1
5.2 Using Indicator Variables	5-2
5.2.1 Input Variables	5-2
5.2.2 Output Variables	5-3
5.2.3 Inserting NULLs	5-3
5.2.4 Handling Returned NULLs	5-4
5.2.5 Fetching NULLs	5-4
5.2.6 Testing for NULLs	5-4



5.2.7 Fe	tching Truncated Values	5-5
5.3 The Basi	ic SQL Statements	5-5
5.3.1 Se	lecting Rows	5-6
5.3.1.1	Available Clauses	5-7
5.3.2 Ins	serting Rows	5-7
5.3.3 DM	1L Returning Clause	5-7
5.3.4 Us	ing Subqueries	5-7
5.3.5 Up	dating Rows	5-8
5.3.6 De	leting Rows	5-8
5.3.7 Us	ing the WHERE Clause	5-8
5.4 Using Bo	polean Data Type	5-9
5.5 Cursors		5-9
5.5.1 De	claring a Cursor	5-10
5.5.2 Op	ening a Cursor	5-11
5.5.3 Fe	tching from a Cursor	5-12
5.5.4 Cld	osing a Cursor	5-13
5.5.5 Us	ing the CURRENT OF Clause	5-13
5.5.6 Re	strictions	5-14
5.5.7 A	Гуріcal Sequence of Statements	5-14
5.5.8 Po	sitioned Update	5-14
5.5.9 Th	e PREFETCH Precompiler Option	5-15
5.6 Scrollabl	e Cursors	5-15
5.6.1 Us	ing Scrollable Cursors	5-16
5.6.1.1	DECLARE SCROLL CURSOR	5-16
5.6.1.2	OPEN	5-16
5.6.1.3	FETCH	5-16
5.6.1.4	CLOSE	5-17
5.6.2 Th	e CLOSE_ON_COMMIT Precompiler Option	5-17
5.6.3 Th	e PREFETCH Precompiler Option	5-17
5.7 Flexible	B Area Length	5-18
5.8 Fix Exec	ution Plans	5-18
5.8.1 SQ	<u>P</u> L File	5-21
5.8.1.1	Examples	5-21
5.8.2 LO	G File	5-23
5.9 Sample I	Program 2: Cursor Operations	5-23
Embedded	I PL/SQL	
6.1 Embeddi	ing PL/SQL	6-1
6.1.1 Ho	st Variables	6-1
6.1.2 VA	RCHAR Variables	6-1



6.1.3	Indic	cator Variables	6-1
6.	1.3.1	Handling NULLs	6-2
6.	1.3.2	Handling Truncated Values	6-2
6.1.4	SQL	CHECK	6-2
6.2 Adv	antage	es of PL/SQL	6-2
6.2.1	Bette	er Performance	6-2
6.2.2	Integ	gration with Oracle	6-3
6.2.3	Curs	sor FOR Loops	6-3
6.2.4	Subp	programs	6-3
6.	2.4.1	Parameter Modes	6-4
6.2.5	Pack	kages	6-4
6.2.6	PL/S	SQL Tables	6-4
6.2.7	User	r-Defined Records	6-5
6.3 Em	beddin	g PL/SQL Blocks	6-6
6.4 Hos	t Varia	bles and PL/SQL	6-6
6.4.1	PL/S	SQL Examples	6-6
6.4.2	A Mo	ore Complex PL/SQL Example	6-7
6.4.3	VAR	CHAR Pseudotype	6-9
6.5 Indi	cator V	ariables and PL/SQL	6-9
6.5.1	Hand	dling NULLs	6-10
6.5.2	Hand	dling Truncated Values	6-10
6.6 Hos	t Table	es and PL/SQL	6-10
6.6.1	ARR	AYLEN Statement	6-12
6.	6.1.1	Optional Keyword EXECUTE to ARRAYLEN Statement	6-13
6.7 Cur	sor Usa	age in Embedded PL/SQL	6-15
6.8 Sto	red PL/	SQL and Java Subprograms	6-16
6.8.1	Crea	ating Stored Subprograms	6-16
6.8.2	Calli	ng a Stored PL/SQL or Java Subprogram	6-17
6.	8.2.1	Anonymous PL/SQL Block	6-17
6.	8.2.2	CALL Statement	6-17
6.	8.2.3	CALL Example	6-18
6.8.3	Usin	g Dynamic PL/SQL	6-18
6.8.4	Subp	programs Restriction	6-19
6.9 San	nple Pr	ogram 9: Calling a Stored Procedure	6-19
6.9.1	Rem	note Access	6-23
6.10 Cu	ırsor Va	ariables	6-23
6.10.1	L Dec	claring a Cursor Variable	6-24
6.10.2	2 Allo	ocating a Cursor Variable	6-24
6.10.3	3 Ope	ening a Cursor Variable	6-24
6.	10.3.1	Opening Indirectly through a Stored PL/SQL Procedure	6-24
6.	10.3.2	Opening Directly from Your Pro*COBOL Application	6-26



6.10.4 Fetching from a Cursor variable	6-26
6.10.5 Closing a Cursor Variable	6-27
6.10.6 Freeing a Cursor Variable	6-27
6.10.7 Restrictions on Cursor Variables	6-27
6.10.8 Sample Program 11: Cursor Variables	6-27
6.10.8.1 SAMPLE11.SQL	6-27
6.10.8.2 SAMPLE11.PCO	6-28
Host Tables	
7.1 Host Tables	7-1
7.2 Advantages of Host Tables	7-1
7.3 Tables in Data Manipulation Statements	7-2
7.3.1 Declaring Host Tables	7-2
7.3.1.1 Restrictions	7-2
7.3.2 Referencing Host Tables	7-3
7.3.3 Using Indicator Tables	7-4
7.3.4 Host Group Item Containing Tables	7-4
7.3.5 Oracle Restrictions	7-5
7.3.6 ANSI Restriction and Requirements	7-5
7.4 Selecting into Tables	7-5
7.4.1 Batch Fetches	7-5
7.4.2 Using SQLERRD(3)	7-6
7.4.3 Number of Rows Fetched	7-6
7.4.4 Restrictions on Using Host Tables	7-7
7.4.5 Fetching NULLs	7-7
7.4.6 Fetching Truncated Values	7-8
7.4.7 Sample Program 3: Fetching in Batches	7-8
7.5 Inserting with Tables	7-10
7.5.1 Restrictions on Host Tables	7-10
7.6 Updating with Tables	7-10
7.6.1 Restrictions in UPDATE	7-11
7.7 Deleting with Tables	7-11
7.7.1 Restrictions in DELETE	7-12
7.8 Using Indicator Tables	7-12
7.9 The FOR Clause	7-13
7.9.1 Restrictions	7-13
7.9.1.1 In a SELECT Statement	7-13
7.9.1.2 With the CURRENT OF Clause	7-14
7.10 The WHERE Clause	7-14
7.11 Mimicking the CURRENT OF Clause	7-15



	7.12 Tables of Group Items as Host Variables	7-16
	7.13 Sample Program 14: Tables of Group Items	7-17
	7.14 Additional Array Insert/Select Syntax	7-19
	7.15 Using Implicit Buffered Insert	7-24
8	Error Handling and Diagnostics	
	8.1 Why Error Handling is Needed	8-1
	8.2 Error Handling Alternatives	8-1
	8.2.1 SQLCA	8-2
	8.2.2 ORACA	8-2
	8.2.3 ANSI SQLSTATE Variable	8-2
	8.2.4 Declaring SQLSTATE	8-3
	8.2.4.1 SQLSTATE Values	8-3
	8.3 Using the SQL Communications Area	8-6
	8.3.1 Contents of the SQLCA	8-6
	8.3.2 Declaring the SQLCA	8-7
	8.3.3 Key Components of Error Reporting	8-7
	8.3.3.1 Status Codes	8-7
	8.3.3.2 Warning Flags	8-7
	8.3.3.3 Rows-Processed Count	8-7
	8.3.3.4 Parse Error Offset	8-7
	8.3.3.5 Error Message Text	8-8
	8.3.4 SQLCA Structure	8-8
	8.3.4.1 SQLCAID	8-8
	8.3.4.2 SQLCABC	8-8
	8.3.4.3 SQLCODE	8-8
	8.3.4.4 SQLERRM	8-9
	8.3.4.5 SQLERRD	8-9
	8.3.4.6 SQLWARN	8-10
	8.3.4.7 SQLEXT	8-11
	8.3.5 PL/SQL Considerations	8-11
	8.3.6 Getting the Full Text of Error Messages	8-11
	8.3.7 DSNTIAR	8-12
	8.3.8 WHENEVER Directive	8-12
	8.3.8.1 Conditions	8-13
	8.3.8.2 SQLWARNING	8-13
	8.3.8.3 SQLERROR	8-13
	8.3.8.4 NOT FOUND or NOTFOUND	8-13
	8.3.8.5 Actions	8-13
	8.3.8.6 CONTINUE	8-13
		0 20



9.2 Advantages and Disadvantages of Dynamic SQL	9-2
9.1 Dynamic SQL	9-1
Oracle Dynamic SQL	
5.5.2 Status Variable Combinations	0 00
8.5.1 Status Variable Combinations	8-30
8.5 How Errors Map to SQLSTATE Codes	8-25
8.4.5.17 ORANEX 8.4.6 ORACA Example Program	8-23 8-23
8.4.5.16 ORANPR 8.4.5.17 ORANEX	8-23 8-23
8.4.5.15 ORANDR	8-23
8.4.5.14 ORACOC	8-23
8.4.5.13 ORAMOC	8-23
8.4.5.12 ORAHOC	8-22
8.4.5.11 Cursor Cache Statistics	8-22
8.4.5.10 ORASLNR	8-22
8.4.5.9 ORASFNM	8-22
8.4.5.8 ORASTXT	8-22
8.4.5.7 Diagnostics	8-22
8.4.5.6 ORASTXTF	8-21
8.4.5.5 ORAHCHF	8-21
8.4.5.4 ORADBGF	8-21
8.4.5.3 ORACCHF	8-21
8.4.5.2 ORACABC	8-20
8.4.5.1 ORACAID	8-20
8.4.5 ORACA Structure	8-20
8.4.4 Choosing Runtime Options	8-20
8.4.3 Enabling the ORACA	8-20
8.4.2 Declaring the ORACA	8-20
8.4.1 Contents of the ORACA	8-19
8.4 Using the Oracle Communications Area	8-19
8.3.10 Getting the Text of SQL Statements	8-18
8.3.9.4 Careless Usage: Examples	8-17
8.3.9.3 Scope	8-17
8.3.9.2 DO CALL	8-16
8.3.9.1 DO PERFORM	8-14
8.3.9 Coding the WHENEVER Statement	8-14
8.3.8.10 STOP	8-14
8.3.8.9 GOTO or GO TO	8-14
8.3.8.8 DO PERFORM	8-14
8.3.8.7 DO CALL	8-14



9.3 When to Use Dynamic SQL	9-2
9.4 Requirements for Dynamic SQL Statements	9-2
9.5 How Dynamic SQL Statements Are Processed	9-3
9.6 Methods for Using Dynamic SQL	9-3
9.6.1 Method 1	9-3
9.6.2 Method 2	9-4
9.6.3 Method 3	9-4
9.6.4 Method 4	9-4
9.6.5 Guidelines	9-5
9.6.5.1 Avoiding Common Errors	9-5
9.7 Using Method 1	9-6
9.7.1 The EXECUTE IMMEDIATE Statement	9-6
9.7.2 An Example	9-7
9.8 Sample Program 6: Dynamic SQL Method 1	9-8
9.9 Using Method 2	9-10
9.9.1 The USING Clause	9-11
9.10 Sample Program 7: Dynamic SQL Method 2	9-11
9.11 Using Method 3	9-14
9.11.1 PREPARE	9-14
9.11.2 DECLARE	9-15
9.11.3 OPEN	9-15
9.11.4 FETCH	9-15
9.11.5 CLOSE	9-16
9.12 Sample Program 8: Dynamic SQL Method 3	9-16
9.13 Using Oracle Method 4	9-18
9.13.1 Need for the SQLDA	9-19
9.13.2 The DESCRIBE Statement	9-19
9.13.3 SQLDA Contents	9-20
9.13.4 Implementing Method 4	9-20
9.14 Using the DECLARE STATEMENT Statement	9-21
9.15 Using Host Tables	9-22
9.16 Using PL/SQL	9-22
9.16.1 With Method 1	9-22
9.16.2 With Method 2	9-22
9.16.3 With Method 3	9-23
9.16.4 With Method 4	9-23
9.16.5 Caution	9-23
9.17 Dynamic SQL Statement Caching	9-23
9.18 About Boolean Data Type Support	9-26



# Part II Applications

10.1 B	asics of ANSI Dynamic SQL	
10.1	.1 Precompiler Options	
10.2 C	Overview of ANSI SQL Statements	
10.3 S	Sample Code	
10.4 C	Oracle Extensions	
10.4	.1 Reference Semantics	
10.4	.2 Using Tables for Bulk Operations	
10.5 A	NSI Dynamic SQL Precompiler Options	
10.6 F	full Syntax of the Dynamic SQL Statements	
10.6	.1 ALLOCATE DESCRIPTOR	
-	10.6.1.1 Variables	
-	10.6.1.2 Examples	
10.6	.2 DEALLOCATE DESCRIPTOR	
10.6	.3 GET DESCRIPTOR	
10.6	.4 SET DESCRIPTOR	
-	10.6.4.1 Example	
10.6	.5 Use of PREPARE	
10.6	.6 DESCRIBE INPUT	
10.6	.7 DESCRIBE OUTPUT	
10.6	.8 EXECUTE	
10.6	.9 Use of EXECUTE IMMEDIATE	
10.6	.10 Use of DYNAMIC DECLARE CURSOR	
10.6	.11 OPEN Cursor	
10.6	.12 FETCH	
10.6	.13 CLOSE a Dynamic Cursor	
10.6	.14 Differences From Oracle Dynamic Method 4	
10.6	.15 Restrictions	
10.7 S	Sample Programs: SAMPLE12.PCO	
Oracle	e Dynamic SQL: Method 4	
11.1 N	leeting the Special Requirements of Method 4	
11.1	.1 Advantages of Method 4	
11.1	.2 Information the Database Needs	
11.1.	.3 Where the Information is Stored	



11.1.4 How Information is Obtained

11.2 Understanding the SQL Descriptor Area (SQLDA)

11-3

11-3

11.2.1	Purpose of the SQLDA	11-3
11.2.2	2 Multiple SQLDAs	11-4
11.2.3	B Declaring a SQLDA	11-4
11.3 Th	e SQLDA Variables	11-7
11.4 Pr	erequisite Knowledge	11-12
11.4.1	Using SQLADR	11-12
11.4.2	2 Converting Data	11-13
11.4.3	B Coercing Datatypes	11-15
11.4.4	Handling NULL/Not NULL Datatypes	11-18
11.5 Th	e Basic Steps	11-19
11.6 A	Closer Look at Each Step	11-19
11.6.1	Declare a Host String	11-20
11.6.2	2 Declare the SQLDAs	11-21
11.6.3	Set the Maximum Number to DESCRIBE	11-21
11.6.4	Initialize the Descriptors	11-21
11.6.5	Store the Query Text in the Host String	11-24
11.6.6	PREPARE the Query from the Host String	11-24
11.6.7	DECLARE a Cursor	11-25
11.6.8	B DESCRIBE the Bind Variables	11-25
11.6.9	Reset Number of Place-Holders	11-26
11.6.1	.0 Get Values for Bind Variables	11-27
11.6.1	1 OPEN the Cursor	11-28
11.6.1	.2 DESCRIBE the Select List	11-29
11.6.1	3 Reset Number of Select-List Items	11-30
11.6.1	.4 Reset Length/Datatype of Each Select-List Item	11-30
11.6.1	.5 FETCH Rows from the Active Set	11-31
11.6.1	.6 Get and Process Select-List Values	11-32
11.6.1	.7 CLOSE the Cursor	11-32
11.7 Ab	out Using Domain Metadata	11-33
11.8 Us	ing Host Tables with Method 4	11-33
11.9 Sa	mple Program 10: Dynamic SQL Method 4	11-36
Multith	readed Applications	
12.1 Int	roduction to Threads	12-1
12.2 Ru	Intime Contexts in Pro*COBOL	12-2
12.3 Ru	ıntime Context Usage Models	12-3
12.3.2	Multiple Threads Sharing a Single Runtime Context	12-3
12.3.2	2 Multiple Threads Sharing Multiple Runtime Contexts	12-4
12.4 Us	ser Interface Features for Multithreaded Applications	12-5
12.4.2	L THREADS Option	12-5



	12.4.2	Embedded SQL Statements and Directives for Runtime Contexts	12-6
	12.4	I.2.1 Host Tables of SQL-CONTEXT Are Not Allowed	12-6
	12.4	I.2.2 EXEC SQL ENABLE THREADS	12-6
	12.4	1.2.3 EXEC SQL CONTEXT ALLOCATE	12-6
	12.4	1.2.4 EXEC SQL CONTEXT USE	12-7
	12.4	1.2.5 EXEC SQL CONTEXT FREE	12-7
	12.4.3	Communication with Pro*C/C++ Programs	12-7
	12.4.4	Multithreading Programming Considerations	12-7
	12.4	I.4.1 Restrictions on Multithreading	12-8
	12.4.5	Multiple Context Examples	12-8
	12.4	I.5.1 Example 1	12-8
	12.4	I.5.2 Example 2	12-8
	12.4	I.5.3 Example 3	12-9
	12.4	I.5.4 Example 4	12-10
	12.4	I.5.5 Example 5	12-11
	12.5 Multi	threaded Example	12-12
13	Large Ob	ojects (LOBs)	
	13.1 Using	g LOBs	13-1
	13.1.1	Internal LOBs	13-1
	13.1.2	External LOBs	13-1
	13.1.3	Security for BFILEs	13-2
	13.1.4	LOBs Compared with LONG and LONG RAW	13-2
	13.1.5	LOB Locators	13-2
	13.1.6	Temporary LOBs	13-3
	13.1.7	LOB Buffering Subsystem	13-3
	13.2 How	to Use LOBs	13-3
	13.2.1	LOB Locators in Your Application	13-5
	13.2.2	Initializing a LOB	13-5
	13.2	2.2.1 Internal LOBs	13-5
	13.2	2.2.2 External LOBs	13-5
	13.2	2.2.3 Temporary LOBs	13-6
	13.2	2.2.4 Freeing LOBs	13-6
	13.3 Rule	s for LOB Statements	13-6
	13.3.1	For All LOB Statements	13-6
	13.3.2	For the LOB Buffering Subsystem	13-7
	13.3.3	For Host Variables	13-7
	13.4 LOB	Statements	13-8
	13.4.1	APPEND	13-8
	13.4.2	ASSIGN	13-9



15.4.5	CLU3E	13-8
13.4.4	COPY	13-10
13.4.5	CREATE TEMPORARY	13-11
13.4.6	DISABLE BUFFERING	13-11
13.4.7	' ENABLE BUFFERING	13-12
13.4.8	B ERASE	13-12
13.4.9	FILE CLOSE ALL	13-13
13.4.1	.0 FILE SET	13-13
13.4.1	1 FLUSH BUFFER	13-14
13.4.1	2 FREE TEMPORARY	13-14
13.4.1	3 LOAD FROM FILE	13-14
13.4.1	4 OPEN	13-15
13.4.1	5 READ	13-16
13.4.1	6 TRIM	13-18
13.4.1	.7 WRITE	13-18
13.4.1	.8 DESCRIBE	13-19
13.4.1	.9 READ and WRITE Using the Polling Method	13-21
13.5 LO	B Sample Program: LOBDEMO1.PCO	13-23
	e procob Command	14-1 14-2
14.1.1		
	tions During Precompilation	14-2
	out the Options	14-2
14.3.1	•	14-3
14.3.2	•	14-3
14.3.3	5	14-4
	tering Precompiler Options	14-5
14.4.1		14-5
14.4.2		14-5
	4.4.2.1 Advantages	14-5
	4.4.2.2 Scope of EXEC ORACLE	14-5
14.4.3	3	14-6
14.4.4		14-7
	out Usage Domain Metadata Support	14-8
	ope of Precompiler Options	14-10
_	lick Reference	14-10
	ing Pro*COBOL Precompiler Options	14-12
14.8.1		14-13
14.8.2	2 ASSUME_SQLCODE	14-13



14.8.3	AUTO_CONNECT	14-14
14.8.4	CHARSET_PICX	14-14
14.8.5	CHARSET_PICN	14-15
14.8.6	CLOSE_ON_COMMIT	14-15
14.8.7	COMMON_PARSER	14-16
14.8.8	CONFIG	14-16
14.8.9	DATE_FORMAT	14-17
14.8.10	DB2_ARRAY	14-17
14.8.11	DBMS	14-18
14.8.12	DECLARE_SECTION	14-18
14.8.13	DEFINE	14-19
14.8.14	DYNAMIC	14-19
14.8.15	END_OF_FETCH	14-19
14.8.16	ERRORS	14-20
14.8.17	File_ID	14-20
14.8.18	FIPS	14-21
14.8.19	FORMAT	14-22
14.8.20	HOLD_CURSOR	14-22
14.8.21	HOST	14-23
14.8.22	IMPLICIT_SVPT	14-23
14.8.23	INAME	14-24
14.8.24	INCLUDE	14-24
14.8.25	IRECLEN	14-25
14.8.26	LITDELIM	14-25
14.8.27	LNAME	14-26
14.8.28	LRECLEN	14-26
14.8.29	LTYPE	14-27
14.8.30	MAX_ROW_INSERT	14-27
14.8.31	MAXLITERAL	14-28
14.8.32	MAXOPENCURSORS	14-28
14.8.33	MODE	14-29
14.8.34	NESTED	14-30
14.8.35	NLS_LOCAL	14-30
14.8.36	ONAME	14-31
14.8.37	ORACA	14-31
14.8.38	ORECLEN	14-32
14.8.39	OUTLINE	14-32
14.8.40	OUTLNPREFIX	14-33
14.8.41	PAGELEN	14-33
14.8.42	PICN_ENDIAN	14-34
14.8.43	PICX	14-34



PLAN_BASELINE	14-35
PLAN_PREFIX	14-35
PLAN_RUN	14-36
PLAN_FIXED	14-36
PLAN_ENABLED	14-36
MEMFORPREFETCH	14-37
PREFETCH	14-37
RELEASE_CURSOR	14-38
RUNOUTLINE	14-39
SELECT_ERROR	14-40
SQLCHECK	14-40
STMT_CACHE	14-43
TYPE_CODE	14-43
UNSAFE_NULL	14-44
USERID	14-44
VARCHAR	14-45
XREF	14-45
Host Variables	A-1
	A-1 A-1
.1 Declaring	A-1
.2 Naming	A-1
NCLUDE Statements	A-1
MAXLITERAL Default	A-2
PIC N or Pic G Clause for Multi-byte Globalization Support Characters	A-2
RETURN-CODE Special Register May Be Unpredictable.	A-2
Byte-Order of Binary Data	A-2
Words, Keywords, and Namespaces	
red Words and Keywords	B-1
red Namespaces	B-4
nce Tuning	
s of Poor Performance	C-1
ing Performance	C-2
Host Tables	C-2
	C-3
	PLAN_PREFIX PLAN_RUN PLAN_FIXED PLAN_ENABLED MEMFORPREFETCH PREFETCH RELEASE_CURSOR RUNOUTLINE SELECT_ERROR SQLCHECK STMT_CACHE TYPE_CODE UNSAFE_NULL USERID VARCHAR XREF  System Dependencies Specific References in this Manual COBOL Versions lost Variables -1 Declaring -2 Naming NCLUDE Statements MAXLITERAL DefaultDIC OF CICLUSE OF COME OF COM



	C.5 Optimizing SQL Statements	C-3
	C.5.1 Optimizer Hints	C-3
	C.5.1.1 Giving Hints	C-4
	C.6 SQL Statement Caching	C-4
	C.7 Using Indexes	C-5
	C.8 Taking Advantage of Row-Level Locking	C-5
	C.9 Eliminating Unnecessary Parsing	C-5
	C.9.1 Handling Explicit Cursors	C-5
	C.9.1.1 Cursor Control	C-6
	C.9.2 Using the Cursor Management Options	C-7
	C.9.2.1 Private SQL Areas and Cursor Cache	C-7
	C.9.2.2 Resource Use	C-8
	C.9.2.3 Infrequent Execution	C-8
	C.9.2.4 Frequent Execution	C-8
	C.9.2.5 Effect on the Shared SQL Area	C-9
	C.9.2.6 Embedded PL/SQL Considerations	C-9
	C.9.2.7 Parameter Interactions	C-9
	C.10 Avoiding Unnecessary Reparsing	C-10
	C.11 About Using Oracle Connection Manager in Traffic Director Mode	C-10
	Syntactic and Semantic Checking  D.1 Syntactic and Semantic Checking Basics	D-1
	D.2 Controlling the Type and Extent of Checking	D-1
	D.3 Specifying SQLCHECK=SEMANTICS	D-2
	D.3.1 Enabling a Semantic Check	D-2
	D.3.1.1 Connecting to Oracle	D-2
	D.3.1.2 Using DECLARE TABLE	D-3
Е	Embedded SQL Statements and Precompiler Directives	
	E.1 Summary of Precompiler Directives and Embedded SQL Statements	E-2
	E.2 About the Statement Descriptions	E-5
	E.3 How to Read Syntax Diagrams	E-5
	E.3.1 Statement Terminator	E-6
	E.3.2 Required Keywords and Parameters	E-6
	E.3.3 Optional Keywords and Parameters	E-6
	E.3.4 Syntax Loops	E-7
	E.3.5 Multi-part Diagrams	E-7
	E.3.6 Oracle Names	E-7
	E.4 ALLOCATE (Executable Embedded SQL Extension)	E-8



E.5	ALLOCATE DESCRIPTOR (Executable Embedded SQL)	E-9
E.6	CALL (Executable Embedded SQL)	E-10
E.7	CLOSE (Executable Embedded SQL)	E-11
E.8	COMMIT (Executable Embedded SQL)	E-12
E.9	CONNECT (Executable Embedded SQL Extension)	E-13
E.10	CONTEXT ALLOCATE (Executable Embedded SQL Extension)	E-15
E.11	CONTEXT FREE (Executable Embedded SQL Extension)	E-15
E.12	CONTEXT USE (Oracle Embedded SQL Directive)	E-16
E.13	DEALLOCATE DESCRIPTOR (Embedded SQL Statement)	E-17
E.14	DECLARE CURSOR (Embedded SQL Directive)	E-18
E.15	DECLARE DATABASE (Oracle Embedded SQL Directive)	E-20
E.16	DECLARE STATEMENT (Embedded SQL Directive)	E-21
E.17	DECLARE TABLE (Oracle Embedded SQL Directive)	E-23
E.18	DELETE (Executable Embedded SQL)	E-24
E.19	DESCRIBE (Executable Embedded SQL)	E-26
E.20	DESCRIBE DESCRIPTOR (Executable Embedded SQL)	E-28
E.21	ENABLE THREADS (Executable Embedded SQL Extension)	E-29
E.22	EXECUTE END-EXEC (Executable Embedded SQL Extension)	E-30
E.23	EXECUTE (Executable Embedded SQL)	E-31
E.24	EXECUTE DESCRIPTOR (Executable Embedded SQL	E-32
E.25	EXECUTE IMMEDIATE (Executable Embedded SQL)	E-34
E.26	FETCH (Executable Embedded SQL)	E-35
E.27	FETCH DESCRIPTOR (Executable Embedded SQL)	E-37
E.28	FREE (Executable Embedded SQL Extension)	E-39
E.29	GET DESCRIPTOR (Executable Embedded SQL)	E-40
E.30	INSERT (Executable Embedded SQL)	E-42
E.31	LOB APPEND (Executable Embedded SQL Extension)	E-45
E.32	LOB ASSIGN (Executable Embedded SQL Extension)	E-45
E.33	LOB CLOSE (Executable Embedded SQL Extension)	E-46
E.34	LOB COPY (Executable Embedded SQL Extension)	E-46
E.35	LOB CREATE TEMPORARY (Executable Embedded SQL Extension)	E-47
E.36	LOB DESCRIBE (Executable Embedded SQL Extension)	E-47
E.37	LOB DISABLE BUFFERING (Executable Embedded SQL Extension)	E-48
E.38	LOB ENABLE BUFFERING (Executable Embedded SQL Extension)	E-49
E.39	LOB ERASE (Executable Embedded SQL Extension)	E-49
E.40	LOB FILE CLOSE ALL (Executable Embedded SQL Extension)	E-50
E.41	LOB FILE SET (Executable Embedded SQL Extension)	E-50
E.42	LOB FLUSH BUFFER (Executable Embedded SQL Extension)	E-51
E.43	LOB FREE TEMPORARY (Executable Embedded SQL Extension)	E-51
E.44	LOB LOAD (Executable Embedded SQL Extension)	E-52
E.45	LOB OPEN (Executable Embedded SQL Extension)	E-52



E.47 LOB TRIM (Executable Embedded SQL Extension) E.48 LOB WRITE (Executable Embedded SQL Extension) E.49 OPEN (Executable Embedded SQL) E.50 OPEN DESCRIPTOR (Executable Embedded SQL) E.51 PREPARE (Executable Embedded SQL) E.52 ROLLBACK (Executable Embedded SQL) E.53 SAVEPOINT (Executable Embedded SQL) E.54 SELECT (Executable Embedded SQL)	E-54 E-56 E-58 E-59
<ul> <li>E.49 OPEN (Executable Embedded SQL)</li> <li>E.50 OPEN DESCRIPTOR (Executable Embedded SQL)</li> <li>E.51 PREPARE (Executable Embedded SQL)</li> <li>E.52 ROLLBACK (Executable Embedded SQL)</li> <li>E.53 SAVEPOINT (Executable Embedded SQL)</li> <li>E.54 SELECT (Executable Embedded SQL)</li> </ul>	E-58 E-59
E.50 OPEN DESCRIPTOR (Executable Embedded SQL)  E.51 PREPARE (Executable Embedded SQL)  E.52 ROLLBACK (Executable Embedded SQL)  E.53 SAVEPOINT (Executable Embedded SQL)  E.54 SELECT (Executable Embedded SQL)	E-56 E-58 E-59
E.51 PREPARE (Executable Embedded SQL) E.52 ROLLBACK (Executable Embedded SQL) E.53 SAVEPOINT (Executable Embedded SQL) E.54 SELECT (Executable Embedded SQL)	E-56 E-58 E-63
E.52 ROLLBACK (Executable Embedded SQL) E.53 SAVEPOINT (Executable Embedded SQL) E.54 SELECT (Executable Embedded SQL)	E-59
E.53 SAVEPOINT (Executable Embedded SQL) E.54 SELECT (Executable Embedded SQL)	
E.54 SELECT (Executable Embedded SQL)	E-61
` ,	
	E-62
E.55 SET DESCRIPTOR (Executable Embedded SQL)	E-65
E.56 UPDATE (Executable Embedded SQL)	E-66
E.57 VAR (Oracle Embedded SQL Directive)	E-70
E.58 WHENEVER (Embedded SQL Directive)	E-71
F.1 Compiling and Linking Pro*COBOL Applications	F-:
	F-:
F.1.1 How to Use the IDE	
F.1.1 How to Use the Animator Products	F-2
F.1.2 How to Use the Animator Products F.1.3 The COBOL and CBLLINK Commands	F-2 F-2
F.1.2 How to Use the Animator Products F.1.3 The COBOL and CBLLINK Commands F.1.4 The COBSQL Command	F-2
F.1.2 How to Use the Animator Products F.1.3 The COBOL and CBLLINK Commands F.1.4 The COBSQL Command F.2 Sample Programs	F-2 F-2
F.1.2 How to Use the Animator Products F.1.3 The COBOL and CBLLINK Commands F.1.4 The COBSQL Command F.2 Sample Programs	F-2 F-2 F-3



#### List of Tables

1-1	Directories	1-5
1-2	Header Files	1-6
2-1	Embedded SQL Statements — Directives	2-3
2-2	Embedded SQL Statements — Executable SQL	2-3
2-3	Relational Operators	2-15
4-1	Summary of Oracle Built-In Datatypes	4-2
4-2	External Data Types	4-3
4-3	Date Format	4-5
4-4	Pseudocolumns and Internal Datatypes	4-9
4-5	Functions and Internal Datatypes	4-9
4-6	Host Variable Declarations	4-13
4-7	Compatible Oracle Internal Datatypes	4-15
4-8	Globalization Support Parameters	4-31
4-9	Conversions Between Internal and External Datatypes	4-36
4-10	Formats for Date Strings	4-38
4-11	Host Variable Equivalencing	4-39
4-12	Parameters for Host Variable Equivalencing	4-41
4-13	VARNUM Examples	4-42
6-1	Legal Datatype Conversions	6-11
7-1	Host Tables Valid in SELECT INTO	7-7
7-2	Host Tables Valid in UPDATE	7-11
7-3	DB2 Array Syntax vs. Oracle Precompiler Syntax	7-20
8-1	Predefined Classes	8-4
8-2	DSNTIAR Error Codes and Their Meanings	8-12
8-3	Parameter Datatypes	8-18
8-4	SQLSTATE Codes	8-25
8-5	Status Variable Behavior with ASSUME_SQLCODE=NO and MODE=ANSI   ANSI14	
	and DECLARE_SECTION=YES	8-30
8-6	Status Variable Behavior with ASSUME_SQLCODE=YES and MODE=ANSI	
	ANSI14 and DECLARE_SECTION=YES	8-31
9-1	Appropriate Method to Use	9-3
10-1	ANSI SQL Datatypes	10-3
10-2	DYNAMIC Option Settings	10-9
10-3	TYPE_CODE Option Settings	10-10
10-4	Definitions of Descriptor Item Names	10-12



10-5	Oracle Extensions to Definitions of Descriptor Item Names	10-13
10-6	Descriptor Item Names for SET DESCRIPTOR	10-15
10-7	Extensions to Descriptor Item Names for SET DESCRIPTOR	10-15
11-1	Internal Datatypes and Related Codes	11-13
11-2	Oracle External and Related COBOL Datatypes	11-14
11-3	PL/SQL Datatype Equivalences with Internal Datatypes	11-15
11-4	Datatype Exceptions to the SQLPR2 Procedure	11-17
13-1	LOB Access Methods	13-4
13-2	Source LOB and Precompiler Datatypes	13-17
13-3	LOB Attributes	13-20
14-1	How Macro Option Values Set Micro Option Values	14-4
14-2	Option List	14-10
14-3	Formats for Date Strings	14-17
14-4	Types of Listings	14-27
14-5	Checking Done by SQLCHECK	14-42
B-1	Reserved Namespaces	B-4
C-1	HOLD_CURSOR and RELEASE _CURSOR Interactions	C-10
E-1	Source/Type Column Meaning	E-3
E-2	Precompiler Directives and Embedded SQL Statements and Clauses	E-3
F-1	Pro*COBOL Sample Programs	F-4



## **Preface**

This manual is a comprehensive user's guide and reference to the Oracle Pro\*COBOL Precompiler. It shows you how to develop COBOL programs that use the database languages SQL and PL/SQL to access and manipulate Oracle data. See *Oracle Database SQL Language Reference* and *PL/SQL User's Guide and Reference* for more information on SQL and PL/SQL.

This Preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Related Documents
- Conventions

#### Intended Audience

The *Pro\*COBOL Developer's Guide* is intended for anyone developing new COBOL applications or converting existing applications to run in the Oracle environment. Written especially for developers, this comprehensive treatment of Pro\*COBOL will also be of value to systems analysts, project managers, and others interested in embedded SQL applications.

To use this manual effectively, you need a working knowledge of the following subjects:

- Applications programming in COBOL
- The SQL database language
- Oracle concepts and terminology

# **Documentation Accessibility**

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

#### **Access to Oracle Support**

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <a href="http://www.oracle.com/pls/topic/lookup?ctx">http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs</a> if you are hearing impaired.



## **Related Documents**

For more information, see these Oracle resources:

- Oracle Database SQL Language Reference
- Oracle C++ Call Interface Developer's Guide
- Oracle Call Interface Developer's Guide

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

## Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.



# Changes in Pro\*COBOL

This section lists out the changes in Oracle Pro\*COBOL starting from Release 19c.

- New Features in Pro\*COBOL Release 23c
- New Features in Pro\*COBOL Release 21c
- Features in Pro\*COBOL Release 19c

## New Features in Pro\*COBOL Release 23c

The following feature is new in this release:

Support for boolean data type

The Pro\*COBOL Precompiler now supports the querying and binding of the SQL BOOLEAN data type. Using the boolean data type makes it easier to represent a boolean state in applications with a true or false value instead of using a character column (with Y or N).

### New Features in Pro\*COBOL Release 21c

Micro Focus Visual COBOL v6.0 compiler supports the following platforms:

- Linux x86 (32-bit and 64-bit) on OL8 & SLES15
- Windows x86 (32-bit and 64-bit)
- HPUX on Itanium (32-bit and 64-bit)

#### Features in Pro\*COBOL Release 19c

Micro Focus Visual COBOL v6.0 compiler supports the following platforms:

- Linux x86 (32-bit and 64-bit) on OL8 & SLES15
- Windows x86 (32-bit and 64-bit)
- HPUX on Itanium (32-bit and 64-bit)
- Solaris SPARC (32-bit and 64-bit)
- IBM AIX (32-bit and 64-bit)
- Linux for IBM System z (31-bit and 64-bit)



# Part I

# **Introduction and Concepts**

#### Part I contains the following chapters:

- Introduction
- Precompiler Concepts
- Database Concepts
- Datatypes and Host Variables
- Embedded SQL
- Embedded PL/SQL
- Host Tables
- Error Handling and Diagnostics
- Oracle Dynamic SQL



# Introduction

This chapter introduces you to the Pro\*COBOL Precompiler. You look at its role in developing application programs that manipulate Oracle data and find out what it enables your applications to do. The following questions are answered:

- What is Pro\*COBOL?
- The Pro\*COBOL Precompiler
- Advantages of the Pro\*COBOL Precompiler
- The SQL Language
- The PL/SQL Language
- Pro\*COBOL Features and Benefits
- Directory Structure

### 1.1 What is Pro\*COBOL?

To access an Oracle database, you use a high-level query language called Structured Query Language (SQL). You often use SQL through an interactive interface, such as SQL\*Plus.

Pro\*COBOL is a programming tool that enables you to embed SQL statements in a COBOL program. The Pro\*COBOL precompiler converts the SQL statements in the COBOL program into standard Oracle run-time library calls. The generated output file can then be compiled, linked, and run in the usual manner.

Use the Pro\*COBOL precompiler when rapid development and compatibility with other systems are your priorities.

# 1.2 The Pro\*COBOL Precompiler

The Pro\*COBOL Precompiler is a programming tool that enables you to embed SQL statements in a host COBOL program. As Figure 1-1 shows, the precompiler accepts the host program as input, translates the embedded SQL statements into standard Oracle runtime library calls, and generates a source program that you can compile, link, and execute in the usual way.



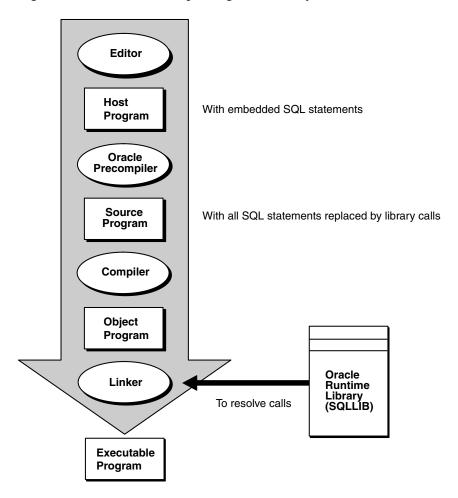


Figure 1-1 Embedded SQL Program Development



Pro\*COBOL supports the MERANT Micro Focus NetExpress version 5.0 for 32-bit Windows 2000.

#### Note:

Pro\*COBOL does not support Object Oriented COBOL (OOCOBOL) specifications.

# 1.2.1 Language Alternatives

Oracle Precompilers are available (but not on all systems) for the following high-level languages:

C/C++

- COBOL
- FORTRAN

# 1.3 Advantages of the Pro\*COBOL Precompiler

The Pro\*COBOL Precompiler lets you pack the power and flexibility of SQL into your application programs. You can embed SQL statements in COBOL. A convenient, easy to use interface lets your application access Oracle directly.

Unlike many application development tools, Pro\*COBOL lets you create highly customized applications. For example, you can create user interfaces that incorporate the latest windowing and mouse technology. You can also create applications that run in the background without the need for user interaction.

Furthermore, with Pro\*COBOL you can fine-tune your applications. It enables close monitoring of resource usage, SQL statement execution, and various run-time indicators. With this information, you can adjust program parameters for maximum performance.

# 1.4 The SQL Language

If you want to access and manipulate Oracle data, you need SQL. Whether you use SQL interactively or embedded in an application program depends on the job at hand. If the job requires the procedural processing power of COBOL, or must be done on a regular basis, use embedded SQL.

SQL has become the database language of choice because it is flexible, powerful, and easy to learn. Being non-procedural, it lets you specify what you want done without specifying how to do it. A few English-like statements make it easy to manipulate Oracle data one row or many rows at a time.

You can execute any SQL (not SQL\*Plus) statement from an application program. For example, you can:

- CREATE, ALTER, and DROP database tables dynamically.
- SELECT, INSERT, UPDATE, and DELETE rows of data.
- COMMIT or ROLLBACK transactions.

Before embedding SQL statements in an application program, you can test them interactively using SQL\*Plus. Usually, only minor changes are required to switch from interactive to embedded SQL.

# 1.5 The PL/SQL Language

An extension to SQL, PL/SQL is a transaction processing language that supports procedural constructs, variable declarations, and robust error handling. Within the same PL/SQL block, you can use SQL and all the PL/SQL extensions.

The main advantage of embedded PL/SQL is better performance. Unlike SQL, PL/SQL enables you to group SQL statements logically and send them to Oracle in a block rather than one by one. This reduces network traffic and processing overhead.

#### **Related Topics**

Embedded PL/SQL



### 1.6 Pro\*COBOL Features and Benefits

As Figure 1-2 shows, Pro\*COBOL offers many features and benefits that help you to develop effective, reliable applications.

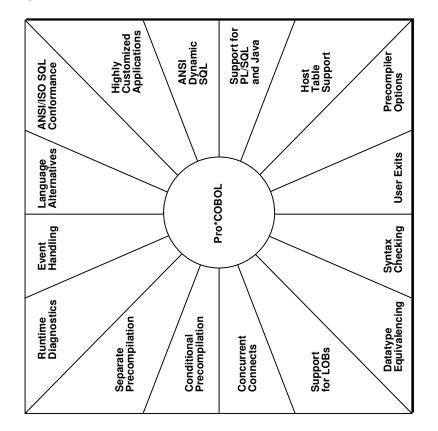


Figure 1-2 Pro\*COBOL Features and Benefits

For example, the Pro\*COBOL Precompiler enables you to:

- Write your application in COBOL.
- Conform to the ANSI/ISO embedded SQL standard.
- Take advantage of ANSI Dynamic SQL Method 4, an advanced programming technique that lets your program accept or build any valid SQL statement at runtime in a COBOL program.
- Design and develop highly customized applications.
- Convert automatically between Oracle internal datatypes and COBOL datatypes.
- Improve performance by embedding PL/SQL transaction processing blocks in your COBOL application program.
- Specify useful precompiler options and change their values during precompilation.
- Use datatype equivalencing to control the way Oracle interprets input data and formats output data.



- Precompile several program modules separately, and then link them into one executable program.
- Check the syntax and semantics of embedded SQL data manipulation statements and PL/SQL blocks.
- Access Oracle databases on multiple nodes concurrently, using Oracle Net Services.
- Use arrays as input and output program variables.
- Precompile sections of code conditionally so that your host program can run in different environments.
- Interface with tools, such as Oracle Forms and Oracle Reports, through user exits written in a high-level language.
- Handle errors and warnings with the ANSI-approved status variables SQLSTATE and SQLCODE, or the SQL Communications Area (SQLCA) and WHENEVER statement. Or both SQLSTATE and SQLCODE, and the SQL Communications Area (SQLCA) and WHENEVER statement.
- Use an enhanced set of diagnostics provided by the Oracle Communications Area (ORACA).
- Access Large Object (LOB) database types.

# 1.7 Directory Structure

When you install Pro\*COBOL, Oracle Universal Installer creates a directory called \precomp in your ORACLE BASE\ORACLE HOME directory.



The \precomp directory can contain files for other products, such as Pro\*C/C++.

The \precomp directory contains the directories listed in Table 1-1.

Table 1-1 Directories

Directory Name	Contents
\admin	Configuration files
\demo\procob2	Sample programs for Pro*COBOL
\demo\sql	SQL scripts for sample programs
\doc\procob2	Readme files for Pro*COBOL
\lib	Library files
\mesg	Message files
\public	Header files



#### 1.7.1 Header Files

The ORACLE\_BASE\ORACLE\_HOME\precomp\public directory contains the Pro\*COBOL header files listed in Table 1-2.

Table 1-2 Header Files

Header File	Description
oraca.cob	Contains the Oracle Communications Area (ORACA), which helps you to diagnose runtime errors and to monitor your program's use of various Oracle resources.
oraca5.cob	ORACA5 is the COMP-5 version of ORACA.
sqlca.cob	Contains the SQL Communications Area (SQLCA), which helps you to diagnose runtime errors. The SQLCA is updated after every executable SQL statement.
sqlca5.cob	SQLCA5 is the COMP-5 version of SQLCA.
sqlda.cob	Contains the SQL Descriptor Area (SQLDA), which is a data structure required for programs that use dynamic SQL Method 4.
sqlda5.cob	This is the COMP-5 version of SQLDA.

## 1.7.2 Library File

The <code>ORACLE\_BASE\ORACLE\_HOME</code> \precomp\lib directory contains the library file that you use when linking Pro\*COBOL applications. The library file is called <code>orasgl12.lib</code>.

Is orasql11.lib correct?

#### 1.7.3 Known Problems, Restrictions, and Workarounds

- 1. Although all Windows operating systems allow spaces in filenames and directory names, the Oracle Pro\*COBOL precompilers do not precompile files that include spaces in the file name or directory name. For example, do not use the following formats:
  - proc iname=test one.pc
  - proc iname=d:\dir1\second dir\sample1.pc
- Users running PROCOB application that are not linked using /LITLINK option and failing at runtime with an error, such as:

```
Load error: file 'ORASQL8'
```

need to copy orasql.dll to orasql8.dll in the same directory where orasql.dll exists.

**3.** The COMP5 precompiler option is not available on SPARC Solaris 64 bit computers. User the COMP option instead.



# 1.8 Compatibility, Upgrading, and Migration

The additional "array insert" and "array select" syntax will help migrating DB2 precompiler applications to the Pro\*Cobol application. This is because you will not need to change DB2 array INSERT and SELECT syntax to that of Oracle Pro\*Cobol.

The ""Implicit Buffered Insert" feature supported by Pro\*Cobol helps you to migrate DB2 precompiler applications to Pro\*Cobol applications without using the array syntax of Pro\*Cobol for better performance.



# **Precompiler Concepts**

This chapter explains how embedded SQL programs do their work. It presents definitions of important words, explanations of basic concepts, and key rules.

#### Topics covered are:

- · Key Concepts of Embedded SQL Programming
- Programming Guidelines
- The Declare Section
- Nested Programs
- Conditional Precompilations
- Separate Precompilations
- Compiling and Linking
- Sample DEPT and EMP Tables
- Sample EMP Program: SAMPLE1.PCO

# 2.1 Key Concepts of Embedded SQL Programming

This section lays the conceptual foundation on which later chapters build.

## 2.1.1 Steps in Developing an Embedded SQL Application

Precompiling results in a source file that can be compiled normally. Although precompiling adds a step to the traditional development process, that step is well worth taking because it lets you write very flexible applications.

Figure 2-1 walks you through the embedded SQL application development process:

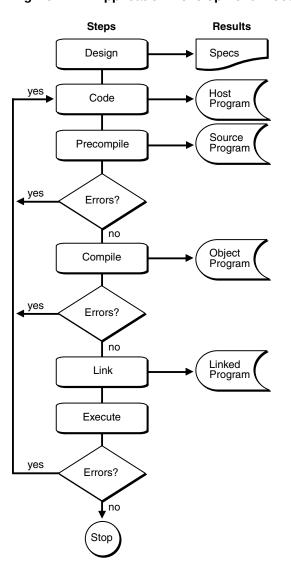


Figure 2-1 Application Development Process

## 2.1.2 Embedded SQL Statements

The term *embedded SQL* refers to SQL statements placed within an application program. Because the application program houses the SQL statements, it is called a *host program*, and the language in which it is written is called the *host language*. For example, with Pro\*COBOL you can embed SQL statements in a COBOL host program.

To manipulate and query Oracle data, you use the INSERT, UPDATE, DELETE, and SELECT statements. INSERT adds rows of data to database tables, UPDATE modifies rows, DELETE removes unwanted rows, and SELECT retrieves rows that meet your search criteria.

Only SQL statements—not SQL\*Plus statements—are valid in an application program. (SQL\*Plus has additional statements for setting environment parameters, editing, and report formatting.)

#### 2.1.2.1 Executable versus Declarative Statements

Embedded SQL includes all the interactive SQL statements plus others that allow you to transfer data between Oracle and a host program. There are two types of embedded SQL statements: *executable statements* and *directives*.

Executable SQL statements generate calls to the database. They include almost all queries, Data Manipulation Language (DML), data definition language (DDL), and Data Control Language (DCL) statements.

*Directives*, on the other hand, do not result in calls to SQLLIB and do not operate on Oracle data.

You use directives to declare Oracle objects, communications areas, and SQL variables. They can be placed wherever COBOL declarations can be placed.

Table 2-1 groups some examples of embedded SQL statements (not a complete list.)

Table 2-1 Embedded SQL Statements — Directives

STATEMENT	PURPOSE
ARRAYLEN*	To use host tables with PL/SQL
BEGIN DECLARE SECTION*	To declare host variables
END DECLARE SECTION*	
DECLARE*	To name Oracle objects
INCLUDE*	To copy in files
VAR*	To equivalence variables
WHENEVER*	To handle runtime errors

Table 2-2 Embedded SQL Statements — Executable SQL

STATEMENT	PURPOSE
ALLOCATE*	To define and control Oracle data
ALTER	
CONNECT*	
CREATE	
DROP	
GRANT	
NOAUDIT	
RENAME	
REVOKE	
TRUNCATE	



Table 2-2 (Cont.) Embedded SQL Statements — Executable SQL

STATEMENT	PURPOSE
CLOSE*	To query and manipulate Oracle data
DELETE	
EXPLAIN PLAN	
FETCH*	
INSERT	
LOCK TABLE	
OPEN*	
SELECT	
UPDATE	
COMMIT	To process transactions
ROLLBACK	
SAVEPOINT	
SET	
TRANSACTION	
DESCRIBE*	To use dynamic SQL
EXECUTE*	
PREPARE*	
ALTER	To control sessions
SESSION	
SET ROLE	

<sup>\*</sup> Has no interactive counterpart



Embedded SQL Statements and Precompiler Directives contains a presentation of the most important statements and directives.

## 2.1.3 Embedded SQL Syntax

In your application program, you can freely intermix SQL statements with host-language statements and use host-language variables in SQL statements. The only special requirement for building SQL statements into your host program is that you begin them with the words EXEC SQL and end them with the token END-EXEC. Pro\*COBOL translates all executable EXEC SQL statements into calls to the runtime library SQLLIB.

Most embedded SQL statements differ from their interactive counterparts only through the addition of a new clause or the use of program variables. Compare the following interactive and embedded ROLLBACK statements:

ROLLBACK WORK; -- interactive

\* embedded EXEC SOL



```
ROLLBACK WORK END-EXEC.
```

A period or any other terminator can follow a SQL statement. Either of the following is allowed:

```
EXEC SQL ... END-EXEC, EXEC SQL ... END-EXEC.
```

### 2.1.4 Static Versus Dynamic SQL Statements

Most application programs are designed to process *static* SQL statements and fixed transactions. In this case, you know the makeup of each SQL statement and transaction before run time. That is, you know which SQL commands will be issued, which database tables might be changed, which columns will be updated, and so on.

However, some applications are required to accept and process any valid SQL statement at run time. In this case you might not know until run time all the SQL commands, database tables, and columns involved.

*Dynamic* SQL is an advanced programming technique that lets your program accept or build SQL statements at run time and take explicit control over datatype conversion.

#### **Related Topics**

- Embedded SOL
- Oracle Dynamic SQL
- ANSI Dynamic SQL
- Oracle Dynamic SQL: Method 4

### 2.1.5 Embedded PL/SQL Blocks

Pro\*COBOL treats a PL/SQL block like a single embedded SQL statement, so you can place a PL/SQL block anywhere in an application program that you can place a SQL statement. To embed PL/SQL in your host program, you simply declare the variables to be shared with PL/SQL and bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END-EXEC.

From embedded PL/SQL blocks, you can manipulate Oracle data flexibly and safely because PL/SQL supports all SQL data manipulation and transaction processing commands.

#### **Related Topics**

Embedded PL/SQL

## 2.1.6 Host Variables and Indicator Variables

A *host variable* is a scalar or table variable or group item declared in the COBOL language and shared with Oracle, meaning that both your program and Oracle can reference its value. Host variables are the key to communication between Oracle and your program.

You use *input* host variables to pass data to the database. You use *output* host variables to pass data and status information from the database to your program.

Host variables can be used anywhere an expression can be used. In SQL statements, host variables must be prefixed with a colon ':' to set them apart from database schema names.



You can associate any host variable with an optional indicator variable. An *indicator variable* is an integer variable that indicates the value or condition of its host variable. A *NULL* is a missing, an unknown, or an inapplicable value. You use indicator variables to assign NULLs to input host variables and to detect NULLs in output variables or truncated values in output character host variables.

A host variable must *not* be:

- prefixed with a colon in COBOL statements
- used in data definition (DDL) statements such as ALTER and CREATE

In SQL statements, an indicator variable must be prefixed with a colon and appended to its associated host variable (to improve readability, you can precede the indicator variable with the optional keyword INDICATOR).

Every program variable used in a SQL statement must be declared according to the rules of the COBOL language. Normal rules of scope apply. COBOL variable names can be any length, but only the first 30 characters are significant for Pro\*COBOL. Any valid COBOL identifier can be used as a host variable identifier, including those beginning with digits.

The external datatype of a host variable and the internal datatype of its source or target database column need not be the same, but they must be compatible.



Datatype Conversion for list of compatible datatypes between which Oracle converts automatically when necessary.

## 2.1.7 Oracle Datatypes

Typically, a host program inputs data to the database, and the database outputs data to the program. Oracle inserts input data into database tables and selects output data into program host variables. To store a data item, Oracle must know its *datatype*, which specifies a storage format and valid range of values.

Oracle recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle stores data in database columns. Oracle also uses internal datatypes to represent database pseudo-columns, which return specific data items but are not actual columns in a table.

External datatypes specify how data is stored in host variables. When your host program inputs data to Oracle, it does any conversion between the external datatype of the input host variable and the internal datatype of the database column. When Oracle outputs data to your host program, if necessary, Oracle converts between the internal datatype of the database column and the external datatype of the output host variable.





You can override default datatype conversions by using dynamic SQL Method 4 or datatype equivalencing. For information about datatype equivalencing, see "Explicit Control Over DATE String Format".

#### 2.1.8 Tables

Pro\*COBOL lets you define table host variables (called *host tables*) and operate on them with a single SQL statement. Using the SELECT, FETCH, DELETE, INSERT, and UPDATE statements, you can query and manipulate large volumes of data with ease.



Host Tables for a complete discussion of host tables.

## 2.1.9 Errors and Warnings

When you execute an embedded SQL statement, it either succeeds or fails, and might result in an error or warning. You need a way to handle these results. Pro\*COBOL provides the following error handling mechanisms:

- SQLCODE status variable
- SQLSTATE status variable
- SQL Communications Area (SQLCA)
- WHENEVER statement
- Oracle Communications Area (ORACA)

### 2.1.9.1 SQLCODE/SQLSTATE Status Variables

After executing a SQL statement, the Oracle Server returns a status code to a variable named SQLCODE or SQLSTATE. The status code indicates whether the SQL statement executed successfully or caused an error or warning condition.

### 2.1.9.2 SQLCA Status Variable

The SQLCA is a data structure that defines program variables used by Oracle to pass runtime status information to the program. With the SQLCA, you can take different actions based on feedback from Oracle about work just attempted. For example, you can check to see if a DELETE statement succeeded and, if so, how many rows were deleted.

The SQLCA provides for diagnostic checking and event handling. At runtime, the SQLCA holds status information passed to your program by Oracle. After executing a SQL statement, Oracle sets SQLCA variables to indicate the outcome, as illustrated in Figure 2-2.



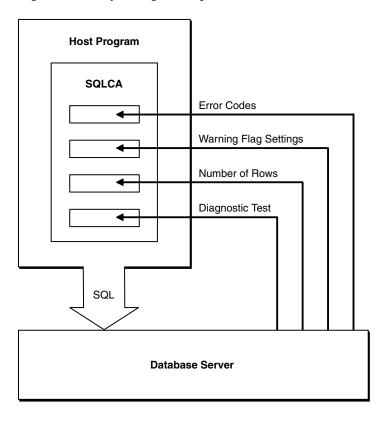


Figure 2-2 Updating the SQLCA

You can check to see if an INSERT, UPDATE, or DELETE statement succeeded and if it did, how many rows were affected. Or, if the statement failed, you can get more information about what happened.

When MODE={ANSI13 | ORACLE}, you must declare the SQLCA by hard-coding it or by copying it into your program with the INCLUDE statement.



"Using the SQL Communications Area" for more information about how to declare and use the SQLCA.

#### 2.1.9.3 WHENEVER Statement

With the WHENEVER statement, you can specify actions to be taken automatically when Oracle detects an error or warning condition. These actions include continuing with the next statement, calling a subprogram, branching to a labeled statement, performing a paragraph, or stopping.

#### 2.1.9.4 ORACA

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA. The ORACA is a data structure that handles Oracle communication. It contains cursor statistics, information about the current SQL statement, option settings, and system statistics.



### 2.1.9.5 Precompiler Options and Error Handling

Oracle returns the success or failure of SQL statements in status variables, SQLSTATE and SQLCODE. With precompiler option MODE=ORACLE, you use SQLCODE, declared by including SQLCA. With MODE=ANSI, either SQLSTATE or SQLCODE must be declared, but SQLCA is not necessary.

#### **Related Topics**

· Error Handling and Diagnostics

## 2.1.10 SQL99 Syntax Support

The SQL standard enables the portability of SQL applications across all conforming software products. Oracle features are compliant with the ANSI/ISO SQL99 standard, including ANSI compliant joins. Pro\*Cobol supports all SQL99 features that are supported by Oracle database, which means that the SQL99 syntax for the SELECT, INSERT, DELETE, and UPDATE statements and the body of the cursor in a DECLARE CURSOR statement are supported.

# 2.2 Programming Guidelines

This section deals with embedded SQL syntax, coding conventions, and Pro\*COBOL-specific features and restrictions. Topics are arranged alphabetically for quick reference.

### 2.2.1 Abbreviations

You can use the standard COBOL abbreviations, such as PIC for PICTURE IS and COMP for USAGE IS COMPUTATIONAL.

## 2.2.2 Case-Insensitivity

Pro\*COBOL precompiler options and values as well as all EXEC SQL statements, inline commands, and COBOL statements are case-insensitive. The precompiler accepts both upper- and lower-case tokens.

## 2.2.3 COBOL Versions Supported

Pro\*COBOL supports the standard implementation of COBOL for your operating system (usually COBOL-85 or COBOL-74). Some platforms may support both COBOL implementations. For more information, see your Oracle system-specific documentation.

## 2.2.4 Coding Areas

The precompiler option FORMAT, specifies the format of your source code. If you specify FORMAT=ANSI (the default), you are conforming as much as possible to the ANSI standard. In this format, columns 1 through 6 can contain an optional sequence number, and column 7 (indicator area) can indicate comments or continuation lines.

Division headers, section headers, paragraph names, FD and 01 statements begin in columns 8 through 11 (area A). Other statements, including EXEC SQL and EXEC ORACLE



statements, must be placed in area B (columns 12 through 72). These guidelines for source code format can be overridden by your compiler's rules.

If you specify FORMAT=TERMINAL, COBOL statements can begin in column 1 (the left-most column), or column 1 can be the indicator area. This format is also subject to the rules of your compiler.

You can specify FORMAT=VARIABLE to allow Flexible B Area Length.

Consult your COBOL compiler documentation for your own platform to determine the actual acceptable formats for COBOL statements.



In this manual, COBOL code examples use the FORMAT=TERMINAL setting. The online sample programs in the demo directory use FORMAT=ANSI.

## 2.2.5 Commas

In SQL, you must use commas to separate list items, as the following example shows:

```
EXEC SQL SELECT ENAME, JOB, SAL
INTO :EMP-NAME, :JOB-TITLE, :SALARY
FROM EMP
WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

In COBOL, you can use commas or blanks to separate list items. For example, the following two statements are equivalent:

```
ADD AMT1, AMT2, AMT3 TO TOTAL-AMT. ADD AMT1 AMT2 AMT3 TO TOTAL-AMT.
```

## 2.2.6 Comments

You can place COBOL comment lines within SQL statements. COBOL comment lines start with an asterisk (\*) in the indicator area.

You can also place ANSI SQL-style comments starting with "--" within SQL statements at the end of a line (but not after the last line of the SQL statement).

COBOL comments continue for the rest of the line after these two characters: "\*>".

You can place C-style comments (/\* ... \*/) in SQL statements.

The following example shows all four styles of comments:

```
MOVE 12 TO DEPT-NUMBER. *> This is the software development group.

EXEC SQL SELECT ENAME, SAL

* assign column values to output host variables

INTO :EMP-NAME, :SALARY -- output host variables

/* column values assigned to output host variables */

FROM EMP

WHERE DEPTNO = :DEPT-NUMBER

END-EXEC. -- illegal Comment
```



You cannot nest comments or place them on the last line of a SQL statement after the terminator END-EXEC.

### 2.2.7 Continuation Lines

You can continue SQL statements from one line to the next, according to the rules of COBOL, as this example shows:

```
EXEC SQL SELECT ENAME, SAL INTO :EMP-NAME, :SALARY FROM EMP
WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

No continuation indicator is needed.

To continue a string literal from one line to the next, code the literal through column 72. On the next line, code a hyphen (-) in column 7, a quote in column 12 or beyond, and then the rest of the literal. An example follows:

```
WORKING STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 UPDATE-STATEMENT PIC X(80) VALUE "UPDATE EMP SET BON
- "US = 500 WHERE DEPTNO = 20".
EXEC SQL END DECLARE SECTION END-EXEC.
```

## 2.2.8 Copy Statements

Copy statements are not parsed by Pro\*COBOL. Therefore, files included with the COPY command should not contain definitions of Host variables or contain embedded SQL statements. Instead, use the INCLUDE precompiler statement which is documented in "Using the INCLUDE Statement". Be careful when using INCLUDE and also using DECLARE\_SECTION=YES. Group items should be either placed all inside or all outside of a Declare Section.

### 2.2.9 Decimal-Point is Comma

Pro\*COBOL supports the DECIMAL-POINT IS COMMA clause in the ENVIRONMENT DIVISION. If the DECIMAL-POINT IS COMMA clause appears in the source file, then the comma will be allowed as the symbol beginning the decimal part of any numeric literals in the VALUE clauses.

For example, the following is allowed:

```
IDENTIFICATION DIVISION.

PROGRAM-ID. FOO

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SPECIAL-NAMES.

DECIMAL-POINT IS COMMA.

DATA DIVISION.

WORKING-STORAGE SECTION.

O1 WDATA1 PIC S9V999 VALUE +,567. *> <--- **

01 WDATA2 PIC S9V999 VALUE -,234. *> <--- **
```



#### 2.2.10 Delimiters

The LITDELIM option specifies the delimiters for COBOL string constants and literals. If you specify LITDELIM=APOST, Pro\*COBOL uses apostrophes when generating COBOL code. If you specify LITDELIM=QUOTE (default), quotation marks are used, as in

```
CALL "SQLROL" USING SQL-TMP0.
```

In SQL statements, you must use quotation marks to delimit identifiers containing special or lowercase characters, as in

```
EXEC SQL CREATE TABLE "Emp2" END-EXEC.
```

However, you must use apostrophes to delimit string constants, as in

```
EXEC SQL SELECT ENAME FROM EMP WHERE JOB = 'CLERK' END-EXEC.
```

Regardless of which delimiter is used in the Pro\*COBOL source file, Pro\*COBOL generates the delimiter specified by the LITDELIM value.

## 2.2.11 Division Headers that are Optional

The following division headers are optional:

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION
- DATA DIVISION

Note that the PROCEDURE DIVISION header is not optional. The following source can be precompiled:

```
*IDENTIFICATION DIVISION header is optional PROGRAM-ID. HELLO.

*ENVIRONMENT DIVISION header is optional CONFIGURATION SECTION.

*DATA DIVISION header is optional WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
DISPLAY "Hello World!".
STOP RUN.
```

## 2.2.12 Embedded SQL Syntax

To use a SQL statement in your Pro\*COBOL program, precede the SQL statement with the EXEC SQL clause, and end the statement with the END-EXEC keyword.



Oracle Database SQL Language Reference. for more information about Embedded SQL syntax.

## 2.2.13 Figurative Constants

Figurative constants, such as HIGH-VALUE, ZERO, and SPACE, cannot be used in SQL statements. For example, the following is *invalid*:

```
EXEC SQL DELETE FROM EMP WHERE COMM = ZERO END-EXEC.
```

Instead, use the following:

```
EXEC SQL DELETE FROM EMP WHERE COMM = 0 END-EXEC.
```

## 2.2.14 File Length

Pro\*COBOL cannot process arbitrarily long source files. Some of the variables used internally limit the size of the generated file. There is no absolute limit to the number of lines allowed, but the following aspects of the source file are contributing factors to the file-size constraint:

- Complexity of the embedded SQL statements (for example, the number of bind and define variables)
- Whether a database name is used (for example, connecting to a database with an AT clause)
- Number of embedded SQL statements

To prevent problems related to this limitation, use multiple program units to sufficiently reduce the size of the source files.

## 2.2.15 FILLER is Allowed

The word FILLER is allowed in host variable declarations. The word FILLER is used to specify an elementary item of a group that cannot be referred to explicitly. The following declaration is valid:

```
01 STOCK.

05 DIVIDEND PIC X(5).

05 FILLER PIC X.

05 PRICE PIC X(6).
```

#### 2.2.16 Host Variable Names

Any valid standard COBOL identifier can be used as a host variable. Variable names can be any length, but only the first 30 characters are significant. The maximum number of significant characters recognized by COBOL compilers is 128.

For portability, you may wish to restrict the length of host variable names to 18 or fewer characters (the length mandated by the SQL standard).

```
✓ See Also:
```

Reserved Words\_ Keywords\_ and Namespaces for a list of words that have restrictions on their use in applications.



## 2.2.17 Hyphenated Names

You can use hyphenated host-variable names in static SQL statements but *not* in dynamic SQL. For example, the following usage is *invalid*:

```
MOVE "DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER" TO SQLSTMT. EXEC SQL PREPARE STMT1 FROM SQLSTMT END-EXEC.
```

#### 2.2.18 Level Numbers

When declaring host variables, you can use level numbers 01 through 49, and 77. Pro\*COBOL does not allow variables containing the VARYING clause or pseudo-type variables (these datatypes are prefixed with "SQL-") to be declared level 49 or 77.

#### 2.2.19 MAXLITERAL Default

With the MAXLITERAL option, you can specify the maximum length of string literals generated by Pro\*COBOL, so that compiler limits are not exceeded. For Pro\*COBOL, the default value is 256, but you might have to specify a lower value. On IBM-proprietary S370 operating systems it is 120.

## 2.2.20 Multibyte Datatypes

ANSI standard National Character Set datatypes are supported for handling multibyte character data. The PIC N or PIC G clause, if supported by your compiler, defines variables that store fixed-length NCHAR strings. You can store variable-length, multibyte National Character Set strings using COBOL group items consisting of a length field and a string field.

The environmental variable NLS\_NCHAR is available to specify a client-side Globalization Support National Character Set.

#### **Related Topics**

VARCHAR Variables

## 2.2.21 NULLs in SQL

In SQL, a NULL represents a missing, unknown, or inapplicable column value; it equates neither to zero nor to a blank. Use the NVL function to convert NULLs to non-NULL values, use the IS [NOT] NULL comparison operator to search for NULLs, and use indicator variables to insert and test for NULLs.

## 2.2.22 Paragraph and Section Names

You can associate standard COBOL paragraph and section names with SQL statements, as shown in the following example:

```
LOAD-DATA.

EXEC SQL

INSERT INTO EMP (EMPNO, ENAME, DEPTNO)

VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)

END-EXEC.
```



Also, you can reference paragraph and section names in a WHENEVER ... DO or WHENEVER ... GOTO statement, as the next example shows:

```
PROCEDURE DIVISION.

MAIN.

EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR END-EXEC.

...

SQL-ERROR SECTION.

...
```

You must begin all paragraph names in area A.

#### 2.2.23 REDEFINES Clause

You can use the COBOL REDEFINES clause to redefine group or elementary items. For example, the following declarations are valid:

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

11 REC-ID PIC X(4).

12 REC-NUM REDEFINES REC-ID PIC S9(4) COMP.

EXEC SQL END DECLARE SECTION END-EXEC.
```

#### And:

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

O1 STOCK.

O5 DIVIDEND PIC X(5).

O5 PRICE PIC X(6).

O1 BOND REDEFINES STOCK.

O5 COUPON-RATE PIC X(4).

O5 PRICE PIC X(7).

EXEC SQL END DECLARE SECTION END-EXEC.
```

Pro\*COBOL issues no warning or error if a single INTO clause uses items from both a group item host variable and from its re-definition.

## 2.2.24 Relational Operators

COBOL relational operators differ from their SQL equivalents, as shown in Relational Operators. Furthermore, COBOL enables the use of words instead of symbols, whereas SQL does not.

Table 2-3 Relational Operators

SQL Operators	COBOL Operators
=	=, EQUAL TO
< >, !=, ^=	NOT=, NOT EQUAL TO
>	>, GREATER THAN
<	<, LESS THAN
>=	>=, GREATER THAN OR EQUAL TO
<=	<=, LESS THAN OR EQUAL TO



#### 2.2.25 Sentence Terminator

A COBOL *sentence* includes one or more COBOL or SQL statements, or both of them, and ends with a period. In conditional sentences, only the last statement must end with a period, as the following example shows.

```
IF EMP-NUMBER = ZERO

MOVE FALSE TO VALID-DATA

PERFORM GET-EMP-NUM UNTIL VALID-DATA = TRUE

ELSE

EXEC SQL DELETE FROM EMP

WHERE EMPNO = :EMP-NUMBER

END-EXEC

ADD 1 TO DELETE-TOTAL.

END-IF.
```

SQL statements may be ended by a comma, a period, or another COBOL statement.

## 2.3 The Declare Section

Passing data between the database server and your application program requires host variables and error handling. This section shows you how to meet these requirements.

#### 2.3.1 Contents of a Declare Section

A Declare Section begins with the statement:

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

and ends with the statement:

```
EXEC SQL END DECLARE SECTION END-EXEC.
```

Between these two statements only the following are allowed:

- Host-variable and indicator-variable declarations
- Non-host COBOL variables
- EXEC SQL DECLARE statements
- EXEC SQL INCLUDE statements
- EXEC SQL VAR statements
- EXEC ORACLE statements
- COBOL comments

#### 2.3.1.1 An Example

In the following example, you declare four host variables for use later in your program.

```
WORKING-STORAGE SECTION.

...

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

...

01 EMP-NUMBER PIC 9(4) COMP VALUE ZERO.

01 EMP-NAME PIC X(10) VARYING.
```



```
01 SALARY PIC S9(5)V99 COMP-3 VALUE ZERO.
01 COMMISSION PIC S9(5)V99 COMP-3 VALUE ZERO.
EXEC SQL END DECLARE SECTION END-EXEC.
```

## 2.3.2 Precompiler Option DECLARE\_SECTION

The Declare Section is optional. For backward compatibility with releases prior to 8.0, for which it was required, Pro\*COBOL provides a command-line precompiler option for explicit control over whether only declarations in the Declare Section are allowed as host variables. This option is:

DECLARE\_SECTION={YES | NO} (default is NO)

You must use the DECLARE\_SECTION option on the command line or in a configuration file.

When MODE=ORACLE and DECLARE\_SECTION=YES, only variables declared inside the Declare Section are allowed as host variables. When MODE=ANSI then DECLARE SECTION is implicitly set to YES.

If DECLARE\_SECTION is set to YES, then you must declare all program variables used in SQL statements inside the Declare Section. If DECLARE\_SECTION is set to NO, then it is optional to use a Declare Section. In this case, declarations of host variables and indicator variables can be made either inside or outside a Declare Section. This optional behavior is a change from Release 8.0 and earlier releases.

Multiple Declare Sections are allowed for each precompiled unit. Furthermore, a host program can contain several independently precompiled units.

#### See Also:

- "Macro and Micro Options" for more information about macro and micro options.
- "DECLARE SECTION" for more details of the option.

### 2.3.3 Using the INCLUDE Statement

The INCLUDE statement lets you copy files into your host program, as the following example shows:

Copy in the SQL Communications Area (SQLCA)
 EXEC SQL INCLUDE SQLCA END-EXEC.
 Copy in the Oracle Communications Area (ORACA)
 EXEC SQL INCLUDE ORACA END-EXEC.

You can INCLUDE any file. When you precompile your Pro\*COBOL program, each EXEC SQL INCLUDE statement is replaced by a copy of the file named in the statement.

#### 2.3.3.1 Filename Extensions

If your system uses file extensions but you do not specify one, Pro\*COBOL assumes the default extension for source files (usually COB). For more information, see your Oracle system-specific documentation.



#### 2.3.3.2 Search Paths

If your system uses directories, you can set a search path for included files using the INCLUDE option, as follows:

INCLUDE=path

where path defaults to the current directory.

Pro\*COBOL first searches the current directory, then the directory specified by the INCLUDE option, and finally the directory for standard INCLUDE files. You need not specify a path for standard files such as the SQLCA and ORACA. However, a path is required for nonstandard files unless they are stored in the current directory.

You can also specify multiple paths on the command line, as follows:

```
... INCLUDE=<path1> INCLUDE=<path2> ...
```

When multiple paths are specified, Pro\*COBOL searches the current directory first, then the *path1* directory, then the *path2* directory, and so on. The directory containing standard INCLUDE files is searched last. The path syntax is system specific. For more information, see your Oracle system-specific documentation.

Remember that Pro\*COBOL searches for a file in the current directory first even if you specify a search path. If the file you want to INCLUDE is in another directory, make sure no file with the same name is in the current directory or any other directory that precedes it in the search path. If your operating system is case sensitive, be sure to specify the same upper/lowercase filename under which the file is stored.

# 2.4 Nested Programs

Nesting programs in COBOL means that you place one program inside another. The contained programs may reference some of the resources of the programs that contain them. The names within the higher-level program and the nested program can be the same, and describe different data items without conflict, because the names are known only within the programs. However, names described in the Configuration Section of the higher-level program can be referenced in the nested program.

Some compilers do not support the GLOBAL clause. Pro\*COBOL supports nested programs by generating code that contains GLOBAL clauses. To avoid generating GLOBAL clauses unconditionally, specify the precompiler option NESTED=NO. NESTED (=YES or NO) defaults to YES and can be used in configuration files, or on the command line, but not inline (EXEC ORACLE statement).

The higher-level program can contain several nested programs. Likewise, nested programs can have programs nested within them. You must place the nested program directly before the END PROGRAM header of the program in which it is nested.

You can call a nested program only by a program in which it is either directly or indirectly nested. If you want a nested program to be called by any program, even one on a different branch of the nested tree structure, you code the COMMON clause in the PROGRAM-ID paragraph of the nested program. You can code COMMON only for nested programs:

PROGRAM-ID. <nested-program-name> COMMON.



You can code the GLOBAL phrase for File Definitions and level 01 data items (any subordinate items automatically become global). This enables them to be referenced in all subprograms directly or indirectly contained within them. You code GLOBAL on the higher-level program. If the nested program defines the same name as one declared GLOBAL in a higher-level program, COBOL uses the declaration within the nested program. If the data item contains a REDEFINES clause, GLOBAL must follow it.

```
FD file-name GLOBAL ...
01 data-name1 GLOBAL ...
01 data-name2 REDEFINES data-name3 GLOBAL ...
```

#### **Related Topics**

NESTED

## 2.4.1 Support for Nested Programs

Pro\*COBOL enables you to store nested programs with embedded SQL within a single source file. All 01 level items, which are marked as global in a containing program and are valid host variables at the containing program level, are usable as valid host variables in any programs directly or indirectly contained by the containing program. Consider the following example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAINPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 REC1 GLOBAL.
         05 VAR1 PIC X(10).
         05 VAR2 PIC X(10).
01 VAR1 PIC X(10) GLOBAL.
    EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    <main program statements>
IDENTIFICATION DIVISION.
   PROGRAM-ID. NESTEDPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 VAR1 PIC S9(4).
PROCEDURE DIVISION.
    EXEC SQL SELECT X, Y INTO : REC1 FROM ... END-EXEC.
    EXEC SQL SELECT X INTO :VAR1 FROM ... END-EXEC.
    EXEC SQL SELECT X INTO :REC1.VAR1 FROM ... END-EXEC.
END PROGRAM NESTEDPROG.
END PROGRAM MAINPROG.
```



The main program declares the host variable REC1 as global, and thus the nested program can use REC1 in the first select statement without having to declare it. Since VAR1 is declared as a global variable and also as a local variable in the nested program, the second select statement will use the VAR1 declared as S9(4), overriding the global declaration. In the third select statement, the global VAR1 of REC1 declared as PIC X(10) is used.

The previous paragraph describes the results when DECLARE\_SECTION=NO is used. When DECLARE\_SECTION=YES, Pro\*COBOL will not recognize host variables unless they are declared inside a Declare Section. If the preceding program is precompiled with DECLARE\_SECTION=YES, then the second select statement would result in an ambiguous host variable error. The first and third select statements would function the same.

Note: Recursive nested programs are not supported

#### 2.4.1.1 Declaring the SQLCA

For information on declaring the SQLCA for nested programs, the included SQLCA definition provided will be declared as global, so the declaration of SQLCA is only required in the higher-level program. The SQLCA can change each time a new SQL statement is executed. The SQLCA provided can always be modified to remove the global specification if you want to declare additional SQLCA areas in the nested programs. This also applies to SQLDA and ORACA.

#### **Related Topics**

SQLCA Status Variable

### 2.4.1.2 Nested Program Example

See SAMPLE13.PCO in the demo directory.

# 2.5 Conditional Precompilations

Conditional precompilation includes (or excludes) sections of code in your host program based on certain conditions. For example, you might want to include one section of code when precompiling under UNIX and another section when precompiling under VMS. Conditional precompilation lets you write programs that can run in different environments.

Conditional sections of code are marked by statements that define the environment and actions to take. You can code host-language statements as well as EXEC SQL statements in these sections. The following statements let you exercise conditional control over precompilation:

```
* -- define a symbol

EXEC ORACLE DEFINE symbol END-EXEC.

* -- if symbol is defined

EXEC ORACLE IFDEF symbol END-EXEC.

* -- if symbol is not defined

EXEC ORACLE IFNDEF symbol END-EXEC.

* -- otherwise

EXEC ORACLE ELSE END-EXEC.

* -- end this control block

EXEC ORACLE ENDIF END-EXEC.
```



A conditional statement must be terminated with END-EXEC.



The conditional compilation feature of your compiler may not be supported by Pro\*COBOL.

## 2.5.1 An Example

In the following example, the SELECT statement is precompiled only when the symbol *SITE2* is defined:

```
EXEC ORACLE IFDEF SITE2 END-EXEC.

EXEC SQL SELECT DNAME

INTO :DEPT-NAME

FROM DEPT

WHERE DEPTNO = :DEPT-NUMBER

EXEC ORACLE ENDIF END-EXEC.
```

Blocks of conditions can be nested as shown in the following example:

```
EXEC ORACLE IFDEF OUTER END-EXEC.

EXEC ORACLE IFDEF INNER END-EXEC.

EXEC ORACLE ENDIF END-EXEC.

EXEC ORACLE ENDIF END-EXEC.
```

You can "Comment out" host-language or embedded SQL code by placing it between IFDEF and ENDIF and *not* defining the symbol.

## 2.5.2 Defining Symbols

You can define a symbol in two ways. Either include the statement

```
EXEC ORACLE DEFINE symbol END-EXEC.
```

in your host program or define the symbol on the command line using the syntax

```
... INAME=filename ... DEFINE=symbol
```

where symbol is not case-sensitive.

Some port-specific symbols are predefined for you when Pro\*COBOL is installed on your system. For example, predefined operating system symbols include CMS, MVS, UNIX, and VMS.



A DEFINE SYMBOL name beginning with a number, or that includes a dash, causes an unrecoverable error during precompilation. You can use underscores, and you can use numbers other than at the beginning of a DEFINE SYMBOL name.

# 2.6 Separate Precompilations

You can precompile several COBOL program modules separately and then link them into one executable program. This supports modular programming, which is required when the functional components of a program are written and debugged by different developers. The individual program modules need not be written in the same language.

#### 2.6.1 Guidelines

The following guidelines will help you avoid some common problems.

### 2.6.1.1 Referencing Cursors

Cursor names are SQL identifiers, whose scope is the precompilation unit. Hence, cursor operations cannot span precompilation units (files). That is, you cannot declare a cursor in one file and open or fetch from it in another file, so when doing a separate precompilation, make sure all definitions and references to a given cursor are in one file

### 2.6.1.2 Specifying MAXOPENCURSORS

When you precompile the program module that connects to Oracle, specify a value for MAXOPENCURSORS that is high enough for any of the program modules. If you use it for another program module, MAXOPENCURSORS is ignored. Only the value in effect for the connect is used at run time.

### 2.6.1.3 Using a Single SQLCA

If you want to use just one memory area for the SQLCA, the process for doing so depends on which version of Pro\*COBOL you are using.

#### 2.6.1.3.1 Version 1.8 and Lower

You must declare the SQLCA globally. You can do this by modifying the SQLCA.COB file, changing the line

```
01 SQLCA.
```

to

01 SQLCA EXTERNAL.

Alternatively, you can include a hard-coded definition for SQLCA, copied from SQLCA.cob and make the aforementioned change. Note that you still have to include a definition of SQLCA in each precompiled unit.

#### 2.6.1.3.2 Version 2 and Later

In later versions of Pro\*COBOL, the SQLCA is not copied from the file system. It is generated by the precompiler. If you need to share the SQLCA structure in these versions, instead of this statement:

EXEC SQL INCLUDE SQLCA END-EXEC



#### you should use the following code:

```
EXEC SQL INCLUDE SQLCA.ANX END-EXEC
```

This causes the precompiler to generate the SQLCA structure with the EXTERNAL keyword added automatically.

### 2.6.1.4 Using a Single DATE FORMAT

You must use the same format string for DATE in each program module.

#### 2.6.2 Restrictions

All references to an explicit cursor must be in the same program file. You cannot perform operations on a cursor that was declared in a different module. See Chapter 4 for more information about cursors.

Also, any program file that contains SQL statements must have a SQLCA that is in the scope of the local SQL statements.

# 2.7 Compiling and Linking

To get an executable program, you must compile the source file(s) produced by Pro\*COBOL, then link the resulting object module with any modules needed from SQLLIB and system-specific Oracle libraries.

The linker resolves symbolic references in the object modules. If these references conflict, the link fails. Such conflicts can happen when you try to link third party software into a precompiled program. Not all third-party software is compatible with Oracle, so you might have problems. Check with Oracle Support Services to see if the software is supported.

Compiling and linking are system-dependent. For example, on some systems, you must turn off compiler optimization when compiling a host language program. For instructions, see your system-specific Oracle manual.

# 2.8 Sample DEPT and EMP Tables

Most of the complete program examples in this guide use two sample database tables: DEPT and EMP. If they do not exist in your demo directory, create them before running the sample programs. Their definitions follow:

```
CREATE TABLE DEPT (DEPTNO NUMBER(2), DNAME VARCHAR2(14), LOC VARCHAR2(13));

CREATE TABLE EMP (EMPNO NUMBER(4) primary key, ENAME VARCHAR2(10), JOB VARCHAR2(9),
MGR NUMBER(4), HIREDATE DATE, SAL NUMBER(7,2), COMM NUMBER(7,2), DEPTNO NUMBER(2));
```

### 2.8.1 Sample DEPT and EMP Data

Respectively, the DEPT and EMP tables contain the following rows of data:

DEPTNO	DNAME	LOC		
10	ACCOUNTING	NEW YORK		
20	RESEARCH	DALLAS		
30	SALES	CHICAGO		
40 OPERATIONS		BOSTON		



EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500		30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

# 2.9 Sample EMP Program: SAMPLE1.PCO

A good way to get acquainted with embedded SQL is to look at a program example. This program is SAMPLE1.PCO in the demo directory.

The program logs on to the database, prompts the user for an employee number, queries the database table EMP for the employee's name, salary, and commission. The selected results are stored in host variables EMP-NAME, SALARY, and COMMISSION. The program uses the host indicator variable, COMM-IND to detect NULL values in column COMMISSION.

The paragraph DISPLAY-INFO then displays the result.

Bug 7225844 - Following para - substituted EMP-NAME for EMP-NUMBER

The COBOL variables USERNAME, PASSWD, and EMP-NAME are declared using the VARYING clause, which enables you to use a variable-length string external Oracle datatype called VARCHAR.

The SQLCA Communications Area is included to handle errors. If an error occurs, paragraph SQL-ERROR is performed.

The BEGIN DECLARE SECTION and END DECLARE SECTION statements used are optional, unless you set the precompiler option DECLARE\_SECTION to YES, or option MODE to ANSI.

The WHENEVER statement is used to handle errors.

The program ends when the user enters a zero employee number.



For simplicity in demonstrating this feature, this example does not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

```
* This program logs on to ORACLE, prompts the user for an employee number,
* queries the database for the employee's name, salary, and commission, then
* displays the result. The program terminates when the user enters a 0.
      ID DIVISION.
      PROGRAM-ID. QUERY.
      ENVIRONMENT DIVISION.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
          EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      01 USERNAME
                           PIC X(10) VARYING.
      01 PASSWD
                            PIC X(10) VARYING.
      01 EMP-REC-VARS.
          05 EMP-NAME
                            PIC X(10) VARYING.
          05 EMP-NUMBER
                            PIC S9(4) COMP VALUE ZERO.
          05 SALARY
                            PIC S9(5) V99 COMP-3 VALUE ZERO.
          05 COMMISSION PIC S9(5)V99 COMP-3 VALUE ZERO.
          05 COMM-IND
                            PIC S9(4) COMP VALUE ZERO.
          EXEC SQL END DECLARE SECTION END-EXEC.
          EXEC SQL INCLUDE SQLCA END-EXEC.
      01 DISPLAY-VARIABLES.
          05 D-EMP-NAME PIC X(10).
          05 D-SALARY
                          PIC Z(4)9.99.
          05 D-COMMISSION PIC Z(4)9.99.
          05 D-EMP-NUMBER PIC 9(4).
      01 D-TOTAL-QUERIED PIC 9(4) VALUE ZERO.
      PROCEDURE DIVISION.
      BEGIN-PGM.
          EXEC SQL WHENEVER SQLERROR
             DO PERFORM SQL-ERROR END-EXEC.
          PERFORM LOGON.
      OUERY-LOOP.
          DISPLAY " ".
          DISPLAY "ENTER EMP NUMBER (0 TO OUIT): "
              WITH NO ADVANCING.
          ACCEPT D-EMP-NUMBER.
          MOVE D-EMP-NUMBER TO EMP-NUMBER.
          IF (EMP-NUMBER = 0)
              PERFORM SIGN-OFF.
          MOVE SPACES TO EMP-NAME-ARR.
          EXEC SOL WHENEVER NOT FOUND GOTO NO-EMP END-EXEC.
          EXEC SQL SELECT ENAME, SAL, NVL(COMM, 0)
              INTO :EMP-NAME, :SALARY, :COMMISSION:COMM-IND
              FROM EMP
              WHERE EMPNO = :EMP-NUMBER
          END-EXEC.
          PERFORM DISPLAY-INFO.
          ADD 1 TO D-TOTAL-QUERIED.
          GO TO QUERY-LOOP.
      NO-EMP.
          DISPLAY "NOT A VALID EMPLOYEE NUMBER - TRY AGAIN.".
```

```
GO TO QUERY-LOOP.
LOGON.
   MOVE "SCOTT" TO USERNAME-ARR.
   MOVE 5 TO USERNAME-LEN.
   MOVE "TIGER" TO PASSWD-ARR.
   MOVE 5 TO PASSWD-LEN.
   EXEC SQL
       CONNECT : USERNAME IDENTIFIED BY : PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
DISPLAY-INFO.
   DISPLAY " ".
   DISPLAY "EMPLOYEE SALARY COMMISSION".
DISPLAY "-----".
   MOVE EMP-NAME-ARR TO D-EMP-NAME.
   MOVE SALARY TO D-SALARY.
   IF COMM-IND = -1
       DISPLAY D-EMP-NAME, D-SALARY, "
                                               NULL"
    ELSE
       MOVE COMMISSION TO D-COMMISSION
                                           ", D-COMMISSION
       DISPLAY D-EMP-NAME, D-SALARY, "
SIGN-OFF.
    DISPLAY " ".
    DISPLAY "TOTAL NUMBER QUERIED WAS ",
      D-TOTAL-QUERIED, ".".
    DISPLAY " ".
   DISPLAY "HAVE A GOOD DAY.".
   DISPLAY " ".
   EXEC SQL COMMIT WORK RELEASE END-EXEC.
    STOP RUN.
SQL-ERROR.
    EXEC SOL WHENEVER SOLERROR CONTINUE END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
   DISPLAY " ".
   DISPLAY SQLERRMC.
   EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
```

#### **Related Topics**

- Using the SQL Communications Area
- MODE
- WHENEVER Directive

# **Database Concepts**

This chapter explains the CONNECT statement and its options, Oracle Net Services, and related network connection statements. Transaction processing is presented. You learn the basic techniques that safeguard the consistency of your database, including how to control whether changes to Oracle data are made permanent or undone.

- Connecting to Oracle
- Default Databases and Connections
- Key Terms
- How Transactions Guard a Database
- Beginning and Ending Transactions
- Using the COMMIT Statement
- Using the ROLLBACK Statement
- Using the SAVEPOINT Statement
- Using the RELEASE Option
- Using the SET TRANSACTION Statement
- Overriding Default Locking
- · Fetching Across Commits
- Handling Distributed Transactions
- Guidelines for Transaction Processing

# 3.1 Connecting to Oracle

Your Pro\*COBOL program must log on to Oracle before querying or manipulating data. To log on, you use the CONNECT statement, as in

```
EXEC SQL

CONNECT : USERNAME IDENTIFIED BY : PASSWD END-EXEC.
```

where USERNAME and PASSWD are PIC X(n) or PIC X(n) VARYING host variables. Alternatively, you can use the statement:

```
EXEC SQL
CONNECT :USR-PWD
END-EXEC.
```

where the host variable USR-PWD contains your username and password separated by a slash (/) followed by an optional tnsnames.ora alias (@TNSALIAS).

The syntax for the CONNECT statement has an optional ALTER AUTHORIZATION clause. The complete syntax for CONNECT is shown here:

The CONNECT statement must be the first SQL statement executed by the program. That is, other executable SQL statements can positionally, but not logically, precede the CONNECT statement. If the precompiler option AUTO\_CONNECT=YES, a CONNECT statement is not needed.)

To supply the username and password separately, you define two host variables as character strings or VARCHAR variables. If you supply a userid containing both username and password, only one host variable is needed.

Make sure to set the username and password variables before the CONNECT is executed or it will fail. Your program can prompt for the values or you can hard-code them, as follows:

```
WORKING STORAGE SECTION.
...
01 USERNAME PIC X(10).
01 PASSWD PIC X(5).
...

PROCEDURE DIVISION.
LOGON.
EXEC SQL WHENEVER SQLERROR GOTO LOGON-ERROR END-EXEC.
MOVE "SCOTT" TO USERNAME.
MOVE "TIGER" TO PASSWD.
EXEC SQL
CONNECT : USERNAME IDENTIFIED BY : PASSWD
END-EXEC.
```

However, you cannot hard-code a username and password into the CONNECT statement or use quoted literals. For example, the following statements are *invalid*:

```
EXEC SQL
CONNECT SCOTT IDENTIFIED BY TIGER
END-EXEC.

EXEC SQL
CONNECT "SCOTT" IDENTIFIED BY "TIGER"
END-EXEC.
```

#### See Also:

- "Changing Passwords at Runtime" for more information about ALTER AUTHORIZATION clause.
- "SYSDBA or SYSOPER Privileges" for more information about SYSDBA and SYSOPER.



## 3.2 Default Databases and Connections

It is possible within a Pro\*COBOL program to maintain more than one database connection at the same time.

## 3.2.1 Concurrent Logons

Pro\*COBOL supports distributed processing through Oracle Net Services. Your application can concurrently access any combination of local and remote databases or make multiple connections to the same database. In Figure 3-1, an application program communicates with one local and three remote Oracle databases. ORA2, ORA3, and ORA4 are logical names used in CONNECT statements.

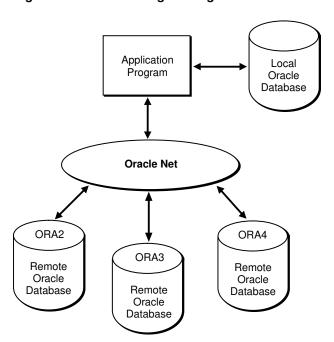


Figure 3-1 Connecting Through Oracle

By eliminating the boundaries in a network between different machines and operating systems, Oracle Net Services provides a distributed processing environment for Oracle tools. This section shows you how the Pro\*COBOL supports distributed processing through Oracle Net Services. You learn how your application can

- Access other databases directly or indirectly
- Concurrently access any combination of local and remote databases
- Make multiple connections to the same database

Normally you would need only a single connection, achieved by EXEC SQL CONNECT : USR-PWD END-EXEC. The database that is connected to is determined by what USR-PWD contains. If it contains the username and password for the default database, it will connect to the database defined as the default for the session. If it contains username/password@REMDB" it will connect through Oracle Net Services to the REMDB database as defined by your Oracle Net



Services configuration. (An alternative is to use the USING clause to specify the Oracle Net Services connection string.) This is the default connection.

To make further concurrent connections to either the same or different databases you make use of the AT clause, that is, EXEC SQL AT DB1 CONNECT : USR-PWD END-EXEC. The name after the AT clause uniquely identifies a "nondefault" connection, and any SQL statements with the same name after the AT clause are executed against that connection. If the AT clause is omitted in an SQL statement then the statement is executed against the default connection.

All database names must be unique, but two or more database names can specify the same connection. That is, you can have multiple connections to any database on any node.

### 3.2.2 Using Username/Password

Usually, you establish a connection to Oracle as follows:

```
EXEC SQL CONNECT : USERNAME IDENTIFIED BY : PASSWORD END-EXEC.
```

Or you can use:

```
EXEC SQL CONNECT : USR-PWD END-EXEC.
```

where USR-PWD contains any valid Oracle connect string.

These are simplified subsets of the CONNECT statement. For all details, read the next sections in this chapter.

#### **Related Topics**

- Automatic Logons
- CONNECT (Executable Embedded SQL Extension)

#### 3.2.2.1 Named Database Connections

In the following example, you connect to a named database. Normally you use a named database connection only for multiple concurrent connections. The following example shows the syntax for a single connection:

```
* -- Declare necessary host variables
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(10) .
01 PASSWORD PIC X(10) .
01 DB-STRING PIC X(20) .
    EXEC SOL END DECLARE SECTION END-EXEC.
    . . .
PROCEDURE DIVISION.
    MOVE "scott" TO USERNAME.
    MOVE "tiger" TO PASSSWORD.
    MOVE "nyremote" TO DB-STRING.
* -- Assign a unique name to the database connection.
    EXEC SQL DECLARE DBNAME DATABASE END-EXEC.
* -- Connect to the nondefault database
    EXEC SQL
```



```
CONNECT :USERNAME IDENTIFIED BY :PASSWORD AT DBNAME USING :DB-STRING END-EXEC.
```

The identifiers in this example serve the following purposes:

- The host variables USERNAME and PASSWORD identify a valid user.
- The host variable DB-STRING contains the Oracle Net Services syntax for logging on to a nondefault database at a remote node.
- The undeclared identifier *DBNAME* names a nondefault connection; it is an identifier used by Oracle, *not* a host or program variable.

The USING clause specifies the network, machine, and database to be associated with *DBNAME*. Later, SQL statements using the AT clause (with *DBNAME*) are executed at the database specified by *DB-STRING*.

Alternatively, you can use a character host variable in the AT clause, as the following example shows:

```
* -- Declare necessary host variables
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(10).
01 PASSWORD PIC X(10).
01 DB-NAME PIC X(10).
01 DB-STRING PIC X(20).
    EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    MOVE "scott" TO USERNAME.
    MOVE "tiger" TO PASSSWORD.
    MOVE "oracle1" TO DB-NAME.
    MOVE "nyremote" TO DB-STRING.
* -- Connect to the nondefault database
    EXEC SQL
    CONNECT : USERNAME IDENTIFIED BY : PASSWORD
    AT :DB-NAME USING :DB-STRING
    END-EXEC.
```

If *DB-NAME* is a host variable, the DECLARE DATABASE statement is not needed. Only if *DBNAME* is an undeclared identifier must you execute a DECLARE *DBNAME* DATABASE statement before executing a CONNECT ... AT *DBNAME* statement.

**SQL Operations.** If granted the privilege, you can execute any SQL data manipulation statement at the nondefault connection. For example, you might execute the following sequence of statements:

```
EXEC SQL AT DBNAME SELECT ... EXEC SQL AT DBNAME INSERT ... EXEC SQL AT DBNAME UPDATE ...
```

In the next example, *DB-NAME* is a host variable:

```
EXEC SQL AT :DB-NAME DELETE ...
```



**Cursor Control.** Cursor control statements such as OPEN, FETCH, and CLOSE are exceptions—they never use an AT clause. If you want to associate a cursor with an explicitly identified database, use the AT clause in the DECLARE CURSOR statement, as follows:

```
EXEC SQL AT :DB-NAME DECLARE emp_cursor CURSOR FOR ...
EXEC SQL OPEN emp_cursor ...
EXEC SQL FETCH emp_cursor ...
EXEC SQL CLOSE emp cursor END-EXEC.
```

If *DB-NAME* is a host variable, its declaration must be within the scope of all SQL statements that refer to the declared cursor. For example, if you open the cursor in one subprogram, then fetch from it in another, you must declare *DB-NAME* globally or pass it to each subprogram.

When opening, closing, or fetching from the cursor, you do not use the AT clause. The SQL statements are executed at the database named in the AT clause of the DECLARE CURSOR statement or at the default database if no AT clause is used in the cursor declaration.

The AT :host-variable clause enables you to change the connection associated with a cursor. However, you cannot change the association while the cursor is open. Consider the following example:

```
EXEC SQL AT :DB-NAME DECLARE emp_cursor CURSOR FOR ...

MOVE "oracle1" TO DB-NAME.

EXEC SQL OPEN emp_cursor END-EXEC.

EXEC SQL FETCH emp_cursor INTO ...

MOVE "oracle2" TO DB-NAME.

* -- illegal, cursor still open

EXEC SQL OPEN emp_cursor END-EXEC.

EXEC SQL FETCH emp cursor INTO ...
```

This is illegal because *emp\_cursor* is still open when you try to execute the second OPEN statement. Separate cursors are not maintained for different connections; there is only one *emp\_cursor*, which must be closed before it can be reopened for another connection. To debug the last example, simply close the cursor before reopening it, as follows:

```
* -- close cursor first

EXEC SQL CLOSE emp_cursor END-EXEC.

MOVE "oracle2" TO DB-NAME.

EXEC SQL OPEN EMP-CUROR END-EXEC.

EXEC SQL FETCH emp_cursor INTO ...
```

**Dynamic SQL**. Dynamic SQL statements are similar to cursor control statements in that some never use the AT clause. For dynamic SQL Method 1, you must use the AT clause if you want to execute the statement at a nondefault connection. An example follows:

```
EXEC SQL AT :DB-NAME EXECUTE IMMEDIATE :SQL-STMT END-EXEC.
```

For Methods 2, 3, and 4, you use the AT clause only in the DECLARE STATEMENT statement if you want to execute the statement at a nondefault connection. All other dynamic SQL statements such as PREPARE, DESCRIBE, OPEN, FETCH, and CLOSE never use the AT clause. The next example shows Method 2:



```
EXEC SQL AT :DB-NAME DECLARE SQL-STMT STATEMENT END-EXEC. EXEC SQL PREPARE SQL-STMT FROM :SQL-STRING END-EXEC. EXEC SQL EXECUTE SQL-STMT END-EXEC.
```

The following example shows Method 3:

```
EXEC SQL AT :DB-NAME DECLARE SQL-STMT STATEMENT END-EXEC.

EXEC SQL PREPARE SQL-STMT FROM :SQL-STRING END-EXEC.

EXEC SQL DECLARE emp_cursor CURSOR FOR SQL-STMT END-EXEC.

EXEC SQL OPEN emp_cursor ...

EXEC SQL FETCH emp_cursor INTO ...

EXEC SQL CLOSE emp_cursor END-EXEC.
```

You need not use the AT clause when connecting to a remote database unless you open two or more connections simultaneously (in which case the AT clause is needed to identify the active connection). To make the default connection to a remote database, use the following syntax:

```
EXEC SQL

CONNECT :USERNAME IDENTIFIED BY :PASSWORD USING :DB-STRING END-EXEC.
```

## 3.2.3 Automatic Logons

You can log on to Oracle automatically with the userid:

where *prefix* is the value of the Oracle initialization parameter OS\_AUTHENT\_PREFIX (the default value is OPS\$) and *username* is your operating system user or task name. For example, if the prefix is OPS\$, your user name is TBARNES, and OPS\$TBARNES is a valid Oracle userid, then you log on to Oracle as user OPS\$TBARNES.

To take advantage of the automatic logon feature, you simply pass a slash (/) character to Pro\*COBOL, as follows:

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

01 ORACLEID PIC X.

EXEC SQL END DECLARE SECTION END-EXEC.

MOVE '/' TO ORACLEID.

EXEC SQL CONNECT :ORACLEID END-EXEC.
```

This automatically connects you as user OPS\$username. For example, if your operating system username is RHILL, and OPS\$RHILL is a valid Oracle username, connecting with a slash (/) automatically logs you on to Oracle as user OPS\$RHILL.

You can also pass a character string to Pro\*COBOL. However, the string cannot contain trailing blanks. For example, the following CONNECT statement will fail:

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

01 ORACLEID PIC X(5).

EXEC SQL END DECLARE SECTION END-EXEC.

MOVE '/ ' TO ORACLEID.

EXEC SQL CONNECT :ORACLEID END-EXEC.
```



#### 3.2.3.1 The AUTO CONNECT Precompiler Option

Pro\*COBOL lets your program log on to the default database without using the CONNECT statement. Simply specify the precompiler option AUTO\_CONNECT on the command line.

Assume that the default value of OS\_AUTHENT\_PREFIX is OPS\$, your username is TBARNES, and OPS\$TBARNES is a valid Oracle userid. When AUTO\_CONNECT=YES, as soon as Pro\*COBOL encounters an executable SQL statement, your program logs on to Oracle automatically with the userid OPS\$TBARNES.

When AUTO\_CONNECT=NO (the default), you must use the CONNECT statement to log on to Oracle.

## 3.2.4 Changing Passwords at Runtime

Pro\*COBOL provides client applications with a convenient way to change a user password at runtime through the optional ALTER AUTHORIZATION clause.

The syntax for the ALTER AUTHORIZATION clause is shown here:

```
EXEC SQL CONNECT .. ALTER AUTHORIZATION : NEWPSWD END-EXEC.
```

Using this clause indicates that you want to change the account password to the value indicated by NEWPSWD. After the change is made, an attempt is made to connect as USER/NEWPSWD. This can have the following results:

- The application will connect without issue.
- The application will fail to connect. This could be due to either of the following:
  - Password verification failed for some reason. In this case the password remains unchanged.
  - The account is locked. Changes to the password are not permitted.

### 3.2.5 Connect Without Alter Authorization

This section describes the possible outcomes of different variations of the CONNECT statement.

#### 3.2.5.1 Standard CONNECT

If an application issues the following statement:

```
EXEC SOL CONNECT ... /* No ALTER AUTHORIZATION clause */
```

it performs a normal connection attempt. The possible results include the following:

- The application will connect without issue.
- The application will connect, but will receive a password warning. The warning
  indicates that the password has expired but is in a grace period which will allow
  logons. At this point, the user is encouraged to change the password before the
  account becomes locked.
- The application will fail to connect. Possible causes include the following:



- The password is incorrect.
- The account has expired, and is possibly in a locked state.

### 3.2.5.2 SYSDBA or SYSOPER Privileges

Before Oracle release 8.1 you did not have to use this clause to have the SYSOPER or SYSDBA system privilege, but now you must.

Append the following optional string to the CONNECT statement after all other clauses if you want to log on with either SYSDBA or SYSOPER system privileges:

```
IN { SYSDBA | SYSOPER | SYSBACKUP} MODE
```

#### For example:

```
EXEC SQL CONNECT ... IN SYSDBA MODE END-EXEC.
```

Here are the restrictions that apply to this option:

- This option is not supported when using the AUTO\_CONNECT=YES precompiler option setting.
- The option is not permitted when using the ALTER AUTHORIZATION keywords in the CONNECT statement.

### 3.2.6 Using Links

Database links are supported through the Oracle distributed database option. For example, a distributed query allows a single SELECT statement to access data on one or more nondefault databases.

The distributed query facility depends on database links, which assign a name to a CONNECT statement rather than to the connection itself. At runtime, the embedded SELECT statement is executed by the specified database server, which connects implicitly to the nondefault database(s) to get the required data.

#### **Related Topics**

Connecting to a Database Service

# 3.3 Key Terms

Before delving into the subject of transactions, you should know the terms defined in this section.

The jobs or tasks that the database manages are called *sessions*. A *user session* is started when you run an application program or a tool such as Oracle Forms and connect to the database. Oracle enables user sessions to work simultaneously and share computer resources. To do this, Oracle must control *concurrence*, the accessing of the same data by many users. Without adequate concurrence controls, there might be a loss of *data integrity*. That is, changes to data or structures might be made in the wrong order.

Oracle uses *locks* to control concurrent access to data. A lock gives you temporary ownership of a database resource such as a table or row of data. Thus, data cannot be changed by other users until you finish with it. You need never explicitly lock a resource, because default locking mechanisms protect table data and structures. However, you can



request *data locks* on tables or rows when it is to your advantage to override default locking. You can choose from several *modes* of locking such as *row share* and *exclusive*.

A *deadlock* can occur when two or more users try to access the same database object. For example, two users updating the same table might wait if each tries to update a row currently locked by the other. Because each user is waiting for resources held by another user, neither can continue until the server breaks the deadlock. The server signals an error to the participating transaction that had completed the least amount of work, and the "deadlock detected while waiting for resource" error code is returned to SQLCODE in the SQLCA.

When a table is queried by one user and updated by another at the same time, the database generates a *read consistent* view of the table's data for the query. That is, once a query begins (and proceeds), the data read by the query does not change. As update activity continues, the database takes *snapshots* of the table's data and records changes in a *rollback segment*. The database uses information in the rollback segment to build read consistent query results and to undo changes if necessary.

### 3.4 How Transactions Guard a Database

The database is transaction oriented; it uses transactions to ensure data integrity. A transaction is a series of one or more logically related SQL statements you define to accomplish some task. The database treats the series of SQL statements as a unit so that all the changes brought about by the statements are either *committed* (made permanent) or *rolled back* (undone) at the same time. If your application program fails in the middle of a transaction, the database is automatically restored to its former (pretransaction) state.

The coming sections show you how to define and control transactions. Specifically, it shows how to:

- Begin and end transactions
- Use the COMMIT statement to make transactions permanent
- Use the SAVEPOINT statement with the ROLLBACK TO statement to undo parts of transactions
- Use the ROLLBACK statement to undo whole transactions
- Specify the RELEASE option to free resources and log off the database
- Use the SET TRANSACTION statement to set read-only transactions
- Use the FOR UPDATE clause or LOCK TABLE statement to override default locking



Oracle Database SQL Language Reference for details about the SQL statements.



# 3.5 Beginning and Ending Transactions

You begin a transaction with the first executable SQL statement (other than CONNECT) in your program. When one transaction ends, the next executable SQL statement automatically begins another transaction. Thus, every executable statement is part of a transaction. Because they cannot be rolled back and need not be committed, declarative SQL statements are not considered part of a transaction.

You end a transaction in one of the following ways:

- Code a COMMIT or ROLLBACK statement, with or without the RELEASE option. This explicitly makes permanent or undoes changes to the database.
- Code a data definition statement (ALTER, CREATE, or GRANT, for example) that issues an automatic commit before and after executing. This implicitly makes permanent changes to the database.

A transaction also ends when there is a system failure or your user session stops unexpectedly because of software problems, hardware problems, or a forced interrupt.

If your program fails in the middle of a transaction, Oracle detects the error and rolls back the transaction. If your operating system fails, Oracle restores the database to its former (pretransaction) state.

# 3.6 Using the COMMIT Statement

The COMMIT statement is used to make changes to the database permanent. Until changes are committed, other users cannot access the changed data; they see it as it was before your transaction began. The COMMIT statement has no effect on the values of host variables or on the flow of control in your program. Specifically, the COMMIT statement:

- Makes permanent all changes made to the database during the current transaction.
- Makes these changes visible to other users.
- Erases all savepoints (see the next section).
- Releases all row and table locks, but not parse locks.
- Closes cursors declared using the FOR UPDATE clause or referenced elsewhere in the code with the CURRENT OF clause. If MODE=ANSI | ANSI14 or CLOSE ON COMMIT=YES is used, then all explicit cursors are closed.
- Ends the transaction.

When MODE={ANSI13 | ORACLE}, explicit cursors not referenced in a CURRENT OF clause remain open across commits. This can boost performance. For an example, see "Fetching Across Commits".

Because they are part of normal processing, COMMIT statements should be placed inline, on the main path through your program. Before your program terminates, it must explicitly commit pending changes. Otherwise, Oracle rolls them back. In the following example, you commit your transaction and disconnect:

```
EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all resources (locks and cursors) held by your program and logs off the database.



You need not follow a data definition statement with a COMMIT statement because data definition statements issue an automatic commit before *and* after executing. So, whether they succeed or fail, the prior transaction is committed.

### 3.6.1 WITH HOLD Clause in DECLARE CURSOR Statements

Any cursor that has been declared with the clause WITH HOLD after the word CURSOR remains open after a COMMIT. The following example shows how to use this clause:

```
EXEC SQL

DECLARE C1 CURSOR WITH HOLD

FOR SELECT ENAME FROM EMP

WHERE EMPNO BETWEEN 7600 AND 7700

END-EXEC.
```

The cursor must not be declared for UPDATE. The WITH HOLD clause is used in DB2 to override the default, which is to close all cursors on commit. Pro\*COBOL provides this clause in order to ease migrations of applications from DB2 to Oracle. When MODE=ANSI, Oracle uses the DB2 default, but all host variables must be declared in a Declare Section. To avoid having a Declare Section, use the precompiler option <code>CLOSE\_ON\_COMMIT</code> described next.

#### **Related Topics**

DECLARE CURSOR (Embedded SQL Directive)

## 3.6.2 CLOSE ON COMMIT Precompiler Option

The precompiler option CLOSE\_ON\_COMMIT is available to override the default behavior of MODE=ANSI (if you specify MODE=ANSI on the command line, any cursors not declared with the WITH HOLD clause are closed on commit):

```
CLOSE ON COMMIT = {YES | NO}
```

The default is NO. This option must be entered only on the command line or in a configuration file.



Use this option carefully; applications may be slowed if cursors are opened and closed many times because of the need to re-parse for each OPEN statement. See "CLOSE ON COMMIT".

# 3.7 Using the ROLLBACK Statement

You use the ROLLBACK statement to undo pending changes made to the database. For example, if you make a mistake, such as deleting the wrong row from a table, you can use ROLLBACK to restore the original data. The ROLLBACK statement has no effect on the values of host variables or on the flow of control in your program. Specifically, the ROLLBACK statement

Undoes all changes made to the database during the current transaction

- Erases all savepoints
- Ends the transaction
- Releases all row and table locks, but not parse locks
- Closes cursors declared using the FOR UPDATE clause or referenced elsewhere in the code with the CURRENT OF clause. If MODE={ANSI | ANSI14}, then all explicit cursors are closed.

When MODE={ANSI13 | ORACLE}, explicit cursors not referenced in a CURRENT OF clause remain open across rollbacks.

Because they are part of exception processing, ROLLBACK statements should be placed in error handling routines, off the main path through your program. In the following example, you roll back your transaction and disconnect:

```
EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all resources held by your program and logs off the database.

If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes a ROLLBACK statement, your program might enter an infinite loop if the rollback fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the ROLLBACK statement.

For example, consider the following:

```
WHENEVER SQLERROR GOTO SQL-ERROR
    END-EXEC.
   DISPLAY 'Employee number? '.
   ACCEPT EMP-NUMBER.
   DISPLAY 'Employee name? '.
   ACCEPT EMP-NAME.
   EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
      VALUES (:EMP-NUMBER, :EMP-NAME)
    END-EXEC.
    . . .
SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
   DISPLAY 'PROCESSING ERROR.'.
   DISPLAY 'ERROR CODE : ', SQLCODE.
   DISPLAY 'MESSAGE :', SQLERRMC.
   EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
```

Oracle rolls back transactions if your program terminates abnormally.

## 3.7.1 Statement-Level Rollbacks

Before executing any SQL statement, Oracle marks an implicit savepoint (not available to you). Then, if the statement fails, Oracle rolls it back automatically and returns the applicable error code to SQLCODE in the SQLCA. For example, if an INSERT statement causes an error by trying to insert a duplicate value in a unique index, the statement is rolled back.

Only work started by the failed SQL statement is lost; work done before that statement in the current transaction is kept. Thus, if a data definition statement fails, the automatic commit that precedes it is not undone.

#### Note:

Before executing a SQL statement, Oracle must parse it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. Errors detected while executing a SQL statement cause a rollback, but errors detected while parsing the statement do not.

Oracle can also roll back single SQL statements to break deadlocks. Oracle signals an error to one of the participating transactions and rolls back the current statement in that transaction.

# 3.8 Using the SAVEPOINT Statement

The SAVEPOINT embedded SQL statement marks and names the current point in processing a transaction. Each marked point is called a *savepoint*. For example, the following statement marks a savepoint named *start\_delete*:

```
EXEC SQL SAVEPOINT start delete END-EXEC.
```

Savepoints let you divide long transactions, giving you more control over complex procedures. For example, if a transaction performs several functions, you can mark a savepoint before each function. Then, if a function fails, you can easily restore the data to its former state, recover, and then reexecute the function.

To undo part of a transaction, you can use savepoints with the ROLLBACK statement and its TO SAVEPOINT clause. The TO SAVEPOINT clause lets you roll back to an intermediate statement in the current transaction. With it, you do not have to undo all your changes. Specifically, the ROLLBACK TO SAVEPOINT statement:

- Undoes changes made to the database since the specified savepoint was marked
- Erases all savepoints marked after the specified savepoint
- Releases all row and table locks acquired since the specified savepoint was marked

In the following example, you access the table MAIL\_LIST to insert new listings, update old listings, and delete (a few) inactive listings. After the delete, you check SQLERRD(3) in the SQLCA for the number of rows deleted. If the number is unexpectedly large, you roll back to the savepoint <code>start\_delete</code>, undoing just the delete.

```
* -- For each new customer

DISPLAY 'New customer number? '.

ACCEPT CUST-NUMBER.

IF CUST-NUMBER = 0

GO TO REV-STATUS

END-IF.

DISPLAY 'New customer name? '.

ACCEPT CUST-NAME.

EXEC SQL INSERT INTO MAIL-LIST (CUSTNO, CNAME, STAT)

VALUES (:CUST-NUMBER, :CUST-NAME, 'ACTIVE').

END-EXEC.

...

* -- For each revised status

REV-STATUS.

DISPLAY 'Customer number to revise status? '.
```



```
ACCEPT CUST-NUMBER.
    IF CUST-NUMBER = 0
        GO TO SAVE-POINT
    END-IF.
    DISPLAY 'New status? '.
    ACCEPT NEW-STATUS.
    EXEC SOL UPDATE MAIL-LIST
       SET STAT = :NEW-STATUS WHERE CUSTNO = :CUST-NUMBER
    END-EXEC.
* -- mark savepoint
SAVE-POINT.
    EXEC SOL SAVEPOINT START-DELETE END-EXEC.
    EXEC SQL DELETE FROM MAIL-LIST WHERE STAT = 'INACTIVE'
     END-EXEC.
    IF SQLERRD(3) < 25
* -- check number of rows deleted
        DISPLAY 'Number of rows deleted is ', SQLERRD(3)
    ELSE
        DISPLAY 'Undoing deletion of ', SQLERRD(3), ' rows'
        EXEC SQL
            WHENEVER SQLERROR GOTO SQL-ERROR
        END-EXEC
        EXEC SQL
            ROLLBACK TO SAVEPOINT START-DELETE
        END-EXEC
     EXEC SOL WHENEVER SOLERROR CONTINUE END-EXEC.
    EXEC SQL COMMIT WORK RELEASE END-EXEC.
    STOP RUN.
* -- exit program.
     . . .
SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    DISPLAY 'Processing error'.
* -- exit program with an error.
     STOP RUN.
```

Note that you cannot specify the RELEASE option in a ROLLBACK TO SAVEPOINT statement.

Rolling back to a savepoint erases any savepoints marked after that savepoint. The savepoint to which you roll back, however, is not erased. For example, if you mark five savepoints, then roll back to the third, only the fourth and fifth are erased. A COMMIT or ROLLBACK statement erases all savepoints.

# 3.9 Using the RELEASE Option

Oracle rolls back changes automatically if your program terminates abnormally. Abnormal termination occurs when your program does not explicitly commit or roll back work and disconnect using the RELEASE embedded SQL statement.

Normal termination occurs when your program runs its course, closes open cursors, explicitly commits or rolls back work, disconnects, and returns control to the user. Your program will exit gracefully if the last SQL statement it executes is either

```
EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

or

EXEC SOL ROLLBACK WORK RELEASE END-EXEC.

where the token WORK is optional. Otherwise, locks and cursors acquired by your user session are held after program termination until Oracle recognizes that the user session is no longer active. This might cause other users in a multiuser environment to wait longer than necessary for the locked resources.

## 3.10 Using the SET TRANSACTION Statement

You can use the SET TRANSACTION statement to begin a read-only or read/write transaction, or to assign your current transaction to a specified rollback segment. A COMMIT, ROLLBACK, or data definition statement ends a read-only transaction.

Because they allow "repeatable reads," read-only transactions are useful for running multiple queries against one or more tables while other users update the same tables. During a read-only transaction, all queries refer to the same snapshot of the database, providing a multitable, multiquery, read-consistent view. Other users can continue to query or update data as usual. An example of the SET TRANSACTION statement follows:

```
EXEC SQL SET TRANSACTION READ ONLY END-EXEC.
```

The SET TRANSACTION statement must be the first SQL statement in a read-only transaction and can appear only once in a transaction. The READ ONLY parameter is required. Its use does not affect other transactions. Only the SELECT (without FOR UPDATE), LOCK TABLE, SET ROLE, ALTER SESSION, ALTER SYSTEM, COMMIT, and ROLLBACK statements are allowed in a read-only transaction.

In the following example, a store manager checks sales activity for the day, the past week, and the past month by using a read-only transaction to generate a summary report. The report is unaffected by other users updating the database during the transaction.

```
EXEC SQL SET TRANSACTION READ ONLY END-EXEC.

EXEC SQL SELECT SUM(SALEAMT) INTO :DAILY FROM SALES

WHERE SALEDATE = SYSDATE END-EXEC.

EXEC SQL SELECT SUM(SALEAMT) INTO :WEEKLY FROM SALES

WHERE SALEDATE > SYSDATE - 7 END-EXEC.

EXEC SQL SELECT SUM(SALEAMT) INTO :MONTHLY FROM SALES

WHERE SALEDATE > SYSDATE - 30 END-EXEC.

EXEC SQL COMMIT WORK END-EXEC.

* -- simply ends the transaction since there are no changes

* -- to make permanent

* -- format and print report
```

# 3.11 Overriding Default Locking

By default, Oracle implicitly (automatically) locks many data structures for you. However, you can request specific data locks on rows or tables when it is to your advantage to override default locking. Explicit locking lets you share or deny access to a table for the duration of a transaction or ensure multitable and multiquery read consistency.



With the SELECT FOR UPDATE OF statement, you can explicitly lock specific rows of a table to make sure they do not change before an update or delete is executed. However, Oracle automatically obtains row-level locks at update or delete time. So, use the FOR UPDATE OF clause only if you want to lock the rows *before* the update or delete.

You can explicitly lock entire tables using the LOCK TABLE statement.

## 3.11.1 Using the FOR UPDATE OF Clause

When you DECLARE a cursor, you can meanwhile optionally specify the FOR UPDATE clause, which has the effect of acquiring an exclusive lock on all rows defined by the cursor. This is useful, for example, when you want to base an update on existing rows in a table and want to ensure that they are not meanwhile changed by anyone else.

Note that if you refer to a cursor with the CURRENT OF clause, that the precompiler will automatically add the FOR UPDATE clause to the cursor definition and the word OF is optional. For instance, instead of:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO = 20 FOR UPDATE OF SAL END-EXEC.
```

you can drop the OF part of the clause and simply code:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO = 20 FOR UPDATE END-EXEC.
```



"Using the CURRENT OF Clause" for an example.

### 3.11.1.1 Restrictions

You cannot use FOR UPDATE with multiple tables, but you must use FOR UPDATE OF to identify a column in the table that you want locked. Row locks obtained by a FOR UPDATE statement are cleared by a COMMIT, which explains why the cursor is closed for you. If you try to fetch from a FOR UPDATE cursor after a commit, Oracle generates a Fetch out of Sequence error.

# 3.12 Fetching Across Commits

If you want to mix commits and fetches, do not use the CURRENT OF clause. Instead, select the ROWID of each row, and then use that value to identify the current row during the update or delete. Consider the following example:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME, SAL, ROWID FROM EMP WHERE JOB = 'CLERK'
END-EXEC.
...
EXEC SQL OPEN emp_cursor END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO ...
```



```
PERFORM

EXEC SQL

FETCH emp_cursor INTO :EMP_NAME, :SALARY, :ROW-ID

END-EXEC

...

EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY

WHERE ROWID = :ROW-ID

END-EXEC

EXEC SQL COMMIT END-EXEC

END-PERFORM.
```

Note, however, that the fetched rows are *not* locked. So, you can receive inconsistent results if another user modifies a row after you read it but before you update or delete it.

## 3.12.1 Using the LOCK TABLE Statement

Use the LOCK TABLE statement locks one or more tables in a specified lock mode. For example, the following statement locks the EMP table in *row share* mode. Row share locks allow concurrent access to a table. They prevent other users from locking the entire table for exclusive use.

```
EXEC SQL

LOCK TABLE EMP IN ROW SHARE MODE NOWAIT
END-EXEC
```

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an *exclusive* lock. While one user has an exclusive lock on a table, no other users can insert, update, or delete rows in that table.

The optional keyword NOWAIT tells Oracle not to wait for a table if it has been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. (You can check SQLCODE in the SQLCA to see if the table lock failed.) If you omit NOWAIT, Oracle waits until the table is available; the wait has no set limit.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. Consequently, a query never blocks another query or an update, and an update never blocks a query. Only if two different transactions try to update the same row will one transaction wait for the other to complete. Table locks are released when your transaction issues a COMMIT or ROLLBACK.



Using Oracle Lock Management Services for more information about lock modes.

# 3.13 Handling Distributed Transactions

A distributed database is a single logical database comprising multiple physical databases at different nodes. A distributed statement is any SQL statement that accesses a remote node using a database link. A distributed transaction includes at

least one distributed statement that updates data at multiple nodes of a distributed database. If the update affects only one node, the transaction is non-distributed.

When you issue a commit, changes to each database affected by the distributed transaction are made permanent. If instead you issue a rollback, all the changes are undone. However, if a network or machine fails during the commit or rollback, the state of the distributed transaction might be unknown or *in doubt*. In such cases, if you have FORCE TRANSACTION system privileges, you can manually commit or roll back the transaction at your local database by using the FORCE clause. The transaction must be identified by a quoted literal containing the transaction ID, which can be found in the data dictionary view DBA 2PC PENDING. Some examples follow:

```
EXEC SQL COMMIT FORCE '22.31.83' END-EXEC. ...
EXEC SQL ROLLBACK FORCE '25.33.86'END-EXEC.
```

FORCE commits or rolls back only the specified transaction and does not affect your current transaction. Note that you cannot manually roll back in-doubt transactions to a savepoint.

The COMMENT clause in the COMMIT statement lets you specify a Comment to be associated with a distributed transaction. If ever the transaction is in doubt, the server stores the text specified by COMMENT in the data dictionary view DBA\_2PC\_PENDING along with the transaction ID. The text must be a quoted literal of no more than 50 characters in length. An example follows:

```
EXEC SQL

COMMIT COMMENT 'In-doubt trans; notify Order Entry'
END-EXEC.
```



Oracle Database Concepts. for more information about distributed transactions.

# 3.14 Guidelines for Transaction Processing

The following guidelines will help you avoid some common problems.

## 3.14.1 Designing Applications

When designing your application, group logically related actions together in one transaction. A well-designed transaction includes all the steps necessary to accomplish a given task—no more and no less.

Data in the tables you reference must be left in a consistent state. Thus, the SQL statements in a transaction should change the data in a consistent way. For example, a transfer of funds between two bank accounts should include a debit to one account and a credit to another. Both updates should either succeed or fail together. An unrelated update, such as a new deposit to one account, should not be included in the transaction.

## 3.14.2 Obtaining Locks

If your application programs include SQL locking statements, make sure the users requesting locks have the privileges needed to obtain the locks. Your DBA can lock any table. Other

users can lock tables they own or tables for which they have a privilege, such as ALTER, SELECT, INSERT, UPDATE, or DELETE.

## 3.14.3 Using PL/SQL

If a PL/SQL block is part of a transaction, commits and rollback operations inside the block affect the whole transaction. In the following example, the rollback operation undoes changes made by the update *and* the insert:

```
EXEC SQL INSERT INTO EMP ...

EXEC SQL EXECUTE

BEGIN UPDATE emp

...

EXCEPTION

WHEN DUP_VAL_ON_INDEX THEN

ROLLBACK;

END;

END-EXEC.
```

## 3.14.4 X/Open Applications

For instructions on using the XA interface in X/Open applications, see X/Open Distributed Transaction Processing (DTP).



4

# Datatypes and Host Variables

This chapter provides the basic information you need to write a Pro\*COBOL program, including:

- The Oracle Database Datatypes
- Datetime and Interval Datatype Descriptors
- Host Variables
- Indicator Variables
- VARCHAR Variables
- Handling Boolean Data
- Handling Character Data
- Universal ROWIDs
- Globalization Support
- Unicode Support for Pro\*COBOL
- Datatype Conversion
- Explicit Control Over DATE String Format
- Datatype Equivalencing
- Platform Endianness Support
- Sample Program 4: Datatype Equivalencing

# 4.1 The Oracle Database Datatypes

Oracle Database recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle Database stores data in database columns.

Oracle Database also uses internal datatypes to represent database pseudocolumns. An external datatype specifies how data is stored in a host variable.



Data Types for complete descriptions of the Oracle internal (also called *built-in*) datatypes.

## 4.1.1 Internal Datatypes

Table 4-1 summarizes the information about each Oracle built-in datatype.

Table 4-1 Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length and Default
CHAR (size)	Fixed-length character data of length <i>size</i> in characters or bytes, depending on the national character set	Fixed for every row in the table (with trailing blanks.) Column size is the number of characters for a fixed-width national character set or the number of bytes required to store one character, with an upper limit of 2000 bytes for each row. Default size is 1 character or 1 byte for each row, depending on the national character set. Consider the character set (one-byte or multibyte) before setting size.
VARCHAR2 (size)	Fixed-length character data of length <i>size</i> in characters or bytes, depending on the national character set. A maximum size must be specified.	Variable for each row. Column size is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum size is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes for each row. Default size is 1 character or 1 byte, depending on the national character set.
NCHAR (size)	Fixed-length character data of length <i>size</i> in characters or bytes, depending on national character set.	Fixed for every row in the table (with trailing blanks). Column <i>size</i> is the number of bytes for a national character set or the number of bytes for a varying-width national character set. Maximum <i>size</i> is determined by the number of bytes required to store one character, with an upper limit of 2000 bytes for each row. Default is 1 character or 1 byte, depending on the character set.
NVARCHAR2 (size)	Variable-length character data of length <i>size</i> in characters or bytes, depending on national character set. A maximum <i>size</i> must be specified.	Variable for each row. Column <i>size</i> is the number of bytes for a national character set. Maximum <i>size</i> is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes for each row. Default is 1 character or 1 byte, depending on the character set.
CLOB	Single-byte character data	Up to 2^32 - 1 bytes, or 4 gigabytes.
NCLOB	Single-byte or fixed- length multibyte national character set (NCHAR) data	Up to 2^32 - 1 bytes, or 4 gigabytes.
LONG	Variable-length character data	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, for each row. Provided for backward compatibility.
NUMBER(p,s)	Variable-length numeric data.: Maximum precision $p$ , or scale $s$ is 38, or both.	Variable for each row. The maximum space required for a given column is 21 bytes for each row.
DATE	Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E.	Fixed at 7 bytes for each row in the table.  Default format is a string (such as DD-MON-YY) specified by NLS_DATE_FORMAT parameter.



Table 4-1 (Cont.) Summary of Oracle Built-In Datatypes

Datatype	Description	Column Length and Default
BLOB	Unstructured binary data	Up to 2^32 - 1 bytes, or 4 gigabytes.
BFILE	Binary data stored in an external file	Up to 2^32 - 1 bytes, or 4 gigabytes.
RAW (size)	Variable-length raw binary data	Variable for each row in the table, up to 2000 bytes for each row. A maximum <i>size</i> must be specified. Provided for backward compatibility.
LONG RAW	Variable-length raw binary data	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, for each row. Provided for backward compatibility.
ROWID	Binary data representing row addresses	Fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table.
BOOLEAN	Logical data	The BOOLEAN data type comprises the distinct truth values: True and False.



All forms of LONG data types (LONG, LONG RAW, LONG VARCHAR, LONG VARRAW) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the LONG data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use CLOB and NCLOB data types for large amounts of character data.

For more information, see:

Migrating Columns from LONGs to LOBs

## 4.1.2 External Data Types

The external data types include all the internal data types plus several data types found in other supported host languages. Use the data type names in data type equivalencing, and the data type codes in dynamic SQL Method 4. The following table lists external data types.

Table 4-2 External Data Types

Name	Code	Description
CHAR	1	<= 65535-byte, variable-length character string ()
	96	<= 65535-byte, fixed-length character string ()
CHARF	96	<= 65535-byte, fixed-length character string
CHARZ	97	<= 65535-byte, fixed-length, null-terminated string ()
DATE	12	7-byte, fixed-length date/time value
DECIMAL	7	COBOL packed decimal
DISPLAY	91	COBOL numeric character string with leading sign
DISPLAY TRAILING	152	COBOL numeric with trailing sign

Table 4-2 (Cont.) External Data Types

Name	Code	Description
FLOAT	4	4-byte or 8-byte floating-point number
INTEGER	3	2-byte, 4-byte, or 8-byte signed integer.
LONG	8	<= 2147483647-byte, fixed-length string
LONG RAW	24	<= 217483647-byte, fixed-length binary data
LONG VARCHAR	94	<= 217483643-byte, variable-length string
LONG VARRAW	95	<= 217483643-byte, variable-length binary data
NUMBER	2	Internal Oracle Format Number represented in binary coded decimal format.
OVERPUNCH LEADING	172	COBOL numeric character string with embedded leading sign
OVERPUNCH TRAILING	154	COBOL numeric character string with embedded trailing sign (equivalent to declarations of the form PIC S9(n)V9(m) DISPLAY)
RAW	23	<= 65535-byte, fixed-length binary data ()
ROWID	11	fixed-length binary value (system-specific)
STRING	5	<= 65535-byte, null-terminated character string ()
UNSIGNED	68	2-byte or 4-byte unsigned integer
UNSIGNED DISPLAY	153	COBOL unsigned numeric
VARCHAR	9	<= 65533-byte, variable-length character string
VARCHAR2	1	<= 65535-byte, variable-length character string ()
VARNUM	6	variable-length binary number
VARRAW	15	<= 65533-byte, variable-length binary data
BOOLEAN		Boolean value (True, False, or Unknown)

#### Notes:

CHAR is datatype 1 when PICX=VARCHAR2 and datatype 96 when PICX=CHARF.

Maximum size is 32767 (32K) on some platforms.



All forms of LONG data types (LONG, LONG RAW, LONG VARCHAR, LONG VARRAW) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the LONG data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use CLOB and NCLOB data types for large amounts of character data.

For more information, see:

Migrating Columns from LONGs to LOBs



### 4.1.2.1 CHAR

CHAR behavior depends on the settings of the option PICX. See "PICX".

### 4.1.2.2 CHARF

By default, the CHARF datatype represents all non-varying character host variables. You use the CHARF datatype to store fixed-length character strings. On most platforms, the maximum length of a CHARF value is 65535 (64K) bytes. See "PICX".

**On Input**. Oracle reads the number of bytes specified for the input host variable, does *not* strip trailing blanks, then stores the input value in the target database column.

If the input value is longer than the defined width of the database column, Oracle generates an error. If the input value is all-blank, then a string of spaces is stored.

**On Output**. Oracle returns the number of bytes specified for the output host variable, blank-padding if necessary, then assigns the output value to the target host variable. If a NULL is returned, then the original value of the variable is not overwritten.

If the output value is longer than the declared length of the host variable, Oracle truncates the value before assigning it to the host variable. If an indicator variable is available, Oracle sets it to the original length of the output value.

### 4.1.2.3 CHARZ

The CHARZ datatype represents fixed-length, null-terminated character strings. On most platforms, the maximum length of a CHARZ value is 65535 bytes. You usually will not need this external type in Pro\*COBOL.

#### 4.1.2.4 DATE

The DATE datatype represents dates and times in 7-byte, fixed-length fields. As Table 4-3 shows, the century, year, month, day, hour (in 24-hour format), minute, and second are stored in that order from left to right.

Table 4-3 Date Format

Byte	1	2	3	4	5	6	7
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example 17-Oct-1994 at 1:23:12 PM	119	194	10	17	14	24	13

The century and year bytes are in excess-100 notation. The hour, minute, and second are in excess-1 notation. Dates before the Common Era (B.C.E.) are less than 100. The epoch is January 1, 4712 B.C.E. For this date, the century byte is 53 and the year byte is 88. The hour byte ranges from 1 to 24. The minute and second bytes range from 1 to 60. The time defaults to midnight (1, 1, 1). Pro\*COBOL also supports five additional datetime datetypes, as described in "Datetime and Interval Datatype Descriptors".



### 4.1.2.5 DECIMAL

The DECIMAL datatype represents packed decimal numbers for calculation. In COBOL, the host variable must be a signed COMP-3 field with an implied decimal point. If significant digits are lost during data conversion, the value is truncated to the declared length.

### 4.1.2.6 DISPLAY

The DISPLAY datatype represents numeric character data. The DISPLAY datatype refers to a COBOL "DISPLAY SIGN LEADING SEPARATE" number, which requires n + 1 bytes of storage for PIC S9(n), and n + d + 1 bytes of storage for PIC S9(n)V9(d).

#### 4.1.2.7 FLOAT

The FLOAT datatype represents numbers that have a fractional part or that exceed the capacity of the INTEGER datatype. FLOAT relates to the COBOL datatypes COMP-1 (4-byte floating point) and COMP-2 (8-byte floating point).

Oracle can represent numbers with greater precision than floating point implementations because the internal format of Oracle numbers is decimal.

**Note:** In SQL statements, when comparing FLOAT values, use the SQL function ROUND because FLOAT stores binary (not decimal) numbers; so, fractions do not convert exactly.

### 4.1.2.8 INTEGER

The INTEGER datatype represents numbers that have no fractional part. An integer is a signed, 2-byte, 4-byte, or 8-byte binary number. The order of the bytes in a word is platform-dependent. You must specify a length for input and output host variables. On output, if the column has a fractional part, the digits after the decimal point are truncated.

#### 4.1.2.9 LONG

The LONG datatype represents fixed-length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG value is 2147483647 bytes (two gigabytes).

### 4.1.2.10 LONG RAW

The LONG RAW datatype represents fixed-length, binary data or byte strings. The maximum length of a LONG RAW value is 2147483647 bytes (two gigabytes).

LONG RAW data is like LONG data, except that Oracle assumes nothing about the meaning of LONG RAW data and does no character set conversions when you transmit LONG RAW data from one system to another.

### 4.1.2.11 LONG VARCHAR

The LONG VARCHAR datatype represents variable-length character strings. LONG VARCHAR variables have a 4-byte length field followed by a string field. The maximum



length of the string field is 2147483643 bytes. In an EXEC SQL VAR statement, do *not* include the 4-byte length field.

### 4.1.2.12 LONG VARRAW

The LONG VARRAW datatype represents binary data or byte strings. LONG VARRAW variables have a 4-byte length field followed by a data field. The maximum length of the data field is 2147483643 bytes. In an EXEC SQL VAR statement, do *not* include the 4-byte length field.

### 4.1.2.13 NUMBER

The NUMBER datatype represents the internal Oracle NUMBER format which cannot be represented by a COBOL datatype.

### 4.1.2.14 OVER-PUNCH

OVER-PUNCH is the default signed numeric for the COBOL language. Digits are held in ASCII or EBCDIC format in radix 10, with one digit for each byte of computer storage. The sign is held in the high order nibble of one of the bytes. It is called OVER-PUNCH because the sign is "punched-over" the digit in either the first or last byte. The default sign position will be over the trailing byte. PIC S9(n)V9(m) TRAILING or PIC S9(n)V9(m) LEADING is used to specify the over-punch.

### 4.1.2.15 RAW

The RAW datatype represents fixed-length binary data or byte strings. On most platforms, the maximum length of a RAW value is 65535 bytes.

RAW data is like CHAR data, except that Oracle assumes nothing about the meaning of RAW data and does no character set conversions when you transmit RAW data from one system to another.

#### 4.1.2.16 ROWID

The ROWID datatype is the database row identifier in COBOL. To support both logical and physical ROWIDs (as well as ROWIDs of non-Oracle tables) the Universal ROWID (UROWID) was defined. Use the SQL-ROWID pseudotype for this datatype (see "Universal ROWIDs").

You can use VARCHAR2 host variables to store ROWIDs in a readable format. When you select or fetch a ROWID into a VARCHAR2 host variable, Oracle converts the binary value to an 18-byte character string and returns it in the format:

#### BBBBBBBB.RRRR.FFFF

where BBBBBBB is the block in the database file, RRRR is the row in the block (the first row is 0), and FFFF is the database file. These numbers are hexadecimal. For example, the ROWID:

0000000E.000A.0007

points to the 11th row in the 15th block in the 7th database file.



Typically, you fetch a ROWID into a VARCHAR2 host variable, and hen compare the host variable to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement. That way, you can identify the latest row fetched by a cursor. For an example, see "Mimicking the CURRENT OF Clause".

**Note:** If you need full portability or your application communicates with a non-Oracle database through Transparent Gateway, specify a maximum length of 256 (not 18) bytes when declaring the VARCHAR2 host variable. If your application communicates with a non-Oracle data source through Oracle Open Gateway, specify a maximum length of 256 bytes. Though you can assume nothing about its contents, the host variable will behave normally in SQL statements.

#### 4.1.2.17 STRING

The STRING datatype is like the VARCHAR2 datatype except that a STRING value is always terminated by a LOW-VALUE character. This datatype is usually not used in Pro\*COBOL.

#### 4.1.2.18 UNSIGNED

The UNSIGNED datatype represents unsigned integers. This datatype is usually not used in Pro\*COBOL.

#### 4.1.2.19 VARCHAR

The VARCHAR datatype represents variable-length character strings. VARCHAR variables have a 2-byte length field followed by a 65533-byte string field. However, for VARCHAR array elements, the maximum length of the string field is 65530 bytes. When you specify the length of a VARCHAR variable, be sure to include 2 bytes for the length field. For longer strings, use the LONG VARCHAR datatype. In an EXEC SQL VAR statement, do *not* include the 2-byte length field.

#### 4.1.2.20 VARCHAR2

The VARCHAR2 datatype represents variable-length character strings. On most platforms, the maximum length of a VARCHAR2 value is 65535 bytes.

Specify the maximum length of a VARCHAR2(n) value in bytes, not characters. So, if a VARCHAR2(n) variable stores multibyte characters, its maximum length is less than n characters.

**On Input.** Oracle reads the number of bytes specified for the input host variable, strips any trailing blanks, and then stores the input value in the target database column.

If the input value is longer than the defined width of the database column, Oracle generates an error. If the input value is all SPACES, Oracle treats it like a NULL.

Oracle can convert a character value to a NUMBER column value if the character value represents a valid number. Otherwise, Oracle generates an error.

**On Output.** Oracle returns the number of bytes specified for the output host variable, blank-padding if necessary, and then assigns the output value to the target host variable. If a NULL is returned, Oracle fills the host variable with blanks.



If the output value is longer than the declared length of the host variable, Oracle truncates the value before assigning it to the host variable. If an indicator variable is available, Oracle sets it to the original length of the output value.

Oracle can convert NUMBER column values to character values. The length of the character host variable determines precision. If the host variable is too short for the number, scientific notation is used. For example, if you select the column value 123456789 into a host variable of length 6, Oracle returns the value 1.2E08 to the host variable.

#### 4.1.2.21 VARNUM

The VARNUM datatype is similar in format to NUMBER and is usually not used in Pro\*COBOL.

#### 4.1.2.22 VARRAW

The VARRAW datatype represents variable-length binary data or byte strings. The VARRAW datatype is like the RAW datatype, except that VARRAW variables have a 2-byte length field followed by a <= 65533-byte data field. For longer strings, use the LONG VARRAW datatype. In an EXEC SQL VAR statement, do *not* include the 2-byte length field. To get the length of a VARRAW variable, simply refer to its length field.

## 4.1.2.23 SQL Pseudocolumns and Functions

SQL recognizes the pseudocolumns listed in Table 4-4, which return specific data items.

Table 4-4 Pseudocolumns and Internal Datatypes

Pseudocolumn	Internal Datatype
CURRVAL	NUMBER
LEVEL	NUMBER
NEXTVAL	NUMBER
ROWID	ROWID
ROWNUM	NUMBER

Pseudocolumns are not actual columns in a table. However, pseudocolumns are treated like columns, so their values must be SELECTed from a table. Sometimes it is convenient to select pseudocolumn values from a dummy table.

In addition, SQL recognizes the functions without parameters listed in Table 4-5, which also return specific data items.

Table 4-5 Functions and Internal Datatypes

Function	Internal Datatype
SYSDATE	DATE
UID	NUMBER
USER	VARCHAR2



You can refer to SQL pseudocolumns and functions in SELECT, INSERT, UPDATE, and DELETE statements. In the following example, you use SYSDATE to compute the number of months since an employee was hired:

```
EXEC SQL SELECT MONTHS_BETWEEN(SYSDATE, HIREDATE)

INTO :MONTHS-OF-SERVICE

FROM EMP

WHERE EMPNO = :EMP-NUMBER

END EXEC.
```

Brief descriptions of the SQL pseudocolumns and functions follow.

**CURRVAL** returns the current number in a specified sequence. Before you can reference CURRVAL, you must use NEXTVAL to generate a sequence number.

**LEVEL** returns the level number of a node in a tree structure. The root is level 1, children of the root are level 2, grandchildren are level 3, and so on.

LEVEL is used in the SELECT CONNECT BY statement to incorporate some or all the rows of a table into a tree structure. In an ORDER BY or GROUP BY clause, LEVEL segregates the data at each level in the tree.

Specify the direction in which the query walks the tree (down from the root or up from the branches) with the PRIOR operator. In the START WITH clause, you can specify a condition that identifies the root of the tree.

**NEXTVAL** returns the next number in a specified sequence. After creating a sequence, you can use it to generate unique sequence numbers for transaction processing. In the following example, the sequence named *partno* assigns part numbers:

```
EXEC SQL INSERT INTO PARTS

VALUES (PARTNO.NEXTVAL, :DESCRIPTION, :QUANTITY, :PRICE END EXEC.
```

If a transaction generates a sequence number, the sequence is incremented when you commit or roll back the transaction. A reference to NEXTVAL stores the current sequence number in CURRVAL.

**ROWNUM** returns a number indicating the sequence in which a row was selected from a table. The first row selected has a ROWNUM of 1, the second row has a ROWNUM of 2, and so on. If a SELECT statement includes an ORDER BY clause, ROWNUMs are assigned to the selected rows *before* the sort is done.

You can use ROWNUM to limit the number of rows returned by a SELECT statement. Also, you can use ROWNUM in an UPDATE statement to assign unique values to each row in a table. Using ROWNUM in the WHERE clause does not stop the processing of a SELECT statement; it just limits the number of rows retrieved. The only meaningful use of ROWNUM in a WHERE clause is:

```
... WHERE ROWNUM < constant END-EXEC.
```

because the value of ROWNUM increases only when a row is retrieved. The following search condition can never be met because the first four rows are not retrieved:

```
... WHERE ROWNUM = 5 END-EXEC.
```

**SYSDATE** returns the current date and time. SYSDATE uses the time zone of either the database host system or the database, depending on the setting of the



 ${\tt TIME\_AT\_DBTIMEZONE} \ initialization\ parameter.\ See\ Oracle\ Database\ Reference: \\ {\tt TIME\_AT\_DBTIMEZONE} \ for\ more\ information.$ 

**UID** returns the unique ID number assigned to an Oracle user.

**USER** returns the username of the current Oracle user.

#### 4.1.2.24 BOOLEAN

Applications can insert TRUE or FALSE values in BOOLEAN type column while inserting data in the database. The values can be:

- "TRUE" or "true"
- "FALSE" or "false"

Any non-zero integer represents TRUE and the integer zero (0) represents FALSE.

Unless prohibited by a  $\mathtt{NOT}$   $\mathtt{NULL}$  constraint, the boolean data type also supports the truth value  $\mathtt{UNKNOWN}$  as the null value.

# 4.2 Datetime and Interval Datatype Descriptors

The OCI datetime and interval datatypes supported by Pro\*COBOL are briefly summarized here.



*Oracle Database SQL Language Reference for more a more complete discussion datetime datatype descriptors* 

#### **ANSI DATE**

The ANSI DATE is based on the DATE, but contains no time portion. (Therefore, it also has no time zone.) ANSI DATE follows the ANSI specification for the DATE datatype. When assigning an ANSI DATE to a DATE or a timestamp datatype, the time portion of the Oracle DATE and the timestamp are set to zero. When assigning a DATE or a timestamp to an ANSI DATE, the time portion is ignored.

You are encouraged to instead use the TIMESTAMP datatype which contains both date and time.

#### **TIMESTAMP**

The TIMESTAMP datatype is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, plus the hour, minute, and second values. It has no time zone. The TIMESTAMP datatype has the form:

TIMESTAMP(fractional seconds precision)

where <code>fractional\_seconds\_precision</code> (which is optional) specifies the number of digits in the fractional part of the <code>SECOND</code> datetime field and can be a number in the range 0 to 9. The default is 6.



#### TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE (TSTZ) is a variant of TIMESTAMP that includes an explicit time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The TIMESTAMP WITH TIME ZONE datatype has the form:

TIMESTAMP (fractional seconds precision) WITH TIME ZONE

where <code>fractional\_seconds\_precision</code> optionally specifies the number of digits in the fractional part of the <code>SECOND</code> datetime field and can be a number in the range 0 to 9. The default is 6.

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the data.

#### TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ) is another variant of TIMESTAMP that includes a time zone displacement in its value. Storage is in the same format as for TIMESTAMP. This type differs from TIMESTAMP WITH TIME ZONE in that data stored in the database is normalized to the database time zone, and the time zone displacement is not stored as part of the column data. When users retrieve the data, Oracle returns it in the users' local session time zone.

The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The TIMESTAMP WITH LOCAL TIME ZONE datatype has the form:

TIMESTAMP(fractional seconds precision) WITH LOCAL TIME ZONE

where <code>fractional\_seconds\_precision</code> optionally specifies the number of digits in the fractional part of the <code>SECOND</code> datetime field and can be a number in the range 0 to 9. The default is 6.

#### INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. The INTERVAL YEAR TO MONTH datatype has the form:

INTERVAL YEAR (year precision) TO MONTH

where the optional <code>year\_precision</code> is the number of digits in the <code>YEAR</code> datetime field. The default value of <code>year\_precision</code> is 2.

#### INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds. The INTERVAL DAY TO SECOND datatype has the form:

INTERVAL DAY (day precision) TO SECOND(fractional seconds precision)

#### where:

day\_precision is the number of digits in the DAY datetime field. It is optional.
 Accepted values are 0 to 9. The default is 2.



• fractional\_seconds\_precision is the number of digits in the fractional part of the SECOND datetime field. It is optional. Accepted values are 0 to 9. The default is 6.



To avoid unexpected results in your DML operations on datetime data, you can verify the database and session time zones by querying the built-in SQL functions DBTIMEZONE and SESSIONTIMEZONE. If the time zones have not been set manually, Oracle uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, Oracle uses UTC as the default value.

## 4.3 Host Variables

Host variables are the key to communication between your host program and the server. Typically, a host program inputs data to the server, and the server outputs data to the program. The server stores input data in database columns and stores output data in program host variables.

## 4.3.1 Declaring Host Variables

Host variables are declared according to COBOL rules, using the COBOL datatypes that are supported by Pro\*COBOL. COBOL datatypes must be compatible with the source/target database column.

The supported COBOL variable declarations, descriptions, corresponding external datatypes, and Oracle datatype codes are shown in Table 4-6.

**Table 4-6 Host Variable Declarations** 

Variable Declaration	Description	External Datatype	Type Code
PIC XX	fixed-length string of 1-byte characters (1)	CHARF	96
PIC X(n)	n-length string of 1-byte characters	VARCHAR	9
PIC XX VARYING	variable-length string of 1-byte characters (1,2)		
PIC X(n) VARYING	variable-length ( $n$ max.) string of 1-byte characters (2)		
PIC NN	fixed-length string of multibyte NCHAR	CHARF	96
PIC GG	characters (1,3)		
PIC N(n)	n-length string of multibyte NCHAR characters	VARCHAR	9
PIC G(n)	(3)		
PIC NN VARYING	variable-length string of multibyte characters (2,3)		
PIC N(n) VARYING	variable-length (n max.) string of multibyte		
PIC GG VARYING	characters (2,3)		
PIC G(n) VARYING	(2,0)		



Table 4-6 (Cont.) Host Variable Declarations

Variable Declaration	Description	External Datatype	Type Code
PIC S99 BINARY PIC S9(n) BINARY PIC S99 COMP PIC S9(n) COMP PIC S99 COMP-4 PIC S9(n) COMP-4	integer (4,5,7)	INTEGER	3
PIC S99 COMP-5 PIC S9( <i>n</i> ) COMP-5	byte-swapped integer (4,5,6,7)	INTEGER	3
COMP-1 COMP-2	floating-point number (5)	FLOAT	4
PIC S99[V99] COMP-3 PIC S9(n)[V9(n)] COMP-3 PIC S99[V99] PACKED-DECIMAL PIC S9(n)[V9(n)] PACKED-DECIMAL	packed-decimal (4,5)	DECIMAL	7
PIC S99[V99] DISPLAY	display leading (8,11)	DISPLAY	91
SIGN LEADING SEPARATE PIC S9(n)[V9(m)] DISPLAY SIGN LEADING SEPARATE PIC S99[V99] DISPLAY SIGN TRAILING SEPARATE PIC S9(n)[V9(m)] DISPLAY SIGN TRAILING SEPARATE	display trailing (8)	DISPLAY TRAILING	152
PIC 99 DISPLAY PIC 9(n)[V9(m)] DISPLAY	unsigned display (9)	UNSIGNED DISPLAY	153
PIC S99[V99] DISPLAY SIGN TRAILING PIC S9(n)[V9(m)] DISPLAY	over-punch trailing (9) over-punch leading (9)	OVER-PUNCH TRAILING	154 172
SIGN TRAILING PIC S99[V99] DISPLAY SIGN LEADING PIC S9(n)[V9(m)] DISPLAY SIGN LEADING		OVER-PUNCH LEADING	
SQL-CURSOR	cursor variable		
SQL-CONTEXT SQL-ROWID	runtime context universal ROWID	UROWID	104
SQL-ROWID SQL-BFILE	**************************************	BFILE	104
SQL-BLOB	external binary file	BLOB	112
SQL-BLOB	binary LOB character LOB	CLOB	113



#### Notes:

- 1. X...X and 9...9 stand for a given number (*n*) of Xs or 9s. For variable-length strings, *n* is the maximum length.
- 2. The keyword VARYING assigns the VARCHAR external datatype to a character string. For more information, see "Declaring VARCHAR Variables".
- 3. Before using the PIC N or PIC G datatype in your Pro\*COBOL source files, verify that it is supported by your COBOL compiler.
- **4.** Only signed numbers (PIC S...) are allowed. For floating-point numbers, however, PIC strings are not accepted.
- Not all COBOL compilers support all of these datatypes.
- 6. With COMP or COMP-5, the number cannot have a fractional part; scaled binary numbers are not supported. COMP-5 is not supported on the SPARC Solaris 64 bit platform, use COMP instead.
- 7. The maximum value of *the integer* is *n* to 18. This may vary, depending on the operating system and the compilers for Pro\*Cobol and Cobol.
- 8. Both DISPLAY and SIGN are optional.
- 9. DISPLAY is optional
- 10. If TRAILING is omitted, the embedded sign position is operating-system dependent.
- 11. LEADING is optional.

In Table 4-6 and Table 4-7 the symbols '[' and ']' denote that an optional entry is contained inside. The symbols '{' and '}' denote that a choice must be made between tokens separated by the symbol '|'.

Table 4-7 shows all the COBOL datatypes that can be converted to and from each internal datatype.

Table 4-7 Compatible Oracle Internal Datatypes

Internal Datatype	Notes	COBOL Datatype	Description
CHAR(x)	(1)	PIC X(n)	character string
VARCHAR2(y)	<b>(1)</b>	PIC XX	n-character string
		PIC X(n) VARYING	variable-length string
		PIC XX VARYING	integer
		PIC S99 COMP	integer
		PIC S9(n) COMP	
		PIC S99 BINARY	integer
		PIC S9(n) BINARY	floating point number
		PIC S99 COMP-5	packed decimal
		PIC S9(n) COMP-5	display
		COMP-1	5.52.53
		COMP-2	
		PIC S99[V99] COMP-3	
		PIC S9(n)[V9(n)] COMP-3	
		PIC S99[V99] DISPLAY	
		PIC S9(n)[V9(n)] DISPLAY	



Table 4-7 (Cont.) Compatible Oracle Internal Datatypes

	,		
Internal Datatype	Notes	COBOL Datatype	Description
NCHAR(u)	(2)	PIC {NN   GG}	national character string
NVARCHAR2(v)	(2)	PIC { N(n)   G(n)}	n-national character string
BLOB		SQL-BLOB	binary LOB
CLOB		SQL-CLOB	character LOB
NCLOB		SQL-NCLOB	national character LOB
BFILE		SQL-BFILE	external binary file
NUMBER	(3)	PIC S99 COMP	integer
NUMBER $(p,s)$		PIC S9(n) COMP	integer
		PIC S99 BINARY	integer
		PIC S9( <i>n</i> ) BINARY PIC S99 COMP-5	floating point number
		PIC S9(n) COMP-5	packed decimal
		COMP-1	display
		COMP-2 PIC S99V99 COMP-3 PIC S9(n)V9(n) COMP-3 PIC S99V99 DISPLAY PIC S9(n)V9(n) DISPLAY	character string (4)  n-character string (4)  variable-length string  n-byte variable-length string
		PIC [XX   NN   GG]  PIC [X(n)   N(n)   G(n)]  PIC XX VARYING  PIC X(n) VARYING	
DATE	<b>(5)</b>	PIC X(n)	n-byte character string
LONG	<b>(1)</b>	PIC XX	character string
RAW		PIC X(n)	<i>n</i> -byte variable-length string
LONG RAW ROWID	(6)	PIC X(n) VARYING PIC XX VARYING	universal rowid
		SQL-ROWID	
BOOLEAN		PIC 9(1)	BINARY integer
		PIC X(5)	BINARY character string

#### Notes:

Bug 7225844 - Following para - substituted  $1 \le x \le 2000$  bytes, default is 1 for

<= x < =2000 bytes, default is 1.

- 1.  $1 \le x \le 2000$  bytes, default is 1.  $1 \le y \le 4000$  bytes, default is 1.
- 2. 1<=u<=2000 bytes, default is 1. 1<=v<=4000 bytes, default is 1.
- **3.** *p* ranges from 2 to 38. s ranges from -84 to 127.
- **4.** Strings can be converted to NUMBERs only if they consist of convertible characters 0 to 9, period (.), +, -, E, e. The Globalization Support (formerly



- called National Language Support or NLS) settings for your system might change the decimal point from a period (.) to a comma (,).
- 5. When converted to a string type, the default size of a DATE depends on the NCHAR settings in effect on your system. When converted to a binary value, the length is 7 bytes.
- 6. When converted to a string type, a ROWID requires from 18 to 4000 bytes. ROWID can also be converted to a character type. Oracle recommends the use of SQL-ROWID for all new programs.

## 4.3.1.1 Example Declarations

The following example declares several host variables for later use:

```
01 STR1 PIC X(3).

01 STR2 PIC X(3) VARYING.

01 NUM1 PIC S9(5) COMP.

01 NUM2 COMP-1.

01 NUM3 COMP-2.
```

You can also declare one-dimensional tables of simple COBOL types, as the next example shows:

```
01 XMP-TABLES.
05 TAB1 PIC XXX OCCURS 3 TIMES.
05 TAB2 PIC XXX VARYING OCCURS 3 TIMES.
05 TAB3 PIC S999 COMP-3 OCCURS 3 TIMES.
```

### 4.3.1.2 Initialization

You can initialize host variables, except pseudotype host variables, using the VALUE clause, as shown in the following example:

```
01 USERNAME PIC X(10) VALUE "SCOTT".
01 MAX-SALARY PIC S9(4) COMP VALUE 5000.
```

If a string value assigned to a character variable is shorter than the declared length of the variable, the string is blank-padded on the right. If the string value assigned to a character variable is longer than the declared length, the string is truncated.

No error or warning is issued, but any VALUES clause on a pseudotype variable is ignored and discarded.

### 4.3.1.3 Restrictions

You cannot use alphabetic character (PIC A) variables or edited data items as host variables. Therefore, the following variable declarations cannot be made for *host* variables:

```
01 AMOUNT-OF-CHECK PIC ****9.99.
01 FIRST-NAME PIC A(10).
01 BIRTH-DATE PIC 99/99/99.
```



## 4.3.2 Referencing Host Variables

Host variables are used in SQL data manipulation statements. A host variable must be prefixed with a colon (:) in SQL statements but must not be prefixed with a colon in COBOL statements, as this example shows:

```
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

1 EMP-NUMBER PIC S9(4) COMP VALUE ZERO.

1 EMP-NAME PIC X(10) VALUE SPACE.

1 SALARY PIC S9(5) V99 COMP-3.

EXEC SQL END DECLARE SECTION END-EXEC.

...

PROCEDURE DIVISION.

DISPLAY "Employee number? " WITH NO ADVANCING.
ACCEPT EMP-NUMBER.

EXEC SQL SELECT ENAME, SAL

INTO :EMP-NAME, :SALARY FROM EMP

WHERE EMPNO = :EMP-NUMBER

END-EXEC.

COMPUTE BONUS = SALARY / 10.
```

Though it might be confusing, you can give a host variable the same name as a table or column, as the following example shows:

```
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

11 EMPNO PIC S9(4) COMP VALUE ZERO.

11 ENAME PIC X(10) VALUE SPACE.

11 COMM PIC S9(5) V99 COMP-3.

EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.

EXEC SQL SELECT ENAME, COMM

INTO :ENAME, :COMM FROM EMP

WHERE EMPNO = :EMPNO

END-EXEC.
```

## 4.3.2.1 Group Items as Host Variables

Pro\*COBOL allows the use of group items in embedded SQL statements. Group items with elementary items (containing only one level) can be used as host variables. The host group items (also referred to as host structures) can be referenced in the INTO clause of a SELECT or a FETCH statement, and in the VALUES list of an INSERT statement. When a group item is used as a host variable, only the group name is used in the SQL statement. For example, given the following declaration

```
01 DEPARTURE.

05 HOUR PIC X(2).

05 MINUTE PIC X(2).
```

the following statement is valid:



```
EXEC SQL SELECT DHOUR, DMINUTE
INTO :DEPARTURE
FROM SCHEDULE
WHERE ...
```

The order that the members are declared in the group item must match the order that the associated columns occur in the SQL statement, or in the database table if the column list in the INSERT statement is omitted. Using a group item as a host variable has the semantics of substituting the group item with elementary items. In the preceding example, it would mean substituting: DEPARTURE with: DEPARTURE.HOUR, :DEPARTURE.MINUTE.

Group items used as host variables can contain host tables. In the following example, the group item containing tables is used to INSERT three entries into the SCHEDULE table:

```
01 DEPARTURE.

05 HOUR PIC X(2) OCCURS 3 TIMES.

05 MINUTE PIC X(2) OCCURS 3 TIMES.

...

EXEC SQL INSERT INTO SCHEDULE (DHOUR, DMINUTE)

VALUES (:DEPARTURE) END-EXEC.
```

If VARCHAR=YES is specified, Pro\*COBOL will recognize implicit VARCHARs. If the nested group item declaration resembles a VARCHAR host variable, then the entire group item is treated like an elementary item of VARYING type. See "VARCHAR".

When referencing elementary items instead of the group items as host variables elementary names need not be unique because you can qualify them using the following syntax:

```
group item.elementary item
```

This naming convention is allowed only in SQL statements. It is similar to the IN (or OF) clause in COBOL, examples of which follow:

```
MOVE MINUTE IN DEPARTURE TO MINUTE-OUT. DISPLAY HOUR OF DEPARTURE.
```

The COBOL IN (or OF) clause is *not* allowed in SQL statements. Qualify elementary names to avoid ambiguity. For example:

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

01 DEPARTURE.

05 HOUR PIC X(2).

05 MINUTE PIC X(2).

01 ARRIVAL.

05 HOUR PIC X(2).

05 MINUTE PIC X(2).

EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL SELECT DHR, DMIN INTO :DEPARTURE.HOUR, :DEPARTURE.MINUTE FROM TIMETABLE

WHERE ...
```

### 4.3.2.2 Restrictions

A host variable cannot substitute for a column, table, or other object in a SQL statement and must not be an Oracle reserved word. See Reserved Words\_ Keywords\_ and Namespaces for a list of reserved words and keywords.



## 4.4 Indicator Variables

You can associate any host variable with an optional indicator variable. Each time the host variable is used in a SQL statement, a result code is stored in its associated indicator variable. Thus, indicator variables let you monitor host variables.

You use indicator variables in the VALUES or SET clause to assign NULLs to input host variables and in the INTO clause to detect NULLs (or truncated values for character columns) in output host variables.

## 4.4.1 Using Indicator Variables

Here are the values indicator variables can take on.

## 4.4.1.1 On Input

The values your program can assign to an indicator variable have the following meanings:

Indicator Variables	Description
-1	Oracle will assign a NULL to the column, ignoring the value of the host variable.
>=0	Oracle will assign the value of the host variable to the column.

## 4.4.1.2 On Output

The values Oracle can assign to an indicator variable have the following meanings:

Indicator Variables	Description
-1	The column value is NULL, so the value of the host variable is indeterminate.
0	Oracle assigned an intact column value to the host variable.
>0	Oracle assigned a truncated column value to the host variable. The integer returned by the indicator variable is the original length of the column value, and SQLCODE in SQLCA is set to zero.
-2	Oracle assigned a truncated column variable to the host variable, but the original column value could not be determined (a LONG column, for example).

## 4.4.2 Declaring Indicator Variables

An indicator variable must be explicitly declared as PIC S9(4) COMP and must not be a reserved word. In the following example, you declare an indicator variable named COMM-IND (the name is arbitrary):

```
WORKING-STORAGE SECTION.
...
01 EMP-NAME PIC X(10) VALUE SPACE.
01 SALARY PIC S9(5)V99 COMP-3.
01 COMMISSION PIC S9(5)V99 COMP-3.
```



```
01 COMM-IND PIC S9(4) COMP.
```

## 4.4.3 Referencing Indicator Variables

In SQL statements, an indicator variable must be prefixed with a colon and appended to its associated host variable. In COBOL statements, an indicator variable must *not* be prefixed with a colon or appended to its associated host variable. An example follows:

```
EXEC SQL SELECT SAL, COMM

INTO :SALARY, :COMMISSION:COMM-IND FROM EMP
WHERE EMPNO = :EMP-NUMBER

END-EXEC.

IF COMM-IND = -1
COMPUTE PAY = SALARY

ELSE

COMPUTE PAY = SALARY + COMMISSION.
```

To improve readability, you can precede any indicator variable with the optional keyword INDICATOR. You must still prefix the indicator variable with a colon. The correct syntax is

```
: host\_variable \verb|INDICATOR|: indicator\_variable|
```

#### and is equivalent to

```
:host variable:indicator variable
```

You can use both forms of expression in your host program.

### 4.4.3.1 Use in Where Clauses

Indicator variables *cannot* be used in the WHERE clause to search for NULLs. For example, the following DELETE statement triggers an error at run time:

```
* Set indicator variable.

COMM-IND = -1

EXEC SQL

DELETE FROM EMP WHERE COMM = :COMMISSION:COMM-IND
END-EXEC.
```

#### The correct syntax follows:

```
EXEC SQL DELETE FROM EMP WHERE COMM IS NULL END-EXEC.
```

## 4.4.3.2 Avoid Error Messages

If you SELECT or FETCH a NULL into a host variable that has no indicator, Oracle issues an error message.

You can disable the error message by also specifying UNSAFE\_NULL=YES on the command line

#### **Related Topics**

Precompiler Options



## 4.4.3.3 ANSI Requirements

When MODE=ORACLE, if you SELECT or FETCH a truncated column value into a host variable that is not associated with an indicator variable, Oracle issues an error message.

However, when MODE={ANSI | ANSI14 | ANSI13}, no error is generated.



Embedded SQL for values for indicator variables.

## 4.4.3.4 Indicator Variables for Multibyte NCHAR Variables

Indicator variables for multibyte NCHAR character variables can be used as with any other host variable. However, a positive value (the result of a SELECT or FETCH was truncated) represents the string length in multibyte characters instead of 1-byte characters.

## 4.4.3.5 Indicator Variables with Host Group Items

To use indicator variables with a host group item, either setup a second group item that contains an indicator variable for each nullable variable in the group item or use a table of half-word integer variables. You do NOT have to have an indicator variable for each variable in the group item, but the nullable fields which you wish to use indicators for must be placed at the beginning of the data group item. The following indicator group item can be used with the DEPARTURE group item:

```
01 DEPARTURE-IND.

05 HOUR-IND PIC S9(4) COMP.

05 MINUTE-IND PIC S9(4) COMP.
```

If you use an indicator table, you do NOT have to declare a table of as many elements as there are members in the host group item. The following indicator table can be used with the DEPARTURE group item:

```
01 DEPARTURE-IND PIC S9(4) COMP OCCURS 2 TIMES.
```

Reference the indicator group item in the SQL statement in the same way that a host indicator variable is referenced:

```
EXEC SQL SELECT DHOUR, DMINUTE
INTO :DEPARTURE:DEPARTURE-IND
FROM SCHEDULE
WHERE ...
```

When the query completes, the NULL/NOT NULL status of each selected component is available in the host indicator group item. The restrictions on indicator host variables and the ANSI requirements also apply to host indicator group items.



## 4.5 VARCHAR Variables

COBOL string datatypes are fixed length. However, Pro\*COBOL lets you declare a variable-length string pseudotype called VARCHAR. A VARCHAR variable is a pseudotype that enables you to specify the exact length of the data stored in the database and to specify the exact length of the data to be passed to the database.

## 4.5.1 Declaring VARCHAR Variables

You define a VARCHAR host variable by adding the keyword VARYING to its declaration, as shown in the following example:

```
01 ENAME PIC X(15) VARYING.
```



PIC N and PIC G are not allowed in definitions that use VARYING. To see how to correctly use PIC N and PIC G in VARCHAR variables, see "Implicit VARCHAR Group Items".

#### Note:

Pro\*COBOL adds 4 characters to variable names declared with VARYING. So, if your COBOL compiler supports a variable name length of n, then Pro\*COBOL supports a variable name that is four characters less (n-4).

The COBOL VARYING phrase is used in PERFORM and SEARCH statements to increment subscripts and indexes. Do not confuse this with the Pro\*COBOL VARYING clause in the preceding example.

VARCHAR is an extended Pro\*COBOL datatype or pre-declared group item. For example, Pro\*COBOL expands the VARCHAR declaration

```
01 ENAME PIC X(15) VARYING.
```

into a group item with length and string fields, as follows:

```
01 ENAME.

05 ENAME-LEN PIC S9(4) COMP.

05 ENAME-ARR PIC X(15).
```

The *length* field (suffixed with -LEN) holds the current length of the value stored in the *string* field (suffixed with -ARR). The maximum length in the VARCHAR host-variable declaration must be in the range of 1 to 9,999 bytes.

The advantage of using VARCHAR variables is that you can explicitly set and reference the length field. With input host variables, Pro\*COBOL reads the value of the length field and uses that many characters of the string field. With output host variables, Pro\*COBOL sets the length value to the length of the character string stored in the string field.

## 4.5.2 Implicit VARCHAR Group Items

Pro\*COBOL implicitly recognizes some group items as VARCHAR host variables when the precompiler option VARCHAR=YES is specified on the command line. For variable-length single-byte character types, use the following structure (*length* expressed in single-byte characters):

```
nn data-name-1.
49 data-name-2 PIC S9(4) COMP.
49 data-name-3 PIC X(length).
```

nn must be 01 through 48.

For variable-length multibyte NCHAR characters, use these formats (length is expressed in double-byte characters):

```
nn DATA-NAME-1.
49 DATA-NAME-2 PIC S9(4) COMP.
49 DATA-NAME-3 PIC N(length).
nn DATA-NAME-1.
49 DATA-NAME-2 PIC S9(4) COMP.
49 DATA-NAME-3 PIC G(length).
```

The elementary items in these group-item structures must be declared as level 49 for Pro\*COBOL to recognize them as VARCHAR host variables.

The VARCHAR=YES command line option must be specified for Pro\*COBOL to recognize the extended form of the VARCHAR group items. If VARCHAR=NO, then any declarations that resemble the preceding formats will be interpreted as regular group items. If VARCHAR=YES and a group item declaration format looks similar (but not identical) to the extended VARCHAR format, then the item will be interpreted as a regular group item rather than a VARCHAR group item. For example, if VARCHAR=YES is specified and you write the following:

```
01 LASTNAME.
48 LASTNAME-LEN PIC S9(4) COMP.
48 LASTNAME-TEXT PIC X(15).
```

then, since level 48 instead of 49 is used for the group item elements, the item is interpreted as a regular group item rather than a VARCHAR group item.



Precompiler Options for more information about the Pro\*COBOL VARCHAR option.

## 4.5.3 Referencing VARCHAR Variables

In SQL statements, you reference a VARCHAR variable using the group name prefixed with a colon, as the following example shows:

```
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```



```
01 PART-NUMBER PIC X(5).
01 PART-DESC PIC X(20) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
...

PROCEDURE DIVISION.
...

EXEC SQL
SELECT PDESC INTO :PART-DESC FROM PARTS
WHERE PNUM = :PART-NUMBER
END-EXEC.
```

After the query executes, PART-DESC-LEN holds the actual length of the character string retrieved from the database and stored in PART-DESC-ARR.

# 4.6 Handling Boolean Data

With boolean data type support, applications can:

- Bind boolean, integer, and char types of data in BOOLEAN columns.
- Fetch BOOLEAN column data in boolean, integer, and char data type variables.
- Fetch metadata about a BOOLEAN column.

### See Also:

- About Boolean Data Type Support
- Using Boolean Data Type

# 4.7 Handling Character Data

This section explains how Pro\*COBOL handles character host variables. There are two kinds of single-byte character host variables and two kinds of multibyte Globalization Support (formerly called NLS) character host variables:

- PIC X(n) (or PIC X...X)
- PIC X(n) VARYING (or PIC X...X VARYING)
- PIC N(n) (or PIC N...N) or PIC G(n) (or PIC G...G)

#### Note:

Before using multibyte NCHAR datatypes, verify that the PIC N or PIC G datatype is supported by your COBOL compiler.



### 4.7.1 Default for PIC X

The default datatype of PIC X variables is CHARF (was VARCHAR2 before release 8.0.) The precompiler command line option, PICX, is provided for backward compatibility. PICX can be entered only on the command line or in a configuration file. See "PICX" for more details.

## 4.7.2 Effects of the PICX Option

The PICX option determines how Pro\*COBOL treats data in character strings. The PICX option enables your program to use ANSI fixed-length strings or to maintain compatibility with previous versions of the database server and Pro\*COBOL.

You must use PICX=VARCHAR2 (not the default) to obtain the same results as releases of Pro\*COBOL before 8.0. Or, use

```
EXEC SQL varname IS VARCHAR2 END-EXEC.
```

for each variable.

## 4.7.3 Fixed-Length Character Variables

Fixed-length character variables are declared using the PIC X(n) and PIC G(n) and PIC N(n) datatypes. These types of variables handle character data based on their roles as input or output variables.

## 4.7.3.1 On Input

When PICX=VARCHAR2, the program interface strips trailing blanks before sending the value to the database. If you insert into a fixed-length CHAR column, Pro\*COBOL re-appends trailing blanks up to the length of the database column. However, if you insert into a variable-length VARCHAR2 column, Pro\*COBOL never appends blanks.

When PICX=CHARF, trailing blanks are never stripped.

Host input variables for multibyte Globalization Support data are *not* stripped of trailing double-byte spaces. The length component is assumed to be the length of the data in characters, not bytes.

Make sure that the input value is not trailed by extraneous characters. Normally, this is not a problem because when a value is ACCEPTed or MOVEd into a PIC X(n) variable, COBOL appends blanks up to the length of the variable.

The following example illustrates the point:

```
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

O1 EMPLOYEES.

O5 EMP-NAME PIC X(10).

O5 DEPT-NUMBER PIC S9(4) VALUE 20 COMP.

O5 EMP-NUMBER PIC S9(9) VALUE 9999 COMP.

O5 JOB-NAME PIC X(8).

EXEC SQL END DECLARE SECTION END-EXEC.
```



```
PROCEDURE DIVISION.

...

DISPLAY "Employee name? " WITH NO ADVANCING.

ACCEPT EMP-NAME.

* Assume that the name MILLER was entered

* EMP-NAME contains "MILLER " (4 trailing blanks)

MOVE "SALES" TO JOB-NAME.

* JOB-NAME now contains "SALES " (3 trailing blanks)

EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO, JOB)

VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER, :JOB-NAME END-EXEC.
```

If you precompile the last example with PICX=VARCHAR2 and the target database columns are VARCHAR2, the program interface strips the trailing blanks on input and inserts just the 6-character string "MILLER" and the 5-character string "SALES" into the database. However, if the target database columns are CHAR, the strings are blank-padded to the width of the columns.

If you precompile the last example with PICX=CHARF and the JOB column is defined as CHAR(10), the value inserted into that column is "SALES####" (five trailing blanks). However, if the JOB column is defined as VARCHAR2(10), the value inserted is "SALES###" (three trailing blanks), because the host variable is declared as PIC X(8). This might not be what you want, so be careful.

### 4.7.3.2 On Output

The PICX option has no effect on output to fixed-length character variables. When you use a PIC X(n) variable as an output host variable, Pro\*COBOL blank-pads it. In our example, when your program fetches the string "MILLER" from the database, EMP-NAME contains the value "MILLER####" (with four trailing blanks). This character string can be used without change as input to another SQL statement.

# 4.7.4 Varying-Length Variables

VARCHAR variables handle character data based on their roles as input or output variables.

### 4.7.4.1 On Input

When you use a VARCHAR variable as an input host variable, your program must assign values to the length and string fields of the expanded VARCHAR declaration, as shown in the following example:

```
IF ENAME-IND = -1
   MOVE "NOT AVAILABLE" TO ENAME-ARR
   MOVE 13 TO ENAME-LEN.
```

You need not blank-pad the string variable. In SQL operations, Pro\*COBOL uses exactly the number of characters given by the length field, counting any spaces.

### 4.7.4.2 On Output

When you use a VARCHAR variable as an output host variable, Pro\*COBOL sets the length field. An example follows:

```
WORKING-STORAGE SECTION.
```



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

01 EMPNO PIC S9(4) COMP.

01 ENAME PIC X(15) VARYING.

...

EXEC SQL END DECLARE SECTION END-EXEC.

...

PROCEDURE DIVISION.

...

EXEC SQL

SELECT ENAME INTO :ENAME FROM EMP

WHERE EMPNO = :EMPNO

END-EXEC.

IF ENAME-LEN = 0

MOVE FALSE TO VALID-DATA.
```

An advantage of VARCHAR variables over fixed-length strings is that the length of the value returned by Pro\*COBOL is available right away. With fixed-length strings, to get the length of the value, your program must count the number of characters.

Host output variables for multibyte NCHAR data are *not* padded at all. The length of the buffer is set to the length in bytes, not characters.



When PICN\_ENDIAN=OS, the length of the buffer for multibyte NCHAR data is set to the length in characters, not bytes.

### 4.8 Universal ROWIDs

There are two kinds of table organization used in the database server: *heap tables* and *index-organized tables*.

Heap tables are the default. This is the organization used in all tables before Oracle. The physical row address (ROWID) is a permanent property that is used to identify a row in a heap table. The external character format of the physical ROWID is an 18-byte character string in base-64 encoding.

An index-organized table does not have physical row addresses as permanent identifiers. A *logical* ROWID is defined for these tables. When you use a SELECT ROWID ... statement from an index-organized table the ROWID is an opaque structure that contains the primary key of the table, control information, and an optional physical "guess". You can use this ROWID in a SQL statement containing a clause such as "WHERE ROWID = ..." to retrieve values from the table.

The *universal* ROWID was introduced in the Oracle 8.1 Database release. Universal ROWID can be used for both physical ROWID and logical ROWID. You can use universal ROWIDs to access data in heap tables, or index-organized tables, since the table organization can change with no effect on applications. The column datatype used for ROWID is UROWID(*length*), where length is optional.

Use the universal ROWID in all new applications.

Declare a universal ROWID, which uses the pseudotype SQL-ROWID, this way:

```
01 MY-ROWID SQL-ROWID.
```



Memory for the universal ROWID is allocated with the ALLOCATE statement:

```
EXEC SQL ALLOCATE :MY-ROWID END-EXEC.
```

Use MY-ROWID in SQL DML statements like this:

```
EXEC SQL SELECT ROWID INTO :MY-ROWID FROM MYTABLE WHERE ... END-EXEC.

EXEC SQL UPDATE MYTABLE SET ... WHERE ROWID = :MY-ROWID END-EXEC.

...
```

Free the memory when you no longer need it with the FREE directive:

```
EXEC SOL FREE : MY-ROWID END-EXEC.
```

You also have the option of using a character host variable of width between 18 and 4000 as the host bind variable for universal ROWID. Character-based universal ROWIDs are supported for heap tables only for backward compatibility. Because a universal ROWID can be variable length there can be truncation when it is selected.

Use the character variable like this:

```
01 MY-ROWID-CHAR PIC X(4000) VARYING.

...

EXEC SQL ALLOCATE :MY-ROWID-CHAR END-EXEC.

EXEC SQL SELECT ROWID INTO :MY-ROWID-CHAR FROM MYTABLE WHERE ... END-EXEC.

...

EXEC SQL UPDATE MYTABLE SET ... WHERE ROWID = :MY-ROWID-CHAR END-EXEC.

...

EXEC SQL FREE :MY-ROWID-CHAR END-EXEC.
```

#### See Also:

- Oracle Database Concepts for more information on universal ROWIDs.
- "Positioned Update" for an example of a positioned update using the universal ROWID.

### 4.8.1 Subprogram SQLROWIDGET

where

The Oracle subprogram SQLROWIDGET enables you to retrieve the ROWID of the last row inserted, updated, or selected. SQLROWIDGET requires a context or NULL and a ROWID as its arguments.

To use the default context, pass the figurative constant NULL as the first parameter in the call to SQLROWIDGET.

Note that the universal ROWID must be declared and allocated **before** the call. The context, if used must be declared and allocated before the call. Here is the syntax:

```
CALL "SQLROWIDGET" USING NULL rowid.

Or

CALL "SQLROWIDGET" USING context rowid.
```



#### context (IN)

is the runtime context variable, of pseudotype SQL-CONTEXT, or the figurative constant NULL for the default context. For a discussion of runtime contexts, see "Embedded SQL Statements and Directives for Runtime Contexts".

```
rowid (OUT)
```

is a universal ROWID variable, of pseudotype SQL-ROWID. When a normal execution finishes, this will point to a valid universal ROWID. In case of an error, *rowid* is undefined.

Here is a sample showing its use with the default context:

```
01 MY-ROWID SQL-ROWID.
...
EXEC SQL ALLOCATE :MY-ROWID END-EXEC.

* INSERT, or UPDATE or DELETE Goes here:
...
CALL "SQLROWIDGET" USING NULL MY-ROWID.

* MY-ROWID now has the universal rowid descriptor for the last row
...
EXEC SQL FREE :MY-ROWID END-EXEC.
```

If your compiler does not allow using the figurative constant NULL in a CALL statement, you can declare a variable with picture S9(9) COMP VALUE 0 and use that with the BY VALUE clause in the call to SQLROWIDGET, as follows:

```
01 NULL-CONTEXT PIC S9(9) COMP VALUE ZERO.
01 MY-ROWID SQLROWID.
....
CALL "SQLROWIDGET" USING BY VALUE NULL-CONTEXT BY REFERENCE MY-ROWID.
```

# 4.9 Globalization Support

Although the widely-used 7-bit or 8-bit ASCII and EBCDIC character sets are adequate to represent the Roman alphabet, some Asian languages, such as Japanese, contain thousands of characters. These languages require 16 bits or more, to represent each character. How does Oracle deal with such dissimilar languages?

Oracle provides Globalization Support (formerly called National Language Support or NLS), which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments. With Globalization Support, number and date formats adapt automatically to the language conventions specified for a user session. Thus, Globalization Support enables users around the world to interact with Oracle in their native languages.

You control the operation of language-dependent features by specifying various Globalization Support parameters. You can set default parameter values in the initialization file. Table 4-8 shows what each Globalization Support parameter specifies.



**Table 4-8 Globalization Support Parameters** 

Globalization Support Parameter	Specifies
NLS_LANGUAGE	language-dependent conventions
NLS_TERRITORY	territory-dependent conventions
NLS_DATE_FORMAT	date format
NLS_DATE_LANGUAGE	language for day and month names
NLS_NUMERIC_CHARACTERS	decimal character and group separator
NLS_CURRENCY	local currency symbol
NLS_ISO_CURRENCY	ISO currency symbol
NLS_SORT	sort sequence

The main parameters are NLS\_LANGUAGE and NLS\_TERRITORY. NLS\_LANGUAGE specifies the default values for language-dependent features, which include

- language for Server messages
- language for day and month names
- sort sequence

NLS TERRITORY specifies the default values for territory-dependent features, which include

- Date format
- Decimal character
- Group separator
- Local currency symbol
- ISO currency symbol

You can control the operation of language-dependent Globalization Support features for a user session by specifying the parameter NLS\_LANG as follows

```
NLS LANG = language territory.character set
```

where *language* specifies the value of NLS\_LANGUAGE for the user session, territory specifies the value of NLS\_TERRITORY, and *character set* specifies the encoding scheme used for the terminal. An *encoding scheme* (usually called a character set or code page) is a range of numeric codes that corresponds to the set of characters a terminal can display. It also includes codes that control communication with the terminal.

You define NLS\_LANG as an environment variable (or the equivalent on your system). For example, on UNIX using the C shell, you might define NLS\_LANG as follows:

```
setenv NLS LANG French France.WE8ISO8859P1
```

To change the values of Globalization Support parameters during a session, you use the ALTER SESSION statement as follows:

```
ALTER SESSION SET nls parameter = value
```

Pro\*COBOL fully supports all the Globalization Support features that allow your applications to process multilingual data stored in an Oracle database. For example, you can declare



foreign-language character variables and pass them to string functions such as INSTRB, LENGTHB, and SUBSTRB. These functions have the same syntax as the INSTR, LENGTH, and SUBSTR functions, respectively, but operate on a per-byte basis rather than a per-character basis.

You can use the functions NLS\_INITCAP, NLS\_LOWER, and NLS\_UPPER to handle special instances of case conversion. And, you can use the function NLSSORT to specify WHERE-clause comparisons based on linguistic rather than binary ordering. You can even pass Globalization Support parameters to the TO\_CHAR, TO\_DATE, and TO\_NUMBER functions.



Oracle Database Globalization Support Guide for more information about Globalization Support.

# 4.10 Unicode Support for Pro\*COBOL

Pro\*COBOL does not support source code written with Unicode. If you want to set UTF8 to locale, source code should be written with only ASCII code.

If you want to use Unicode variables, you need to remember the following:

- Pro\*COBOL does not support "USAGE NATIONAL". If you want to use Unicode variables, use -C NSYMBOL="NATIONAL" for compiling Cobol programs with MF Cobol.
- Unicode can be used with PIC X and PIC N:
  - AL32UTF8 and UTF8 can be used with PIC X.
  - AL16UTF16 can be used with PIC N.
- Equate NLS\_NCHAR environment variable to the client-side character set used in PIC N. If NLS\_NCHAR is not set, then NLS\_LANG is used.
- Equate NLS\_LANG environment variable to the client-side character set used in PIC X.
- AL16UTF16, JA16SJISFIXED and JA16EUCFIXED can be set in NLS NCHAR.
- JA16SJIS and JA16SJISTILDE can also be set in NLS NCHAR.
- JA16EUC and JA16EUCTILDE cannot be used with NLS NCHAR.

#### See Also:

For the character sets that can be used in NLS\_LANG, refer to the Globalization Support Guide.

PICX=charf/varchar2 works for unicode PIC X variables and for PIC N it works with the restrictions mentioned in the aforementioned list.



### 4.10.1 NLS\_LOCAL=YES Restrictions

When the precompiler option NLS\_LOCAL is YES, the runtime library (SQLLIB) performs blank-padding and blank-stripping for Globalization Support multibyte datatypes.

When NLS\_LOCAL=YES, multibyte NCHAR features are not supported within a PL/SQL block. These features include N-quoted character literals and fixed-length character variables.

These restrictions then apply:

**Tables Disallowed.** Host variables declared using the PIC N or PIC G datatype must not be tables.

**No Odd-Byte Widths.** Oracle CHAR columns should not be used to store multibyte NCHAR characters. A run-time error is generated if data with an odd number of bytes is FETCHed from a single-byte column into a multibyte NCHAR host variable.

**No Host Variable Equivalencing.** Multibyte NCHAR character variables cannot be equivalenced using an EXEC SQL VAR statement.

**No Dynamic SQL.** Dynamic SQL is not available for NCHAR multibyte character string host variables in Pro\*COBOL.

Functions should not be used on columns that store multibyte Globalization Support data.

# 4.10.2 Character Strings in Embedded SQL

A multibyte Globalization Support character string in an embedded SQL statement consists of the letter N, followed by the string enclosed in single quotes.

For example,

```
EXEC SQL

SELECT EMPNO INTO :EMP-NUM FROM EMP

WHERE ENAME=N'NLS_string'

END-EXEC.
```

### 4.10.3 Embedded DDL

When the precompiler option, NLS\_LOCAL=YES, columns storing NCHAR data cannot be used in embedded data definition language (DDL) statements. This restriction cannot be enforced when precompiling, so the use of extended column types, such as NCHAR, within embedded DDL statements results in an execution error rather than a precompile error.

#### **Related Topics**

Precompiler Options

### 4.10.4 Blank Padding

When a Pro\*COBOL character variable is defined as a multibyte Globalization Support variable, the following blank padding and blank stripping rules apply, depending on the external datatype of the variable.

**CHARF.** Input data is stripped of any trailing double-byte spaces. However, if a string consists only of multibyte spaces, a single multibyte space is left in the buffer to act as a sentinel.

Output host variables are blank padded with multibyte spaces.

**VARCHAR.** On input, host variables are *not* stripped of trailing double-byte spaces. The length component is assumed to be the length of the data in characters, not bytes.

On output, the host variable is not blank padded at all. The length of the buffer is set to the length of the data in characters, not bytes.

**STRING/LONG VARCHAR.** These host variables are not supported for Globalization Support data, since they can only be specified using dynamic SQL or datatype equivalencing, neither of which is supported for Globalization Support data.

#### **Related Topics**

Handling Character Data

#### 4.10.5 Indicator Variables

You can use indicator variables with multibyte Globalization Support character variables as use you would with any other variable, except column length values are expressed in characters instead of bytes.

#### **Related Topics**

Using Indicator Variables

### 4.10.6 Various Combinations of PIC X/PIC N Variables and NCHAR/ CHAR Columns

Pro\*Cobol supports various combinations of PIC X /PIC N variables and NCHAR/ CHAR columns for select, insert and update statements through command line options charset\_picx and charset\_picn.

### 4.10.6.1 PIC X and NCHAR Column

By default, PIC X variables are converted to the server-side Database Character Set so that these variables can be used with CHAR columns. Therefore, when you use the PIC X variables with NCHAR columns, there can be some data loss. To avoid this, use the following command to set the character set form used by PIC X variables to NCHAR:

charset picx=nchar charset

#### 4.10.6.2 PIC N and CHAR column

By default, PIC N variables are converted to the server-side National Character Set so that these variables can be used with NCHAR columns. Therefore, when you use the PIC N variables with CHAR columns, there may be some performance impact. To avoid this, use the following command to set the character set form used by PIC N variables to CHAR:

charset picn=db charset



#### See Also:

Using Pro\*COBOL Precompiler Options for more information on CHARSET\_PICX and CHARSET\_PICN.

### 4.10.7 Working With New Options

The following table shows how to work with the new options:

Host var : Host variables type

DB column type : CHAR/NCHAR

charset\_picx : value for charset\_picx optioncharset\_picn : value for charset\_picn option

NLS\_NCHAR : value for NLS\_NCHAR

data charset: character set of host variables

Host var	DB col type	charset_picx	charset_picN	NLS_NCHAR	Data charset
PIC X	CHAR	nchar_charset	-	set	NLS_NCHAR
PIC X	CHAR	db_charset	-	set	NLS_LANG
PIC X	CHAR	nchar_charset	-	not set	NLS_LANG
PIC X	CHAR	db_charset	-	not set	NLS_LANG
PIC X	NCHAR	nchar_charset	-	set	NLS_NCHAR
PIC X	NCHAR	db_charset	-	set	NLS_LANG
PIC X	NCHAR	nchar_charset	-	not set	NLS_LANG
PIC X	NCHAR	db_charset	-	not set	NLS_LANG
PIC N	CHAR	-	nchar_charset	set	NLS_NCHAR
PIC N	CHAR	-	db_charset	set	NLS_LANG
PIC N	CHAR	-	nchar_charset	not set	NLS_LANG
PIC N	CHAR	-	db_charset	not set	NLS_LANG
PIC N	NCHAR	-	nchar_charset	set	NLS_NCHAR
PIC N	NCHAR	-	db_charset	set	NLS_LANG
PIC N	NCHAR	-	nchar_charset	not set	NLS_LANG
PIC N	NCHAR	-	db_charset	not set	NLS_LANG

# 4.11 Datatype Conversion

At precompile time, an external datatype is assigned to each host variable. For example, Pro\*COBOL assigns the INTEGER external datatype to host variables of type PIC S9(n) COMP. At run time, the datatype code of every host variable used in a SQL statement is passed to Oracle. Oracle uses the codes to convert between internal and external datatypes.

Before assigning a SELECTed column value to an output host variable, Oracle must convert the internal datatype of the source column to the datatype of the host variable. Likewise, before assigning or comparing the value of an input host variable to a column, Oracle must



convert the external datatype of the host variable to the internal datatype of the target column.

Conversions between internal and external datatypes follow the usual data conversion rules. For example, you can convert a CHAR value of 1234 to a PIC S9(4) COMP value. You cannot, however, convert a CHAR value of 123465543 (number too large) or 10F (number not decimal) to a PIC S9(4) COMP value. Likewise, you cannot convert a PIC X(n) value that contains alphabetic characters to a NUMBER value.

The datatype of the host variable must be compatible with that of the database column. It is your responsibility to make sure that values are convertible. For example, if you try to convert the string value YESTERDAY to a DATE column value, you get an error. Conversions between internal and external datatypes follow the usual data conversion rules. For instance, you can convert a CHAR value of 1234 to a 2-byte integer. But, you cannot convert a CHAR value of 65543 (number too large) or 10F (number not decimal) to a 2-byte integer. Likewise, you cannot convert a string value that contains alphabetic characters to a NUMBER value.

Number conversion follows the conventions specified by Globalization Support parameters in the Oracle initialization file. For example, your system might be configured to recognize a comma (,) instead of a period (.) as the decimal character.

The following table shows the supported conversions between internal and external datatypes.

Table 4-9 Conversions Between Internal and External Datatypes

External	Internal	-	-	-	-	-	-	-
CHAR	I/O	I/O (2)	I/O	I(3)		I/O	I/O (3)	I/O (1)
CHARF	I/O	I/O (2)	I/O	I (3)		I/O	I/O (3)	I/O (1)
CHARZ	I/O	I/O (2)	I/O	I (3)		I/O	I/O (3)	I/O (1)
DATE	I/O	I/O	I	-	-	-		
DECIMAL	I/O (4)	-	I	-	-	I/O	-	-
DISPLAY	I/O (4))	-	I	-	-	I/O	-	-
FLOAT	I/O (4)	-	I	-	-	I/O	-	-
INTEGER	I/O (4)	-	I	-	-	I/O	-	-
LONG	I/O	I/O (2)	I/O	I (3.5)	-	I/O	I/O (3)	I/O (1)
LONG RAW	O(6)	-	I (5,6)	I/O	-	-	I/O	-
LONG VARCHAR	I/O	I/O(2)	I/O	I (3,5)	-	I/O	I/O(3))	I/O (1)
LONG VARRAW	I/O (6)	-	I (5,6)	I/O	-	-	I/O	-
NUMBER	I/O (4)	-	I	-	-	I/O	-	-
RAW	I/O (6)	-	I (5,6)	I/O	-	-	I/O	-
ROWID	1	-	1	-	-	-	-	I/O
STRING	I/O	I/O (2)	I/O	I (3.5)	-	I/O	I/O (3)	I/O (1)
UNSIGNED	I/O (4)	-	I	-	-	I/O	-	-
VARCHAR	I/O	I/O (2)	I/O	I (3,5)	-	I/O	I/O (3)	-
VARCHAR2	I/O	I/O (2)	I/O	I (3)	-	I/O	I/O (3)	I/O (1)



Table 4-9 (Cont.) Conversions Between Internal and External Datatypes

External	Internal	-	-	-	-	-	-	-	
VARNUM	I/O (4)	-	I	-	-	I/O	-	-	
VARRAW	I/O (6)	-	I (5,6)	I/O	-	-	I/O	-	

#### In the preceding table:

- I = input only
- O = output only
- I/O = input or output

#### Note:

On input, host string must be in Oracle'BBBBBBBBB.RRRR.FFFF' format.

On output, column value is returned in same format.

On input, host string must be the default DATE character format.

On output, column value is returned in same format

On input, host string must be in hex format.

On output, column value is returned in same format.

On output, column value must represent a valid number.

On input, length must be less than or equal to 2000.

#### See Also:

Oracle Database Globalization Support Guide for more information about Globalization Support.

# 4.12 Explicit Control Over DATE String Format

When you select a DATE column value into a character host variable, Oracle must convert the internal binary value to an external character value. So, Oracle implicitly calls the SQL function TO\_CHAR, which returns a character string in the default date format. The default is set by the Oracle initialization parameter NLS\_DATE\_FORMAT. To get other information such as the time or Julian date, you must explicitly call TO\_CHAR with a format mask.

A conversion is also necessary when you insert a character host value into a DATE column. Oracle implicitly calls the SQL function TO\_DATE, which expects the default date format. To insert dates in other formats, you must explicitly call TO\_DATE with a format mask.

For compatibility with other versions of SQL Pro\*COBOL now provides the following precompiler option to specify date strings:

#### DATE\_FORMAT={ISO | USA | EUR | JIS | LOCAL | 'fmt'

The DATE\_FORMAT option must be used on the command line or in a configuration file. The date strings are shown in the following table:

Table 4-10 Formats for Date Strings

Format Name	Abbreviation	Date Format
International Standards Organization	ISO	yyyy-mm-dd
USA standard	USA	mm/dd/yyyy
European standard	EUR	dd.mm.yyyy
Japanese Industrial Standard	JIS	yyyy-mm-dd
installation-defined	LOCAL	Any installation-defined form.

'fmt' is a date format model, such as 'Month dd, yyyy'. Note that all separately compiled units linked together must use the same DATE FORMAT value.



Datetime Format Elements for the list of date format model elements.

# 4.13 Datatype Equivalencing

Datatype equivalencing lets you control the way Oracle interprets input data and the way Oracle formats output data. You can equivalence supported COBOL datatypes to external datatypes on a variable-by-variable basis.

### 4.13.1 Usefulness of Equivalencing

Datatype equivalencing is useful in several ways. For example, suppose you want to use a variable-length string in a COBOL program. You can declare a PIC X host variable, then equivalence it to the external datatype VARCHAR2.

In addition, you can use datatype equivalencing to override default datatype conversions. Unless Globalization Support parameters in the initialization file specify otherwise, if you select a DATE column value into a character host variable, Oracle returns a 9-byte string formatted as follows:

DD-MON-YY

However, if you equivalence the character host variable to the DATE external datatype, Oracle returns a 7-byte value in the internal format.

### 4.13.2 Host Variable Equivalencing

By default, Pro\*COBOL assigns a specific external datatype to every host variable. You can override the default assignments by equivalencing host variables to external datatypes. This is called *host variable equivalencing*.

The syntax of the VAR embedded SQL statement is:



There must be at least one of the two clauses, or both.

where:

Table 4-11 Host Variable Equivalencing

Variable	Description
precision and scale	Integer literals that represent, respectively, the number of significant digits and the point at which rounding will occur. For example, a scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46); a scale of -3 means the number is rounded to the nearest thousand (3456 becomes 3000).
	You can specify a <i>precision</i> of 1 99 and a <i>scale</i> of -84 99. However, the maximum precision and scale of a database column are 38 and 127, respectively. So, if <i>precision</i> exceeds 38, you cannot insert the value of <i>host_variable</i> into a database column. On the other hand, if the scale of a column value exceeds 99, you cannot select or fetch the value into <i>host_variable</i> .
	Specify precision and scale only when type_name is DECIMAL or DISPLAY
length	Input or output host variable (or host table) declared earlier.
	The VARCHAR and VARRAW external datatypes have a 2-byte length field followed by an <i>n</i> -byte data field, where <i>n</i> lies in the range 1 65533. So, if <i>type_name</i> is VARCHAR or VARRAW, <i>host_variable</i> must be at least 3 bytes long.
	The LONG VARCHAR and LONG VARRAW external datatypes have a 4-byte length field followed by an <i>n</i> -byte data field, where <i>n</i> lies in the range 1 2147483643. So, if <i>type_name</i> is LONG VARCHAR or LONG VARRAW, <i>host_variable</i> must be at least 5 bytes long.
size	An integer which is the size, in bytes, of a buffer used to perform conversion of the specified host_variable to another character set.
SQL datatype	Name of a valid external datatype such as RAW or STRING.
host_variable	Input or output host variable or host table declared earlier.
	The VARCHAR and VARRAW external datatypes have a 2-byte length field followed by an <i>n</i> -byte data field, where <i>n</i> lies in the range 1 65533. So, if <i>type_name</i> is VARCHAR or VARRAW, <i>host_variable</i> must be at least 3 bytes long.
	The LONG VARCHAR and LONG VARRAW external datatypes have a 4-byte length field followed by an <i>n</i> -byte data field, where <i>n</i> lies in the range 1 2147483643. So, if <i>type_name</i> is LONG VARCHAR or LONG VARRAW, <i>host_variable</i> must be at least 5 bytes long.

An Example shows which parameters to use with each external datatype.

You cannot use EXEC SQL VAR with NCHAR host variables (those containing PIC G or PIC N clauses).

If DECLARE\_SECTION=TRUE then you must have a Declare Section and you must place EXEC SQL VAR statements in the Declare Section.

When ext\_type\_name is FLOAT, use *length*; when ext\_type\_name is DECIMAL, you must specify *precision* and *scale* instead of *length*.

Host variable equivalencing is useful in several ways. For example, you can use it when you want Oracle to store but not interpret data. Suppose you want to store a host table of 4-byte integers in a RAW database column. Simply equivalence the host table to the RAW external datatype, as follows:

```
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

O1 EMP-TABLES.

O5 EMP-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.

...

Reset default datatype (INTEGER) to RAW.

EXEC SQL VAR EMP-NUMBER IS RAW (200) END-EXEC.

EXEC SQL END DECLARE SECTION END-EXEC.
```

With host tables, the length you specify must match the buffer size required to hold the table. In the last example, you specified a length of 200, which is the buffer size needed to hold 50 4-byte integers.

You can also declare a group item to be used as a LONG VARCHAR:

```
01 MY-LONG-VARCHAR.
05 UC-LEN PIC S9(9) COMP.
05 UC-ARR PIC X(6000).
EXEC SQL VAR MY-LONG-VARCHAR IS LONG VARCHAR(6000).
```

#### See Also:

- "CONVBUFSZ Clause in VAR Statement" for more information about CONVBUFSZ clause.
- "VAR (Oracle Embedded SQL Directive) " for a syntax diagram of EXEC SQL VAR statement.

#### 4.13.2.1 CONVBUFSZ Clause in VAR Statement

The EXEC SQL VAR statement can have an optional *CONVBUFSZ* clause. You specify the size, in bytes, of the buffer in the runtime library used to perform conversion of the specified host variable between character sets.

When you have not used the CONVBUFSZ clause, the runtime automatically determines a buffer size based on the ratio of the host variable character size (determined by NLS\_LANG) and the character size of the database character set. This can sometimes result in the creation of a buffer of LONG size. Databases are allowed to have only one LONG column. An error is raised if there is more than one LONG value.

To avoid such errors, you use a length shorter than the size of a LONG. If a character set conversion results in a value longer than the length specified by CONVBUFSZ, then Pro\*COBOL returns an error.



### 4.13.2.2 An Example

Suppose you want to select employee names from the EMP table, then pass them to a Clanguage routine that expects null-terminated strings. You need not explicitly null-terminate the names. Simply equivalence a host variable to the STRING external datatype, as follows:

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

...

01 EMP-NAME PIC X(11).

EXEC SQL VAR EMP-NAME IS STRING (11) END-EXEC.

EXEC SQL END DECLARE SECTION END-EXEC.
```

The width of the ENAME column is 10 characters, so you allocate the new *EMP-NAME* 11 characters to accommodate the null terminator. (Here, *length* is optional because it defaults to the length of the host variable.) When you select a value from the ENAME column into *EMP-NAME*, Oracle null-terminates the value for you.

Table 4-12 Parameters for Host Variable Equivalencing

External Datatype	Length	Precision	Scale	Default Length
CHAR	optional	n/a	n/a	declared length of variable
CHARZ	optional	n/a	n/a	declared length of variable
DATE	n/a	n/a	n/a	7 bytes
DECIMAL	n/a	required	required	none
DISPLAY	n/a	required	required	none
DISPLAY TRAILING	n/a	required	required	none
UNSIGNED DISPLAY	n/a	required	required	none
OVERPUNCH TRAILING	n/a	required	required	none
OVERPUNCH LEADING	n/a	required	required	none
FLOAT	optional (4 or 8)	n/a	n/a	declared length of variable
INTEGER	optional (1, 2, 4 or 8)	n/a	n/a	declared length of variable
LONG	optional	n/a	n/a	declared length of variable
LONG RAW	optional	n/a	n/a	declared length of variable
LONG VARCHAR	required (note 1)	n/a	n/a	none
LONG VARRAW	required (note 1)	n/a	n/a	none
NUMBER	n/a	n/a	n/a	not available
STRING	optional	n/a	n/a	declared length of variable
RAW	optional	n/a	n/a	declared length of variable

External Datatype	Length	Precision	Scale	Default Length
ROWID	n/a	n/a	n/a	18 bytes (see note 2)
UNSIGNED	optional (1, 2, or 4)	n/a	n/a	declared length of variable
VARCHAR	required	n/a	n/a	none
VARCHAR2	optional	n/a	n/a	declared length of variable
VARNUM	n/a	n/a	n/a	22 bytes
VARRAW	optional	n/a	n/a	none

Table 4-12 (Cont.) Parameters for Host Variable Equivalencing

### 4.13.3 Using the CHARF Datatype Specifier

You can use the datatype specifier CHARF in VAR statements to equivalence COBOL datatypes to the fixed-length ANSI datatype CHAR.s

When PICX=CHARF, specifying the datatype CHAR in a VAR statement equivalences the host-language datatype to the fixed-length ANSI datatype CHAR (Oracle external datatype code 96). However, when PICX=VARCHAR2, the host-language datatype is equivalenced to the variable-length datatype VARCHAR2 (code 1).

However, you can always equivalence host-language datatypes to the fixed-length ANSI datatype CHAR. Simply specify the datatype CHARF in the VAR statement. If you use CHARF, the host-language datatype is equivalenced to the fixed-length ANSI datatype CHAR even when PICX=VARCHAR2.

### 4.13.4 Guidelines

To input VARNUM or DATE values, you must use the Oracle internal format. Keep in mind that Oracle uses the internal format to output VARNUM and DATE values.

After selecting a column value into a VARNUM host variable, you can check the first byte to get the length of the value. Internal Datatypes gives some examples of returned VARNUM values.

**Table 4-13 VARNUM Examples** 

Decimal Value	Length Byte	<b>Exponent Byte</b>	Mantissa Bytes	<b>Terminator Byte</b>
5	2	193	6	n/a
-5	3	62	96	102
2767	3	194	28, 68	n/a
-2767	4	61	74, 34	102
100000	2	195	11	n/a
1234567	5	196	2, 24, 46, 68	n/a



<sup>1.</sup> If the data field exceeds 65533 bytes, pass -1.

<sup>2.</sup> This length is typical but the default is port-specific.

If no Oracle external datatype suits your needs exactly, use a VARCHAR2-based or RAW-based external datatype.



"Explicit Control Over DATE String Format" for converting DATE values.

### 4.13.5 RAW and LONG RAW Values

When you select a RAW or LONG RAW column value into a character host variable, Oracle must convert the internal binary value to an external character value. In this case, Oracle returns each binary byte of RAW or LONG RAW data as a pair of characters. Each character represents the hexadecimal equivalent of a nibble (half a byte). For example, Oracle returns the binary byte 11111111 as the pair of characters "FF". The SQL function RAWTOHEX performs the same conversion.

A conversion is also necessary when you insert a character host value into a RAW or LONG RAW column. Each pair of characters in the host variable must represent the hexadecimal equivalent of a binary byte. If a character does not represent the hexadecimal value of a nibble, Oracle issues an error message.



"Sample Program 4: Datatype Equivalencing" for more information about datatype conversion

# 4.14 Platform Endianness Support

Oracle stored unicode data (UTF16) is always in big endian. Currently, client applications run on different platforms. Linux and Windows have little endian representation and Solaris has big endian representation. When UTF16 data is inserted or selected, Pro\*Cobol doesn't convert endian form between server and the client. This leads to corrupted UTF16 (UCS2) strings in the PIC N variable.

Platform endianness (Little endian for Linux and Windows, Big endian for Solaris) in PIC N variables can be maintained using the command line option picn endian.

#### **New Command Line Option**

```
picn_endian={BIG|OS}
```

If picn\_endian=big, then PIC N variables are bound with character set ID AL16UTF16.

If picn\_endian=os then PIC N variables are bound with character set ID UCS2.

The default value for this option is "big" to preserve the current behavior. This option is ignored if NLS NCHAR is not AL16UTF16.

Character set form for PIC N variables can be set by using the existing Pro\*Cobol command line option:

charset picn={nchar charset|db charset}





"Varying-Length Variables" for more information about varchar host variable if PICN ENDIAN=0S.

# 4.15 Sample Program 4: Datatype Equivalencing

After connecting to Oracle, this program creates a database table named IMAGE in the SCOTT account, then simulates the insertion of bitmap images of employee numbers into the table. Datatype equivalencing lets the program use the Oracle external datatype LONG RAW to represent the images. Later, when the user enters an employee number, the number's "bitmap" is selected from the IMAGE table and displayed on the terminal screen.

- \* This program simulates the storage and retrieval of bitmap images into table
- $^{\star}$  IMAGE, which is created in the SCOTT account after logging on to ORACLE.
- \* Datatype equivalencing allows an ORACLE external type of LONG RAW to be
- \* specified for the programs representation of the images.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DTY-EQUIV.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(10) VARYING.
01 PASSWD
                    PIC X(10) VARYING.
01 EMP-REC-VARS.
   05 EMP-NUMBER PIC S9(4) COMP.
   05 EMP-NAME PIC X(10) VARYING.
   05 SALARY
                  PIC S9(6)V99
                    DISPLAY SIGN LEADING SEPARATE.
   05 COMMISSION PIC S9(6)V99
                    DISPLAY SIGN LEADING SEPARATE.
   05 COMM-IND PIC S9(4) COMP.
   EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
   EXEC SQL VAR COMMISSION IS DISPLAY(8,2) END-EXEC.
01 BUFFER-VAR.
   05 BUFFER
                  PIC X(8192).
   EXEC SQL VAR BUFFER IS LONG RAW END-EXEC.
01 INEMPNO
                    PIC S9(4) COMP.
   EXEC SQL END DECLARE SECTION END-EXEC.
   EXEC SQL INCLUDE SQLCA END-EXEC.
01 DISPLAY-VARIABLES.
   05 D-EMP-NAME PIC X(10).
   05 D-SALARY
                  PIC $Z(4)9.99.
   05 D-COMMISSION PIC $Z(4)9.99.
   05 D-INEMPNO PIC 9(4).
01 REPLY PIC X(10).
                  PIC S9(9) COMP.
01 INDX
01 PRT-QUOT PIC S9(9) COMP.
01 PRT-MOD PIC S9(9) COMP.
```



```
PROCEDURE DIVISION.
BEGIN-PGM.
    EXEC SOL WHENEVER SOLERROR
        DO PERFORM SQL-ERROR END-EXEC.
    PERFORM LOGON.
    DISPLAY "OK TO DROP THE IMAGE TABLE? (Y/N) "
        WITH NO ADVANCING.
    ACCEPT REPLY.
    IF (REPLY NOT = "Y") AND (REPLY NOT = "y")
       GO TO SIGN-OFF-EXIT.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL DROP TABLE IMAGE END-EXEC.
    DISPLAY " ".
    IF (SQLCODE = 0) DISPLAY
        "TABLE IMAGE DROPPED - CREATING NEW TABLE."
    ELSE IF (SQLCODE = -942) DISPLAY
        "TABLE IMAGE DOES NOT EXIST - CREATING NEW TABLE."
    ELSE PERFORM SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR
       DO PERFORM SQL-ERROR END-EXEC.
    EXEC SQL CREATE TABLE IMAGE
        (EMPNO NUMBER (4) NOT NULL, BITMAP LONG RAW)
    END-EXEC.
    EXEC SQL DECLARE EMPCUR CURSOR FOR
        SELECT EMPNO, ENAME FROM EMP
    END-EXEC.
    EXEC SQL OPEN EMPCUR END-EXEC.
    DISPLAY " ".
    DISPLAY
      "INSERTING BITMAPS INTO IMAGE FOR ALL EMPLOYEES ...".
    DISPLAY " ".
INSERT-LOOP.
    EXEC SOL WHENEVER NOT FOUND GOTO NOT-FOUND END-EXEC.
    EXEC SOL FETCH EMPCUR
        INTO :EMP-NUMBER, :EMP-NAME
    END-EXEC.
   MOVE EMP-NAME-ARR TO D-EMP-NAME.
    DISPLAY "EMPLOYEE ", D-EMP-NAME WITH NO ADVANCING.
    PERFORM GET-IMAGE.
    EXEC SQL INSERT INTO IMAGE
       VALUES (:EMP-NUMBER, :BUFFER)
    END-EXEC.
    DISPLAY " IS DONE!".
   MOVE SPACES TO EMP-NAME-ARR.
    GO TO INSERT-LOOP.
NOT-FOUND.
    EXEC SQL CLOSE EMPCUR END-EXEC.
    EXEC SQL COMMIT WORK END-EXEC.
    DISPLAY " ".
    DISPLAY
      "DONE INSERTING BITMAPS. NEXT, LET'S DISPLAY SOME.".
DISP-LOOP.
   MOVE 0 TO INEMPNO.
```

```
DISPLAY " ".
    DISPLAY "ENTER EMPLOYEE NUMBER (0 TO QUIT): "
        WITH NO ADVANCING.
    ACCEPT D-INEMPNO.
    MOVE D-INEMPNO TO INEMPNO.
    IF (INEMPNO = 0)
       GO TO SIGN-OFF.
    EXEC SQL WHENEVER NOT FOUND GOTO NO-EMP END-EXEC.
    EXEC SQL SELECT EMP.EMPNO, ENAME, SAL, NVL(COMM, 0), BITMAP
        INTO :EMP-NUMBER, :EMP-NAME, :SALARY,
             :COMMISSION:COMM-IND, :BUFFER
        FROM EMP, IMAGE
        WHERE EMP.EMPNO = :INEMPNO
         AND EMP.EMPNO = IMAGE.EMPNO
    END-EXEC.
    DISPLAY " ".
    PERFORM SHOW-IMAGE.
   MOVE EMP-NAME-ARR TO D-EMP-NAME.
   MOVE SALARY TO D-SALARY.
   MOVE COMMISSION TO D-COMMISSION.
    DISPLAY "EMPLOYEE ", D-EMP-NAME, " HAS SALARY ", D-SALARY
       WITH NO ADVANCING.
    IF COMM-IND = -1
        DISPLAY " AND NO COMMISSION."
       DISPLAY " AND COMMISSION ", D-COMMISSION, "."
    END-TF.
    MOVE SPACES TO EMP-NAME-ARR.
    GO TO DISP-LOOP.
NO-EMP
   DISPLAY "NOT A VALID EMPLOYEE NUMBER - TRY AGAIN.".
    GO TO DISP-LOOP.
LOGON.
   MOVE "SCOTT" TO USERNAME-ARR.
   MOVE 5 TO USERNAME-LEN.
   MOVE "TIGER" TO PASSWD-ARR.
   MOVE 5 TO PASSWD-LEN.
    EXEC SQL
        CONNECT : USERNAME IDENTIFIED BY : PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
    DISPLAY " ".
GET-IMAGE.
    PERFORM MOVE-IMAGE
       VARYING INDX FROM 1 BY 1 UNTIL INDX > 8192.
MOVE-IMAGE.
    STRING '*' DELIMITED BY SIZE
        INTO BUFFER
        WITH POINTER INDX.
    DIVIDE 256 INTO INDX
        GIVING PRT-QUOT REMAINDER PRT-MOD.
    IF (PRT-MOD = 0) DISPLAY "." WITH NO ADVANCING.
SHOW-IMAGE.
```

```
PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > 10
                             ********
      DISPLAY "
   END-PERFORM.
   DISPLAY " ".
SIGN-OFF.
   EXEC SQL DROP TABLE IMAGE END-EXEC.
SIGN-OFF-EXIT.
   DISPLAY " ".
   DISPLAY "HAVE A GOOD DAY.".
   DISPLAY " ".
   EXEC SQL COMMIT WORK RELEASE END-EXEC.
   STOP RUN.
SOL-ERROR.
   EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
   DISPLAY " ".
   DISPLAY "ORACLE ERROR DETECTED: ".
   DISPLAY " ".
   DISPLAY SQLERRMC.
   EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
   STOP RUN.
```



5

# **Embedded SQL**

This chapter describes the basic techniques of embedded SQL programming. Topics are:

- Using Host Variables
- Using Indicator Variables
- The Basic SQL Statements
- Cursors
- Scrollable Cursors
- Flexible B Area Length
- Fix Execution Plans
- Sample Program 2: Cursor Operations
- Using Boolean Data Type

# 5.1 Using Host Variables

Use host variables to pass data and status information to your program from the database, and to pass data to the database.

### 5.1.1 Output Versus Input Host Variables

Depending on how they are used, host variables are called output or input host variables. Host variables in the INTO clause of a SELECT or FETCH statement are called *output* host variables because they hold column values output by Oracle. Oracle assigns the column values to corresponding output host variables in the INTO clause.

All other host variables in a SQL statement are called *input* host variables because your program inputs their values to Oracle. For example, you use input host variables in the VALUES clause of an INSERT statement and in the SET clause of an UPDATE statement. They are also used in the WHERE, HAVING, and FOR clauses. In fact, input host variables can appear in a SQL statement wherever a value or expression is allowed.

You cannot use input host variables to supply SQL keywords or the names of database objects. Thus, you cannot use input host variables in data definition statements (sometimes called *DDL*) such as ALTER, CREATE, and DROP. In the following example, the DROP TABLE statement is *invalid*:

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

01 TABLE-NAME PIC X(30) VARYING.

...

EXEC SQL END DECLARE SECTION END-EXEC.

...

DISPLAY 'Table name? '.

ACCEPT TABLE-NAME.

EXEC SQL DROP TABLE :TABLE-NAME END-EXEC.

-- host variable not allowed
```



#### Note:

In an ORDER BY clause, you *can* use a host variable, but it is treated as a constant or literal, and hence the contents of the host variable have no effect. For example, the SQL statement:

```
EXEC SQL SELECT ENAME, EMPNO INTO :NAME, :NUMBER FROM EMP ORDER BY :ORD END-EXEC.
```

appears to contain an input host variable, *ORD*. However, the host variable in this case is treated as a constant, and regardless of the value of ORD, no ordering is done.

Before Oracle executes a SQL statement containing input host variables, your program must assign values to them. Consider the following example:

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

01 EMP-NUMBER PIC S9(4) COMP.

01 EMP-NAME PIC X(20) VARYING.

EXEC SQL END DECLARE SECTION END-EXEC.

* -- get values for input host variables

DISPLAY 'Employee number? '.

ACCEPT EMP-NUMBER.

DISPLAY 'Employee name? '.

ACCEPT EMP-NAME.

EXEC SQL INSERT INTO EMP (EMPNO, ENAME)

VALUES (:EMP-NUMBER, :EMP-NAME)

END-EXEC.
```

Notice that the input host variables in the VALUES clause of the INSERT statement are prefixed with colons.

# 5.2 Using Indicator Variables

You can associate any host variable with an optional indicator variable. Each time the host variable is used in a SQL statement, a result code is stored in its associated indicator variable. Thus, indicator variables let you monitor host variables.

You use indicator variables in the VALUES or SET clause to assign NULLs to input host variables and in the INTO clause to detect NULLs or truncated values in output host variables.

# 5.2.1 Input Variables

For input host variables, the values your program can assign to an indicator variable have the following meanings:

Variable	Meaning
-1	Oracle will assign a NULL to the column, ignoring the value of the host variable.



Variable	Meaning
>= 0	Oracle will assigns the value of the host variable to the column.

### 5.2.2 Output Variables

For output host variables, the values Oracle can assign to an indicator variable have the following meanings:

Variable	Meaning
-2	Oracle assigned a truncated column value to the host variable, but could not assign the original length of the column value to the indicator variable because the number was too large.
-1	The column value is NULL, so the value of the host variable is indeterminate.
0	Oracle assigned an intact column value to the host variable.
> 0	Oracle assigned a truncated column value to the host variable, assigned the original column length (expressed in characters, instead of bytes, for multibyte Globalization Support host variables) to the indicator variable, and set SQLCODE in the SQLCA to zero.

Remember, an indicator variable must be declared as a 2-byte integer and, in SQL statements, must be prefixed with a colon and appended to its host variable (unless you use the keyword INDICATOR).

### 5.2.3 Inserting NULLs

You can use indicator variables to insert NULLs. Before the insert, for each column you want to be NULL, set the appropriate indicator variable to -1, as shown in the following example:

```
MOVE -1 TO IND-COMM.

EXEC SQL INSERT INTO EMP (EMPNO, COMM)

VALUES (:EMP-NUMBER, :COMMISSION:IND-COMM)

END-EXEC.
```

The indicator variable *IND-COMM* specifies that a NULL is to be stored in the COMM column.

You can hard-code the NULL instead, as follows:

```
EXEC SQL INSERT INTO EMP (EMPNO, COMM)

VALUES (:EMP-NUMBER, NULL)

END-EXEC.
```

But this is less flexible.

Typically, you insert NULLs conditionally, as the next example shows:

```
DISPLAY 'Enter employee number or 0 if not available: '
WITH NO ADVANCING.

ACCEPT EMP-NUMBER.

IF EMP-NUMBER = 0
MOVE -1 TO IND-EMPNUM

ELSE
MOVE 0 TO IND-EMPNUM

END-IF.
```



```
EXEC SQL INSERT INTO EMP (EMPNO, SAL)

VALUES (:EMP-NUMBER:IND-EMPNUM, :SALARY)
END-EXEC.
```

### 5.2.4 Handling Returned NULLs

You can also use indicator variables to manipulate returned NULLs, as the following example shows:

```
EXEC SQL SELECT ENAME, SAL, COMM

INTO :EMP-NAME, :SALARY, :COMMISSION:IND-COMM
FROM EMP
WHERE EMPNO = :EMP_NUMBER
END-EXEC.
IF IND-COMM = -1
MOVE SALARY TO PAY.

* -- commission is null; ignore it
ELSE
ADD SALARY TO COMMISSION GIVING PAY.
END-IF.
```

### 5.2.5 Fetching NULLs

Using the precompiler option UNSAFE\_NULL=YES, you can select or fetch NULLs into a host variable that lacks an indicator variable, as the following example shows:

```
* -- assume that commission is NULL

EXEC SQL SELECT ENAME, SAL, COMM

INTO :EMP-NAME, :SALARY, :COMMISSION

FROM EMP

WHERE EMPNO = :EMP-NUMBER

END-EXEC.
```

SQLCODE in the SQLCA is set to zero indicating that Oracle executed the statement without detecting an error or exception.

Without an indicator variable there is no way to know whether or not a NULL was returned. The value of the host variable is undefined. If you do not use indicator variables, set the precompiler option UNSAFE\_NULL=YES. Oracle therefore advises that UNSAFE\_NULL=YES only be used to upgrade existing programs and that indicator variables be used for all new programs.

When UNSAFE\_NULL=NO, if you select or fetch NULLs into a host variable that lacks an indicator variable, Oracle issues an error message.

#### **Related Topics**

UNSAFE\_NULL

### 5.2.6 Testing for NULLs

You can use indicator variables in the WHERE clause to test for NULLs, as the following example shows:

```
EXEC SQL SELECT ENAME, SAL
INTO :EMP-NAME, :SALARY
FROM EMP
WHERE :COMMISSION:IND-COMM IS NULL ...
```



However, you cannot use a relational operator to compare NULLs with each other or with other values. For example, the following SELECT statement fails if the COMM column contains one or more NULLs:

```
EXEC SQL SELECT ENAME, SAL
  INTO :EMP-NAME, :SALARY
  FROM EMP
  WHERE COMM = :COMMISSION:IND-COMM
END-EXEC.
```

The next example shows how to compare values for equality when some of them might be NULLs:

```
EXEC SQL SELECT ENAME, SAL

INTO :EMP_NAME, :SALARY

FROM EMP

WHERE (COMM = :COMMISSION) OR ((COMM IS NULL) AND

(:COMMISSION:IND-COMM IS NULL))

END-EXEC.
```

### 5.2.7 Fetching Truncated Values

If a value is truncated when fetched into a host variable, no error is generated. In all cases a WARNING will be signaled (see "Warning Flags"). if an indicator variable is used with a character string, when a value is truncated, the indicator variable is set to the length of the value in the database. Note that no warning is flagged if a numeric value is truncated.

# 5.3 The Basic SQL Statements

Executable SQL statements let you query, manipulate, and control Oracle data and create, define, and maintain Oracle objects such as tables, views, and indexes. This chapter focuses on statements which manipulate data in database tables (sometimes called *DML*) and cursor control statements.

The following SQL statements let you query and manipulate Oracle data:

SQL Statements	Description
SELECT	Returns rows from one or more tables.
INSERT	Adds new rows to a table.
UPDATE	Modifies rows in a table.
DELETE	Removes rows from a table.

When executing a data manipulation statement such as INSERT, UPDATE, or DELETE, you want to know how many rows have been updated as well as whether it succeeded. To find out, you simply check the SQLCA. (Executing any SQL statement sets the SQLCA variables.) You can check in the following two ways:

- Implicit checking with the WHENEVER statement
- Explicit checking of SQLCA variables

Alternatively, when MODE={ANSI | ANSI14}, you can check the status variable SQLSTATE or SQLCODE. For more information, see "ANSI SQLSTATE Variable".

When executing a SELECT statement (query), however, you must also deal with the rows of data it returns. Queries can be classified as follows:

- queries that return no rows (that is, merely check for existence)
- queries that return only one row
- queries that return more than one row

Queries that return more than one row require an explicitly declared cursor or cursor variable. The following embedded SQL statements let you define and control an explicit cursor:

SQL Statements	Description
DECLARE	Names the cursor and associates it with a query.
OPEN	Executes the query and identifies the active set.
FETCH	Advances the cursor and retrieves each row in the active set, one by one.
CLOSE	Disables the cursor (the active set becomes undefined.)

In the coming sections, first you learn how to code INSERT, UPDATE, DELETE, and single-row SELECT statements. Then, you progress to multirow SELECT statements. , see the



Oracle Database SQL Language Reference for more information about statements and their clauses.

### 5.3.1 Selecting Rows

Querying the database is a common SQL operation. To issue a query you use the SELECT statement. In the following example, you query the EMP table:

```
EXEC SQL SELECT ENAME, JOB, SAL + 2000
INTO :emp_name, :JOB-TITLE, :SALARY
FROM EMP
WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

The column names and expressions following the keyword SELECT make up the *select list*. The select list in our example contains three items. Under the conditions specified in the WHERE clause (and following clauses, if present), Oracle returns column values to the host variables in the INTO clause. The number of items in the select list should equal the number of host variables in the INTO clause, so there is a place to store every returned value.

In the simplest case, when a query returns one row, its form is that shown in the last example (in which EMPNO is a unique key). However, if a query can return more than one row, you must fetch the rows using a cursor or select them into a host array.

If a query is written to return only one row but might actually return several rows, the result depends on how you specify the option SELECT\_ERROR. When SELECT\_ERROR=YES (the default), Oracle issues an message if more than one row is returned.

When SELECT ERROR=NO, a row is returned and Oracle generates no error.

#### 5.3.1.1 Available Clauses

You can use all of the following standard SQL clauses in your SELECT statements: INTO, FROM, WHERE, CONNECT BY, START WITH, GROUP BY, HAVING, ORDER BY, and FOR UPDATE OF.

### 5.3.2 Inserting Rows

You use the INSERT statement to add rows to a table or view. In the following example, you add a row to the EMP table:

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, SAL, DEPTNO)

VALUES (:EMP_NUMBER, :EMP-NAME, :SALARY, :DEPT-NUMBER)

END-EXEC.
```

Each column you specify in the column list must belong to the table named in the INTO clause. The VALUES clause specifies the row of values to be inserted. The values can be those of constants, host variables, SQL expressions, or pseudocolumns, such as USER and SYSDATE.

The number of values in the VALUES clause must equal the number of names in the column list. You can omit the column list if the VALUES clause contains a value for each column in the table in the same order they were defined by CREATE TABLE, but this is not recommended because a table's definition can change.

### 5.3.3 DML Returning Clause

The INSERT, UPDATE, and DELETE statements can have an optional *DML returning clause* which returns column value expressions *expr*, into host variables *hv*, with host indicator variables *iv*. The returning clause has this syntax:

```
{RETURNING | RETURN} {expr [,expr]}
INTO {:hv [[INDICATOR]:iv] [, :hv [[INDICATOR]:iv]]}
```

The number of expressions must equal the number of host variables. This clause eliminates the need for selecting the rows after an INSERT or UPDATE, and before a DELETE when you need to record that information for your application. The DML returning clause eliminates inefficient network round-trips, extra processing, and server memory. You will also note, for example, when a trigger inserts default values or a primary key value.

The returning\_clause is not allowed with a subquery. It is only allowed after the VALUES clause.

For example, our INSERT could have a clause at its end such as:

```
RETURNING EMPNO, ENAME, DEPTNO INTO :NEW-EMP-NUMBER, :NEW-EMP-NAME, :DEPT
```

#### **Related Topics**

Embedded SQL Statements and Precompiler Directives

### 5.3.4 Using Subqueries

A *subquery* is a nested SELECT statement. Subqueries let you conduct multi-part searches. They can be used to

- supply values for comparison in the WHERE, HAVING, and START WITH clauses of SELECT, UPDATE, and DELETE statements
- define the set of rows to be inserted by a CREATE TABLE or INSERT statement
- define values for the SET clause of an UPDATE statement

For example, to copy rows from one table to another, replace the VALUES clause in an INSERT statement with a subquery, as follows:

```
EXEC SQL INSERT INTO EMP2 (EMPNO, ENAME, SAL, DEPTNO)
SELECT EMPNO, ENAME, SAL, DEPTNO FROM EMP
WHERE JOB = :JOB-TITLE
END-EXEC.
```

Notice how the INSERT statement uses the subquery to obtain intermediate results.

### 5.3.5 Updating Rows

You use the UPDATE statement to change the values of specified columns in a table or view. In the following example, you update the SAL and COMM columns in the EMP table:

```
EXEC SQL UPDATE EMP

SET SAL = :SALARY, COMM = :COMMISSION

WHERE EMPNO = :EMP-NUMBER

END-EXEC.
```

You can use the optional WHERE clause to specify the conditions under which rows are updated. See "Using the WHERE Clause".

The SET clause lists the names of one or more columns for which you must provide values. You can use a subquery to provide the values, as the following example shows:

```
EXEC SQL UPDATE EMP
SET SAL = (SELECT AVG(SAL)*1.1 FROM EMP WHERE DEPTNO = 20)
WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

### 5.3.6 Deleting Rows

You use the DELETE statement to remove rows from a table or view. In the following example, you delete all employees in a given department from the EMP table:

```
EXEC SQL DELETE FROM EMP
WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

You can use the optional WHERE clause to specify the condition under which rows are deleted.

### 5.3.7 Using the WHERE Clause

You use the WHERE clause to select, update, or delete only those rows in a table or view that meet your search condition. The WHERE-clause *search condition* is a Boolean expression, which can include scalar host variables, host arrays (not in SELECT statements), and subqueries.

If you omit the WHERE clause, all rows in the table or view are processed. If you omit the WHERE clause in an UPDATE or DELETE statement, Oracle sets SQLWARN(5) in the SQLCA to 'W' to warn that all rows were processed.

# 5.4 Using Boolean Data Type

Oracle Database supports storing of native boolean data type values. Pro\*COBOL applications can perform bind and fetch operations on boolean data in the database. Using the boolean data type makes it easier to represent boolean state in applications instead of using a character type column to indicate "Y" or "N."

An application can include the boolean data type with the EXEC SQL statements in the following ways:

Declare a variable of boolean data type in the EXEC SQL variable declaration statement.

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

01 var1 PIC 9 BINARY VALUE 0

01 var2 PIC X(5) BINARY VALUE 1

EXEC SQL END DECLARE SECTION END-EXEC.
```

Define an existing table with boolean data type columns.

```
EEXEC SQL DECLARE table name table (column name boolean) END-EXEC.
```

Use boolean data type to define user-defined data type (equivalent of C typedef).

```
EXEC SQL var <variable name> IS boolean END-EXEC.
```

Insert a boolean value into a table.

```
EXEC SQL INSERT INTO table name values(:var1);
```

Fetch data from a boolean data type column.

If data is fetched in a PIC X() type variable (out variable), the data from boolean data type column returns a true or false value. If the output variable type is PIC 9 then a 1 or 0 integer value is returned. If the database column value is NULL then the declared variable is untouched.

The following example gives the output as a boolean type, true or false:

```
b1 PIC X(5) BINARY VALUE 0;
EXEC SQL SELECT col_name INTO :b1
FROM table_name;
```

The following example gives the output as an integer type, 1 or 0:

```
c1 PIC 9(1) BINARY VALUE 0;
EXEC SQL SELECT col_name INTO :c1
FROM table name;
```

### 5.5 Cursors

To process a SQL statement, Oracle opens a work area called a *private SQL area*. The private SQL area stores information needed to execute the SQL statement. An identifier called a *cursor* lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.



For static SQL statements, there are two types of cursors: *implicit* and *explicit*. Oracle implicitly declares a cursor for all data definition and data manipulation statements, including SELECT statements that use the INTO clause.

The set of rows retrieved is called the *results set*; its size depends on how many rows meet the query search condition. You use an explicit cursor to identify the row currently being processed, which is called the *current row*.

When a query returns multiple rows, you can explicitly define a cursor to

- Process beyond the first row returned by the guery
- Keep track of which row is currently being processed

A cursor identifies the current row in the set of rows returned by the query. This allows your program to process the rows one at a time. The following statements let you define and manipulate a cursor:

- DECLARE
- OPEN
- FETCH
- CLOSE

First you use the DECLARE statement (more precisely, a directive) to name the cursor and associate it with a query.

The OPEN statement executes the query and identifies all the rows that meet the query search condition. These rows form a set called the active set of the cursor. After opening the cursor, you can use it to retrieve the rows returned by its associated query.

Rows of the active set are retrieved one by one (unless you use host arrays). You use a FETCH statement to retrieve the current row in the active set. You can execute FETCH repeatedly until all rows have been retrieved.

When done fetching rows from the active set, you disable the cursor with a CLOSE statement, and the active set becomes undefined.

### 5.5.1 Declaring a Cursor

You use the DECLARE statement to define a cursor by giving it a name, as the following example shows:

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR SELECT ENAME, EMPNO, SAL FROM EMP
WHERE DEPTNO = :DEPT_NUMBER
END-EXEC.
```

The cursor name is an identifier used by the precompiler, *not* a host or program variable, and should not be declared in a COBOL statement. Therefore, cursor names cannot be passed from one precompilation unit to another. Cursor names *cannot* be hyphenated. They can be any length, but only the first 31 characters are significant. For ANSI compatibility, use cursor names no longer than 18 characters.

The precompiler option CLOSE\_ON\_COMMIT is provided for use in the command line or in a configuration file. Any cursor not declared with the WITH HOLD clause is closed after a COMMIT or ROLLBACK when CLOSE ON COMMIT=YES.



If MODE is specified at a higher level than CLOSE\_ON\_COMMIT, then MODE takes precedence. The defaults are MODE=ORACLE and CLOSE\_ON\_COMMIT=NO. If you specify MODE=ANSI then any cursors not using the WITH HOLD clause will be closed on COMMIT. The application will run more slowly because cursors are closed and re-opened many times. Setting CLOSE\_ON\_COMMIT=NO when MODE=ANSI results in performance improvement. To see how macro options such as MODE affect micro options such as CLOSE\_ON\_COMMIT, see "Precedence of Option Values".

The SELECT statement associated with the cursor cannot include an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement.

Because it is declarative, the DECLARE statement must physically (not just logically) precede all other SQL statements referencing the cursor. That is, forward references to the cursor are not allowed. In the following example, the OPEN statement is misplaced:

```
EXEC SQL OPEN EMPCURSOR END-EXEC.

* -- MISPLACED OPEN STATEMENT
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
SELECT ENAME, EMPNO, SAL
FROM EMP
WHERE ENAME = :EMP-NAME
END-EXEC.
```

The cursor control statements (DECLARE, OPEN, FETCH, CLOSE) must all occur within the same precompiled unit. For example, you cannot declare a cursor in source file A.PCO, then open it in source file B.PCO.

Your host program can declare as many cursors as it needs. However, in a given file, every DECLARE statement must be unique. That is, you cannot declare two cursors with the same name in one precompilation unit, even across blocks or procedures, because the scope of a cursor is global within a file.

For users of MODE=ANSI or CLOSE\_ON\_COMMIT=YES, the WITH HOLD clause can be used in a DECLARE section to override the behavior defined by the two options. With these options set, the behavior will be for all cursors to be closed when a COMMIT is issued. This can have performance implications due to the overhead of re-opening the cursor to continue processing. The careful use of WITH HOLD can speed up programs that need to conform to the ANSI standard for precompilers in most respects.

#### **Related Topics**

- CLOSE ON COMMIT
- WITH HOLD Clause in DECLARE CURSOR Statements

### 5.5.2 Opening a Cursor

Use the OPEN statement to execute the query and identify the active set. In the following example, a cursor named *EMPCURSOR* is opened.

```
EXEC SOL OPEN EMPCURSOR END-EXEC.
```

OPEN positions the cursor just before the first row of the active set. However, none of the rows is actually retrieved at this point. That will be done by the FETCH statement.

Once you open a cursor, the query's input host variables are not reexamined until you reopen the cursor. Thus, the active set does not change. To change the active set, you must reopen the cursor.



The amount of work done by OPEN depends on the values of three precompiler options: HOLD\_CURSOR, RELEASE\_CURSOR, and MAXOPENCURSORS.

#### **Related Topics**

Using Pro\*COBOL Precompiler Options

### 5.5.3 Fetching from a Cursor

You use the FETCH statement to retrieve rows from the active set and specify the output host variables that will contain the results. Recall that the SELECT statement associated with the cursor cannot include an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement. In the following example, you fetch into three host variables:

```
EXEC SQL FETCH EMPCURSOR
INTO :EMP-NAME, :EMP-NUMBER, :SALARY
END-EXEC.
```

The cursor must have been previously declared and opened. The first time you execute FETCH, the cursor moves from before the first row in the active set to the first row. This row becomes the current row. Each subsequent execution of FETCH advances the cursor to the next row in the active set, changing the current row. The cursor can only move forward in the active set. To return to a row that has already been fetched, you must reopen the cursor, then begin again at the first row of the active set.

If you want to change the active set, you must assign new values to the input host variables in the query associated with the cursor, then reopen the cursor. When MODE=ANSI, you must close the cursor before reopening it.

As the next example shows, you can fetch from the same cursor using different sets of output host variables. However, corresponding host variables in the INTO clause of each FETCH statement must have the same datatype.

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR

SELECT ENAME, SAL FROM EMP WHERE DEPTNO = 20

END-EXEC.

...

EXEC SQL OPEN EMPCURSOR END-EXEC.

EXEC SQL WHENEVER NOT FOUND DO ...

LOOP.

EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME1, :SAL1 END-EXEC

EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME2, :SAL2 END-EXEC

EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME3, :SAL3 END-EXEC

...

GO TO LOOP.

...

END-PERFORM.
```

If the active set is empty or contains no more rows, FETCH returns the "no data found" Oracle warning code to SQLCODE in the SQLCA (if MODE=ANSI then the optional SQLSTATE variable will also be set.) The status of the output host variables is indeterminate. (In a typical program, the WHENEVER NOT FOUND statement detects this error.) To reuse the cursor, you must reopen it.



### 5.5.4 Closing a Cursor

When finished fetching rows from the active set, you close the cursor to free the resources, such as storage, acquired by opening the cursor. When a cursor is closed, parse locks are released. What resources are freed depends on how you specify the options HOLD\_CURSOR and RELEASE\_CURSOR. In the following example, you close the cursor named *EMPCURSOR*:

```
EXEC SOL CLOSE EMPCURSOR END-EXEC.
```

You cannot fetch from a closed cursor because its active set becomes undefined. If necessary, you can reopen a cursor (with new values for the input host variables, for example).

When CLOSE\_ON\_COMMIT=NO (the default when MODE=ORACLE), issuing a COMMIT or ROLLBACK will only close cursors declared using the FOR UPDATE clause or referred to by the CURRENT OF clause. Other cursors that are not affected by the COMMIT or ROLLBACK statement, remain open, if they are open already. However, when CLOSE\_ON\_COMMIT=YES (the default when MODE=ANSI), issuing a COMMIT or ROLLBACK closes *all* cursors.

#### **Related Topics**

CLOSE ON COMMIT

### 5.5.5 Using the CURRENT OF Clause

You use the CURRENT OF *cursor\_name* clause in a DELETE or UPDATE statement to refer to the latest row fetched from the named cursor. The cursor must be open and positioned on a row. If no fetch has been done or if the cursor is not open, the CURRENT OF clause results in an error and processes no rows.

The FOR UPDATE OF clause is optional when you declare a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement. The CURRENT OF clause signals the precompiler to add a FOR UPDATE clause if necessary.

In the following example, you use the CURRENT OF clause to refer to the latest row fetched from a cursor named *EMPCURSOR*:

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR

SELECT ENAME, SAL FROM EMP WHERE JOB = 'CLERK'
END-EXEC.

EXEC SQL OPEN EMPCURSOR END-EXEC.
EXEC SQL WHENEVER NOT FOUND DO ...

LOOP.

EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME, :SALARY
END-EXEC.

EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
WHERE CURRENT OF EMPCURSOR
END-EXEC.
GO TO LOOP.
```

#### **Related Topics**

Mimicking the CURRENT OF Clause



#### 5.5.6 Restrictions

An explicit FOR UPDATE OF or an implicit FOR UPDATE acquires exclusive row locks. All rows are locked at the open, not as they are fetched, and are released when you commit or rollback. If you try to fetch from a FOR UPDATE cursor after a commit, Oracle generates an error.

You cannot use the CURRENT OF clause with a cursor declared with a join since internally, the CURRENT OF mechanism uses the ROWID pseudocolumn and there is no way to specify which table the ROWID relates to. For an alternative, see "Mimicking the CURRENT OF Clause". Finally, you cannot use the CURRENT OF clause in dynamic SQL.

### 5.5.7 A Typical Sequence of Statements

The following example shows the typical sequence of cursor control statements using the CURRENT OF clause and the FOR UPDATE OF clause:

```
* -- Define a cursor.
     EXEC SOL DECLARE EMPCURSOR CURSOR FOR
        SELECT ENAME, JOB FROM EMP
        WHERE EMPNO = :EMP-NUMBER
        FOR UPDATE OF JOB
    END-EXEC.
* -- Open the cursor and identify the active set.
    EXEC SQL OPEN EMPCURSOR END-EXEC.
* -- Exit if the last row was already fetched.
        WHENEVER NOT FOUND GOTO NO-MORE
    END-EXEC.
* -- Fetch and process data in a loop.
LOOP.
     EXEC SQL FETCH EMPCURSOR INTO : EMP-NAME, : JOB-TITLE
      END-EXEC.
* -- host-language statements that operate on the fetched data
      EXEC SQL UPDATE EMP
         SET JOB = :NEW-JOB-TITLE
         WHERE CURRENT OF EMPCURSOR
      END-EXEC.
      GO TO LOOP.
     . . .
MO-MORE.
 -- Disable the cursor.
      EXEC SQL CLOSE EMPCURSOR END-EXEC.
      EXEC SQL COMMIT WORK RELEASE END-EXEC.
    STOP RUN.
```

### 5.5.8 Positioned Update

The following skeletal example demonstrates positioned update using the universal ROWID, which is defined in "Universal ROWIDs":

```
O1 MY-ROWID SQL-ROWID.

EXEC SQL ALLOCATE :MY-ROWID END-EXEC.

EXEC SQL DECLARE C CURSOR FOR
```



```
SELECT ROWID, ... FROM MYTABLE FOR UPDATE OF ... END-EXEC.
EXEC SQL OPEN C END-EXEC.
EXEC SQL FETCH C INTO :MY-ROWID ... END-EXEC.

* Process retrieved data.
...

EXEC SQL UPDATE MYTABLE SET ... WHERE ROWID = :MY-ROWID END-EXEC.
...

NO-MORE-DATA:
EXEC SQL CLOSE C END-EXEC.
EXEC SQL FREE :MY-ROWID END-EXEC.
```

### 5.5.9 The PREFETCH Precompiler Option

The precompiler option PREFETCH allows for more efficient queries by pre-fetching rows. This decreases the number of server round-trips needed and reduces memory required. The number of rows set by the PREFETCH option value in a configuration file or on the command line is used for all queries involving explicit cursors, subject to the standard precedence rules.

When used inline, the PREFETCH option must precede any of these cursor statements:

- EXEC SQL OPEN cursor
- EXEC SQL OPEN cursor USING host\_var\_list
- EXEC SQL OPEN cursor USING DESCRIPTOR desc\_name

When an OPEN is executed, the value of PREFETCH gives the number of rows to be prefetched when the query is executed. You can set the value from 0 (no pre-fetching) to 9999. The default value is 1.



The PREFETCH precompiler option is specifically designed for enhancing the performance of single row fetches. PREFETCH values have no effect when doing array fetches, regardless of which value is assigned.

### 5.6 Scrollable Cursors

A scrollable cursor is a work area where Oracle executes SQL statements and stores information that is processed during execution.

When a cursor is executed, the results of the query are placed into a a set of rows called the result set. The result set can be fetched either sequentially or non-sequentially. Non-sequential result sets are called scrollable cursors.

A scrollable cursor enables users to access the rows of a database result set in a forward, backward, and random manner. This enables the program to fetch any row in the result set.

#### **Related Topics**

About Using Scrollable Cursors in OCI



### 5.6.1 Using Scrollable Cursors

The following statements let you define and manipulate a scrollable cursor.

#### 5.6.1.1 DECLARE SCROLL CURSOR

You can use the DECLARE <cursor name > SCROLL CURSOR statement to name the scrollable cursor and associate it with a query.

#### 5.6.1.2 OPEN

You can use the OPEN statement in the same way as in the case of a non-scrollable cursor.

#### 5.6.1.3 FETCH

You can use the FETCH statement to fetch required rows in a random manner. An application can fetch rows up or down, first or last row directly, or fetch any single row in a random manner.

The following options are available with the FETCH statement.

FETCH FIRST

Fetches the first row from the result set.

2. FETCH PRIOR

Fetches the row prior to the current row.

3. FETCH NEXT

Fetches the next row from the current position. This is same as the non-scrollable cursor FETCH.

4. FETCH LAST

Fetches the last row from the result set.

5. FETCH CURRENT

Fetches the current row.

6. FETCH RELATIVE n

Fetches the *n*th row relative to the current row, where *n* is the offset.

7. FETCH ABSOLUTE n

Fetches the *n*th row, where *n* is the offset from the start of the result set.

The following example describes how to FETCH the last record from a result set.

```
EXEC SQL DECLARE emp_cursor SCROLL CURSOR FOR
SELECT ename, sal FROM emp WHERE deptno=20;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH LAST emp_cursor INTO :emp_name, :sal;
EXEC SQL CLOSE emp cursor;
```



#### 5.6.1.4 CLOSE

You can use the CLOSE statement in the same way as in the case of a non-scrollable cursor.



You cannot use scrollable cursors for REF cursors.

## 5.6.2 The CLOSE\_ON\_COMMIT Precompiler Option

The CLOSE\_ON\_COMMIT micro precompiler option provides the ability to choose whether or not to close all cursors when a COMMIT is executed and the macro option MODE=ANSI. When MODE=ANSI, CLOSE\_ON\_COMMIT has the default value YES. Explicitly setting CLOSE\_ON\_COMMIT=NO results in better performance because cursors will not be closed when a COMMIT is executed, removing the need to re-open the cursors and incur extra parsing.

### 5.6.3 The PREFETCH Precompiler Option

The precompiler option PREFETCH allows for more efficient queries by pre-fetching a given number of rows. This decreases the number of server round trips needed and reduces overall memory usage. The number of rows set by the PREFETCH option value is used for all queries involving explicit cursors, subject to the standard precedence rules. When used inline, the PREFETCH option must precede any of these cursor statements:

- EXEC SQL OPEN cursor
- EXEC SQL OPEN cursor USING host var list
- EXEC SQL OPEN cursor USING DESCRIPTOR desc\_name

When an OPEN is executed, the value of PREFETCH gives the number of rows to be prefetched when the query is executed. You can set the value from 0 (no pre-fetching) to 65535. The default value is 1.



The default value of the PREFETCH option is 1 - return a single row for each round-trip. If you choose not to use the PREFETCH option, using the command line, you must explicitly disable it by setting the PREFETCH option to 0.

PREFETCH is automatically disabled when LONG or LOB columns are being retrieved.



PREFETCH is used primarily to enhance the performance of single row fetches. PREFETCH has no effect when array fetches are used.

#### Note:

The PREFETCH option should be used wisely, and on a case-by-case basis. Select an appropriate prefetch value that will optimize performance of a specific FETCH statement. To accomplish this, use the inline prefetch option instead of the command line prefetch option.

#### Note:

The performance of many large applications can be improved simply by using indicator variables with host variables in FETCH statements.

To enable precompiler applications to obtain the maximum advantage from the use of the PREFETCH option on single row fetches, it is strongly recommended that you use indicator variables.

## 5.7 Flexible B Area Length

The length of B Area for a Pro\*Cobol program is limited to 72 when the format is set to ANSI. Cobol compilers now can support B Area length up to 253. This provides a developer with the flexibility to type a line that is longer than 72 columns. Pro\*Cobol now supports B area length up to 253 when a Pro\*Cobol application is precompiled with the

FORMAT=VARIABLE IRECLEN=300

options.

# 5.8 Fix Execution Plans

In application development environments where modules are developed in one environment, and then integrated and deployed into another, the performance of the applications are affected. At times, the performance of the precompiler applications are affected by changes in the database environment. These may include changes in the optimizer statistics, changes to the optimizer settings, or changes to parameters affecting the sizes of memory structures.

To fix execution plans for SQL's used in Pro\*Cobol in the development environment, you need to use the outline feature of Oracle at the time of precompiling. An outline is implemented as a set of optimizer hints that are associated with the SQL statement. If you enable the use of the outline for the statement, Oracle automatically considers the stored hints and tries to generate an execution plan in accordance with those hints. In

this way, you can ensure that the performance is not affected when the modules are integrated or deployed into different environments.

You can use the following SQL statements to create outlines in Pro\*Cobol:

- SELECT
- DELETE
- UPDATE
- INSERT... SELECT
- CREATE TABLE... AS SELECT

If the outline option is set, then the precompiler generates two files, a SQL file and a LOG file at the end of successful precompilation. Command line options outline and outlnprefix control the generation of the outlines.



Precompiler Options for more details on these command line options

Each generated outline name is unique. Because the file names used in the application are unique, this information is used in generating the outline name. In addition, the category name is also prefixed.

#### A

#### Caution:

DATA DIVISION.

WORKING-STORAGE SECTION.

Oracle allows only 128 bytes for the outline name. If you exceed the limit, the precompiler will flag an error. You can restrict the length of the outline name by using the <code>outlnprefix</code> option.

#### **Example 5-1** Generating a SQL File Containing Outlines

You need to precompile the following program by using the outline option to generate SQL files containing the outlines for all the outline-supported SQL statements in this program.



\* EMBEDDED COBOL (file "OUTNDEMO.PCO")

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01 USERNAME PIC X(10) VARYING.

01 PASSWD PIC X(10) VARYING.

01 ENAME PIC X(10).

01 JOB PIC X(9).

01 SAL PIC X(6).

01 COMM PIC X(6).

EXEC SQL END DECLARE SECTION END-EXEC.

01 STRINGFIELDS.

02 STR PIC X(18) VARYING.

EXEC SQL INCLUDE SQLCA END-EXEC.

PROCEDURE DIVISION.

BEGIN-PGM.

EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR END-EXEC.

PERFORM LOGON.

EXEC SQL INSERT INTO BONUS
SELECT ENAME, JOB, SAL, COMM FROM EMP
WHERE JOB LIKE 'SALESMAN'
END-EXEC.

EXEC SQL UPDATE BONUS
SET SAL = SAL \* 1.1 WHERE SAL < 1500
END-EXEC.

EXEC SQL DECLARE C1 CURSOR FOR SELECT ENAME, JOB, SAL, COMM FROM BONUS ORDER BY SAL END-EXEC.

EXEC SQL OPEN C1 END-EXEC.

DISPLAY "Contents of updated BONUS table".
DISPLAY "ENAME JOB SAL COMM".
DISPLAY " ".

EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.

FETCH-DATA.

EXEC SQL FETCH C1 INTO :ENAME, :JOB, :SAL, :COMM END-EXEC.

DISPLAY ENAME, JOB, SAL, COMM.

GO TO FETCH-DATA.

END-FETCH.

EXEC SQL CLOSE C1 END-EXEC.

EXEC SQL WHENEVER NOT FOUND DO PERFORM SQL-ERROR END-EXEC.

EXEC SQL DELETE FROM BONUS END-EXEC.

EXEC SQL CREATE TABLE OUTLNDEMO\_TAB AS SELECT EMPNO, ENAME, SAL FROM EMP WHERE DEPTNO = 10 END-EXEC.

EXEC SQL DROP TABLE OUTLNDEMO TAB END-EXEC.

EXEC SQL ROLLBACK WORK RELEASE END-EXEC.



```
STOP RUN.
LOGON.
    MOVE "scott" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "tiger" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL
        CONNECT : USERNAME IDENTIFIED BY : PASSWD
    END-EXEC.
* HANDLES SQL ERROR CONDITIONS
SOL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
```

### 5.8.1 SQL File

The generated file name has the following format:

```
<filename> <filetype>.sql
```

In Pro\*Cobol, for the file "abc.pco", the generated SQL file will be abc pco.sql.

#### Generated file format

If the outInprefix option is not used, then the format of the unique identifier used as outline name and comment is:

```
<category_name>_<filename>_<filetype>_<sequence no.>
```

If the outInprefix option is used (outInprefix=cprefix\_name>), then the format of the unique identifier used as outline name and comment is:

```
fix name> <sequence no.>
```

If outline=yes, which is the default category, then <category\_name> will be DEFAULT and outline name will be:

```
DEFAULT_<filename>_<filetype>_<sequence no.>

Or

<prefix_name>_<sequence no.>
```

The allowed range for <sequence no. > is 0000 to 9999.

SQL in the generated precompiled file will have the comment appended to it as it appears in the outline for that SQL.

### 5.8.1.1 Examples

Consider the following examples.



#### **Example 1**

If abc.pco has the statements

```
EXEC SQL select * from emp where empno=:var END-EXEC. EXEC SQL select * from dept END-EXEC.
```

and if outline=mycat1 and outlnprefix is not used, then:

#### Contents of abc\_pco.sql

```
create or replace outline mycat1_abc_pco_0000 for category mycat1 on select *
from emp where empno=:b1 /* mycat1_abc_pco_0000 */;
```

create or replace outline mycat1\_abc\_pco\_0001 for category mycat1 on select \*
from dept /\* mycat1 abc pco 0001 \*/;

#### Contents of abc.cob

```
01 SQ0001 GLOBAL.

02 FILLER PIC X(60) VALUE "select * from emp where empno=:b1

- /* mycat1_abc_pco_0000 */

- "".
```

#### **Example 2**

If abc.pco has the statements

```
EXEC SQL select * from emp where empno=:var END-EXEC. EXEC SQL select * from dept END-EXEC.
```

and if outline=mycat1 and outlnprefix=myprefix, then:

#### Contents of abc\_pco.sql

```
create or replace outline myprefix_0000 for category mycat1 on select * from emp where empno=:b1 /* myprefix_0000 */;
```

create or replace outline myprefix\_0001 for category mycat1 on select \* from
dept /\* myprefix 0001 \*/;

#### Contents of abc.cob

```
01 SQ0001 GLOBAL.

02 FILLER PIC X(60) VALUE "select * from emp where empno=:b1
- /* myprefix_0000 */
- "".
```

#### **Example 3**

If abc.pco has the statements

```
EXEC SQL select * from emp where empno=:var END-EXEC. EXEC SQL select * from dept END-EXEC.
```

and if outline=yes and outInprefix=myprefix, then:



#### Contents of abc\_pco.sql

```
create or replace outline myprefix_0000 on select * from emp where empno=:b1 /*
myprefix_0000 */;
create or replace outline myprefix_0001 on select * from dept /* myprefix 0001 */;
```

#### Contents of abc.cob

```
01 SQ0001 GLOBAL.

03 FILLER PIC X(60) VALUE "select * from emp where empno=:b1

- /* myprefix_0000 */
_ ""
```

#### 5.8.2 LOG File

The generated file name has the following format:

```
<filename> <filetype>.log
```

In Pro\*Cobol, for the file "abc.pco", the generated LOG file will be abc\_pco.log.

Consider the following example.

#### Example 1

If abc.pco has the statement

```
EXEC SQL select * from emp END-EXEC.
```

#### Contents of abc\_pco.log

```
CATEGORY <Category_name>
Source SQL_0
SELECT * FROM emp
OUTLINE NAME
abc_pco_0000
OUTLINE SQL_0
Select * from emp /* abc pco 0000 */
```

# 5.9 Sample Program 2: Cursor Operations

This program logs on to Oracle, declares and opens a cursor, fetches the names, salaries, and commissions of all salespeople, displays the results, then closes the cursor

All fetches except the final one return a row and, if no errors were detected during the fetch, a success status code. The final fetch fails and returns the "no data found" Oracle warning code to SQLCODE in the SQLCA. The cumulative number of rows actually fetched is found in SQLERRD(3) in the SQLCA.



```
IDENTIFICATION DIVISION.
PROGRAM-ID. CURSOR-OPS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME
                   PIC X(10) VARYING.
01 PASSWD
                   PIC X(10) VARYING.
01 EMP-REC-VARS.
   05 EMP-NAME
                    PIC X(10) VARYING.
   05 SALARY
                    PIC S9(6)V99
                     DISPLAY SIGN LEADING SEPARATE.
   05 COMMISSION
                    PIC S9(6)V99
                    DISPLAY SIGN LEADING SEPARATE.
   EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
   EXEC SQL VAR COMMISSION IS DISPLAY(8,2) END-EXEC.
   EXEC SQL END DECLARE SECTION END-EXEC.
   EXEC SQL INCLUDE SQLCA END-EXEC.
01 DISPLAY-VARIABLES.
   05 D-EMP-NAME PIC X(10).
   05 D-SALARY PIC Z(4)9.99.
   05 D-COMMISSION PIC Z(4)9.99.
PROCEDURE DIVISION.
BEGIN-PGM.
   EXEC SQL WHENEVER SQLERROR
       DO PERFORM SQL-ERROR END-EXEC.
   PERFORM LOGON.
   EXEC SQL DECLARE SALESPEOPLE CURSOR FOR
       SELECT ENAME, SAL, COMM
       FROM EMP
       WHERE JOB LIKE 'SALES%'
   END-EXEC.
   EXEC SOL OPEN SALESPEOPLE END-EXEC.
   DISPLAY " ".
   DISPLAY "SALESPERSON SALARY
                                   COMMISSION".
   DISPLAY "----".
FETCH-LOOP.
   EXEC SQL WHENEVER NOT FOUND
      DO PERFORM SIGN-OFF END-EXEC.
   EXEC SQL FETCH SALESPEOPLE
       INTO :EMP-NAME, :SALARY, :COMMISSION
   END-EXEC.
   MOVE EMP-NAME-ARR TO D-EMP-NAME.
   MOVE SALARY TO D-SALARY.
   MOVE COMMISSION TO D-COMMISSION.
   DISPLAY D-EMP-NAME, " ", D-SALARY, " ", D-COMMISSION.
   MOVE SPACES TO EMP-NAME-ARR.
   GO TO FETCH-LOOP.
LOGON.
   MOVE "SCOTT" TO USERNAME-ARR.
   MOVE 5 TO USERNAME-LEN.
   MOVE "TIGER" TO PASSWD-ARR.
   MOVE 5 TO PASSWD-LEN.
   EXEC SOL
```

```
CONNECT : USERNAME IDENTIFIED BY : PASSWD
   END-EXEC.
   DISPLAY " ".
   DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
SIGN-OFF.
   EXEC SQL CLOSE SALESPEOPLE END-EXEC.
   DISPLAY " ".
   DISPLAY "HAVE A GOOD DAY.".
   DISPLAY " ".
   EXEC SQL COMMIT WORK RELEASE END-EXEC.
   STOP RUN.
SQL-ERROR.
   EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
   DISPLAY " ".
   DISPLAY "ORACLE ERROR DETECTED:".
   DISPLAY " ".
   DISPLAY SQLERRMC.
   EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
   STOP RUN.
```



6

# Embedded PL/SQL

This chapter shows you how to improve performance by embedding PL/SQL transaction processing blocks in your program. This chapter has the following sections:

- Embedding PL/SQL
- · Advantages of PL/SQL
- Embedding PL/SQL Blocks
- Host Variables and PL/SQL
- Indicator Variables and PL/SQL
- Host Tables and PL/SQL
- Cursor Usage in Embedded PL/SQL
- Stored PL/SQL and Java Subprograms
- Sample Program 9: Calling a Stored Procedure
- Cursor Variables

# 6.1 Embedding PL/SQL

Pro\*COBOL treats a PL/SQL block like a single embedded SQL statement. You can place a PL/SQL block anywhere in a host program that you can place a SQL statement.

To embed a PL/SQL block in your host program, declare the variables to be shared with PL/SQL and bracket the PL/SQL block with the EXEC SQL EXECUTE and END-EXEC keywords.

### 6.1.1 Host Variables

Inside a PL/SQL block, host variables are global to the entire block and can be used anywhere a PL/SQL variable is allowed. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon. The colon sets host variables apart from PL/SQL variables and database objects.

## 6.1.2 VARCHAR Variables

When entering a PL/SQL block, Oracle automatically checks the length fields of VARCHAR host variables. Therefore, you must set the length fields *before* the block is entered. For input variables, set the length field to the length of the value stored in the string field. For output variables, set the length field to the maximum length allowed by the string field.

## 6.1.3 Indicator Variables

In a PL/SQL block, you cannot refer to an indicator variable by itself; it must be appended to its associated host variable. Further, if you refer to a host variable with its indicator variable, you must always refer to it that way in the same block.

### 6.1.3.1 Handling NULLs

When entering a block, if an indicator variable has a value of -1, PL/SQL automatically assigns a NULL to the host variable. When exiting the block, if a host variable is NULL, PL/SQL automatically assigns a value of -1 to the indicator variable.

### 6.1.3.2 Handling Truncated Values

PL/SQL does not raise an exception when a truncated string value is assigned to a host variable. However, if you use an indicator variable, PL/SQL sets it to the original length of the string.

## 6.1.4 SQLCHECK

You must specify SQLCHECK=SEMANTICS when precompiling a program with an embedded PL/SQL block. You must also use the USERID option.

#### **Related Topics**

· Precompiler Options

# 6.2 Advantages of PL/SQL

This section looks at some of the features and benefits offered by PL/SQL, such as:

- Better performance
- Integration with Oracle
- Cursor FOR loops
- Procedures and functions
- Packages
- PL/SQL tables
- · User-defined records



Oracle Database PL/SQL Language Reference for more information about PL/SQL.

### 6.2.1 Better Performance

PL/SQL can help you reduce overhead, improve performance, and increase productivity. For example, without PL/SQL, Oracle must process SQL statements one at a time. Each SQL statement results in another call to the Server and higher overhead. However, with PL/SQL, you can send an entire block of SQL statements to the server. This minimizes communication between your application and the server.



## 6.2.2 Integration with Oracle

PL/SQL is tightly integrated with the server. For example, most PL/SQL datatypes are native to the data dictionary. Furthermore, you can use the %TYPE attribute to base variable declarations on column definitions stored in the data dictionary, as the following example shows:

```
job title emp.job%TYPE;
```

That way, you need not know the exact datatype of the column. Furthermore, if a column definition changes, the variable declaration changes accordingly and automatically. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes.

### 6.2.3 Cursor FOR Loops

With PL/SQL, you need not use the DECLARE, OPEN, FETCH, and CLOSE statements to define and manipulate a cursor. Instead, you can use a cursor FOR loop, which implicitly declares its loop index as a record, opens the cursor associated with a given query, repeatedly fetches data from the cursor into the record and then closes the cursor. An example follows:

```
DECLARE
...

BEGIN

FOR emprec IN (SELECT empno, sal, comm FROM emp) LOOP

IF emprec.comm / emprec.sal > 0.25 THEN ...

END LOOP;

END;
```

Notice that you use dot notation to reference fields in the record.

### 6.2.4 Subprograms

PL/SQL has two types of subprograms called *procedures* and *functions*, which aid application development by letting you isolate operations. Generally, you use a procedure to perform an action and a function to compute a value.

Procedures and functions provide *extensibility*. That is, they let you tailor the PL/SQL language to suit your needs. For example, if you need a procedure that creates a new department, you can write your own, such as follows:

```
PROCEDURE create_dept
    (new_dname IN CHAR(14),
        new_loc IN CHAR(13),
        new_deptno OUT NUMBER(2)) IS

BEGIN
    SELECT deptno_seq.NEXTVAL INTO new_deptno FROM dual;
    INSERT INTO dept VALUES (new_deptno, new_dname, new_loc);

END create dept;
```

When called, this procedure accepts a new department name and location, selects the next value in a department-number database sequence, inserts the new number, name, and location into the *dept* table and then returns the new number to the caller.

You can store subprograms in the database (using CREATE FUNCTION and CREATE PROCEDURE) that can be called from multiple applications without needing to be recompiled each time.

#### 6.2.4.1 Parameter Modes

You use *parameter modes* to define the behavior of formal parameters. There are three parameter modes: IN (the default), OUT, and IN OUT. An IN parameter lets you pass values to the subprogram being called. An OUT parameter lets you return values to the caller of a subprogram. An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

The datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Table 6-1 shows the legal conversions between datatypes.

## 6.2.5 Packages

PL/SQL lets you bundle logically related types, program objects, and subprograms into a *package*. Packages can be compiled and stored in a database, where their contents can be shared by multiple applications.

Packages usually have two parts: a specification and a body. The *specification* is the interface to your applications; it declares the types, constants, variables, exceptions, cursors, and subprograms available for use. The *body* defines cursors and subprograms and so implements the specification. The following example "packages" two employment procedures:

```
PACKAGE emp_actions IS -- package specification

PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);

PROCEDURE fire_employee (emp_id NUMBER);

END emp_actions;

PACKAGE BODY emp_actions IS -- package body

PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS

BEGIN

INSERT INTO emp VALUES (empno, ename, ...);

END hire_employee;

PROCEDURE fire_employee (emp_id NUMBER) IS

BEGIN

DELETE FROM emp WHERE empno = emp_id;

END fire_employee;

END emp actions;
```

Only the declarations in the package specification are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible.

### 6.2.6 PL/SQL Tables

PL/SQL provides a composite datatype named TABLE. Objects of type TABLE are called *PL/SQL tables*, which are modeled as (but not the same as) database tables. PL/SQL tables have only one column and use a primary key to give you array-like access to rows. The column can belong to any scalar type (such as CHAR, DATE, or NUMBER), but the primary key must belong to type BINARY\_INTEGER, PLS\_INTEGER or VARCHAR2.



You can declare PL/SQL table types in the declarative part of any block, procedure, function, or package. The following example declares a TABLE type called *NumTabTyp*:

```
DECLARE

TYPE NumTabTyp IS TABLE OF NUMBER

INDEX BY BINARY_INTEGER;

...

BEGIN
...
END;
```

Once you define type *NumTabTyp*, you can declare PL/SQL tables of that type, as the next example shows:

```
num tab NumTabTyp;
```

The identifier *num\_tab* represents an entire PL/SQL table.

You reference rows in a PL/SQL table using array-like syntax to specify the primary key value. For example, you reference the ninth row in the PL/SQL table named *num\_tab* as follows:

```
num_tab(9) ...
```

### 6.2.7 User-Defined Records

You can use the %ROWTYPE attribute to declare a record that represents a row in a database table or a row fetched by a cursor. However, you cannot specify the datatypes of fields in the record or define fields of your own. The composite datatype RECORD lifts those restrictions.

Objects of type RECORD are called *records*. Unlike PL/SQL tables, records have uniquely named fields, which can belong to different datatypes. For example, suppose you have different kinds of data about an employee such as name, salary, hire date, and so on. This data is dissimilar in type but logically related. A record that contains such fields as the name, salary, and hire date of an employee would let you treat the data as a logical unit.

You can declare record types and objects in the declarative part of any block, procedure, function, or package. The following example declares a RECORD type called *DeptRecTyp*:

```
DECLARE
   TYPE DeptRecTyp IS RECORD
        (deptno NUMBER(4) NOT NULL := 10, -- must initialize
        dname CHAR(9),
        loc CHAR(14));
```

Notice that the field declarations are like variable declarations. Each field has a unique name and specific datatype. You can add the NOT NULL option to any field declaration and so prevent the assigning of NULLs to that field. However, you must initialize NOT NULL fields.

Once you define type *DeptRecTyp*, you can declare records of that type, as the next example shows:

```
dept rec DeptRecTyp;
```

The identifier *dept\_rec* represents an entire record.

You use dot notation to reference individual fields in a record. For example, you reference the *dname* field in the *dept\_rec* record as follows:



```
dept_rec.dname ...
```

# 6.3 Embedding PL/SQL Blocks

Pro\*COBOL treats a PL/SQL block like a single embedded SQL statement. Thus, you can place a PL/SQL block anywhere in a host program that you can place a SQL statement.

To embed a PL/SQL block in your host program, simply bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END-EXEC as follows:

```
EXEC SQL EXECUTE
DECLARE
...
BEGIN
...
END;
END-EXEC.
```

When your program embeds PL/SQL blocks, you must specify the precompiler option SQLCHECK=SEMANTICS because PL/SQL must be parsed by Pro\*COBOL. To connect to the server, you must also specify the option USERID.

#### **Related Topics**

Using Pro\*COBOL Precompiler Options

# 6.4 Host Variables and PL/SQL

Host variables are the key to communication between a host language and a PL/SQL block. Host variables can be shared with PL/SQL, meaning that PL/SQL can set and reference host variables.

For example, you can prompt a user for information and use host variables to pass that information to a PL/SQL block. Then, PL/SQL can access the database and use host variables to pass the results back to your host program.

Inside a PL/SQL block, host variables are treated as global to the entire block and can be used anywhere a PL/SQL variable is allowed. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon. The colon sets host variables apart from PL/SQL variables and database objects.

## 6.4.1 PL/SQL Examples

The following example illustrates the use of host variables with PL/SQL. The program prompts the user for an employee number and then displays the job title, hire date, and salary of that employee.

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

01 USERNAME PIC X(20) VARYING.

01 PASSWORD PIC X(20) VARYING.

01 EMP-NUMBER PIC S9(4) COMP.

01 JOB-TITLE PIC X(20) VARYING.

01 HIRE-DATE PIC X(9) VARYING.

01 SALARY PIC S9(6)V99

DISPLAY SIGN LEADING SEPARATE.

EXEC SQL END DECLARE SECTION END-EXEC.
```



```
EXEC SQL INCLUDE SQLCA END-EXEC.
   DISPLAY 'Connected to Oracle'.
   DISPLAY 'Employee Number (0 to end)? 'WITH NO ADVANCING
   ACCEPT EMP-NUMBER
   IF EMP-NUMBER = 0
       EXEC SOL COMMIT WORK RELEASE END-EXEC
       DISPLAY 'Exiting program'
       STOP RUN
   END-IF.
   ----- begin PL/SQL block -----
   EXEC SQL EXECUTE
      BEGIN
          SELECT job, hiredate, sal
             INTO : JOB-TITLE, : HIRE-DATE, : SALARY
             FROM EMP
             WHERE EMPNO = :EMP-NUMBER;
      END:
   END-EXEC.
   ----- end PL/SQL block -----
   DISPLAY 'Number Job Title Hire Date Salary'.
   DISPLAY '----'.
   DISPLAY EMP-NUMBER, JOB-TITLE, HIRE-DATE, SALARY.
END-PERFORM.
   . . .
SQL-ERROR.
   EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
   EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
   DISPLAY 'Processing error'.
   STOP RUN.
```

Notice that the host variable *EMP-NUMBER* is set before the PL/SQL block is entered, and the host variables *JOB-TITLE*, *HIRE-DATE*, and *SALARY* are set inside the block.

### 6.4.2 A More Complex PL/SQL Example

In the following example the user is prompted for a bank account number, transaction type, and transaction amount. The account is then debited or credited. If the account does not exist, an exception is raised. When the transaction is complete its status is displayed.

```
EXEC SOL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(20) VARYING.
01 ACCT-NUM PIC S9(4) COMP.
01 TRANS-TYPE PIC X(1).
01 TRANS-AMT PIC PIC S9(6)V99
                      DISPLAY SIGN LEADING SEPARATE.
            PIC X(80) VARYING.
01 STATUS
   EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
    DISPLAY 'Username? 'WITH NO ADVANCING.
    ACCEPT USERNAME.
    DISPLAY 'Password? '.
    ACCEPT PASSWORD.
    EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR.
    EXEC SQL CONNECT : USERNAME IDENTIFIED BY : PASSWORD.
    PERFORM
    DISPLAY 'Account Number (0 to end)? '
        WITH NO ADVANCING
    ACCEPT ACCT NUM
```

```
IF ACCT-NUM = 0
       EXEC SQL COMMIT WORK RELEASE END-EXEC
       DISPLAY 'Exiting program' WITH NO ADVANCING
    END-IF.
    DISPLAY 'Transaction Type - D)ebit or C)redit? '
       WITH NO ADVANCING
    ACCEPT TRANS-TYPE
    DISPLAY 'Transaction Amount? '
    ACCEPT trans amt
   ----- begin PL/SQL block -----
    EXEC SQL EXECUTE
      DECLARE
          oru_pal NUMBER(9,2);
err_msg CHAP(77)
          nonexistent EXCEPTION;
      BEGIN
         IF :TRANS-TYP-TYPE = 'C' THEN
                                            -- credit the account
              UPDATE accts SET bal = bal + :TRANS-AMT
                  WHERE acctid = :acct-num;
              IF SQL\ROWCOUNT = 0 THEN -- no rows affected
                  RAISE nonexistent;
              ELSE
                 :STATUs := 'Credit applied';
              END IF;
          ELSIF :TRANS-TYPe = 'D' THEN
                                       -- debit the account
              SELECT bal INTO old bal FROM accts
                  WHERE acctid = :ACCT-NUM;
              IF old bal >= :TRANS-AMT THEN   -- enough funds
                  UPDATE accts SET bal = bal - :TRANS-AMT
                      WHERE acctid = :ACCT-NUM;
                  :STATUS := 'Debit applied';
              ELSE
                 :STATUS := 'Insufficient funds';
              END IF;
            :STATUS := 'Invalid type: ' | :TRANS-TYPE;
          END IF;
          COMMIT;
      EXCEPTION
          WHEN NO DATA FOUND OR nonexistent THEN
              :STATUS := 'Nonexistent account';
          WHEN OTHERS THEN
              err msg := SUBSTR(SQLERRM, 1, 70);
             :STATUS := 'Error: ' || err msg;
      END;
    ----- end PL/SQL block -----
    DISPLAY 'Status: ', STATUS
END-PERFORM.
SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    DISPLAY 'Processing error'.
    STOP RUN.
```

## 6.4.3 VARCHAR Pseudotype

Recall that you can use the VARCHAR pseudotype to declare variable-length character strings. If the VARCHAR is an input host variable, you must tell Pro\*COBOL what length to expect. Therefore, set the length field to the actual length of the value stored in the string field.

If the VARCHAR is an output host variable, Pro\*COBOL automatically sets the length field. However, to use a VARCHAR output host variable in your PL/SQL block, you must initialize the length field *before* entering the block. Therefore, set the length field to the declared (maximum) length of the VARCHAR, as shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01 EMP-NUM PIC S9(4) COMP.
       01 EMP-NAME PIC X(10) VARYING.
       01 SALARY PIC S9(6)V99
                   DISPLAY SIGN LEADING SEPARATE.
    EXEC SOL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
* -- initialize length field
    MOVE 10 TO EMP-NAME-LEN.
    EXEC SQL EXECUTE
    BEGIN
       SELECT ename, sal INTO : EMP-NAME, : SALARY
           FROM emp
           WHERE empno = :EMP-NUM;
    END;
    END-EXEC.
```

# 6.5 Indicator Variables and PL/SQL

PL/SQL does not need indicator variables because it can manipulate NULLs. For example, within PL/SQL, you can use the IS NULL operator to test for NULLs, as follows:

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

You can use the assignment operator (:=) to assign NULLs, as follows:

```
variable := NULL;
```

However, host languages need indicator variables because they cannot manipulate NULLs. Embedded PL/SQL meets this need by letting you use indicator variables to:

- Accept NULLs input from a host program
- Output NULLs or truncated values to a host program

When used in a PL/SQL block, indicator variables are subject to the following rule:

• If you refer to a host variable with an indicator variable, you must always refer to it that way in the same block.

In the following example, the indicator variable *IND-COMM* appears with its host variable *COMMISSION* in the SELECT statement, so it must appear that way in the IF statement:

```
EXEC SQL EXECUTE

BEGIN

SELECT ename, comm

INTO :EMP-NAME, :COMMISSION:IND-COMM FROM emp

WHERE empno = :EMP-NUM;

IF :COMMISSION:IND-COMM IS NULL THEN ...

END;

END;

END, EXEC.
```

Notice that PL/SQL treats :COMMISSION:IND-COMM like any other simple variable. Though you cannot refer directly to an indicator variable inside a PL/SQL block, PL/SQL checks the value of the indicator variable when entering the block and sets the value correctly when exiting the block.

## 6.5.1 Handling NULLs

When entering a block, if an indicator variable has a value of -1, PL/SQL automatically assigns a NULL to the host variable. When exiting the block, if a host variable is NULL, PL/SQL automatically assigns a value of -1 to the indicator variable. In the next example, if *IND-SAL* had a value of -1 before the PL/SQL block was entered, the *salary missing* exception is raised. An *exception* is a named error condition.

```
EXEC SQL EXECUTE
BEGIN
    IF :SALARY:IND-SAL IS NULL THEN
        RAISE salary_missing;
    END IF;
    ...
END;
END-EXEC.
```

### 6.5.2 Handling Truncated Values

PL/SQL does not raise an exception when a truncated string value is assigned to a host variable. However, if you use an indicator variable, PL/SQL sets it to the original length of the string. The following example the host program will be able to tell, by checking the value of *IND-NAME*, if a truncated value was assigned to *EMP-NAME*:

```
EXEC SQL EXECUTE
DECLARE
...
new_name CHAR(10);
BEGIN
...
:EMP_NAME:IND-NAME := new_name;
...
END;
END-EXEC.
```

# 6.6 Host Tables and PL/SQL

You can pass input host tables and indicator tables to a PL/SQL block. They can be indexed by a PL/SQL variable of type BINARY\_INTEGER or PLS\_INTEGER; VARCHAR2 key types are not permitted. Normally, the entire host table is passed to PL/SQL, but you can use the ARRAYLEN statement (discussed later) to specify a smaller table dimension.

Furthermore, you can use a subprogram call to assign all the values in a host table to rows in a PL/SQL table. Given that the table subscript range is  $m \dots n$ , the corresponding PL/SQL table index range is always  $1 \dots (n - m + 1)$ . For example, if the table subscript range is  $5 \dots 10$ , the corresponding PL/SQL table index range is  $1 \dots (10 - 5 + 1)$  or  $1 \dots 6$ .

**Note:** Pro\*COBOL does not check your usage of host tables. For instance, no index range checking is done.

In the following example, you pass a host table named *salary* to a PL/SQL block, which uses the host table in a function call. The function is named *median* because it finds the middle value in a series of numbers. Its formal parameters include a PL/SQL table named *num\_tab*. The function call assigns all the values in the actual parameter *salary* to rows in the formal parameter *num\_tab*.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SALARY OCCURS 100 TIMES PIC S9(6)V99 COMP-3.
01 MEDIAN-SALARY PIC S9(6)V99 COMP-3.
    EXEC SQL END DECLARE SECTION END-EXEC.
* -- populate the host table
    EXEC SQL EXECUTE
    DECLARE
       TYPE NumTabTyp IS TABLE OF REAL
          INDEX BY BINARY INTEGER;
       n BINARY INTEGER;
       FUNCTION median (num tab NumTabTyp, n INTEGER)
           RETURN REAL IS
       BEGIN
* -- compute median
       END;
    BEGIN
       n := 100;
       :MEDIAN-SALARY := median(:SALARY
                                            END:
    END-EXEC.
```

You can also use a subprogram call to assign all row values in a PL/SQL table to corresponding elements in a host table. For an example, see "Stored PL/SQL and Java Subprograms".

The interface between Host Tables and PL/SQL strictly controls datatypes. The default external type for PIC X is CHARF (fixed length character string) and this can only be mapped to PL/SQL tables of type CHAR.

Table 6-1 shows the legal conversions between row values in a PL/SQL table and elements in a host table. The most notable fact is that you cannot pass a PIC X variable to a table of type VARCHAR2 without using datatype equivalencing to equivalence the variable to VARCHAR2, or using PICX=VARCHAR2 on the command line.

Table 6-1 Legal Datatype Conversions

PL/SQL Table	-	-	-	-	-	-	-	-
Host Table	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	X	-	-	-	-	-	-	-
CHARZ	X	-	-	-	-	-	-	-
DATE	-	Χ	-	_	_	-	-	-



Table 6-1 (Cont.) Legal Datatype Conversions

PL/SQL Table	-		-	-	-	-	-	-
DECIMAL	-	-	-	-	X	-	-	-
DISPLAY	-	-	-	-	Χ	-	-	-
FLOAT	-	-	-	-	Χ	-	-	-
INTEGER	-	-	-	-	-	-	-	-
LONG	Χ	-	Χ	-	-	-	-	-
LONG VARCHAR	-	-	X	X	-	Χ	-	X
LONG VARRAW	-	-	-	X	-	Χ	-	-
NUMBER	-	-	-	-	Χ		-	-
RAW	-	-	-	Χ	-	X	-	-
ROWID	-	-	-	-	-	-	Χ	-
STRING	-	-	Χ	Χ	-	X	-	X
UNSIGNED	-	-	-	-	X	-	-	-
VARCHAR	-	-	Χ	Χ	-	X	-	X
VARCHAR2	-	-	Χ	Χ	-	Χ	-	X
VARNUM	-	-	-	-	Χ	-	-	-
VARRAW	-	-	-	Χ	-	Χ	-	-

#### 6.6.1 ARRAYLEN Statement

Suppose you must pass an input host table to a PL/SQL block for processing. By default, when binding such a host table, Pro\*COBOL use its declared dimension. However, you might not want to process the entire table. In that case, you can use the ARRAYLEN statement to specify a smaller table dimension. ARRAYLEN associates the host table with a host variable, which stores the smaller dimension. The statement syntax is:

```
EXEC SQL ARRAYLEN host array (dimension) EXECUTE END-EXEC.
```

where *dimension* is a 4-byte, integer host variable, *not* a literal or an expression.

The ARRAYLEN statement must appear somewhere after the declarations of *host\_array* and *dimension*. You cannot specify an offset into the host table. However, you might be able to use COBOL features for that purpose.

The following example uses ARRAYLEN to override the default dimension of a host table named *BONUS*:

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

01 BONUS OCCURS 100 TIMES PIC S9(6)V99

DISPLAY SIGN LEADING SEPARATE.

01 MY-DIM PIC S9(9) COMP.

...

EXEC SQL ARRAYLEN BONUS (MY-DIM) END-EXEC.

EXEC SQL END DECLARE SECTION END-EXEC.

* -- populate the host table
```



```
* -- set smaller table dimension
    MOVE 25 TO MY-DIM.
    EXEC SQL EXECUTE
    DECLARE
       TYPE NumTabTyp IS TABLE OF REAL
           INDEX BY BINARY INTEGER;
       median bonus REAL;
       FUNCTION median (num_tab NumTabTyp, n INTEGER)
            RETURN REAL IS
       BEGIN
* -- compute median
       END;
    BEGIN
       median bonus := median(:BONUS, :MY-DIM);
    END;
     END-EXEC.
```

Only 25 table elements are passed to the PL/SQL block because ARRAYLEN reduces the host table from 100 to 25 elements. As a result, when the PL/SQL block is sent to the server for execution, a much smaller host table is sent along. This saves time and, in a networked environment, reduces network traffic.

#### 6.6.1.1 Optional Keyword EXECUTE to ARRAYLEN Statement

The use of host tables used in a dynamic SQL Method 2 statement (see "Using Method 2") may have two different interpretations based on the presence or absence of the keyword to EXECUTE the ARRAYLEN statement. .

If the EXECUTE keyword is absent:

- The PL/SQL block will be executed multiple times, with the actual number determined by the minimum dimension of ARRAYLEN used.
- The host array cannot be bound to a PL/SQL table.

#### If EXECUTE is present:

- The host table must be bound to an index table.
- The PL/SQL block will be executed once.
- All host variables specified in the EXEC SQL EXECUTE statement must:
  - Be specified in an ARRAYLEN ... EXECUTE statement, or
  - Be a scalar.

The following Pro\*COBOL example demonstrates how host tables can be used to determine how many times a given PL/SQL block is executed. In this case, the PL/SQL block will be execute 3 times resulting in 3 new rows in the *emp* table.

```
01 DYNSTMT PIC X(80) VARYING.
01 EMPNOTAB PIC S9(4) COMPUTATIONAL OCCURS 5 TIMES.
01 ENAMETAB PIC X(10) OCCURS 3 TIMES.

...

MOVE 1111 TO EMPNOTAB(1).

MOVE 2222 TO EMPNOTAB(2).

MOVE 3333 TO EMPNOTAB(3).

MOVE 4444 TO EMPNOTAB(4).
```



```
MOVE 5555 TO EMPNOTAB(5).

MOVE "MICKEY" TO ENAMETAB(1).

MOVE "MINNIE" TO ENAMETAB(2).

MOVE "GOOFY" TO ENAMETAB(3).

MOVE "BEGIN INSERT INTO emp(empno, ename) VALUES :b1, :b2; END;"

TO DYNSTMT-ARR.

MOVE 57 TO DYNSTMT-LEN.

EXEC SQL PREPARE s1 FROM :DYNSTMT END-EXEC.

EXEC SQL EXECUTE s1 USING :EMPNOTAB, :ENAMETAB END-EXEC.
```

#### Given the following PL/SQL procedure:

```
CREATE OR REPLACE PACKAGE pkg AS

TYPE tab IS TABLE OF NUMBER(5) INDEX BY BINARY_INTEGER;

PROCEDURE proc1 (parm1 tab, parm2 NUMBER, parm3 tab);

END;
```

The following Pro\*COBOL example demonstrates how to bind a host table to a PL/SQL index table through dynamic method 2. Note the presence of the ARRAYLEN...EXECUTE statement for all host arrays specified in the EXEC SQL EXECUTE statement.

```
01 DYNSTMT PIC X(80) VARYING.
01 II
            PIC S9(4) COMP VALUE 2.
01 INTTAB
           PIC S9(9) COMP OCCURS 3 TIMES.
01 DIM PIC S9(9) COMP VALUE 3.
         EXEC SQL ARRAYLEN INTTAB (DIM) EXECUTE END-EXEC.
         MOVE 1 TO INTTAB(1).
         MOVE 2 TO INTTAB(2).
         MOVE 3 TO INTTAB(3).
         MOVE "BEGIN pkg.proc1 (:v1, :v2, :v3); end;";
          TO DYNSTMT-ARR.
         MOVE 37 TO DYNSTMT-LEN.
         EXEC SQL PREPARE s1 FROM : DYNSTMT END-EXEC.
         EXEC SQL EXECUTE s1 USING :INTTAB, :II, :INTTAB END-EXEC.
      . . .
```

However, the following Pro\*COBOL example will result in a precompile-time error because there is no ARRAYLEN...EXECUTE statement for INTTAB2.

```
01 DYNSTMT PIC X(80) VARYING.
01 INTTAB PIC S9(9) COMP OCCURS 3 TIMES.
01 INTTAB2 PIC S9(9) COMP OCCURS 3 TIMES.
01 DIM PIC S9(9) COMP VALUE 3.

EXEC SQL ARRAYLEN INTTAB (DIM) EXECUTE END-EXEC.
...
MOVE 1 TO INTTAB(1).
```



```
MOVE 2 TO INTTAB(2).

MOVE 3 TO INTTAB(3).

MOVE "BEGIN pkg.proc1 (:v1, :v2, :v3); end;";

TO DYNSTMT-ARR.

MOVE 37 TO DYNSTMT-LEN.

EXEC SQL PREPARE s1 FROM :DYNSTMT END-EXEC.

EXEC SQL EXECUTE s1 USING :INTTAB, :INTTAB2, :INTTAB END-EXEC.
```

# 6.7 Cursor Usage in Embedded PL/SQL

The maximum number of cursors your program can simultaneously use is determined by the database initialization parameter OPEN\_CURSORS. Normally, to prevent OPEN\_CURSORS being exceeded, the precompiler allows management of statement cursors. The precompiler options HOLD\_CURSOR, RELEASE\_CURSOR and MAXOPENCURSORS are used. While executing an embedded PL/SQL block there will be one cursor, the parent cursor, associated with the entire PL/SQL block and a separate child cursor for each statement executed during the execution of the PL/SQL block. Because the PL/SQL block is passed to the server for execution, only the parent cursor can be tracked by the precompiler runtime library. Thus, it is possible for applications that use a lot of cursors in this way to exceed OPEN\_CURSORS. Figure 6-1 shows how to calculate the maximum number of cursors used.

Figure 6-1 Maximum Cursors in Use

SQL statement cursors
PL/SQL parent cursors
PL/SQL child cursors
+ 6 cursors for overhead

Sum of cursors in use

Must not exceed OPEN\_CURSORS

Developers should be aware of this situation and plan for this in the setting of OPEN CURSORS and MAXOPENCURSORS.

If there are problems with this, you may wish to free all child cursors after a SQL statement is executed.

This can be achieved by setting RELEASE\_CURSOR=YES and HOLD\_CURSOR=NO. Because the use of the first setting for the entire program is likely to have an impact on performance, you can set these options in line as follows:

```
EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.

* -- first embedded PL/SQL block
    EXEC ORACLE OPTION (RELEASE_CURSOR=NO)END-EXEC.

* -- embedded SQL statements
    EXEC ORACLE OPTION (RELEASE_CURSOR=YES)END-EXEC.

* -- second embedded PL/SQL block
    EXEC ORACLE OPTION (RELEASE_CURSOR=NO)END-EXEC.

* -- embedded SQL statements
```



#### **Related Topics**

• Embedded PL/SQL Considerations

# 6.8 Stored PL/SQL and Java Subprograms

Unlike anonymous blocks, PL/SQL subprograms (procedures and functions) and Java methods can be compiled separately, stored in the database, and invoked.

A subprogram explicitly created using an Oracle tool such as SQL\*Plus is called a *stored* subprogram. Once compiled and stored in the data dictionary, it is a database object can be re-executed without being re-compiled.

When a subprogram within a PL/SQL block or stored subprogram is sent to the database by your application, it is called an **inline** subprogram and is compiled by the database. Pro\*COBOL sends the statement to the server for execution.

Subprograms defined within a package are considered part of the package, and so are called **packaged** subprograms. Stored subprograms not defined within a package are called **standalone** subprograms.

## 6.8.1 Creating Stored Subprograms

You can embed the SQL statements CREATE FUNCTION, CREATE PROCEDURE, and CREATE PACKAGE in a COBOL program, as the following example shows:

Notice that the embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement is a hybrid. Like all other embedded CREATE statements, it begins with the keywords EXEC SQL (not EXEC SQL EXECUTE).

If an embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement fails, Oracle generates a warning, not an error.



Oracle Database SQL Language Reference. for the full syntax of the CREATE statement.



## 6.8.2 Calling a Stored PL/SQL or Java Subprogram

To call a stored subprogram from your host program, you can use either an anonymous PL/SQL block or the CALL embedded SQL statement.

### 6.8.2.1 Anonymous PL/SQL Block

The following example calls a standalone procedure named raise\_salary:

```
EXEC SQL EXECUTE
BEGIN
    raise_salary(:emp_id, :increase);
END;
END-EXEC.
```

Notice that stored subprograms can take parameters. In this example, the actual parameters *emp\_id* and *increase* are host variables.

In the next example, the procedure *raise\_salary* is stored in a package named *emp\_actions*, so you must use dot notation to fully qualify the procedure call:

```
EXEC SQL EXECUTE
BEGIN
    emp_actions.raise_salary(:emp_id, :increase);
END;
END-EXEC.
```

An actual IN parameter can be a literal, host variable, host table, PL/SQL constant or variable, PL/SQL table, PL/SQL user-defined record, subprogram call, or expression. However, an actual OUT parameter cannot be a literal, subprogram call, or expression.

You must use precompiler option SQLCHECK=SEMANTICS with an embedded PL/SQL block.

### 6.8.2.2 CALL Statement

The concepts presented earlier for the embedded PL/SQL block holds true for the CALL statement. The CALL embedded SQL statement has the form:

```
EXEC SQL
   CALL [schema.][package.]stored_proc[@db_link](arg1, ...)
   [INTO :ret_var[[INDICATOR]:ret_ind]]
END-EXEC.
```

#### where:

schema

the schema containing the procedure

package

the package containing the procedure

stored proc

is the Java or PL/SQL stored procedure to be called



```
db link
```

is the optional remote database link

```
arg1...
```

is the list of arguments (variables, literals, or expressions) passed,

```
ret_var
```

is the optional host variable which receives the result

```
ind_var
```

the optional indicator variable for ret\_var.

You can use either SQLCHECK=SYNTAX, or SQLCHECK=SEMANTICS with the CALL statement.

### 6.8.2.3 CALL Example

If you have created a PL/SQL function fact (stored in the package mathpkg) that takes an integer as input and returns its factorial in an integer:

```
EXEC SQL CREATE OR REPLACE PACKAGE BODY mathpkg as
  function fact(n IN INTEGER) RETURN INTEGER AS
   BEGIN
    IF (n <= 0) then return 1;
    ELSE return n * fact(n - 1);
    END IF;
   END fact;
END mathpkge;
END-EXEC.</pre>
```

then to use fact in a Pro\*COBOL application:

```
...

01 N PIC S9(4) COMP.

01 FACT PIC S9(9) COMP.

...

EXEC SQL CALL mathpkge.fact(:N) INTO :FACT END-EXEC.
...
```

#### See Also:

- "CALL (Executable Embedded SQL)" for more information about the CALL statement.
- Developing Applications with Multiple Programming Languages for a complete explanation of passing arguments and other issues.

## 6.8.3 Using Dynamic PL/SQL

Recall that Pro\*COBOL treats an entire PL/SQL block like a single SQL statement. Therefore, you can store a PL/SQL block in a string host variable. Then, if the block

contains no host variables, you can use dynamic SQL Method 1 to execute the PL/SQL string. Or, if the block contains a known number of host variables, you can use dynamic SQL Method 2 to prepare and execute the PL/SQL string. If the block contains an unknown number of host variables, you must use dynamic SQL Method 4.

#### **Related Topics**

- Oracle Dynamic SQL
- ANSI Dynamic SQL
- Oracle Dynamic SQL: Method 4

### 6.8.4 Subprograms Restriction

In dynamic SQL Method 4, a host table cannot be bound to a PL/SQL procedure with a parameter of type TABLE.

# 6.9 Sample Program 9: Calling a Stored Procedure

Before trying the sample program, you must create a PL/SQL package named *calldemo*, by running the following script, titled CALLDEMO.SQL, which is supplied with Pro\*COBOL. The script can be found in the Pro\*COBOL demo library. Check your system-specific Oracle documentation for exact spelling of the the name of the script.

```
CREATE OR REPLACE PACKAGE calldemo AS
  TYPE name array IS TABLE OF emp.ename%type
      INDEX BY BINARY INTEGER;
  TYPE job array IS TABLE OF emp.job%type
      INDEX BY BINARY INTEGER;
  TYPE sal array IS TABLE OF emp.sal%type
      INDEX BY BINARY INTEGER;
  PROCEDURE get employees(
    batch size IN INTEGER, -- rows at a time
    found IN OUT INTEGER, -- rows actually returned
    done fetch OUT INTEGER, -- all done flag
    emp_name OUT name array,
    job OUT job_array, sal OUT sal_array);
END calldemo;
CREATE OR REPLACE PACKAGE BODY calldemo AS
  CURSOR get emp (dept number IN number) IS
      SELECT ename, job, sal FROM emp
          WHERE deptno = dept number;
  -- Procedure "get employees" fetches a batch of employee
  -- rows (batch size is determined by the client/caller
  -- of the procedure). It can be called from other
  -- stored procedures or client application programs.
  -- The procedure opens the cursor if it is not
  -- already open, fetches a batch of rows, and
```



```
-- returns the number of rows actually retrieved. At
  -- end of fetch, the procedure closes the cursor.
  PROCEDURE get employees (
    dept number IN number,
    batch size IN
                      INTEGER,
    found
           IN OUT INTEGER,
    done_fetch OUT INTEGER,
    emp name OUT name array,
         OUT job_array,
    job
               OUT sal array) IS
    sal
  BEGIN
                                     -- open the cursor if
      IF NOT get emp%ISOPEN THEN
         OPEN get_emp(dept_number); -- not already open
      END IF;
      -- Fetch up to "batch size" rows into PL/SQL table,
      -- tallying rows found as they are retrieved. When all
      -- rows have been fetched, close the cursor and exit
      -- the loop, returning only the last set of rows found.
      done fetch := 0; -- set the done flag FALSE
      found := 0;
      FOR i IN 1..batch size LOOP
          FETCH get emp INTO emp_name(i), job(i), sal(i);
          IF get emp%NOTFOUND THEN -- if no row was found
              CLOSE get_emp;
              done fetch := 1; -- indicate all done
              EXIT;
          FLSE
              found := found + 1; -- count row
          END IF;
      END LOOP;
  END;
END;
```

The following sample program connects to the database, prompts the user for a department number and then calls a PL/SQL procedure named <code>get\_employees</code>, which is stored in package <code>calldemo</code>. The procedure declares three PL/SQL tables as OUT formal parameters and then fetches a batch of employee data into the PL/SQL tables. The matching actual parameters are host tables. When the procedure finishes, row values in the PL/SQL tables are automatically assigned to the corresponding elements in the host tables. The program calls the procedure repeatedly, displaying each batch of employee data, until no more data is found.

```
* is found.
* Use option picx=varchar2 when precompiling this sample program.
IDENTIFICATION DIVISION.
PROGRAM-ID. CALL-STORED-PROC.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME
                   PIC X(15) VARYING.
01 PASSWD
                    PIC X(15) VARYING.
 01 DEPT-NUM
                    PIC S9(9) COMP.
 01 EMP-TABLES.
    05 EMP-NAME
                    OCCURS 10 TIMES PIC X(10).
    05 JOB-TITLE OCCURS 10 TIMES PIC X(10).
    05 SALARY
                    OCCURS 10 TIMES COMP-2.
01 DONE-FLAG
                   PIC S9(9) COMP.
 01 TABLE-SIZE
                   PIC S9(9) COMP VALUE 10.
 01 NUM-RET
                    PIC S9(9) COMP.
01 SQLCODE
                    PIC S9(9) COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.
01 COUNTER
                    PIC S9(9) COMP.
 01 DISPLAY-VARIABLES.
    05 D-EMP-NAME PIC X(10).
    05 D-JOB-TITLE PIC X(10).
    05 D-SALARY PIC Z(5)9.
    05 D-DEPT-NUM PIC 9(2).
    EXEC SQL INCLUDE SQLCA END-EXEC.
PROCEDURE DIVISION.
BEGIN-PGM.
    EXEC SOL WHENEVER SOLERROR DO
       PERFORM SQL-ERROR END-EXEC.
    PERFORM LOGON.
    PERFORM INIT-TABLES VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER > 10.
    PERFORM GET-DEPT-NUM.
    PERFORM DISPLAY-HEADER.
    MOVE ZERO TO DONE-FLAG.
    MOVE ZERO TO NUM-RET.
    PERFORM FETCH-BATCH UNTIL DONE-FLAG = 1.
    PERFORM LOGOFF.
 INIT-TABLES.
    MOVE SPACE TO EMP-NAME (COUNTER).
    MOVE SPACE TO JOB-TITLE (COUNTER) .
    MOVE ZERO TO SALARY (COUNTER) .
GET-DEPT-NUM.
    MOVE ZERO TO DEPT-NUM.
    DISPLAY " ".
```

```
DISPLAY "ENTER DEPARTMENT NUMBER: "
       WITH NO ADVANCING.
   ACCEPT D-DEPT-NUM.
   MOVE D-DEPT-NUM TO DEPT-NUM.
DISPLAY-HEADER.
   DISPLAY " ".
    DISPLAY "EMPLOYEE JOB TITLE SALARY".
    DISPLAY "----".
FETCH-BATCH.
   EXEC SQL EXECUTE
       BEGIN
           CALLDEMO.GET EMPLOYEES
               (:DEPT-NUM, :TABLE-SIZE,
                :NUM-RET, :DONE-FLAG,
                :EMP-NAME, :JOB-TITLE, :SALARY);
        END;
   END-EXEC.
    PERFORM PRINT-ROWS VARYING COUNTER FROM 1 BY 1
       UNTIL COUNTER > NUM-RET.
PRINT-ROWS.
   MOVE EMP-NAME (COUNTER) TO D-EMP-NAME.
   MOVE JOB-TITLE (COUNTER) TO D-JOB-TITLE.
   MOVE SALARY (COUNTER) TO D-SALARY.
   DISPLAY D-EMP-NAME, " ",
           D-JOB-TITLE, " ",
           D-SALARY.
LOGON.
   MOVE "SCOTT" TO USERNAME-ARR.
   MOVE 5 TO USERNAME-LEN.
   MOVE "TIGER" TO PASSWD-ARR.
   MOVE 5 TO PASSWD-LEN.
   EXEC SOL
       CONNECT : USERNAME IDENTIFIED BY : PASSWD
   END-EXEC.
   DISPLAY " ".
   DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
LOGOFF.
   DISPLAY " ".
   DISPLAY "HAVE A GOOD DAY.".
   DISPLAY " ".
   EXEC SQL COMMIT WORK RELEASE END-EXEC.
   STOP RUN.
SOL-ERROR.
   EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    DISPLAY " ".
   DISPLAY "ORACLE ERROR DETECTED:".
   DISPLAY " ".
   DISPLAY SQLERRMC.
   EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
```

Remember that the datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Further, before a stored subprogram

exits, all OUT formal parameters must be assigned values. Otherwise, the values of corresponding actual parameters are indeterminate.

#### 6.9.1 Remote Access

PL/SQL lets you access remote databases through *database links*. Typically, database links are established by your DBA and stored in the data dictionary. A database link tells your program where the remote database is located, the path to it, and what username and password to use. The following example uses the database link *dallas* to call the *raise\_salary* procedure:

```
EXEC SQL EXECUTE
   BEGIN
   raise_salary@dallas(:emp_id, :increase);
   END;
END-EXEC.
```

You can create synonyms to provide location transparency for remote subprograms, as the following example shows:

```
CREATE PUBLIC SYNONYM raise salary FOR raise salary@dallas;
```

### 6.10 Cursor Variables

You can use *cursor variables* in your Pro\*COBOL programs to process multi-row queries using static embedded SQL. A cursor variable identifies a *cursor reference* that is defined and opened on the database server, using PL/SQL.

Like a cursor, a cursor variable points to the current row in the active set of a multi-row query. Cursors differ from cursor variables the way constants differ from variables. While a cursor is static, a cursor variable is dynamic, because it is not tied to a specific query. You can open a cursor variable for any type-compatible query.

You can assign new values to a cursor variable and pass it as a parameter to subprograms, including subprograms stored in a database. This gives you a convenient way to centralize data retrieval.

First, you declare the cursor variable. After declaring the variable, you use these statements to control a cursor variable:

- ALLOCATE
- OPEN ... FOR
- FETCH
- CLOSE
- FREE

After you declare the cursor variable and allocate memory for it, you must pass it as an input host variable (bind variable) to PL/SQL, OPEN it FOR a multi-row query on the server side, FETCH from it on the client side and then CLOSE it on either side.

The advantages of cursor variables are

• Ease of maintenance. Queries are centralized, in the stored procedure that opens the cursor variable. If you need to change the cursor, you only need to make the change in one place: the stored procedure. There is no need to change each application.



 Increased Security. The user of the application (the username when the Pro\*COBOL application connected to the database) must have execute permission on the stored procedure that opens the cursor. This user, however, does not need to have read permission on the tables used in the query. This capability can be used to limit access to the columns in the table.



Cursor Variables for complete information about cursor variables.

### 6.10.1 Declaring a Cursor Variable

You declare a Pro\*COBOL cursor variable using the SQL-CURSOR pseudotype. For example:

```
WORKING-STORAGE SECTION.
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 CUR-VAR SQL-CURSOR.
...
EXEC SQL END DECLARE SECTION END-EXEC.
```

A SQL-CURSOR variable is implemented as a COBOL group item in the code that Pro\*COBOL generates. A cursor variable is just like any other Pro\*COBOL host variable.

### 6.10.2 Allocating a Cursor Variable

Before you can OPEN or FETCH from a cursor variable, you must initialize it using the Pro\*COBOL ALLOCATE command. For example, to initialize the cursor variable CURVAR that was declared in the previous section, write the following statement:

```
EXEC SOL ALLOCATE : CUR-VAR END-EXEC.
```

Allocating a cursor variable does *not* require a call to the server, either at precompile time or at runtime.

The AT clause cannot be used in an ALLOCATE statement.

**Caution:** Allocating a cursor variable *does* cause heap memory to be used. For this reason, avoid allocating a cursor variable in a program loop.

## 6.10.3 Opening a Cursor Variable

You must use an embedded anonymous PL/SQL block to open a cursor variable on the database server. The anonymous PL/SQL block may open the cursor either indirectly by calling a PL/SQL stored procedure that opens the cursor (and defines it in the same statement) or directly from the Pro\*COBOL program.

### 6.10.3.1 Opening Indirectly through a Stored PL/SQL Procedure

Consider the following PL/SQL package stored in the database:



```
CREATE PACKAGE demo cur pkg AS
   TYPE EmpName IS RECORD (name VARCHAR2(10));
   TYPE cur type IS REF CURSOR RETURN EmpName;
    PROCEDURE open_emp_cur (
              curs IN OUT curtype,
              dept num IN
                           number);
END;
CREATE PACKAGE BODY demo cur pkg AS
   CREATE PROCEDURE open emp cur (
             curs IN OUT curtype,
              dept_num IN number) IS
    BEGIN
       OPEN curs FOR
           SELECT ename FROM emp
               WHERE deptno = dept num
               ORDER BY ename ASC;
    END;
END;
```

After this package has been stored, you can open the cursor *curs* by first calling the *open\_emp\_cur* stored procedure from your Pro\*COBOL program and then issuing a FETCH from the cursor variable emp\_cursor in the program. For example:

```
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 emp_cursor sql-cursor.
01 DEPT-NUM PIC S9(4).
01 EMP-NAME PIC X(10) VARYING.
    EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    Allocate the cursor variable.
    EXEC SQL
        ALLOCATE :emp-cursor
    END-EXEC.
    MOVE 30 TO dept num.
    Open the cursor on the Oracle Server.
    EXEC SQL EXECUTE
            demo cur pkg.open emp cur(:emp-cursor, :dept-num);
        END;
    END-EXEC.
    EXEC SQL
        WHENEVER NOT FOUND DO PERFORM SIGN-OFF
    END-EXEC.
FETCH-LOOP.
    EXEC SQL
        FETCH :emp_cursor INTO :EMP-NAME
    END-EXEC.
    DISPLAY "Employee Name: ",: EMP-NAME.
    GO TO FETCH-LOOP.
SIGN-OFF.
    . . .
```



### 6.10.3.2 Opening Directly from Your Pro\*COBOL Application

To open a cursor using a PL/SQL anonymous block in a Pro\*COBOL program, define the cursor in the anonymous block. Consider the following example:

```
PROCEDURE DIVISION.

...

EXEC SQL EXECUTE

begin

OPEN :emp_cursor FOR SELECT ename FROM emp

WHERE deptno = :DEPT-NUM;

end;

END-EXEC.
```

## 6.10.4 Fetching from a Cursor Variable

After opening a cursor variable for a multi-row query, you use the FETCH statement to retrieve rows as you would from a static cursor. The syntax follows:

```
EXEC SQL FETCH cursor_variable_name
   INTO {record_name | variable_name[, variable_name, ...]}
END-EXEC.
```

Each column value returned by the cursor variable is assigned to a corresponding field or variable in the INTO clause, providing that their datatypes are compatible.

The FETCH statement must be executed on the client side. The following example fetches rows into a host record named *EMP-REC*:

```
* -- exit loop when done fetching

EXEC SQL

WHENEVER NOT FOUND DO PERFORM NO-MORE

END-EXEC.

PERFORM

* -- fetch row into record

EXEC SQL FETCH :EMP-CUR INTO :EMP-REC END-EXEC

* -- test for transfer out of loop

...

* -- process the data
...

END-PERFORM.
...

NO-MORE.
```

Use the embedded SQL FETCH .... INTO command to retrieve the rows selected when you opened the cursor variable. For example:

```
EXEC SQL
   FETCH :emp_cursor INTO :EMP-INFO:EMP-INFO-IND
END-EXEC.
```

Before you can FETCH from a cursor variable, the variable must be initialized and opened. You cannot FETCH from an unopened cursor variable.

## 6.10.5 Closing a Cursor Variable

Use the embedded SQL CLOSE statement to close a cursor variable. At this point its active set becomes undefined. The syntax follows:

```
EXEC SQL CLOSE cursor variable name END-EXEC.
```

The CLOSE statement can be executed on the client side or the server side. The following example closes the cursor variable *CUR-VAR* when the last row is processed:

```
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

Declare the cursor variable.

01 CUR-VAR SQL-CURSOR.

...

EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.

* Allocate and open the cursor variable, then fetch one or more rows.

...

* Close the cursor variable.

EXEC SQL

CLOSE :CUR-VAR

END-EXEC.
```

### 6.10.6 Freeing a Cursor Variable

To free memory allocated for the cursor variable, *CUR-VAR*, use the FREE statement after the CLOSE:

```
* Free the cursor variable memory.

EXEC SQL

FREE :CUR-VAR

END-EXEC.
```

#### 6.10.7 Restrictions on Cursor Variables

The following restrictions apply to the use of cursor variables:

- Cursor variables are not supported in dynamic SQL.
- You can only use cursor variables with the ALLOCATE, FETCH, FREE, and CLOSE commands. The DECLARE CURSOR command does not apply to cursor variables.
- You cannot use the AT clause with the ALLOCATE command.

## 6.10.8 Sample Program 11: Cursor Variables

The following sample programs—a SQL script (SAMPLE11.sql) and a Pro\*COBOL program (SAMPLE11.pco)—demonstrate how you can use cursor variables in Pro\*COBOL.

## 6.10.8.1 SAMPLE11.SQL

Following is the PL/SQL source code for a creating a package that declares and opens a cursor variable:



```
CONNECT SCOTT/TIGER
CREATE OR REPLACE PACKAGE emp demo pkg AS
   TYPE emp cur type IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open cur (
        cursor IN OUT emp_cur_type,
        dept num IN
                       number);
END emp_demo_pkg;
CREATE OR REPLACE PACKAGE BODY emp demo pkg AS
    PROCEDURE open_cur (
       cursor IN OUT emp cur type,
        dept num IN number) IS
        OPEN cursor FOR SELECT * FROM emp
        WHERE deptno = dept num
       ORDER BY ename ASC;
   END:
END emp demo pkg;
```

#### 6.10.8.2 SAMPLE11.PCO

Following is a Pro\*COBOL sample program, SAMPLE11.PCO, that uses the cursor variable declared in the SAMPLE11.sql example to fetch employee names, salaries, and commissions from the EMP table:

```
****************
* Sample Program 11: Cursor Variable Operations
^{\star} This program logs on to ORACLE, allocates and opens a cursor ^{\star}
^{\star} variable fetches the names, salaries, and commissions of all ^{\star}
* salespeople, displays the results, then closes the cursor.
*************
IDENTIFICATION DIVISION.
PROGRAM-ID. CURSOR-VARIABLES.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(15) VARYING.
01 PASSWD PIC X(15) VARYING.
 01 HOST
                      PIC X(15) VARYING.
 01 EMP-CUR
                      SQL-CURSOR.
 01 EMP-INFO.
    05 EMP-NUM PIC S9(4) COMP.
05 EMP-NAM PIC X(10) VARYING.
05 EMP-JOB PIC X(10) VARYING.
05 EMP-MGR PIC S9(4) COMP.
05 EMP-DAT PIC X(10) VARYING.
05 EMP-SAL PIC S9(6)V99
                         DISPLAY SIGN LEADING SEPARATE.
     05 EMP-COM PIC S9(6)V99
                         DISPLAY SIGN LEADING SEPARATE.
     05 EMP-DEP
                      PIC S9(4) COMP.
 01 EMP-INFO-IND.
     05 EMP-NUM-IND PIC S9(4) COMP.
```

```
05 EMP-NAM-IND PIC S9(4) COMP.
   05 EMP-JOB-IND PIC S9(4) COMP.
   05 EMP-MGR-IND PIC S9(4) COMP.
   05 EMP-DAT-IND PIC S9(4) COMP.
   05 EMP-SAL-IND PIC S9(4) COMP.
   05 EMP-COM-IND PIC S9(4) COMP.
   05 EMP-DEP-IND PIC S9(4) COMP.
   EXEC SQL END DECLARE SECTION END-EXEC.
   EXEC SQL INCLUDE SQLCA END-EXEC.
01 DISPLAY-VARIABLES.
   05 D-DEP-NUM PIC Z(3)9.
   05 D-EMP-NAM
                    PIC X(10).
                  PIC Z(4)9.99.
   05 D-EMP-SAL
   05 D-EMP-COM
                    PIC Z(4)9.99.
   05 D-EMP-DEP
                  PIC 9(2).
PROCEDURE DIVISION.
BEGIN-PGM.
   EXEC SQL
       WHENEVER SQLERROR DO PERFORM SQL-ERROR
   END-EXEC.
   PERFORM LOGON.
   EXEC SQL
       ALLOCATE : EMP-CUR
   END-EXEC.
   DISPLAY "Enter department number (0 to exit): "
       WITH NO ADVANCING.
   ACCEPT D-EMP-DEP.
   MOVE D-EMP-DEP TO EMP-DEP.
   IF EMP-DEP <= 0
       GO TO SIGN-OFF
   END-IF.
   MOVE EMP-DEP TO D-DEP-NUM.
   EXEC SOL EXECUTE
       BEGIN
           emp demo pkg.open cur(:EMP-CUR, :EMP-DEP);
       END:
   END-EXEC.
   DISPLAY " ".
   DISPLAY "For department ", D-DEP-NUM, ":".
   DISPLAY " ".
   DISPLAY "EMPLOYEE SALARY COMMISSION".
   DISPLAY "----".
FETCH-LOOP.
   EXEC SQL
       WHENEVER NOT FOUND GOTO CLOSE-UP
   END-EXEC.
   MOVE SPACES TO EMP-NAM-ARR.
   EXEC SQL FETCH : EMP-CUR
       INTO :EMP-NUM:EMP-NUM-IND,
            :EMP-NAM:EMP-NAM-IND,
            :EMP-JOB:EMP-JOB-IND,
            :EMP-MGR:EMP-MGR-IND,
            :EMP-DAT:EMP-DAT-IND,
            :EMP-SAL:EMP-SAL-IND,
            :EMP-COM:EMP-COM-IND,
```

```
:EMP-DEP:EMP-DEP-IND
   END-EXEC.
   MOVE EMP-SAL TO D-EMP-SAL.
    IF EMP-COM-IND = 0
       MOVE EMP-COM TO D-EMP-COM
       DISPLAY EMP-NAM-ARR, " ", D-EMP-SAL,
              " ", D-EMP-COM
   ELSE
      DISPLAY EMP-NAM-ARR, " ", D-EMP-SAL,
                       N/A"
   END-IF.
   GO TO FETCH-LOOP.
LOGON.
   MOVE "SCOTT" TO USERNAME-ARR.
   MOVE 5 TO USERNAME-LEN.
   MOVE "TIGER" TO PASSWD-ARR.
   MOVE 5 TO PASSWD-LEN.
   MOVE "INST1 ALIAS" TO HOST-ARR.
   MOVE 11 TO HOST-LEN.
   EXEC SQL
      CONNECT : USERNAME IDENTIFIED BY : PASSWD
   END-EXEC.
   DISPLAY " ".
   DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
CLOSE-UP.
   EXEC SOL
       CLOSE : EMP-CUR
   END-EXEC.
   EXEC SQL
      FREE :EMP-CUR
   END-EXEC.
SIGN-OFF.
   DISPLAY " ".
   DISPLAY "HAVE A GOOD DAY.".
   DISPLAY " ".
   EXEC SOL
       COMMIT WORK RELEASE
   END-EXEC.
   STOP RUN.
SQL-ERROR.
   EXEC SQL
      WHENEVER SQLERROR CONTINUE
   END-EXEC.
   DISPLAY " ".
   DISPLAY "ORACLE ERROR DETECTED:".
   DISPLAY " ".
   DISPLAY SQLERRMC.
   EXEC SQL
      ROLLBACK WORK RELEASE
   END-EXEC.
   STOP RUN.
```

7

## **Host Tables**

This chapter looks at using host tables to simplify coding and improve program performance. You learn how to manipulate Oracle data using host tables, how to operate on all the elements of a host table with a single SQL statement, how to limit the number of table elements processed, and how to use tables of group items.

#### The main sections are:

- Host Tables
- Advantages of Host Tables
- Selecting into Tables
- Selecting into Tables
- Inserting with Tables
- Updating with Tables
- · Deleting with Tables
- Using Indicator Tables
- The FOR Clause
- The WHERE Clause
- Mimicking the CURRENT OF Clause
- Tables of Group Items as Host Variables
- Additional Array Insert/Select Syntax
- Using Implicit Buffered Insert

#### 7.1 Host Tables

A host table (also known as an array) is a set of related data items, called *elements*, associated with a single variable. An indicator variable defined as a table is called an *indicator table*. An indicator table can be associated with any host table that is NULLABLE.

# 7.2 Advantages of Host Tables

Host tables can ease programming and can offer greatly improved performance. When writing an application, you are usually faced with the problem of storing and manipulating large amounts of data. Host tables simplify the task of accessing multiple return values.

Host tables let you manipulate multiple rows with a single SQL statement. Thus, communications overhead is reduced markedly, especially in a networked environment. For example, suppose you want to insert information about 300 employees into the EMP table. Without host tables your program must do 300 individual INSERTs—one for each employee. With host tables, only one INSERT need be done.

# 7.3 Tables in Data Manipulation Statements

Pro\*COBOL allows the use of host tables in data manipulation statements. You can use host tables as input variables in the INSERT, UPDATE, and DELETE statements and as output variables in the INTO clause of SELECT and FETCH statements.

The syntax used for host tables and for simple host variables is nearly the same. One difference is the optional FOR clause, which lets you control table processing. Also, there are restrictions on mixing host tables and simple host variables in a SQL statement.

## 7.3.1 Declaring Host Tables

You declare and dimension host tables in the Data Division. In the following example, three host tables are declared, each dimensioned with 50 elements:

```
01 EMP-TABLES.
05 EMP-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.
05 EMP-NAME OCCURS 50 TIMES PIC X(10.
05 SALARY OCCURS 50 TIMES PIC S9(5)V99 COMP-3.
```

You can use the INDEXED BY phrase in the OCCURS clause to specify an index, as the next example shows:

```
01 EMP-TABLES.
05 EMP-NUMBER PIC X(10) OCCURS 50 TIMES
INDEXED BY EMP-INDX.
...
```

The INDEXED BY phrase implicitly declares the index item EMP-INDX.

#### 7.3.1.1 Restrictions

Multi-dimensional host tables are not allowed. Thus, the two-dimensional host table declared in the following example is *invalid*:

```
01 NATION.
05 STATE OCCURS 50 TIMES.
10 STATE-NAME PIC X(25).
10 COUNTY OCCURS 25 TIMES.
15 COUNTY-NAME PIX X(25).
```

Variable-length host tables are not allowed either. For example, the following declaration of EMP-REC is *invalid for a host variable*:

```
01 EMP-FILE.
05 REC-COUNT PIC S9(3) COMP.
05 EMP-REC OCCURS 0 TO 250 TIMES
DEPENDING ON REC-COUNT.
```



The maximum number of host table elements in a SQL statement that is accessible in one fetch is 4GB (or possibly greater, depending on your platform and the available memory). If you try to access a number that exceeds the maximum, you get a "parameter out of range" runtime error. If the statement is an anonymous PL/SQL block, the number of elements accessible is limited to 4GB divided by the size of the datatype.

## 7.3.2 Referencing Host Tables

If you use multiple host tables in a single SQL statement, their dimensions should be the same. This is not a requirement, however, because Pro\*COBOL always uses the *smallest* dimension for the SQL operation. In the following example, only 25 rows are inserted

```
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01 EMP-TABLES.

05 EMP-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.

05 EMP-NAME PIC X(10) OCCURS 50 TIMES.

05 DEPT-NUMBER PIC S9(4) COMP OCCURS 25 TIMES.

EXEC SQL END DECLARE SECTION END-EXEC.

...

PROCEDURE DIVISION.

...

* Populate host tables here.

...

EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)

VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)

END-EXEC.
```

Host tables must *not* be subscripted in SQL statements. For example, the following INSERT statement is *invalid*:

```
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01 EMP-TABLES.

05 EMP-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.

05 EMP-NAME PIC X(10) OCCURS 50 TIMES.

05 DEPT-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.

EXEC SQL END DECLARE SECTION END-EXEC.

...

PROCEDURE DIVISION.

...

PERFORM LOAD-EMP VARYING J FROM 1 BY 1 UNTIL J > 50.

...

LOAD-EMP.

EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)

VALUES (:EMP-NUMBER(J), :EMP-NAME(J),

:DEPT-NUMBER(J))
```

You need not process host tables in a PERFORM VARYING statement. Instead, use the unsubscripted table names in your SQL statement. Pro\*COBOL treats a SQL statement containing host tables of dimension *n* like the same statement executed *n* times with *n* different scalar host variables, but more efficiently.



## 7.3.3 Using Indicator Tables

You can use indicator tables to assign NULLs to elements in input host tables and to detect NULLs or truncated values (of character columns only) in output host tables. The following example shows how to conduct an INSERT with indicator tables:

```
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01 EMP-TABLES.

05 EMP-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.

05 DEPT-NUMBER PIC S9(4) COMP OCCURS 50 TIMES.

05 COMMISSION PIC S9(5)V99 COMP-3 OCCURS 50 TIMES.

05 COMM-IND PIC S9(4) COMP OCCURS 50 TIMES.

EXEC SQL END DECLARE SECTION END-EXEC.

...

PROCEDURE DIVISION.

* Populate the host and indicator tables.

* Set indicator table to all zeros.

...

EXEC SQL INSERT INTO EMP (EMPNO, DEPTNO, COMM)

VALUES (:EMP-NUMBER, :DEPT-NUMBER,

:COMMISSION:COMM-IND)

END-EXEC.
```

The dimension of the indicator table must be greater than or equal to the dimension of the host table.

When using host table SELECT and FETCH, it is recommended that you use indicator variables. That way you can test for NULLs in the associated output host table.

If a NULL is selected or fetched into a host variable that has no associated indicator variable, your program stops processing, sets *sqlca.sqlerrd(3)* to the number of rows processed, and returns an error.

NULL is selected by default, but you can switch it off by using the UNSAFE\_NULL = YES option.

When DBMS=V7 or V8, your program does not consider truncation to be an error.

## 7.3.4 Host Group Item Containing Tables

**Note:** If you have a host group item containing tables, then you must use a corresponding group item of tables for an indicator. For example, if your group item is the following:

```
01 DEPARTURE.

05 HOUR PIC X(2) OCCURS 3 TIMES.

05 MINUTE PIC X(2) OCCURS 3 TIMES.
```

the following indicator variable cannot be used:

```
01 DEPARTURE-IND PIC S9(4) COMP OCCURS 6 TIMES.
```

The indicator variable you use with the group item of tables must itself be a group item of tables such as the following:



```
01 DEPARTURE-IND.
05 HOUR-IND PIC S9(4) COMP OCCURS 3 TIMES.
05 MINUTE-IND PIC S9(4) COMP OCCURS 3 TIMES.
```

#### 7.3.5 Oracle Restrictions

Mixing scalar host variables with host tables in the VALUES, SET, INTO, or WHERE clause is *not* allowed. If any of the host variables is a host table, all must be host tables.

You cannot use host tables with the CURRENT OF clause in an UPDATE or DELETE statement.

## 7.3.6 ANSI Restriction and Requirements

The array interface is an Oracle extension to the ANSI/ISO embedded SQL standard. However, when you precompile with MODE=ANSI, array SELECTs and FETCHes are still allowed. The use of arrays can be flagged using the FIPS flagger precompiler option, if desired.

# 7.4 Selecting into Tables

You can use host tables as output variables in the SELECT statement. If you know the maximum number of rows the select will return, simply define the host tables with that number of elements. In the following example, you select directly into three host tables. The table was defined with 50 rows, with the knowledge that the select will return no more than 50 rows.

```
01 EMP-REC-TABLES.

05 EMP-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.

05 EMP-NAME OCCURS 50 TIMES PIC X(10) VARYING.

05 SALARY OCCURS 50 TIMES PIC S9(6)V99

DISPLAY SIGN LEADING SEPARATE.

...

EXEC SQL SELECT ENAME, EMPNO, SAL

INTO :EMP-NAME, :EMP-NUMBER, :SALARY

FROM EMP

WHERE SAL > 1000

END-EXEC.
```

In this example, the SELECT statement returns up to 50 rows. If there are fewer than 50 eligible rows or you want to retrieve only 50 rows, this method will suffice. However, if there are more than 50 eligible rows, you cannot retrieve all of them this way. If you reexecute the SELECT statement, it just returns the first 50 rows again, even if more are eligible. You must either define a larger table or declare a cursor for use with the FETCH statement.

If a SELECT INTO statement returns more rows than the size of the table you defined, Oracle issues an error message unless you specify SELECT\_ERROR=NO.

#### Related Topics

SELECT ERROR

#### 7.4.1 Batch Fetches

Use batch fetches when the size of data you are processing is large (greater than about 100 rows) as well as when you do not know how many rows will be returned.

If you do not know the maximum number of rows a select will return, you can declare and open a cursor, and then fetch from it in "batches." Batch fetches within a loop let you retrieve a large number of rows with ease. Each fetch returns the next batch of rows from the current active set. In the following example, you fetch in 20-row batches:

```
01 EMP-REC-TABLES.
    05 EMP-NUMBER OCCURS 20 TIMES PIC S9(4) COMP.
    05 EMP-NAME OCCURS 20 TIMES PIC X(10) VARYING.
    05 SALARY OCCURS 20 TIMES PIC S9(6)V99
                    DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL DECLARE EMPCURSOR CURSOR FOR
    SELECT EMPNO, SAL FROM EMP
    END-EXEC.
    EXEC SQL OPEN EMPCURSOR END-EXEC.
    EXEC SOL WHENEVER NOT FOUND DO PERFORM END-IT.
LOOP.
    EXEC SQL FETCH EMPCURSOR INTO : EMP-NUMBER, : SALARY END-EXEC.
* -- process batch of rows
    GO TO LOOP.
END-IT.
. . .
```

Do not forget to check how many rows were actually returned in the last fetch and to process them.

#### **Related Topics**

Sample Program 3: Fetching in Batches

#### 7.4.2 Using SQLERRD(3)

For INSERT, UPDATE, and DELETE statements, SQLERRD(3) records the number of rows processed.

SQLERRD(3) is also useful when an error occurs during a table operation. Processing stops at the row that caused the error, so SQLERRD(3) gives the number of rows processed successfully.

#### 7.4.3 Number of Rows Fetched

Each fetch returns, at most, the number of entries in the table. Fewer rows are returned in the following cases:

- The end of the active set is reached. The "no data found" warning code is returned to SQLCODE in the SQLCA. For example, this happens if you fetch into a table of number of entries 100, but only 20 rows are returned.
- Fewer than a full batch of rows remain to be fetched. For example, this happens if you fetch 70 rows into a table of number of entries 20 because after the third fetch, only 10 rows remain to be fetched.
- An error is detected while processing a row. The fetch fails and the applicable error code is returned to SQLCODE.

The cumulative number of rows returned can be found in the third element of SQLERRD in the SQLCA, called SQLERRD(3) in this guide. This applies to each open cursor. In the following example, notice how the status of each cursor is maintained separately:

```
EXEC SQL OPEN CURSOR1 END-EXEC.

EXEC SQL OPEN CURSOR2 END-EXEC.

EXEC SQL FETCH CURSOR1 INTO :TABLE-OF-20 END-EXEC.

* -- now running total in SQLERRD(3) is 20

EXEC SQL FETCH CURSOR2 INTO :TABLE-OF-30 END-EXEC.

* -- now running total in SQLERRD(3) is 30, not 50

EXEC SQL FETCH CURSOR1 INTO :TABLE-OF-20 END-EXEC.

* -- now running total in SQLERRD(3) is 40 (20 + 20)

EXEC SQL FETCH CURSOR2 INTO :TABLE-OF-30 END-EXEC.

* -- now running total in SQLERRD(3) is 60 (30 + 30)
```

## 7.4.4 Restrictions on Using Host Tables

Using host tables in the WHERE clause of a SELECT statement is allowed only in a subquery. Also, since Pro\*COBOL always takes the smallest dimension of table, do not mix simple host variables with host tables in the INTO clause of a SELECT or FETCH statement because only one row will be retrieved. If any of the host variables is a table, then all must be tables.

Host Tables Valid in SELECT INTO shows which uses of host tables are valid in a SELECT INTO statement.

Table 7-1 Host Tables Valid in SELECT INTO

INTO Clause	WHERE Clause	Valid?
table	table	no
scalar	scalar	yes
table	scalar	yes
scalar	table	no

#### **Related Topics**

The WHERE Clause

#### 7.4.5 Fetching NULLs

When UNSAFE\_NULL=YES, if you select or fetch a NULL into a host table that lacks an indicator table, no error is generated. So, when doing table selects and fetches, Oracle recommends that you use indicator tables. This is because this makes it NULLs easier to find in the associated output host table.

When UNSAFE\_NULL=NO, if you select or fetch a NULL into a host table that lacks an indicator table, Oracle stops processing, sets SQLERRD(3) to the number of rows processed, and issues an error message:



Using Indicator Variables to learn how to find NULLs and truncated values.

## 7.4.6 Fetching Truncated Values

If you select or fetch a truncated column value into a host table that lacks an indicator table, Oracle sets SQLWARN(2).

You can check SQLERRD(3) for the number of rows processed before the truncation occurred. The rows-processed count includes the row that caused the truncation error.

When doing table selects and fetches, you can use indicator tables. That way, if Oracle assigns one or more truncated column values to an output host table, you can find the original lengths of the column values in the associated indicator table.

## 7.4.7 Sample Program 3: Fetching in Batches

The following host table sample program can be found in the demo directory.

```
* Sample Program 3: Host Tables
* This program logs on to ORACLE, declares and opens a cursor,
* fetches in batches using host tables, and prints the results. *
******************
IDENTIFICATION DIVISION.
PROGRAM-ID. HOST-TABLES.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(15) VARYING.
01 PASSWD
                     PIC X(15) VARYING.
 01 EMP-REC-TABLES.
    05 EMP-NUMBER OCCURS 5 TIMES PIC S9(4) COMP.
    05 EMP-NAME OCCURS 5 TIMES PIC X(10) VARYING.
05 SALARY OCCURS 5 TIMES PIC S9(6)V99
                     DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
01 NUM-RET PIC S9(9) COMP VALUE ZERO.
01 PRINT-NUM PIC S9(9) COMP VALUE ZERO.
01 COUNTER PIC S9(9) COMP.
 01 DISPLAY-VARIABLES.
     05 D-EMP-NAME PIC X(10).
     05 D-EMP-NUMBER PIC 9(4).
     05 D-SALARY PIC Z(4)9.99.
 PROCEDURE DIVISION.
BEGIN-PGM.
        WHENEVER SQLERROR DO PERFORM SQL-ERROR
    END-EXEC.
    PERFORM LOGON.
     EXEC SQL
         DECLARE C1 CURSOR FOR
         SELECT EMPNO, SAL, ENAME
```



```
FROM EMP
   END-EXEC.
    EXEC SQL
       OPEN C1
    END-EXEC.
FETCH-LOOP.
   EXEC SOL
      WHENEVER NOT FOUND DO PERFORM SIGN-OFF
   END-EXEC.
   EXEC SQL
       FETCH C1
       INTO : EMP-NUMBER, : SALARY, : EMP-NAME
   END-EXEC.
    SUBTRACT NUM-RET FROM SOLERRD(3) GIVING PRINT-NUM.
   PERFORM PRINT-IT.
   MOVE SQLERRD(3) TO NUM-RET.
   GO TO FETCH-LOOP.
LOGON.
   MOVE "SCOTT" TO USERNAME-ARR.
   MOVE 5 TO USERNAME-LEN.
   MOVE "TIGER" TO PASSWD-ARR.
   MOVE 5 TO PASSWD-LEN.
   EXEC SQL
      CONNECT : USERNAME IDENTIFIED BY : PASSWD
   END-EXEC.
   DISPLAY " ".
   DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
PRINT-IT.
   DISPLAY " ".
   DISPLAY "EMPLOYEE NUMBER SALARY EMPLOYEE NAME".
   DISPLAY "-----"
    PERFORM PRINT-ROWS
       VARYING COUNTER FROM 1 BY 1
       UNTIL COUNTER > PRINT-NUM.
PRINT-ROWS.
   MOVE EMP-NUMBER (COUNTER) TO D-EMP-NUMBER.
   MOVE SALARY (COUNTER) TO D-SALARY.
   DISPLAY "
                ", D-EMP-NUMBER, " ", D-SALARY, " ",
       EMP-NAME-ARR IN EMP-NAME (COUNTER).
   MOVE SPACES TO EMP-NAME-ARR IN EMP-NAME (COUNTER).
SIGN-OFF.
   SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
   IF (PRINT-NUM > 0) PERFORM PRINT-IT.
   EXEC SQL
      CLOSE C1
   END-EXEC.
   EXEC SQL
      COMMIT WORK RELEASE
   END-EXEC.
    DISPLAY " ".
   DISPLAY "HAVE A GOOD DAY.".
   DISPLAY " ".
    STOP RUN.
SOL-ERROR.
   EXEC SOL
```

```
WHENEVER SQLERROR CONTINUE
END-EXEC.
DISPLAY " ".
DISPLAY "ORACLE ERROR DETECTED:".
DISPLAY " ".
DISPLAY SQLERRMC.
EXEC SQL
ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
```

## 7.5 Inserting with Tables

You can use host tables as input variables in an INSERT statement. Just make sure your program populates the tables with data before executing the INSERT statement. If some elements in the tables are irrelevant, you can use the FOR clause to control the number of rows inserted.

An example of inserting with host tables follows:

```
01 EMP-REC-TABLES.

05 EMP-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.

05 EMP-NAME OCCURS 50 TIMES PIC X(10) VARYING.

05 SALARY OCCURS 50 TIMES PIC S9(6)V99

DISPLAY SIGN LEADING SEPARATE.

* -- populate the host tables

EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)

VALUES (:EMP-NAME, :EMP-NUMBER, :SALARY)

END-EXEC.
```

The number of rows inserted will be available in SQLERRD(3).

Host tables must *not* be subscripted in SQL statements. For example the following INSERT statement is invalid:

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I = TABLE-DIMENSION.

EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)

VALUES (:EMP-NAME(I), :EMP-NUMBER(I), :SALARY(I))

END_EXEC
END_PERFORM.
```

#### Related Topics

The FOR Clause

#### 7.5.1 Restrictions on Host Tables

Mixing simple host variables with host tables in the VALUES clause of an INSERT, UPDATE, or DELETE statement causes only the first element of any host table to be processed because simple host variables are treated as host tables of dimension one and Pro\*COBOL always uses the smallest declared dimension. You receive a warning when this occurs.

# 7.6 Updating with Tables

You can also use host tables as input variables in an UPDATE statement, as the following example shows:

```
01 EMP-REC-TABLES.

05 EMP-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.

05 SALARY OCCURS 50 TIMES PIC S9(6)V99

DISPLAY SIGN LEADING SEPARATE.

* -- populate the host tables
EXEC SQL

UPDATE EMP SET SAL = :SALARY WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

The number of rows updated by issuing this statement is available in SQLERRD(3). This is not necessarily the number of rows in the host table. The number does *not* include rows processed by an update cascade (which causes subsequent updates.)

If some elements in the tables are irrelevant, you can use the FOR clause to limit the number of rows updated.

The last example showed a typical update using a unique key (*EMP-NUMBER*). Each table element qualified just one row for updating. In the following example, each table element qualifies multiple rows:

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

...

05 JOB-TITLE OCCURS 10 TIMES PIC X(10) VARYING.

05 COMMISSION OCCURS 50 TIMES PIC S9(6)V99

DISPLAY SIGN LEADING SEPARATE.

EXEC SQL END DECLARE SECTION END-EXEC.

* -- populate the host tables

EXEC SQL

UPDATE EMP SET COMM = :COMMISSION WHERE JOB = :JOB-TITLE END-EXEC.
```

#### 7.6.1 Restrictions in UPDATE

You cannot use host tables with the CURRENT OF clause in an UPDATE statement. For an alternative, see "Mimicking the CURRENT OF Clause".

Table 7-2 shows which uses of host tables are valid in an UPDATE statement:

Table 7-2 Host Tables Valid in UPDATE

SET Clause	WHERE Clause	Valid?
table	table	yes
scalar	scalar	yes
table	scalar	no
scalar	table	no

## 7.7 Deleting with Tables

You can also use host tables as input variables in a DELETE statement. Doing so is like executing the DELETE statement repeatedly using successive elements of the host table in the WHERE clause. Thus, each execution might delete zero, one, or more rows from the table. An example of deleting with host tables follows:



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

...

05 EMP-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.

EXEC SQL END DECLARE SECTION END-EXEC.

* -- populate the host table

EXEC SQL

DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER

END-EXEC.
```

The cumulative number of rows deleted can be found in SQLERRD(3). That number does *not* include rows processed by a delete cascade.

The last example showed a typical delete using a unique key (*EMP-NUMBER*). Each table element qualified just one row for deletion. In the following example, each table element qualifies multiple rows:

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

...

05 JOB-TITLE OCCURS 10 TIMES PIC X(10) VARYING.

EXEC SQL END DECLARE SECTION END-EXEC.

* -- populate the host table

EXEC SQL

DELETE FROM EMP WHERE JOB = :JOB-TITLE

END-EXEC.
```

#### 7.7.1 Restrictions in DELETE

You cannot use host tables with the CURRENT OF clause in a DELETE statement. For an alternative, see "Mimicking the CURRENT OF Clause".

## 7.8 Using Indicator Tables

You use indicator tables to assign NULLs to input host tables and to detect NULL or truncated values in output host tables. The following example shows how to insert with indicator tables:

```
01 EMP-REC-VARS.

05 EMP-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.

05 DEPT-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.

05 COMMISSION OCCURS 50 TIMES PIC S9(6)V99

DISPLAY SIGN LEADING SEPARATE.

* -- indicator table:

05 COMM-IND OCCURS 50 TIMES PIC S9(4) COMP.

* -- populate the host tables

* -- populate the indicator table; to insert a NULL into

* -- the COMM column, assign -1 to the appropriate element in

* -- the indicator table

EXEC SQL

INSERT INTO EMP (EMPNO, DEPTNO, COMM)

VALUES (:EMP_NUMBER, :DEPT-NUMBER, :COMMISSION:COMM-IND)

END-EXEC.
```

The number of entries of the indicator table cannot be smaller than the number of entries of the host table.

#### 7.9 The FOR Clause

You can use the optional FOR clause to set the number of table elements processed by any of the following SQL statements:

- DELETE
- EXECUTE (See information on Oracle dynamic SQL in Oracle Dynamic SQL: Method 4.
- FFTCH
- INSERT
- OPEN
- UPDATE

The FOR clause is especially useful in UPDATE, INSERT, and DELETE statements. With these statements you might not want to use the entire table. The FOR clause lets you limit the elements used to just the number you need, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   EMP-REC-VARS.
01
    05 EMP-NAME OCCURS 1000 TIMES PIC X(20) VARYING.
    05 SALARY OCCURS 100 TIMES PIC S9(6)V99
                 DISPLAY SIGN LEADING SEPARATE.
01 ROWS-TO-INSERT PIC S9(4) COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.
* -- populate the host tables
    MOVE 25 TO ROWS-TO-INSERT.
 -- set FOR-clause variable
 -- will process only 25 rows
    EXEC SQL FOR : ROWS-TO-INSERT
        INSERT INTO EMP (ENAME, SAL)
        VALUES (:EMP-NAME, :SALARY)
    END-EXEC.
```

The FOR clause must use an integer host variable to count table elements. For example, the following statement is illegal:

```
* -- illegal

EXEC SQL FOR 25

INSERT INTO EMP (ENAME, EMPNO, SAL)

VALUES (:EMP-NAME, :EMP-NUMBER, :SALARY)

END-EXEC.
```

The FOR clause variable specifies the number of table elements to be processed. Make sure the number does not exceed the smallest table dimension. Internally, the value is treated as an unsigned quantity. An attempt to pass a negative value through the use of a signed host variable will result in unpredictable behavior.

#### 7.9.1 Restrictions

Two restrictions keep FOR clause semantics clear: you cannot use the FOR clause in a SELECT statement or with the CURRENT OF clause.

#### 7.9.1.1 In a SELECT Statement

If you use the FOR clause in a SELECT statement, you receive an error message.



The FOR clause is not allowed in SELECT statements because its meaning is unclear. Does it mean "execute this SELECT statement *n* times"? Or, does it mean "execute this SELECT statement once, but return *n* rows"? The problem in the former case is that each execution might return multiple rows. In the latter case, it is better to declare a cursor and use the FOR clause in a FETCH statement, as follows:

```
EXEC SOL FOR :LIMIT FETCH EMPCURSOR INTO ...
```

#### 7.9.1.2 With the CURRENT OF Clause

You can use the CURRENT OF clause in an UPDATE or DELETE statement to refer to the latest row returned by a FETCH statement, as the following example shows:

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
SELECT ENAME, SAL FROM EMP WHERE EMPNO = :EMP-NUMBER
END-EXEC.
...
EXEC SQL OPEN EMPCURSOR END-EXEC.
...
EXEC SQL FETCH emp_cursor INTO :EM-NAME, :SALARY END-EXEC.
...
EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
WHERE CURRENT OF EMPCURSOR
END-EXEC.
```

However, you cannot use the FOR clause with the CURRENT OF clause. The following statements are invalid because the only logical value of *LIMIT* is 1 (you can only update or delete the current row once):

```
EXEC SQL FOR :LIMIT UPDA-CURSOR END-EXEC.
...

EXEC SQL FOR :LIMIT DELETE FROM EMP
WHERE CURRENT OF emp_cursor
END-EXEC.
```

## 7.10 The WHERE Clause

Pro\*COBOL treats a SQL statement containing host tables of number of entries n like the same SQL statement executed n times with n different scalar variables (the individual table elements). The precompiler issues an error message only when such treatment is ambiguous:

For example, assuming the declarations:

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

...

05 MGRP-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.

05 JOB-TITLE OCCURS 50 TIMES PIC X(20) VARYING.

01 I PIC S9(4) COMP.

EXEC SQL END DECLARE SECTION END-EXEC.
```

it would be ambiguous if the statement:

```
EXEC SQL SELECT MGR INTO :MGR-NUMBER FROM EMP WHERE JOB = :JOB-TITLE END-EXEC.
```

were treated like the following statement



```
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 50

SELECT MGR INTO :MGR-NUMBER(I) FROM EMP

WHERE JOB = :JOB_TITLE(I)

END-EXEC

END-PERFORM.
```

because multiple rows might meet the WHERE-clause search condition, but only one output variable is available to receive data. Therefore, an error message is issued.

On the other hand, it would not be ambiguous if the statement

```
EXEC SQL

UPDATE EMP SET MGR = :MGR_NUMBER

WHERE EMPNO IN (SELECT EMPNO FROM EMP WHERE

JOB = :JOB-TITLE)

END-EXEC.
```

were treated like the following statement

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 50

UPDATE EMP SET MGR = :MGR_NUMBER(I)

WHERE EMPNO IN

(SELECT EMPNO FROM EMP WHERE JOB = :JOB-TITLE(I))

END-EXEC

END-PERFORM.
```

because there is a *MGR-NUMBER* in the SET clause for each row matching *JOB-TITLE* in the WHERE clause, even if each *JOB-TITLE* matches multiple rows. All rows matching each *JOB-TITLE* can be SET to the same *MGR-NUMBER*, so no error message is issued.

# 7.11 Mimicking the CURRENT OF Clause

The CURRENT OF clause enables you to do UPDATEs or DELETEs of the most recent row in the cursor. Use of the CURRENT OF clause causes the FOR UPDATE clause to be added to the cursor. Adding this clause has the effect of locking all rows identified by the cursor in exclusive mode. Note that you cannot use CURRENT OF with host tables. Instead, append FOR UPDATE to the definition of the cursor and explicitly select the ROWID column, then use that value to identify the current row during the update or delete. An example follows:

```
05 EMP-NAME OCCURS 25 TIMES PIC X(20) VARYING.
   05 JOB-TITLE OCCURS 25 TIMES PIC X(15) VARYING.
   05 OLD-TITLE OCCURS 25 TIMES PIC X(15) VARYING.
   05 ROW-ID OCCURS 25 TIMES PIC X(18) VARYING.
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
   SELECT ENAME, JOB, ROWID FROM EMP
   FOR UPDATE
END-EXEC.
. . .
EXEC SQL OPEN EMPCURSOR END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO ...
. . .
PERFORM
   EXEC SQL
       FETCH EMPCURSOR
       INTO : EMP-NAME, : JOB-TITLE, : ROW-ID
   END-EXEC
   EXEC SQL
```

```
DELETE FROM EMP

WHERE JOB = :OLD-TITLE AND ROWID = :ROW-ID

END-EXEC

EXEC SQL COMMIT WORK END-EXEC

END-PERFORM.
```

# 7.12 Tables of Group Items as Host Variables

Pro\*COBOL allows the use of tables of group items (also called *records*) in embedded SQL statements. The tables of group items can be referenced in the INTO clause of a SELECT or a FETCH statement, and in the VALUES list of an INSERT statement.

For example, given the following declaration:

```
01 TABLES.

05 EMP-TABLE OCCURS 20 TIMES.

10 EMP-NUMBER PIC S9(4) COMP.

10 EMP-NAME PIC X(10).

10 DEPT-NUMBER PIC S9(4) COMP.
```

the following statement is valid:

```
EXEC SQL INSERT INTO EMP(EMPNO, ENAME, DEPTNO) VALUES(:EMP-TABLE)
END-EXEC.
```

Assuming that the group item has been filled with data already, the statement bulk inserts 20 rows consisting of the employee number, employee name, and department number into the EMP table.

Make sure that the order of the group items corresponds to the order in the SQL statement.

To use an indicator variable, set up a second table of a group item that contains an indicator variable for each variable in the group item:

```
01 TABLES-IND.

05 EMP-TABLE-IND OCCURS 20 TIMES.

10 EMP-NUMBER-IND PIC S9(4) COMP.

10 EMP-NAME-IND PIC S9(4) COMP.

10 DEPT-NUMBER_IND PIC S9(4) COMP.
```

The host indicator table of a group item can be used as follows:

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)

VALUES (:EMP-TABLE:EMP-TABLE-IND)

END-EXEC.
```

If the exact characteristics of the data are known, it is convenient to specify an elementary item indicator for a group item:

```
05 EMP-TABLE-IND PIC S9(4) COMP OCCURS 20 TIMES.
```

Host tables of group items cannot have group items that are tables. For example:

```
01 TABLES.

05 EMP-TABLE OCCURS 20 TIMES.

10 EMP-NUMBER PIC S9(4) COMP OCCURS 10 TIMES.

10 EMP-NAME PIC X(10).

10 DEPT-NUMBER PIC S9(4) COMP.
```



EMP-TABLE cannot be used as a host variable because EMP-NUMBER is a table.

Host tables of nested group items are not allowed. For example:

```
01 TABLES.

05 TEAM-TABLE OCCURS 20 TIMES

10 EMP-TABLE

15 EMP-NUMBER PIC S9(4) COMP.

15 EMP-NAME PIC X(10).

10 DEPT-TABLE.

15 DEPT-NUMBER PIC S9(4) COMP.

15 DEPT-NAME PIC X(10).
```

TEAM-TABLE cannot be used as a host variable because its members (EMP-TABLE and DEPT-TABLE) are group items themselves.

Finally, the restrictions that apply to host tables in Pro\*COBOL also apply to tables of group items:

- Multi-dimensional and variable-length tables are not allowed.
- If multiple tables are used in a single SQL statement, their dimensions should be the same.
- Host tables in SQL statements must not be subscripted.

# 7.13 Sample Program 14: Tables of Group Items

This program logs on, declares and opens a cursor, fetches in batches using a table of group items. Read the initial comments for details.

```
* Sample Program 14: Tables of group items
* This program logs on to ORACLE, declares and opens a cursor,
* fetches in batches using a table of group items , and prints
* the results. This sample is identical to sample3 except that
* instead of using three separate host tables of five elements
* each, it uses a five-element table of three group items.
* The output should be identical.
******************
IDENTIFICATION DIVISION.
PROGRAM-ID. TABLE-OF-GROUP-ITEMS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SOL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME
              PIC X(15) VARYING.
01 PASSWD
                  PIC X(15) VARYING.
01 EMP-REC-TABLE OCCURS 5 TIMES.
    05 EMP-NUMBER PIC S9(4) COMP.
    05 SALARY
                   PIC S9(6)V99
                   DISPLAY SIGN LEADING SEPARATE.
    05 EMP-NAME
                 PIC X(10) VARYING.
    EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
01 NUM-RET PIC S9(9) COMP VALUE ZERO.
01 PRINT-NUM
                  PIC S9(9) COMP VALUE ZERO.
```

```
01 COUNTER
                   PIC S9(9) COMP.
01 DISPLAY-VARIABLES.
   05 D-EMP-NAME PIC X(10).
    05 D-EMP-NUMBER PIC 9(4).
    05 D-SALARY
                   PIC Z(4)9.99.
PROCEDURE DIVISION.
BEGIN-PGM.
   EXEC SQL
      WHENEVER SQLERROR DO PERFORM SQL-ERROR
   END-EXEC.
    PERFORM LOGON.
   EXEC SQL
       DECLARE C1 CURSOR FOR
       SELECT EMPNO, SAL, ENAME
       FROM EMP
   END-EXEC.
   EXEC SQL
       OPEN C1
    END-EXEC.
FETCH-LOOP.
       WHENEVER NOT FOUND DO PERFORM SIGN-OFF
   END-EXEC.
    EXEC SQL
       FETCH C1
       INTO :EMP-REC-TABLE
    END-EXEC.
    SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
   PERFORM PRINT-IT.
   MOVE SQLERRD(3) TO NUM-RET.
   GO TO FETCH-LOOP.
LOGON.
   MOVE "SCOTT" TO USERNAME-ARR.
   MOVE 5 TO USERNAME-LEN.
   MOVE "TIGER" TO PASSWD-ARR.
   MOVE 5 TO PASSWD-LEN.
   EXEC SOL
      CONNECT : USERNAME IDENTIFIED BY : PASSWD
   END-EXEC.
   DISPLAY " ".
   DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
PRINT-IT.
   DISPLAY " ".
   DISPLAY "EMPLOYEE NUMBER SALARY EMPLOYEE NAME".
   DISPLAY "----".
    PERFORM PRINT-ROWS
       VARYING COUNTER FROM 1 BY 1
       UNTIL COUNTER > PRINT-NUM.
PRINT-ROWS.
   MOVE EMP-NUMBER (COUNTER) TO D-EMP-NUMBER.
   MOVE SALARY (COUNTER) TO D-SALARY.
   DISPLAY " ", D-EMP-NUMBER, " ", D-SALARY, " ",
       EMP-NAME-ARR IN EMP-NAME (COUNTER).
   MOVE SPACES TO EMP-NAME-ARR IN EMP-NAME (COUNTER).
```

```
SIGN-OFF.
    SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
    IF (PRINT-NUM > 0) PERFORM PRINT-IT.
    EXEC SQL
        CLOSE C1
    END-EXEC.
    EXEC SQL
        COMMIT WORK RELEASE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    STOP RUN.
SOL-ERROR.
    EXEC SQL
       WHENEVER SQLERROR CONTINUE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL
       ROLLBACK WORK RELEASE
    END-EXEC.
    STOP RUN.
```

# 7.14 Additional Array Insert/Select Syntax

The Oracle precompiler also supports the DB2 insert and fetch syntax for the host tables. The supported additional array insert and fetch syntax are shown in the following images, respectively.

Figure 7-1 Additional Insert Syntax

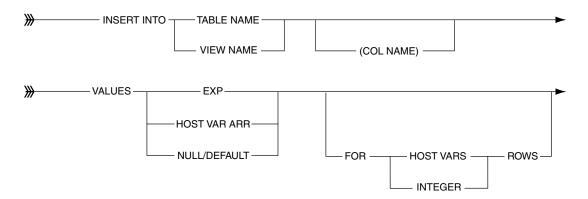
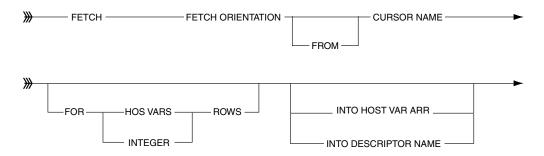


Figure 7-2 Additional Fetch Syntax



The optional ROWSET and ROWSET STARTING AT clauses are used in the fetch-orientation (FIRST, PRIOR, NEXT, LAST, CURRENT, RELATIVE and ABSOLUTE). Consider the following examples:

- FIRST ROWSET
- PRIOR ROWSET
- NEXT ROWSET
- LAST ROWSET
- CURRENT ROWSET
- ROWSET STARTING AT RELATIVE
- ROWSET STARTING AT ABSOLUTEn

Examples of the DB2 array insert/fetch syntax and their comparison with the corresponding Oracle precompiler syntax are shown in Table 7-3:

Table 7-3 DB2 Array Syntax vs. Oracle Precompiler Syntax

DB2 Array Syntax	Oracle Precompiler Syntax
EXEC SQL INSERT INTO DSN8810.ACT (ACTNO, ACTKWD, ACTDESC) VALUES (:HVA1, :HVA2, :HVA3) FOR :NUM_ROWS ROWS END-EXEC.	EXEC SQL FOR :NUM_ROWS INSERT INTO DSN8810.ACT (ACTNO, ACTKWD, ACTDESC) VALUES (:HVA1, :HVA2, :HVA3) END-EXEC.
EXEC SQL  FETCH NEXT ROWSET FROM C1  FOR 20 ROWS  INTO :HVA_EMPNO, :HVA_LASTNAME,  :HVA_SALARY  END-EXEC.	EXEC SQL  FOR :TWENTY  FETCH c1  INTO :HVA_EMPNO, :HVA_LASTNAME,  :HVA_SALARY  END-EXEC.

In DB2 syntax, a row-set positioned cursor should be first declared before retrieving row sets of data. To enable a cursor to fetch row sets, 'WITH ROWSET POSITIONING' clause has to be used in the DECLARE CURSOR statement, which is not required and relevant in the Oracle precompiler syntax, as shown in the following table.



# DB2 Array Syntax Cracle Precompiler Syntax EXEC SQL DECLARE C1 CURSOR WITH ROWSET POSITIONING FOR SELECT EMPNO, LASTNAME, SALARY FROM DSN8810.EMP END-EXEC. END-EXEC.

This additional array syntax support can be enabled with the precompiler option "db2\_array", whose default option is "no". The DB2 array syntax support cannot be used together with the Oracle precompiler syntax; only one of the syntax, either Oracle precompiler, or DB2 syntax, will be supported at a time.

#### Example 7-1 Inserting and Fetching Rows by Using the DB2 Array Syntax

This program inserts INSCNT rows into the EMP table by using the DB2 array insert syntax, and then fetches the inserted rows by using the DB2 array fetch syntax.

```
*****************
* db2arrdemo:
**********************
IDENTIFICATION DIVISION.
PROGRAM-ID. db2arrdemo.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
* EMBEDDED COBOL (file "DB2ARRDEMO.PCO")
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME
                 PIC X(10) VARYING.
01 PASSWD
                   PIC X(10) VARYING.
01 EMPINDATA.
    02 EMPIN OCCURS 25 TIMES.
      03 EMPNO PIC 9(4) COMP.
      03 ENAME PIC X(10).
      03 JOB PIC X(9).
      03 MGR PIC 9(4).
      03 HIREDATE PIC X(9).
      03 SAL PIC X(6).
      03 COMM PIC X(6).
      03 DEPTNO PIC 9(2).
01 EMPOUTDATA.
    02 EMPOUT OCCURS 5 TIMES.
      03 EMPNO1 PIC 9(4) COMP.
      03 ENAME1 PIC X(10).
      03 JOB1 PIC X(9).
      03 MGR1 PIC 9(4).
      03 HIREDATE1 PIC X(9).
      03 SAL1 PIC X(6).
      03 COMM1 PIC X(6).
      03 DEPTNO1 PIC 9(2).
    EXEC SQL END DECLARE SECTION END-EXEC.
```



```
01 INSCNT PIC 9(3) COMP VALUE 25.
    01 FETCHCNT PIC 9(3) COMP VALUE 5.
    01 CNT PIC 9(4).
    01 CNT2 PIC 9(2).
    01 STRINGFIELDS.
       02 STR PIC X(18) VARYING.
        EXEC SQL INCLUDE SQLCA END-EXEC.
    PROCEDURE DIVISION.
    BEGIN-PGM.
        EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR END-EXEC.
        PERFORM LOGON.
   * Fill the array elements to insert.
PERFORM FILL-DATA VARYING CNT FROM 1 BY 1
       UNTIL CNT > INSCNT.
   * Inserting data using DB2 array insert syntax.
DISPLAY "Inserting data using DB2 array insert syntax".
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, JOB, MGR, HIREDATE,
         SAL, COMM, DEPTNO) VALUES (:EMPIN)
FOR : INSCNT ROWS
END-EXEC.
EXEC SQL SELECT COUNT(*) INTO :CNT FROM EMP
        WHERE ENAME LIKE 'EMP %'
END-EXEC.
DISPLAY "Number of rows successfully inserted into EMP "
        "table:", CNT.
DISPLAY " ".
   * Declares scrollable cursor to fetch data.
        EXEC SQL DECLARE C1 SCROLL CURSOR FOR
                 SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL,
 COMM, DEPTNO
 FROM EMP
        WHERE ENAME LIKE 'EMP %'
ORDER BY EMPNO
END-EXEC.
        EXEC SQL OPEN C1 END-EXEC.
DISPLAY "Fetching data using DB2 array fetch syntax ".
     PERFORM FETCH-TAB.
    ENDFETCH-TAB.
EXEC SQL CLOSE C1 END-EXEC.
        EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
        STOP RUN.
    LOGON.
       MOVE "scott" TO USERNAME-ARR.
       MOVE 5 TO USERNAME-LEN.
       MOVE "tiger" TO PASSWD-ARR.
       MOVE 5 TO PASSWD-LEN.
        EXEC SOL
```

```
CONNECT : USERNAME IDENTIFIED BY : PASSWD
        END-EXEC.
   * FILLS ARRAY TO INSERT INTO EMP TABLE
    FILL-DATA.
       MOVE CNT TO EMPNO(CNT).
MOVE " " TO STR.
STRING "EMP_", CNT INTO STR
END-STRING.
MOVE STR TO ENAME (CNT) .
MOVE " " TO STR.
STRING "JOB ", CNT INTO STR
END-STRING.
MOVE STR TO JOB(CNT).
MOVE 100 TO MGR (CNT).
IF CNT > 30 THEN
   COMPUTE CNT2 = 30
   MOVE CNT TO CNT2
END-IF
MOVE " " TO STR.
STRING CNT2, "-JAN-06" INTO STR
END-STRING.
MOVE STR TO HIREDATE (CNT).
MOVE " " TO STR.
STRING CNT2, "000" INTO STR
END-STRING.
MOVE STR TO SAL(CNT).
MOVE 1000 TO COMM(CNT).
MOVE 10 TO DEPTNO(CNT).
   * FETCHES DATA FROM EMP TABLE
    FETCH-TAB.
       EXEC SQL WHENEVER NOT FOUND GOTO ENDFETCH-TAB END-EXEC.
DISPLAY "Fetch using FETCH FIRST ROWSET".
       EXEC SQL FETCH FIRST ROWSET FROM C1 FOR :FETCHCNT ROWS
        INTO : EMPOUT
END-EXEC.
PERFORM PRINTDATA.
DISPLAY " ".
DISPLAY "Fetch using FETCH NEXT ROWSET".
        EXEC SQL FETCH NEXT ROWSET FROM C1 FOR 5 ROWS
        INTO : EMPOUT END-EXEC.
PERFORM PRINTDATA.
DISPLAY " ".
DISPLAY "Fetch using FETCH CURRENT ROWSET".
        EXEC SQL FETCH CURRENT ROWSET FROM C1 FOR :FETCHCNT ROWS
         INTO : EMPOUT
END-EXEC.
PERFORM PRINTDATA.
```

```
DISPLAY " ".
DISPLAY "Fetch using FETCH LAST ROWSET".
       EXEC SQL FETCH LAST ROWSET FROM C1 FOR :FETCHCNT ROWS
        INTO : EMPOUT
END-EXEC.
PERFORM PRINTDATA.
DISPLAY " ".
DISPLAY "Fetch using FETCH ROWSET STARTING AT ABSOLUTE".
COMPUTE CNT = 4 * FETCHCNT.
       EXEC SQL FETCH ROWSET STARTING AT ABSOLUTE :CNT FROM C1
        FOR 5 ROWS INTO : EMPOUT
END-EXEC.
PERFORM PRINTDATA.
DISPLAY " ".
DISPLAY "Fetch using FETCH ROWSET STARTING AT RELATIVE".
       EXEC SQL FETCH ROWSET STARTING AT RELATIVE -3 FROM C1
        FOR :FETCHCNT ROWS INTO :EMPOUT
END-EXEC.
PERFORM PRINTDATA.
DISPLAY " ".
DISPLAY "Fetch using FETCH PRIOR ROWSET ".
       EXEC SQL FETCH PRIOR ROWSET FROM C1 FOR : FETCHCNT ROWS
        INTO : EMPOUT
END-EXEC.
PERFORM PRINTDATA.
   * Prints fetched data
    PRINTDATA
PERFORM VARYING CNT FROM 1 BY 1 UNTIL CNT > FETCHCNT
          DISPLAY "Empno=", EMPNO1(CNT), ", Ename=", ENAME1(CNT),
          ", Job=", JOB1(CNT), ", Mgr=", MGR1(CNT),
                  ", Hiredate=", HIREDATE1(CNT)
          DISPLAY "Sal=", SAL1(CNT), ", Comm=", COMM1(CNT),
          ", Deptno=", DEPTNO1(CNT)
        END-PERFORM.
   * HANDLES SOL ERROR CONDITIONS
    SOL-ERROR.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        DISPLAY " ".
        DISPLAY "ORACLE ERROR DETECTED:".
       DISPLAY " ".
        DISPLAY SQLERRMC.
        EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
```

# 7.15 Using Implicit Buffered Insert

For improved performance, Pro\*Cobol application developers can reference host arrays in their embedded SQL statements. This provides a means to execute an array of SQL statements with a single round-trip to the database. Despite the significant performance improvements afforded by array execution, some developers choose not to use this capability because it is not ANSI standard. For example, an application written to exploit array execution in Oracle cannot be precompiled using IBM's precompiler.

One workaround is to use buffered INSERT statements, which enable you to gain performance benefits while retaining ANSI standard embedded SQL syntax.

The command line option "max\_row\_insert" controls the number of rows to be buffered before executing the INSERT statement. By default it is zero and the feature is disabled. To enable this feature, specify any number greater than zero.

If insert bufering is enabled, precompiler runtime will flag the corresponding cursor and:

- Allocate or re-allocate extra memory to hold bind values (first execute only).
- Copy bind values from program host variables to internal runtime bind structures.
- Increment the rows buffered count.
- Flush the buffered INSERT statements if MAX INSERT ROWS has been buffered.
- If MAX\_INSERT\_ROWS has not been hit, then return after copying the values to the internal bind buffers without flushing.

If you are executing a new embedded SQL statement that results in a flush of the buffered insert statements:

- Flush the buffers.
- Continue with the call that prompted the flush

The application is informed of the error through the standard precompiler error mechanisms such as SQLCODE or SQLSTATE in Pro\*Cobol.

The "implicit\_svpt" option controls whether an implicit savepoint is taken prior to the start of a new batched insert.

- If yes, a savepoint is taken prior to the start of a new batch of rows. If an error occurs on the insert, an implicit "rollback to savepoint" is executed.
- If no, there is no implicit savepoint taken. If an error occurs on the buffered insert, then it is reported back to the application, but no rollback is executed. Errors are reported asynchronously for buffer inserts. Errors for inserted rows are not reported when the INSERT statement is executed in the application.
  - Some errors for inserted rows are reported later, when the first statement other than the INSERT is executed. This may include DELETE, UPDATE, INSERT (into different tables), COMMIT, and ROLLBACK. Any statement that closes the buffered insert statement can report an error. In such cases, the statement that reports the error is not executed. You need to first handle the error and also reexecute the statement on which the buffered insert error is reported. Otherwise, you may rollback the transaction and reexecute it.

For example, consider using a COMMIT statement to close a buffered insert loop. COMMIT can report an error because of a duplicate key from an earlier insert. In this scenario, the commit is not executed. You should first handle the error and then reexecute COMMIT. Otherwise, you can rollback the transaction and reexecute it.

Some errors are reported on the insert itself, and may reflect an error of a previously inserted row. In such cases, no further inserts are executed. You need to handle the errors of the previously inserted row and continue inserting the current insert, which is a long process. Instead, you may rollback and reexecute the transaction.

For example, consider that the limit of internal buffer is 10 rows and the application is inserting 15 rows in a loop. Suppose there is an error on the 8th row. The error is reported when the 11th row insert happens and the insert is no more executed further.



The following are some of the possible errors that you might face during buffered insert:

- ORA-00001: duplicate key in index
- ORA-01400: mandatory (not null) column is missing or Null during insert
- ORA-01401: inserted value too large for column
- ORA-01438: value larger than specified precision allows for this column

#### Example 7-2 inserting Buffered Rows into a Table

This program inserts LOOPCNT number of rows into the EMP table. At loop counter=5, this program attempts to insert an invalid empno. Without the max\_row\_insert option, the program inserts all rows except the invalid row. When the max\_row\_insert option is set to LOOPCNT, only the first four rows are inserted.

Using the max\_row\_insert option, when the erroneous statement is removed, the program performs the same way an array insert program would.

```
***************
    * bufinsdemo:
    ^{\star} This program inserts LOOPCNT number of rows into EMP table.
    * At loop counter=5, this program attempts to insert an invalid *
    * empno. Without max row insert option, this program inserts
    * all rows except this invalid row. When max row insert option
    * is set to LOOPCNT, only the first 4 rows are inserted.
    * With max row insert option, when this errorneous statement
    * is removed, the performance of this program is similar to
    * having an array insert in this program.
    *****************
     IDENTIFICATION DIVISION.
     PROGRAM-ID. bufinsdemo.
     ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.
     DATA DIVISION.
     WORKING-STORAGE SECTION.
    * EMBEDDED COBOL (file "BUFINSDEMO.PCO")
        EXEC SOL BEGIN DECLARE SECTION END-EXEC.
     01 USERNAME PIC X(10) VARYING.
     01 PASSWD
                       PIC X(10) VARYING.
     01 EMPIN.
        02 EMPNO PIC 9(6) COMP.
        02 ENAME PIC X(10).
        02 JOB PIC X(9).
        02 MGR PIC 9(4).
        02 HIREDATE PIC X(9).
        02 SAL PIC X(6).
        02 COMM PIC X(6).
        02 DEPTNO PIC 9(2).
     01 EMPOUT.
        02 EMPNO1 PIC 9(4) COMP.
        02 ENAME1 PIC X(10).
        02 JOB1 PIC X(9).
        02 MGR1 PIC 9(4).
```



```
02 HIREDATE1 PIC X(9).
        02 SAL1 PIC X(6).
        02 COMM1 PIC X(6).
        02 DEPTNO1 PIC 9(2).
       EXEC SQL END DECLARE SECTION END-EXEC.
    01 LOOPCNT PIC 9(4) COMP VALUE 100.
    01 CNT PIC 9(4).
    01 CNT2 PIC 9(2).
    01 STRINGFIELDS.
       02 STR PIC X(18) VARYING.
       EXEC SOL INCLUDE SOLCA END-EXEC.
    PROCEDURE DIVISION.
   BEGIN-PGM.
       EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR END-EXEC.
       PERFORM LOGON.
   * When max row insert option is set to LOOPCNT and when the errorneous
   * statement is removed, all the rows will be inserted into the database
   * in one stretch and hence maximum performance gain will be achieved.
DISPLAY "Inserting ", LOOPCNT, " rows into EMP table".
PERFORM INS-TAB VARYING CNT FROM 1 BY 1
       UNTIL CNT > LOOPCNT.
EXEC SQL COMMIT END-EXEC.
EXEC SQL SELECT COUNT(*) INTO :CNT FROM EMP
        WHERE ENAME LIKE 'EMP %'
END-EXEC.
DISPLAY "Number of rows successfully inserted into EMP "
        "table:", CNT.
DISPLAY " ".
       EXEC SQL DECLARE C1 CURSOR FOR
                SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL,
COMM, DEPTNO
FROM EMP
        WHERE ENAME LIKE 'EMP %'
ORDER BY EMPNO
END-EXEC.
       EXEC SQL OPEN C1 END-EXEC.
DISPLAY "Fetching inserted rows from EMP table".
         PERFORM FETCH-TAB.
   ENDFETCH-TAB.
EXEC SQL CLOSE C1 END-EXEC.
EXEC SQL DELETE FROM EMP WHERE EMPNO < 1000 END-EXEC.
        EXEC SQL COMMIT WORK RELEASE END-EXEC.
        STOP RUN.
   LOGON.
       MOVE "scott" TO USERNAME-ARR.
```

```
MOVE 5 TO USERNAME-LEN.
         MOVE "tiger" TO PASSWD-ARR.
         MOVE 5 TO PASSWD-LEN.
         EXEC SQL
             CONNECT : USERNAME IDENTIFIED BY : PASSWD
         END-EXEC.
     * INSERTS DATA INTO EMP TABLE
     INS-TAB.
 IF CNT = 5 THEN
   MOVE 10000 TO EMPNO
 ELSE
           MOVE CNT TO EMPNO
 END-IF
 MOVE " " TO STR.
 STRING "EMP_", CNT INTO STR
 END-STRING.
 MOVE STR TO ENAME.
 MOVE " " TO STR.
 STRING "JOB_", CNT INTO STR
 END-STRING.
 MOVE STR TO JOB.
 MOVE 100 TO MGR.
 IF CNT > 30 THEN
     COMPUTE CNT2 = 30
     MOVE CNT TO CNT2
 END-IF
 MOVE " " TO STR.
 STRING CNT2, "-JAN-06" INTO STR
 END-STRING.
 MOVE STR TO HIREDATE.
 MOVE " " TO STR.
 STRING CNT2, "000" INTO STR
 END-STRING.
 MOVE STR TO SAL.
 MOVE 1000 TO COMM.
 MOVE 10 TO DEPTNO.
 EXEC SQL INSERT INTO EMP (EMPNO, ENAME, JOB, MGR, HIREDATE,
        SAL, COMM, DEPTNO) VALUES (:EMPIN)
 END-EXEC.
     * FETCHES DATA FROM EMP TABLE
     FETCH-TAB.
         EXEC SQL WHENEVER NOT FOUND GOTO ENDFETCH-TAB END-EXEC.
         EXEC SQL FETCH C1 INTO : EMPOUT END-EXEC.
         DISPLAY "Empno=", EMPNO1, ", Ename=", ENAME1,
", Job=", JOB1, ", Mgr=", MGR1,
                 ", Hiredate=", HIREDATE1.
         DISPLAY "Sal=", SAL1, ", Comm=", COMM1, ", Deptno=", DEPTNO1.
GO TO FETCH-TAB.
```

\* HANDLES SQL ERROR CONDITIONS SQL-ERROR.

DISPLAY "ORACLE ERROR DETECTED:".
DISPLAY SQLERRMC.



8

# **Error Handling and Diagnostics**

An application program must anticipate runtime errors and attempt to recover from them. This chapter provides an in-depth discussion of error reporting and recovery. You learn how to handle warnings and errors using the ANSI status variables SQLCODE and SQLSTATE, or the Oracle SQLCA (SQL Communications Area) structure. You also learn how to use the WHENEVER statement and how to diagnose problems using the Oracle ORACA (Oracle Communications Area) structure.

The following topics are discussed:

- · Why Error Handling is Needed
- · Error Handling Alternatives
- Using the SQL Communications Area
- Using the Oracle Communications Area
- How Errors Map to SQLSTATE Codes

# 8.1 Why Error Handling is Needed

A significant part of every application program must be devoted to error handling. The main benefit of error handling is that it enables your program to continue operating in the presence of errors. Errors arise from design faults, coding mistakes, hardware failures, invalid user input, and many other sources

You cannot anticipate all possible errors, but you can plan to handle certain kinds of errors meaningful to your program. For Pro\*COBOL, error handling means detecting and recovering from SQL statement execution errors. You must trap errors because the precompiler will continue regardless of the errors encountered unless you halt processing.

You can also prepare to handle warnings such as "value truncated" and status changes such as "end of data." It is especially important to check for error and warning conditions after every data manipulation statement because an INSERT, UPDATE, or DELETE statement might fail before processing all eligible rows in a table.

# 8.2 Error Handling Alternatives

Pro\*COBOL supports two general methods of error handling:

- The Oracle-specific method with SQLCA and optional ORACA.
- The SQL standard method with SQLSTATE status variable.

The precompiler MODE option governs compliance with the SQL standard. When MODE={ANSI | ANSI14}, you declare the SQLSTATE status variable as PIC X(5). Additionally, the ANSI SQL-89 SQLCODE status variable is still supported, but it is not recommended for new programs because it has been removed from the SQL standard. When MODE={ORACLE | ANSI13}, you *must* include the SQLCA through an EXEC SQL INCLUDE statement. It is possible to use both methods in one program but usually not necessary.

See Also:

Status Variable Combinations for detailed information on mixing methods.

#### 8.2.1 SQLCA

The SQLCA is a record-like, host-language data structure which includes Oracle warnings, error numbers and error text. Oracle updates the SQLCA after every *executable* SQL or PL/SQL statement. (SQLCA values are undefined after a declarative statement.) By checking return codes stored in the SQLCA, your program can determine the outcome of a SQL statement. This can be done in two ways:

- · Implicit checking with the WHENEVER statement
- Explicit checking of SQLCA variables

When you use the WHENEVER statement to implicitly check the status of your SQL statements, Pro\*COBOL automatically inserts error checking code after each executable statement. Alternatively, you can explicitly write your own code to test the value of the SQLCODE member of the SQLCA structure. Include SQLCA by using the embedded SQL INCLUDE statement:

EXEC SQL INCLUDE SQLCA END-EXEC.

#### 8.2.2 ORACA

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA, which contains cursor statistics, SQL statement text, certain option settings and system statistics. Include ORACA by using the embedded SQL INCLUDE statement:

EXEC SOL INCLUDE ORACA END-EXEC.

The ORACA is optional and can be declared regardless of the MODE setting.

✓ See Also:

Using the Oracle Communications Area for more information about the ORACA status variable.

#### 8.2.3 ANSI SQLSTATE Variable

When MODE=ANSI, you can declare the ANSI SQLSTATE variable inside the Declare Section for implicit or explicit error checking. If the option DECLARE\_SECTION is set to NO, then you can also declare it outside of the Declare Section.



When MODE=ANSI, you can also declare the SQLCODE variable with a picture S9(9) COMP. While it can be used instead of or with the SQLSTATE variable, this is not recommended for new programs. You can also use the SQLCA with the SQLSTATE variable. When MODE=ANSI14, then SQLSTATE is not supported and you must declare either SQLCODE or include SQLCA. You cannot declare both SQLCODE and SQLCA for any setting of mode.

### 8.2.4 Declaring SQLSTATE

This section describes how to declare SQLSTATE. SQLSTATE must be declared as a five-character alphanumeric string as in the following example:

```
Declare the SQLSTATE status variable.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

O1 SQLSTATE PIC X(5).

EXEC SQL END DECLARE SECTION END-EXEC.
```

#### 8.2.4.1 SQLSTATE Values

SQLSTATE status codes consist of a two-character *class code* followed by a three-character *subclass code*. Aside from class code 00 (successful completion), the class code denotes a category of exceptions. Aside from subclass code 000 (not applicable), the subclass code denotes a specific exception within that category. For example, the SQLSTATE value '22012' consists of class code 22 (data exception) and subclass code 012 (division by zero).

Each of the five characters in a SQLSTATE value is a digit (0..9) or an uppercase Latin letter (A..Z). Class codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined conditions (those defined in the SQL standard). All other class codes are reserved for implementation-defined conditions. Within predefined classes, subclass codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined sub-conditions. All other subclass codes are reserved for implementation-defined sub-conditions. Figure 8-1 shows the coding scheme:



Figure 8-1 SQLSTATE Coding Scheme

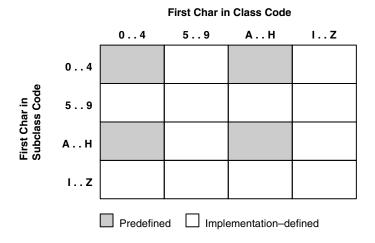


Table 8-1 shows the classes predefined by the SQL standard.

Table 8-1 Predefined Classes

Class	Condition
00	successful completion
01	warning
02	no data
07	dynamic SQL error
08	connection exception
09	triggered action exception
0A	feature not supported
0A	feature not supported
0D	invalid target type specification
0E	invalid schema name list specification
0F	locator exception
0L	invalid grantor
OM	invalid SQL-invoked procedure reference
0P	invalid role specification
0S	invalid transform group name specification
OT	target table disagrees with cursor specification
OU	attempt to assign to non-updatable column
0V	attempt to assign to ordering column
0W	prohibited statement encountered during trigger execution
0Z	diagnostics exception
21	cardinality violation
22	data exception
23	integrity constraint violation



Table 8-1 (Cont.) Predefined Classes

Class	Condition	
24	invalid cursor state	
25	invalid transaction state	
26	invalid SQL statement name	
27	triggered data change violation	
28	invalid authorization specification	
2A	direct SQL syntax error or access rule violation	
2B	dependent privilege descriptors still exist	
2C	invalid character set name	
2D	invalid transaction termination	
2E	invalid connection name	
2F	SQL routine exception	
2H	invalid collation name	
30	invalid SQL statement identifier	
33	invalid SQL descriptor name	
34	invalid cursor name	
35	invalid condition number	
36	cursor sensitivity exception	
37	dynamic SQL syntax error or access rule violation	
38	external routine exception	
39	external routine invocation exception	
3B	savepoint exception	
3C	ambiguous cursor name	
3D	invalid catalog name	
3F	invalid schema name	
40	transaction rollback	
42	syntax error or access rule violation	
44	with check option violation	
HZ	remote database access	



The class code HZ is reserved for conditions defined in International Standard ISO/IEC DIS 9579-2, *Remote Database Access*.

Table 8-4 shows how errors map to SQLSTATE status codes. In some cases, several errors map to the status code. In other cases, no error maps to the status code (so the last column is empty). Status codes in the range 60000..99999 are implementation-defined.



# 8.3 Using the SQL Communications Area

Oracle uses the SQL Communications Area (SQLCA) to store status information passed to your program at run time. The SQLCA is a record-like, COBOL data structure that is a updated after each executable SQL statement, so it always reflects the outcome of the most recent SQL operation. Its fields contain error, warning, and status information updated by Oracle whenever a SQL statement is executed. To determine that outcome, you can check variables in the SQLCA explicitly with your own COBOL code or implicitly with the WHENEVER statement.

When MODE={ORACLE | ANSI13}, the SQLCA is required; if the SQLCA is not declared, compile-time errors will occur. The SQLCA is optional when MODE={ANSI | ANSI14}, but if you want to use the WHENEVER SQLWARNING statement, you must declare the SQLCA. The SQLCA must also be included when using multibyte NCHAR host variables.



When your application uses Oracle Net Services to access a combination of local and remote databases concurrently, all the databases write to one SQLCA. There is *not* a different SQLCA for each database. For more information, see "Concurrent Logons".

# 8.3.1 Contents of the SQLCA

The SQLCA contains runtime information about the execution of SQL statements, such as error codes, warning flags, event information, rows-processed count, and diagnostics.

Figure 8-2 shows all the variables in the SQLCA.

Figure 8-2 SQLCA Variable Declarations for Pro\*COBOL

```
01
   SOLCA.
   05 SQLCAID
                             PIC X(8).
   05 SQLCABC
                             PIC S9(9) COMPUTATIONAL.
   05 SQLCODE
                             PIC S9(9) COMPUTATIONAL.
   05 SQLERRM.
       49 SQLERRML
                            PIC S9(4) COMPUTATIONAL.
       49 SQLERRMC
                            PIC X(70)
                             PIC X(8).
      SOLERRP
   05 SQLERRD OCCURS 6 TIMES
                             PIC S9(9) COMPUTATIONAL.
   05 SQLWARN.
                            PIC X(1).
       10 SOLWARNO
       10 SQLWARN1
                            PIC X(1).
       10 SQLWARN2
                             PIC X(1).
                             PIC X(1).
       10 SOLWARN3
                            PIC X(1).
       10 SQLWARN4
       10 SOLWARNS
                             PIC X(1).
       10 SQLWARN6
                             PIC X(1).
       10 SQLWARN7
                             PIC X(1).
   05 SQLEXT
                             PIC X(8).
```



# 8.3.2 Declaring the SQLCA

To declare the SQLCA, simply include it (using an EXEC SQL INCLUDE statement) in your Pro\*COBOL source file outside the Declare Section as follows:

Include the SQL Communications Area (SQLCA). EXEC SQL INCLUDE SQLCA END-EXEC.

The SOLCA must be declared *outside* the Declare Section.



#### WARNING:

Do not declare SQLCODE if SQLCA is declared. Likewise, do not declare SQLCA if SQLCODE is declared. The status variable declared by the SQLCA structure is also called SQLCODE, so errors will occur if both error-reporting mechanisms are used.

When you precompile your program, the INCLUDE SQLCA statement is replaced by several variable declarations that allow Oracle to communicate with the program.

# 8.3.3 Key Components of Error Reporting

The key components of Pro\*COBOL error reporting depend on several fields in the SQLCA.

### 8.3.3.1 Status Codes

Every executable SQL statement returns a status code in the SQLCA variable SQLCODE, which you can check implicitly with WHENEVER SQLERROR or explicitly with your own COBOL code.

## 8.3.3.2 Warning Flags

Warning flags are returned in the SQLCA variables SQLWARN0 through SQLWARN7, which you can check with WHENEVER SQLWARNING or with your own COBOL code. These warning flags are useful for detecting runtime conditions that are not considered errors.

### 8.3.3.3 Rows-Processed Count

The number of rows processed by the most recently executed SQL statement is returned in the SQLCA variable SQLERRD(3). For repeated FETCHes on an OPEN cursor, SQLERRD(3) keeps a running total of the number of rows fetched.

## 8.3.3.4 Parse Error Offset

Before executing a SQL statement, Oracle must parse it; that is, examine it to make sure it follows syntax rules and refers to valid database objects. If Oracle finds an error, an offset is stored in the SQLCA variable SQLERRD(5), which you can check explicitly. The offset specifies the character position in the SQL statement at which the parse error begins. The first character occupies position zero. For example, if the offset is 9, the parse error begins at the tenth character.



If your SQL statement does not cause a parse error, Oracle sets SQLERRD(5) to zero. Oracle also sets SQLERRD(5) to zero if a parse error begins at the first character (which occupies position zero). So, check SQLERRD(5) only if SQLCODE is negative, which means that an error has occurred.

## 8.3.3.5 Error Message Text

The error code and message for errors are available in the SQLCA variable SQLERRMC. For example, you might place the following statements in an error-handling routine:

\* Handle SQL execution errors. MOVE SQLERRMC TO ERROR-MESSAGE. DISPLAY ERROR-MESSAGE.

At most, the first 70 characters of message text are stored. For messages longer than 70 characters, you must call the SQLGLM subroutine, which is discussed in "Getting the Full Text of Error Messages".

## 8.3.4 SQLCA Structure

This section describes the structure of the SQLCA, its fields, and the values they can store.

## 8.3.4.1 SQLCAID

This string field is initialized to "SQLCA" to identify the SQL Communications Area.

## 8.3.4.2 SQLCABC

This integer field holds the length, in bytes, of the SQLCA structure.

## 8.3.4.3 SQLCODE

This integer field holds the status code of the most recently executed SQL statement. The status code, which indicates the outcome of the SQL operation, can be any of the following numbers:

Status Code	Description	
0	Oracle executed the statement without detecting an error or exception.	
> 0	Oracle executed the statement but detected an exception. This occurs when Oracle cannot find a row that meets your WHERE-clause search condition or when a SELECT INTO or FETCH returns no rows.	



Status Code	Description
< 0	When MODE={ANSI   ANSI14   ANSI113}, +100 is returned to SQLCODE after an INSERT of no rows. This can happen when a subquery returns no rows to process.
	Oracle did not execute the statement because of a database, system, network, or application error. Such errors can be fatal. When they occur, the current transaction should, in most cases, be rolled back.
	Negative return codes correspond to error codes listed in <i>Oracle Database Error Messages</i> .

# 8.3.4.4 SQLERRM

This sub-record contains the following two fields:

Fields	Description	
SQLERRML	This integer field holds the length of the message text stored in SQLERRMC.	
SQLERRMC	This string field holds the message text for the error code stored in SQLCODE and can store up to 70 characters. For the full text of messages longer than 70 characters, use the SQLGLM function.	
	Verify SQLCODE is negative before you reference SQLERRMC. If you reference SQLERRMC when SQLCODE is zero, you get the message text associated with a prior SQL statement.	

### **SQLERRP**

This string field is reserved for future use.

# 8.3.4.5 SQLERRD

This table of binary integers has six elements. Descriptions of the fields in SQLERRD follow:

Fields	Description	
SQLERRD(1)	This field is reserved for future use.	
SQLERRD(2)	This field is reserved for future use.	
SQLERRD(3)	This field holds the number of rows processed by the most recently executed SQL statement. However, if the SQL statement failed, the value of SQLERRD(3) is undefined, with one exception. If the error occurred during a table operation, processing stops at the row that caused the error, so SQLERRD(3) gives the number of rows processed successfully.	
	The rows-processed count is zeroed after an OPEN statement and incremented after a FETCH statement. For the EXECUTE, INSERT, UPDATE, DELETE, and SELECT INTO statements, the count reflects the number of rows processed successfully. The count does <i>not</i> include rows processed by an update or delete cascade. For example, if 20 rows are deleted because they meet WHERE-clause criteria, and 5 more rows are deleted because they now (after the primary delete) violate column constraints, the count is 20 not 25.	



Fields	Description
SQLERRD(4)	This field is reserved for future use.
SQLERRD(5)	This field holds an offset that specifies the character position at which a parse error begins in the most recently executed SQL statement. The first character occupies position zero.
SQLERRD(6)	This field is reserved for future use.

# 8.3.4.6 SQLWARN

This table of single characters has eight elements. They are used as warning flags. Oracle sets a flag by assigning it a 'W' (for warning) character value. The flags warn of exceptional conditions.

For example, a warning flag is set when Oracle assigns a truncated column value to an output host character variable.



Figure 8-2 illustrates SQLWARN implementation in Pro\*COBOL as a group item with elementary PIC X items named SQLWARN0 through SQLWARN7.

Descriptions of the fields in SQLWARN follow:

Fields	Description	
SQLWARN0	This flag is set if another warning flag is set.	
SQLWARN1	This flag is set if a truncated column value was assigned to an output host variable. This applies only to character data. Oracle truncates certain numeric data without setting a warning or returning a negative SQLCODE value.	
	To find out if a column value was truncated and by how much, check the indicator variable associated with the output host variable. The (positive) integer returned by an indicator variable is the original length of the column value. You can increase the length of the host variable accordingly.	
SQLWARN2	This flag is set if one or more NULLs were ignored in the evaluation of a SQL group function such as AVG, COUNT, or MAX. This behavior is expected because, except for COUNT(*), all group functions ignore NULLs. If necessary, you can use the SQL function NVL to temporarily assign values (zeros, for example) to the NULL column entries.	
SQLWARN3	This flag is set if the number of columns in a query select list does not equal the number of host variables in the INTO clause of the SELECT or FETCH statement. The number of items returned is the lesser of the two.	
SQLWARN4	This flag is no longer in use.	
SQLWARN5	This flag is set when an EXEC SQL CREATE {PROCEDURE   FUNCTION   PACKAGE   PACKAGE BODY} statement fails because of a PL/SQL compilation error.	



Fields	Description	
SQLWARN6	This flag is no longer in use.	
SQLWARN7	This flag is no longer in use.	

## 8.3.4.7 SQLEXT

This string field is reserved for future use.

# 8.3.5 PL/SQL Considerations

When your Pro\*COBOL program executes an embedded PL/SQL block, not all fields in the SQLCA are set. For example, if the block fetches several rows, the rows-processed count, SQLERRD(3), is set to 1, *not* the actual number of rows fetched. So, you should rely only on the SQLCODE and SQLERRM fields in the SQLCA after executing a PL/SQL block.

## 8.3.6 Getting the Full Text of Error Messages

Regardless of the setting of MODE, you can use SQLGLM to get the full text of error messages if you have explicitly declared SQLCODE and not included SQLCA. The SQLCA can accommodate error messages up to 70 characters long. To get the full text of longer (or nested) error messages, you need the SQLGLM subroutine.

If connected to a database, you can call SQLGLM using the syntax

```
CALL "SQLGLM" USING MSG-TEXT, MAX-SIZE, MSG-LENGTH
```

where the parameters are:

Parameter	Datatype	Parameter Definition
MSG-TEXT	PIC X(n)	The field in which to store the error message. (Oracle blank-pads to the end of this field.)
MAX-SIZE	PIC S9(9) COMP	An integer that specifies the maximum size of the MSG-TEXT field in bytes.
MSG-LENGTH	PIC S9(9) COMP	An integer variable in which Oracle stores the actual length of the error message.

All parameters must be passed by reference. This is usually the default parameter passing convention; you need not take special action.

The maximum length of an error message is 512 characters including the error code, nested messages, and message inserts such as table and column names. The maximum length of an error message returned by SQLGLM depends on the value specified for MAX-SIZE.

The following example uses SQLGLM to get an error message of up to 200 characters in length:

```
Declare variables for the SQL-ERROR subroutine call.

O1 MSG-TEXT PIC X(200).

O1 MAX-SIZE PIC S9(9) COMP VALUE 200.

O1 MSG-LENGTH PIC S9(9) COMP.

...

PROCEDURE DIVISION.
```



```
MAIN.
```

```
EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC. ... SQL-ERROR.
```

- \* Clear the previous message text.
  MOVE SPACES TO MSG-TEXT.
- \* Get the full text of the error message.

  CALL "SQLGLM" USING MSG-TEXT, MAX-SIZE, MSG-LENGTH.

  DISPLAY MSG-TEXT.

In the example, SQLGLM is called only when a SQL error has occurred. Always make sure SQLCODE is negative *before* calling SQLGLM. If you call SQLGLM when SQLCODE is zero, you get the message text associated with a prior SQL statement.



If your application calls SQLGLM to get message text, the message length must be passed. Do *not* use the SQLCA variable SQLERRML. SQLERRML is a PIC S9(4) COMP integer while SQLGLM and SQLIEM expect a PIC S9(9) COMP integer. Instead, use another variable declared as PIC S9(9) COMP.

## 8.3.7 DSNTIAR

DB2 provides an assembler routine called DSNTIAR to obtain a form of the SQLCA that can be displayed. For users migrating to Oracle from DB2, Pro\*COBOL provides DSNTIAR. The DSNTIAR implementation is a wrapper around SQLGLM. The DSNTIAR interface is as follows

```
CALL 'DSNTIAR' USING SQLCA MESSAGE LRECL
```

where *MESSAGE* is the output message area, in VARCHAR form of size greater than or equal to 240, and *LRECL* is a full word containing the length of the output messages, between 72 and 240. The first half-word of the MESSAGE argument contains the length of the remaining area. The possible error codes returned by DSNTIAR are listed in the following table.

Table 8-2 DSNTIAR Error Codes and Their Meanings

Error Codes	Description
0	Successful execution
4	More data was available than could fit into the provided message
8	The logical record length (LRECL) was not between 72 and 240
12	The message area was not large enough (greater than 240)

## 8.3.8 WHENEVER Directive

By default, Pro\*COBOL ignores error and warning conditions and continues processing, if possible. To do automatic condition checking and error handling, you need the WHENEVER statement.

With the WHENEVER statement you can specify actions to be taken when Oracle detects an error, warning condition, or "not found" condition. These actions include continuing with the next statement, PERFORMing a paragraph, branching to a paragraph, or stopping.

### 8.3.8.1 Conditions

You can have Oracle automatically check the SQLCA for any of the following conditions.

# 8.3.8.2 SQLWARNING

SQLWARN(0) is set because Oracle returned a warning (one of the warning flags, SQLWARN(1) through SQLWARN(7), is also set) or SQLCODE has a positive value other than +1403. For example, SQLWARN(1) is set when Oracle assigns a truncated column value to an output host variable.

Declaring the SQLCA is optional when MODE={ANSI | ANSI14}. To use WHENEVER SQLWARNING, however, you *must* declare the SQLCA.



You have to have included SQLCA for this to work.

## **8.3.8.3 SQLERROR**

SQLCODE has a negative value if Oracle returns an error.

## 8.3.8.4 NOT FOUND or NOTFOUND

SQLCODE has a value of +1403 (or +100 when MODE={ANSI | ANSI14 | ANSI13} or when END\_OF\_FETCH=100) when the end of fetch has been reached. This can happen when all the rows that meet the search criteria have been fetched or no rows meet that criteria.

You may use the END\_OF\_FETCH option to override the value use by the MODE macro option.

```
END OF FETCH = 100 \mid 1403 (default 1403)
```

#### **Related Topics**

END OF FETCH

### 8.3.8.5 Actions

You can use the WHENEVER statement to specify the following actions.

### 8.3.8.6 CONTINUE

Your program continues to run with the next statement if possible. This is the default action, equivalent to not using the WHENEVER statement. You can use it to "turn off" condition checking.



### 8.3.8.7 DO CALL

Your program calls a nested subprogram. When the end of the subprogram is reached, control transfers to the statement that follows the failed SQL statement.

#### 8.3.8.8 DO PERFORM

Your program transfers control to a COBOL section or paragraph. When the end of the section is reached, control transfers to the statement that follows the failed SQL statement.

```
EXEC SQL
    WHENEVER <condition> DO PERFORM <section_name>
END-EXEC.
```

### 8.3.8.9 GOTO or GO TO

Your program branches to the specified paragraph or section.

#### 8.3.8.10 STOP

Your program stops running and uncommitted work is rolled back.

Be careful. The STOP action displays no messages before logging off.



Though in the generated code EXEC SQL WHENEVER SQLERROR STOP is converted to IF SQLCODE IN SQLCA IS EQUAL TO 1403 THEN STOP RUN END-IF, Oracle server will take care of rolling back uncommitted data.

# 8.3.9 Coding the WHENEVER Statement

Code the WHENEVER statement using the following syntax:

```
EXEC SQL
WHENEVER <condition> <action>
END-EYEC
```

### 8.3.9.1 DO PERFORM

When using the WHENEVER ... DO PERFORM statement, the usual rules for PERFORMing a paragraph or section apply. However, you cannot use the THRU, TIMES, UNTIL, or VARYING clauses.

For example, the following WHENEVER ... DO statement is *invalid*:

```
PROCEDURE DIVISION.

* Invalid statement
EXEC SQL WHENEVER SQLERROR DO
PERFORM DISPLAY-ERROR THRU LOG-OFF
END-EXEC.
```



```
DISPLAY-ERROR.
...
LOG-OFF.
...
```

In the following example, WHENEVER SQLERROR DO PERFORM statements are used to handle specific errors:

```
PROCEDURE DIVISION.
MAIN SECTION.
MSTART.
    EXEC SQL
        WHENEVER SQLERROR DO PERFORM INS-ERROR
    END-EXEC.
    EXEC SQL
        INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
        VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
    END-EXEC.
    EXEC SQL
       WHENEVER SQLERROR DO PERFORM DEL-ERROR
    END-EXEC.
    EXEC SQL
        DELETE FROM DEPT
        WHERE DEPTNO = :DEPT-NUMBER
    END-EXEC.
MEXIT.
    STOP RUN.
INS-ERROR SECTION.
INSSTART.
    Check for "duplicate key value" Oracle error
    IF SQLCA.SQLCODE = -1
    Check for "value too large" Oracle error
    ELSE IF SQLCA.SQLCODE = -1401
    . . .
    ELSE
    . . .
    END-IF.
    . . .
INSEXIT.
    EXIT.
DEL-ERROR SECTION.
DSTART.
   Check for the number of rows processed.
    IF SQLCA.SQLERRD(3) = 0
    ELSE
    END-IF.
DEXIT.
    EXIT.
```

Notice how the paragraphs check variables in the SQLCA to determine a course of action.

### 8.3.9.2 DO CALL

This clause calls an action subprogram. Here is the syntax of this clause:

```
EXEC SQL
   WHENEVER <condition> DO CALL <subprogram_name>
   [USING <param1> ...]
END-EXEC.
```

The following restrictions or rules apply:

- You cannot use the RETURNING, ON\_EXCEPTION, or OVER\_FLOW phrases in the USING clause.
- You may have to enter the subprogram name followed by the keyword COMMON in the PROGRAM-ID statement of your COBOL source code.
- You must use a WHENEVER CONTINUE statement in the action subprogram.
- The action subprogram name may have to be in double quotes in the DO CALL clause of the WHENEVER directive.

Here is an example of a program that can call the error subprogram SQL-ERROR from inside the subprogram LOGON, or inside the MAIN program, without having to repeat code in two places, as when using the DO PERFORM clause:

```
IDENTIFICATION DIVISION.
           PROGRAM-ID. MAIN.
           ENVIRONMENT DIVISION.
           PROCEDURE DIVISION.
           BEGIN-PGM.
               EXEC SOL
                  WHENEVER SOLERROR DO CALL "SOL-ERROR"
               END-EXEC.
               CALL "LOGON".
           IDENTIFICATION DIVISION.
           PROGRAM-ID. LOGON.
           DATA DIVISION.
           WORKING-STORAGE SECTION.
           01 USERNAME PIC X(15) VARYING.
           01 PASSWD
                                PIC X(15) VARYING.
           PROCEDURE DIVISION.
               MOVE "SCOTT" TO USERNAME-ARR.
               MOVE 5 TO USERNAME-LEN.
               MOVE "TIGER" TO PASSWD-ARR.
               MOVE 5 TO PASSWD-LEN.
               EXEC SQL
                  CONNECT : USERNAME IDENTIFIED BY : PASSWD
               END-EXEC.
               DISPLAY " ".
               DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.
           END PROGRAM LOGON.
           IDENTIFICATION DIVISION.
           PROGRAM-ID. SOL-ERROR COMMON.
           PROCEDURE DIVISION.
               EXEC SQL
                   WHENEVER SQLERROR CONTINUE
               END-EXEC.
```



```
DISPLAY " ".

DISPLAY SQLERRMC.

EXEC SQL

ROLLBACK WORK RELEASE

END-EXEC.

END PROGRAM SQL-ERROR.

END PROGRAM MAIN.
```

## 8.3.9.3 Scope

Because WHENEVER is a declarative statement, its scope is positional, not logical. It tests all executable SQL statements that follow it in the source file, not in the flow of program logic. So, code the WHENEVER statement before the first executable SQL statement you want to test.

A WHENEVER statement stays in effect until superseded by another WHENEVER statement checking for the same condition.

**Suggestion**: You can place WHENEVER statements at the beginning of each program unit that contains SQL statements. That way, SQL statements in one program unit will not reference WHENEVER actions in another program unit, causing errors at compile or run time.

## 8.3.9.4 Careless Usage: Examples

Careless use of the WHENEVER statement can cause problems. For example, the following code enters an infinite loop if the DELETE statement sets the NOT FOUND condition, because no rows meet the search condition:

```
* Improper use of WHENEVER.

EXEC SQL

WHENEVER NOT FOUND GOTO NO-MORE

END-EXEC.

PERFORM GET-ROWS UNTIL DONE = "YES".

...

GET-ROWS.

EXEC SQL

FETCH emp_cursor INTO :EMP-NAME, :SALARY

END-EXEC.

...

NO-MORE.

MOVE "YES" TO DONE.

EXEC SQL

DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER

END-EXEC.

...
```

In the next example, the NOT FOUND condition is properly handled by resetting the GOTO target:

```
* Proper use of WHENEVER.

EXEC SQL WHENEVER NOT FOUND GOTO NO-MORE END-EXEC.

PERFORM GET-ROWS UNTIL DONE = "YES".

...

GET-ROWS.

EXEC SQL

FETCH emp_cursor INTO :EMP-NAME, :SALARY

END-EXEC.

...
```



```
NO-MORE.

MOVE "YES" TO DONE.

EXEC SQL WHENEVER NOT FOUND GOTO NONE-FOUND END-EXEC.

EXEC SQL

DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER
END-EXEC.

...

NONE-FOUND.
```

## 8.3.10 Getting the Text of SQL Statements

In many Pro\*COBOL applications, it is convenient to know the text of the statement being processed, its length, and the SQL command (such as INSERT or SELECT) that it contains. This is especially true for applications that use dynamic SQL.

The routine SQLGLS, which is part of the SQLLIB runtime library, returns the following information:

- The text of the most recently parsed SQL statement
- The length of the statement
- A function code

You can call SQLGLS after issuing a static SQL statement. With dynamic SQL Method 1, you can call SQLGLS after the SQL statement is executed. With dynamic SQL Method 2, 3, or 4, you can call SQLGLS after the statement is prepared.

To call SQLGLS, you use the following syntax:

```
CALL "SQLGLS" USING SQLSTM STMLEN SQLFC.
```

Table 8-3 shows the host-language datatypes available for the parameters in the SQLGLS argument list.tt

**Table 8-3 Parameter Datatypes** 

Parameter	Datatype
SQLSTM	PIC X(n)
STMLEN	PIC S9(9) COMP
SQLFC	PIC S9(9) COMP

All parameters must be passed by reference. This is usually the default parameter passing convention; you need not take special action.

The parameter SQLSTM is a blank-padded (not null-terminated) character buffer that holds the returned text of the SQL statement. Your program must statically declare the buffer or dynamically allocate memory for it.

The length parameter STMLEN is a four-byte integer. Before calling SQLGLS, set this parameter to the actual size (in bytes) of the SQLSTM buffer. When SQLGLS returns, the SQLSTM buffer contains the SQL statement text blank padded to the length of the buffer. STMLEN returns the actual number of bytes in the returned statement text, not counting the blank padding. However, STMLEN returns a zero if an error occurred.

Some possible errors follow:



- No SQL statement was parsed.
- You passed an invalid parameter (for example, a negative length value).
- An internal exception occurred in SQLLIB.

The parameter SQLFC is a four-byte integer that returns the SQL function code for the SQL command in the statement. A complete table of the function code for each SQL command is found in *Oracle Call Interface Developer's Guide*.

There are no SQL function codes for these statements:

- CONNECT
- COMMIT
- FETCH
- ROLLBACK
- RELEASE

# 8.4 Using the Oracle Communications Area

The SQLCA handles standard SQL communications. The Oracle Communications Area (ORACA) is a similar structure that you can include in your program to handle Oracle-specific communications. When you need more runtime information than the SQLCA provides, use the ORACA.

Besides helping you to diagnose problems, the ORACA lets you monitor your program's use of resources such as the SQL Statement Executor and the *cursor cache*, an area of memory reserved for cursor management.

## 8.4.1 Contents of the ORACA

The ORACA contains option settings, system statistics, and extended diagnostics. Figure 8-3 shows all the variables in the ORACA:

Figure 8-3 ORACA Variable Declarations for Pro\*COBOL

```
ORACA
    ORACA.
01
    05
            ORACAID PIC X(8).
    05
            ORACABC PIC S9(9) COMP.
    0.5
            ORACCHF PIC S9(9) COMP.
    05
            ORADBGF PIC S9(9) COMP.
    05
            ORAHCHF PIC S9(9) COMP.
    05
            ORASTXTF
                                          PIC S9(9) COMP.
    05
            ORASTXT.
                    ORASTXTL
                                          PIC S9(4) COMP.
            49
            49
                    ORASTXTL
                                          PIC X(70).
    05
            ORASFNM.
                    ORASFNML
                                          PIC S9(4) COMP.
            49
            49
                    ORASFNMC
                                          PIC X(70).
            ORASLNR PIC X(8).
    05
            ORAHOC PTC S9(9) COMP.
    0.5
            ORAMOC PIC S9(9) COMP.
    05
    05
            ORACOC PIC S9(9) COMP.
    05
            ORANOR PIC S9(9) COMP.
            ORANPR PIC S9(9) COMP.
            ORANEX PIC S9(9) COMP.
```



# 8.4.2 Declaring the ORACA

To declare the ORACA, simply include it (using an EXEC SQL INCLUDE statement) in your Pro\*COBOL source file outside the Declare Section as follows:

\* Include the Oracle Communications Area (ORACA). EXEC SQL INCLUDE ORACA END-EXEC.

# 8.4.3 Enabling the ORACA

To enable the ORACA, you must set the ORACA precompiler option to YES on the command line or in a configuration file with:

ORACA=YES

or inline with:

EXEC Oracle OPTION (ORACA=YES) END-EXEC.

Then, you must choose appropriate runtime options by setting flags in the ORACA. Enabling the ORACA is optional because it adds to runtime overhead. The default setting is ORACA=NO.

# 8.4.4 Choosing Runtime Options

The ORACA includes several option flags. Setting these flags by assigning them nonzero values enables you to:

- Save the text of SQL statements
- Enable DEBUG operations
- Check cursor cache consistency (the cursor cache is a continuously updated area of memory used for cursor management)
- Check heap consistency (the heap is an area of memory reserved for dynamic variables)
- Gather cursor statistics

The following descriptions will help you choose the options you need.

## 8.4.5 ORACA Structure

This section describes the structure of the ORACA, its fields, and the values they can store.

## 8.4.5.1 ORACAID

This string field is initialized to ORACA to identify the Oracle Communications Area.

### 8.4.5.2 ORACABC

This integer field holds the length, expressed in bytes, of the ORACA data structure.



### 8.4.5.3 ORACCHF

If the master DEBUG flag (ORADBGF) is set, this flag lets you check the cursor cache for consistency before every cursor operation.

The runtime library does the consistency checking and can issue error messages, which are listed in *Oracle Database Error Messages*.

This flag has the following settings:

Settings	Description	
0	Disable cache consistency checking (the default).	
1	Enable cache consistency checking.	

### 8.4.5.4 ORADBGF

This master flag lets you choose all the DEBUG options. It has the following settings:

Settings	Description	
0	Disable all DEBUG operations (the default).	
1	Allow DEBUG operations to be enabled.	

### 8.4.5.5 ORAHCHF

If the master DEBUG flag (ORADBGF) is set, this flag tells the runtime library to check the heap for consistency every time Pro\*COBOL dynamically allocates or frees memory. This is useful for detecting program bugs that upset memory.

This flag must be set before the CONNECT command is issued and, once set, cannot be cleared; subsequent change requests are ignored. It has the following settings:

Settings	Description
0	Enable heap consistency checking (the default).
1	Disable heap consistency checking.

### 8.4.5.6 ORASTXTF

This flag lets you specify when the text of the current SQL statement is saved. It has the following settings:

Settings	Description	
0	Never save the SQL statement text (the default).	
1	Save the SQL statement text on SQLERROR only.	
2	Save the SQL statement text on SQLERROR or SQLWARNING.	
3	Always save the SQL statement text.	

The SQL statement text is saved in the ORACA sub-record named ORASTXT.



## 8.4.5.7 Diagnostics

The ORACA provides an enhanced set of diagnostics; the following variables help you to locate errors quickly.

### 8.4.5.8 ORASTXT

This sub-record helps you find faulty SQL statements. It lets you save the text of the last SQL statement parsed by Oracle. It contains the following two fields:

Settings	Description	
ORASTXTL	This integer field holds the length of the current SQL statement.	
ORASTXTC	This string field holds the text of the current SQL statement. At most, the first 70 characters of text are saved.	

Statements parsed by Pro\*COBOL, such as CONNECT, FETCH, and COMMIT, are *not* saved in the ORACA.

### 8.4.5.9 ORASFNM

This sub-record identifies the file containing the current SQL statement and so helps you find errors when multiple files are precompiled for one application. It contains the following two fields:

Settings	Description
ORASFNML	This integer field holds the length of the filename stored in ORASFNMC.
ORASFNMC	This string field holds the filename. At most, the first 70 characters are stored.

### 8.4.5.10 ORASLNR

This integer field identifies the line at (or near) which the current SQL statement can be found.

## 8.4.5.11 Cursor Cache Statistics

The following variables let you gather cursor cache statistics. They are automatically set by every COMMIT or ROLLBACK statement your program issues. Internally, there is a set of these variables for each CONNECTed database. The current values in the ORACA pertain to the database against which the last commit or rollback was executed.

### 8.4.5.12 ORAHOC

This integer field records the highest value to which MAXOPENCURSORS was set during program execution.



### 8.4.5.13 ORAMOC

This integer field records the maximum number of open cursors required by your program. This number can be higher than ORAHOC if MAXOPENCURSORS was set too low, which forced Pro\*COBOL to extend the cursor cache.

#### 8.4.5.14 ORACOC

This integer field records the current number of open cursors required by your program.

#### 8.4.5.15 ORANOR

This integer field records the number of cursor cache reassignments required by your program. This number shows the degree of "thrashing" in the cursor cache and should be kept as low as possible.

#### 8.4.5.16 ORANPR

This integer field records the number of SQL statement parses required by your program.

#### 8.4.5.17 ORANEX

This integer field records the number of SQL statement executions required by your program. The ratio of this number to the ORANPR number should be kept as high as possible. In other words, avoid unnecessary re-parsing.

#### **Related Topics**

Performance Tuning

## 8.4.6 ORACA Example Program

The following program prompts for a department number, inserts the name and salary of each employee in that department into one of two tables, and then displays diagnostic information from the ORACA:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ORACAEX.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL INCLUDE SQLCA END-EXEC.
    EXEC SQL INCLUDE ORACA END-EXEC.
    EXEC ORACLE OPTION (ORACA=YES) END-EXEC.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(20).
01 PASSWORD PIC X(20).
01 EMP-NAME PIC X(10)
                  PIC X(10) VARYING.
01 DEPT-NUMBER PIC S9(4) COMP.
01 SALARY PIC S9(6)V99
                           DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
```



```
DISPLAY "Username? " WITH NO ADVANCING.
    ACCEPT USERNAME.
    DISPLAY "Password? " WITH NO ADVANCING.
    ACCEPT PASSWORD.
    EXEC SOL
        WHENEVER SQLERROR GOTO SQL-ERROR
    END-EXEC.
    EXEC SOL
        CONNECT : USERNAME IDENTIFIED BY : PASSWORD
    END-EXEC.
    DISPLAY "Connected to Oracle".
* -- set flags in the ORACA
* -- enable debug operations
    MOVE 1 TO ORADBGF.
* -- enable cursor cache consistency check
    MOVE 1 TO ORACCHF.
* -- always save the SQL statement
    MOVE 3 TO ORASTXTF.
    DISPLAY "Department number? " WITH NO ADVANCING.
    ACCEPT DEPT-NUMBER.
    EXEC SQL DECLARE EMPCURSOR CURSOR FOR
        SELECT ENAME, SAL + NVL(COMM, 0)
        FROM EMP
        WHERE DEPTNO = :DEPT-NUMBER
    END-EXEC.
    EXEC SQL OPEN EMPCURSOR END-EXEC.
    EXEC SOL
        WHENEVER NOT FOUND GOTO NO-MORE
    END-EXEC.
LOOP.
    EXEC SQL
        FETCH EMPCURSOR INTO : EMP-NAME, :SALARY
    END-EXEC.
    IF SALARY < 2500
        EXEC SQL
            INSERT INTO PAY1 VALUES (:EMP-NAME, :SALARY)
        END-EXEC
    ELSE
        EXEC SOL
            INSERT INTO PAY2 VALUES (:EMP-NAME, :SALARY)
        END-EXEC
    END-IF.
    GO TO LOOP.
NO-MORE.
    EXEC SQL CLOSE EMPCURSOR END-EXEC.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL COMMIT WORK RELEASE END-EXEC.
    DISPLAY "(NO-MORE.) Last SQL statement: ", ORASTXTC.
    DISPLAY "... at or near line number: ", ORASLNR.
    DISPLAY " ".
    DISPLAY "
                       Cursor Cache Statistics".
    DISPLAY "Maximum value of MAXOPENCURSORS ", ORAHOC.
"", ORAMOC.
"", ORAMOC.
    DISPLAY "----".
                                              ", ORACOC.
    DISPLAY "Current number of open cursors:
    DISPLAY "Number of cache reassignments:
    DISPLAY "Number of SQL statement parses:
    DISPLAY "Number of SQL statement executions: ", ORANEX.
    STOP RUN.
```

# 8.5 How Errors Map to SQLSTATE Codes

The following table describes SQLSTATE the codes, what they signify, and the returned errors.

Table 8-4 SQLSTATE Codes

Code	Condition	Oracle Error
00000	successful completion	ORA-00000
01000	warning	
01001	cursor operation conflict	
1002	disconnect error	
01003	null value eliminated in set function	
1004	string data - right truncation	
1005	insufficient item descriptor areas	
01006	privilege not revoked	
1007	privilege not granted	
1008	implicit zero-bit padding	
01009	search condition too long for info schema	
)100A	query expression too long for info schema	
2000	no data	ORA-01095
		ORA-01403
7000	dynamic SQL error	
7001	using clause does not match parameter specs	
7002	using clause does not match target specs	
7003	cursor specification cannot be executed	
7004	using clause required for dynamic parameters	
7005	prepared statement not a cursor specification	
7006	restricted datatype attribute violation	
7007	using clause required for result fields	
7008	invalid descriptor count	SQL-02126

Table 8-4 (Cont.) SQLSTATE Codes

Code	Condition	Oracle Error
07009	invalid descriptor index	
08000	connection exception	
08001	SQL client unable to establish SQL connection	
08002	connection name in use	
08003	connection does not exist	SQL-02121
08004	SQL server rejected SQL connection	
08006	connection failure	
08007	transaction resolution unknown	
0A000	feature not supported	ORA-03000 03099
0A001	multiple server transactions	
21000	cardinality violation	ORA-01427
		SQL-02112
22000	data exception	
22001	string data - right truncation	ORA-01401
		ORA-01406
22002	null value - no indicator parameter	ORA-01405
		SQL-02124
22003	numeric value out of range	ORA-01426
		ORA-01438
		ORA-01455
		ORA-01457
22005	error in assignment	
22007	invalid datetime format	
22008	datetime field overflow	ORA-01800 01899
22009	invalid time zone displacement value	
22011	substring error	
22012	division by zero	ORA-01476
22015	interval field overflow	
22018	invalid character value for cast	
22019	invalid escape character	ORA-00911
		ORA-01425
22021	character not in repertoire	
22022	indicator overflow	ORA-01411
22023	invalid parameter value	ORA-01025
		ORA-01488
		ORA-04000 04019
22024	unterminated C string	ORA-01479 01480
22025	invalid escape sequence	ORA-01424
22026	string data - length mismatch	



Table 8-4 (Cont.) SQLSTATE Codes

Code	Condition	Oracle Error
22027	trim error	
23000	integrity constraint violation	ORA-1400, ORA-00001
		ORA-02290 02299
24000	invalid cursor state	ORA-01001 01003
		ORA-01410
		ORA-08006
		SQL-02114
		SQL-02117 SQL-02118
		SQL-02118 SQL-02122
25000	invalid transaction state	0 QL 02122
26000	invalid SQL statement name	
27000	triggered data change violation	
28000	invalid authorization specification	
2A000	direct SQL syntax error or access rule violation	
2B000	dependent privilege descriptors still exist	
2C000	invalid character set name	
2D000	invalid transaction termination	
2E000	invalid connection name	
33000	invalid SQL descriptor name	
34000	invalid cursor name	
35000	invalid condition number	
37000	dynamic SQL syntax error or access rule violation	
3C000	ambiguous cursor name	
3D000	invalid catalog name	
3F000	invalid schema name	
40000	transaction rollback	ORA-02091 02092
40001	serialization failure	
40002	integrity constraint violation	
40003	statement completion unknown	



Table 8-4 (Cont.) SQLSTATE Codes

42000   Syntax error or access rule violation   ORA-00022   ORA-00251   ORA-00251   ORA-00251   ORA-00251   ORA-00300 00999   ORA-01031   ORA-01031   ORA-011031   ORA-01190 011493   ORA-011700 01799   ORA-01190 02099   ORA-02140 02289   ORA-02420 02424   ORA-02450 02429   ORA-02450 02429   ORA-02450 02499   ORA-03276 03299   ORA-04040 04059   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-00370 00429   ORA-00370 00429   ORA-00370 00429   ORA-07200 07999   ORA-0	Code	Condition	Oracle Error
ORA-00900 00999   ORA-01031   ORA-01031   ORA-01490 01493   ORA-01490 01493   ORA-01700 01799   ORA-01700 01799   ORA-01700 01799   ORA-02140 02289   ORA-02424 02289   ORA-02420 02424   ORA-02420 02424   ORA-02420 02429   ORA-04040 04059   ORA-04040 04059   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-06000 00899   ORA-06400 00899   ORA-06430 06449   ORA-07200 07999   ORA-07200 07999   ORA-07200 07999   ORA-07070 09099   ORA-00700 09099   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-02700 03199   ORA-02700 0319	42000	syntax error or access rule violation	ORA-00022
ORA-01031   ORA-01490   .01493   ORA-01700   .01493   ORA-01700   .01799   ORA-01490   .02099   ORA-02140   .02299   ORA-02140   .02289   ORA-02420   .02424   ORA-02420   .02424   ORA-02420   .02424   ORA-02420   .03299   ORA-04040   .04059   ORA-04040   .04059   ORA-04070   .04099   ORA-04070   .04099   ORA-04070   .04099   ORA-04070   .04099   ORA-04070   .04099   ORA-06000   ORA-06000   ORA-06000   ORA-06000   ORA-06000   ORA-06000   ORA-06000   ORA-06000   ORA-06430   ORA-064			ORA-00251
ORA-01490 01493   ORA-01700 01799   ORA-01700 01799   ORA-01700 01799   ORA-01700 01799   ORA-01700 01799   ORA-01700 01799   ORA-01700 02099   ORA-02140 022424   ORA-02140 02424   ORA-02240 02424   ORA-02250 02429   ORA-03276 03299   ORA-04040 04059   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-06000 0899   ORA-06000 0899   ORA-06000 0899   ORA-06430 06449   ORA-07200 07999   ORA-06430 06449   ORA-07200 07999   ORA-09050 00068   ORA-02376 02399   ORA-04020 04039   ORA-02376 02399   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-06000 06159   ORA-06000 06159   ORA-06000 06159   ORA-06000 06159   ORA-06000 06149   ORA-06000 06599   ORA-06000 06599   ORA-06250 066429   ORA-06250 06429   ORA-06250 06259   ORA-06250 06429   ORA-06250 06259   ORA-06250 06259   ORA-0625			ORA-00900 00999
ORA-01700 01799   ORA-01900 02099   ORA-01900 02099   ORA-02140 02289   ORA-02140 02289   ORA-02140 02289   ORA-02420 02424   ORA-02450 02429   ORA-02450 02429   ORA-02450 02429   ORA-02450 02429   ORA-03276 03299   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-06430 06429   ORA-06600 08899   ORA-06620 06439   ORA-07200 07999   ORA-072			ORA-01031
ORA-01900 02099   ORA-02140 02289   ORA-02140 02289   ORA-02140 02289   ORA-02140 02429   ORA-02420 02429   ORA-03276 03299   ORA-03276 03299   ORA-04040 04059   ORA-04070 04099   ORA-06000 00899   ORA-06000 00899   ORA-06000 00899   ORA-06000 00899   ORA-06000 00899   ORA-06000 00899   ORA-09700 07999   ORA-09700 07999   ORA-09700 07999   ORA-09700 07999   ORA-09700 009999   ORA-09700 009999   ORA-00000 00068   ORA-02376 002399   ORA-00200 00068   ORA-02376 002399   ORA-004020 04039   ORA-004020 04039   ORA-004000 00120   ORA-00440 00569   ORA-00440 00569   ORA-00400 00159   ORA-00400 00159   ORA-02370 00159   ORA-02370 00159   ORA-02370 00159   ORA-02370 00159   ORA-06000 06599   ORA-06000 06599   ORA-06000 06599   ORA-06000 06149   ORA-06000			ORA-01490 01493
ORA-02140 02289   ORA-02420 02424   ORA-02420 02424   ORA-02420 02424   ORA-02426 02499   ORA-02426 02499   ORA-03276 03299   ORA-04040 04059   ORA-04070 04099   ORA-0600 00899   ORA-06600 00899   ORA-06600 06499   ORA-07200 07999   ORA-07200 07999   ORA-07200 07999   ORA-07200 07999   ORA-07200 07999   ORA-00500 00068   ORA-02376 02399   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04040 00569   ORA-04040 00569   ORA-04040 00569   ORA-02700 02899   ORA-0270			ORA-01700 01799
ORA-02420 02424   ORA-02450 02499   ORA-02450 02499   ORA-02450 02499   ORA-03276 03299   ORA-04040 04059   ORA-04040 04059   ORA-04040 04059   ORA-04040 04059   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-0600 00899   ORA-0630 06449   ORA-07200 07999   ORA-06330 06449   ORA-07200 07999   ORA-07000 09999   ORA-07000 09999   ORA-07000 09999   ORA-00500 00068   ORA-00376 02399   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04040 00569   ORA-00440 00569   ORA-00440 00569   ORA-03100 03199   ORA-06200 02899   ORA-03100 03199   ORA-06200 02899   ORA-06200 06249   ORA-06200 06249   ORA-06200 06249   ORA-06200 0629   ORA-06200			ORA-01900 02099
ORA-02450 02499   ORA-03276 03299   ORA-03276 03299   ORA-04040 04059   ORA-04040 04059   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-06000 00429   ORA-06600 00899   ORA-06630 06449   ORA-0630 06449   ORA-07200 07999   ORA-09700 09999   ORA-09700 09999   ORA-09700 09999   ORA-0018 00035   ORA-0018 00035   ORA-0018 00035   ORA-002376 02399   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-00440 00569   ORA-00440 00569   ORA-00440 00569   ORA-00440 00569   ORA-00440 00569   ORA-00440 00569   ORA-00270 02899   ORA-0270 02899   ORA-0270 02899   ORA-0270 02899   ORA-0270 02899   ORA-0200 06249   ORA-02128   ORA-06200 06249   ORA-06200 06599   ORA-06600 06599   ORA-06500 06599   ORA-06500 06429   ORA-06200 06429   ORA			
ORA-03276 03299   ORA-04040 04059   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 04099   ORA-04070 00429   ORA-00370 00429   ORA-0630			
ORA-04040 04059   ORA-04070 04099			
44000 with check option violation ORA-04070 04099 44000 with check option violation ORA-01402 60000 system errors ORA-00370 00429			
44000       with check option violation       ORA-01402         60000       system errors       ORA-00370 00429         60000       ORA-00600 00899       ORA-00600 00899         ORA-00430 06449       ORA-07200 07999         ORA-09700 09999       ORA-00018 00035         61000       resource error       ORA-00018 00035         ORA-002376 02399       ORA-00420 04039         62000       multithreaded server and detached process errors       ORA-00100 00120         ORA-00440 00569       ORA-00440 00569         63000       Oracle XA and two-task interface errors       ORA-00440 00569         63000       Oracle XA and two-task interface errors       ORA-00400 00159         ORA-02700 02899       ORA-02700 02899         ORA-02700 02899       ORA-06200 06249         SQL-02128       ORA-06200 06249         SQL-02128       ORA-06200 0629         65000       PL/SQL errors       ORA-06500 06599         66000       Oracle Net Services driver errors       ORA-06000 06429         ORA-06600 06999       ORA-12500 12599         67000       licensing errors       ORA-00430 00439         69000       SQL*Connect errors       ORA-00570 0059			
60000 system errors ORA-00370 00429 ORA-00600 00899 ORA-00600 00899 ORA-06430 06449 ORA-07200 07999 ORA-07200 07999 ORA-09700 09999 ORA-09700 09999 ORA-00570 00068 ORA-02376 02399 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-00100 00120 ORA-00150 00569 ORA-02700 00569 ORA-02700 00599 ORA-02700 02899 ORA-02700 02899 ORA-02700 02899 ORA-03100 03199 ORA-03100 03199 ORA-06200 06249 SQL-02128 ORA-02700 06249 SQL-02128 ORA-02700 06249 SQL-02128 ORA-02700 06249 ORA-06000 06599 ORA-06000 06599 ORA-06000 06599 ORA-06000 06599 ORA-06000 06149 ORA-06200 0629 ORA-06000 06399 ORA-12100 12299 ORA-12500 12599 ORA-12500 12599 ORA-12500 12599 ORA-06000 06399 ORA-06000 06399 ORA-06000 06399 ORA-12500 12599 ORA-06000 06399 ORA-06000 .			
ORA-00600 00899 ORA-06430 06449 ORA-07200 07999 ORA-09700 09999 61000 resource error ORA-00018 00035 ORA-00050 00068 ORA-02376 02399 ORA-04020 04039 62000 multithreaded server and detached process errors ORA-00100 00120 ORA-00440 00569 63000 Oracle XA and two-task interface errors ORA-00150 00159 SQL-02128 ORA-02700 02899 ORA-03100 03199 ORA-06200 06249 SQL-02128 64000 control file, database file, and redo file errors; ORA-00200 00369 archival and media recovery errors ORA-06200 06299 ORA-06500 06599 66000 PL/SQL errors ORA-06000 06149 ORA-06250 06429 ORA-06600 06999 ORA-12100 12299 ORA-12500 12599 67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599	44000	with check option violation	ORA-01402
ORA-06430 06449 ORA-07200 07999 ORA-09700 09999  61000 resource error ORA-00018 00035 ORA-00050 00068 ORA-02376 02399 ORA-04020 04039 ORA-04020 04039  62000 multithreaded server and detached process errors ORA-00100 00120 ORA-00440 00569  63000 Oracle XA and two-task interface errors ORA-00150 00159 SQL-02128 ORA-02700 02899 ORA-03100 03199 ORA-06200 06249 SQL-02128  64000 control file, database file, and redo file errors; ORA-0100 00369 archival and media recovery errors ORA-01100 01250  65000 PL/SQL errors ORA-06500 06599  66000 Oracle Net Services driver errors ORA-06000 06149 ORA-06250 06429 ORA-06250 06429 ORA-06600 06999 ORA-12100 12299 ORA-12500 12599  67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599	60000	system errors	ORA-00370 00429
ORA-07200 07999   ORA-09700 09999			
61000       resource error       ORA-09700 09999         61000       resource error       ORA-00018 00035 ORA-00050 00068 ORA-002376 02399 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-00100 00120 ORA-00100 00120 ORA-00100 00120 ORA-00150 00159 SQL-02128 ORA-00150 00159 ORA-00150 00159 ORA-002128 ORA-02700 02899 ORA-02700 02899 ORA-03100 03199 ORA-03100 03199 ORA-06200 06249 SQL-02128         64000       control file, database file, and redo file errors; ORA-01100 01250 ORA-01100 01250 ORA-01100 01250 ORA-06000 06599 ORA-01100 01250 ORA-06600 06599 ORA-06600 06999 ORA-12100 12299 ORA-12500 12299 ORA-12500 12299 ORA-12500 12299 ORA-12500 12299 ORA-12500 12599         67000       licensing errors       ORA-00370 00599         69000       SQL*Connect errors       ORA-00430 00439 ORA-00570 00599			ORA-06430 06449
61000       resource error       ORA-00018 00035 ORA-00050 00068 ORA-002376 02399 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-04020 04039 ORA-00100 00120 ORA-00440 00569 ORA-00440 00569 ORA-00440 00569 ORA-00440 00569 ORA-00440 00569 ORA-02128 ORA-02700 02899 ORA-02700 02899 ORA-02700 02899 ORA-03100 03199 ORA-06200 06249 SQL-02128 ORA-02102 006249 ORA-06200 06249 ORA-06200 06249 ORA-06100 01250 ORA-06500 06599 ORA-01100 01250 ORA-06500 06599 ORA-06500 06429 ORA-06200 06429 ORA-06200 06429 ORA-06200 06429 ORA-06200 06429 ORA-06200 06429 ORA-12100 12299 ORA-12500 12599 ORA-12500 12599 ORA-12500 12599 ORA-00430 00439         67000       licensing errors       ORA-00430 00439 ORA-00570 00599			
ORA-00050 00068   ORA-02376 02399   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-04020 04039   ORA-00100 00120   ORA-00140 00569   ORA-00440 00569   ORA-00440 00569   ORA-00150 00159   SQL-02128   ORA-02700 02899   ORA-03100 03199   ORA-06200 06249   SQL-02128   ORA-06200 06249   SQL-02128   ORA-06200 06249   ORA-06200 00369   ORA-06200			ORA-09700 09999
ORA-02376 02399   ORA-04020 04039	61000	resource error	ORA-00018 00035
ORA-04020 04039			ORA-00050 00068
62000       multithreaded server and detached process errors       ORA-00100 00120 ORA-00440 00569         63000       Oracle XA and two-task interface errors       ORA-00150 00159 SQL-02128 ORA-02700 02899 ORA-03100 03199 ORA-06200 06249 SQL-02128         64000       control file, database file, and redo file errors; ORA-00200 00369 archival and media recovery errors       ORA-01100 01250 ORA-01100 01250         65000       PL/SQL errors       ORA-06500 06599 ORA-06500 06599 ORA-06600 06429 ORA-06600 06429 ORA-06600 06999 ORA-12100 12299 ORA-12100 12299 ORA-12500 12599         67000       licensing errors       ORA-00430 00439 ORA-00570 00599         69000       SQL*Connect errors       ORA-00570 00599			
63000 Oracle XA and two-task interface errors  ORA-00440 00569  ORA-00150 00159  SQL-02128  ORA-02700 02899  ORA-03100 03199  ORA-06200 06249  SQL-02128  64000 control file, database file, and redo file errors;  ORA-00200 00369  archival and media recovery errors  ORA-01100 01250  65000 PL/SQL errors  ORA-06500 06599  66000 Oracle Net Services driver errors  ORA-06000 06429  ORA-06250 06429  ORA-06600 06999  ORA-12100 12299  ORA-12500 12599  67000 licensing errors  ORA-00430 00439  69000 SQL*Connect errors  ORA-00570 00599			ORA-04020 04039
63000       Oracle XA and two-task interface errors       ORA-00150 00159 SQL-02128 ORA-02700 02899 ORA-03100 03199 ORA-06200 06249 SQL-02128         64000       control file, database file, and redo file errors; archival and media recovery errors       ORA-00200 00369 ORA-01100 01250         65000       PL/SQL errors       ORA-06500 06599 ORA-06500 06599 ORA-06250 06429 ORA-06250 06429 ORA-06000 06999 ORA-12100 12299 ORA-12500 12599         67000       licensing errors       ORA-00430 00439 ORA-00570 00599	62000	multithreaded server and detached process errors	
SQL-02128 ORA-02700 02899 ORA-03100 03199 ORA-06200 06249 SQL-02128  64000			ORA-00440 00569
ORA-02700 02899 ORA-03100 03199 ORA-06200 06249 SQL-02128  64000 control file, database file, and redo file errors; ORA-00200 00369 archival and media recovery errors ORA-01100 01250  65000 PL/SQL errors ORA-06500 06599  66000 Oracle Net Services driver errors ORA-06000 06149 ORA-06250 06429 ORA-06600 06999 ORA-12100 12299 ORA-12500 12599  67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599	63000	Oracle XA and two-task interface errors	ORA-00150 00159
ORA-03100 03199 ORA-06200 06249 SQL-02128  64000 control file, database file, and redo file errors; ORA-00200 00369 archival and media recovery errors ORA-01100 01250  65000 PL/SQL errors ORA-06500 06599  66000 Oracle Net Services driver errors ORA-06000 06149 ORA-06250 06429 ORA-06600 06999 ORA-12100 12299 ORA-12500 12599  67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599			SQL-02128
ORA-06200 06249   SQL-02128			
64000       control file, database file, and redo file errors; archival and media recovery errors       ORA-00200 00369 ORA-01100 01250         65000       PL/SQL errors       ORA-06500 06599         66000       Oracle Net Services driver errors       ORA-06000 06149 ORA-06250 06429 ORA-06600 06999 ORA-12100 12299 ORA-12100 12299 ORA-12500 12599         67000       licensing errors       ORA-00430 00439 ORA-00570 00599         69000       SQL*Connect errors       ORA-00570 00599			
archival and media recovery errors  ORA-01100 01250  PL/SQL errors  ORA-06500 06599  ORA-06000 06149  ORA-06250 06429  ORA-06600 06999  ORA-12100 12299  ORA-12500 12599  67000 licensing errors  ORA-00430 00439  69000 SQL*Connect errors  ORA-00570 00599			
65000 PL/SQL errors ORA-06500 06599 66000 Oracle Net Services driver errors ORA-06000 06149 ORA-06250 06429 ORA-06600 06999 ORA-12100 12299 ORA-12500 12599 67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599	64000	control file, database file, and redo file errors;	ORA-00200 00369
66000       Oracle Net Services driver errors       ORA-06000 06149         ORA-06250 06429       ORA-06600 06999         ORA-12100 12299       ORA-12500 12599         67000       licensing errors       ORA-00430 00439         69000       SQL*Connect errors       ORA-00570 00599		archival and media recovery errors	ORA-01100 01250
ORA-06250 06429 ORA-06600 06999 ORA-12100 12299 ORA-12500 12599  67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599	65000	PL/SQL errors	ORA-06500 06599
ORA-06600 06999 ORA-12100 12299 ORA-12500 12599 67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599	66000	Oracle Net Services driver errors	ORA-06000 06149
ORA-12100 12299 ORA-12500 12599  67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599			ORA-06250 06429
ORA-12500 12599 67000 licensing errors ORA-00430 00439 69000 SQL*Connect errors ORA-00570 00599			ORA-06600 06999
67000         licensing errors         ORA-00430 00439           69000         SQL*Connect errors         ORA-00570 00599			ORA-12100 12299
69000 SQL*Connect errors ORA-00570 00599			ORA-12500 12599
	67000	licensing errors	ORA-00430 00439
ORA-07000 07199	69000	SQL*Connect errors	ORA-00570 00599
			ORA-07000 07199



Table 8-4 (Cont.) SQLSTATE Codes

Code	Condition	Oracle Error
72000	SQL execute phase errors	ORA-01000 01099
		ORA-01401 01489
		ORA-01495 01499
		ORA-01500 01699
		ORA-02400 02419
		ORA-02425 02449 ORA-04060 04069
		ORA-08000 08190
		ORA-12000 12019
		ORA-12300 12499
		ORA-12700 21999
82100	out of memory (could not allocate)	SQL-02100
82101	inconsistent cursor cache: unit cursor/global cursor mismatch	SQL-02101
82102	inconsistent cursor cache: no global cursor entry	SQL-02102
82103	inconsistent cursor cache: out of range cursor cache reference	SQL-02103
82104	inconsistent host cache: no cursor cache available	SQL-02104
82105	inconsistent cursor cache: global cursor not found	SQL-02105
82106	inconsistent cursor cache: invalid cursor number	SQL-02106
82107	program too old for runtime library	SQL-02107
82108	invalid descriptor passed to runtime library	SQL-02108
82109	inconsistent host cache: host reference is out of range	SQL-02109
82110	inconsistent host cache: invalid host cache entry type	SQL-02110
82111	heap consistency error	SQL-02111
82112	unable to open message file	SQL-02113
82113	code generation internal consistency failed	SQL-02115
82114	reentrant code generator gave invalid context	SQL-02116
82115	invalid hstdef argument	SQL-02119
82116	first and second arguments to sqlrcn both null	SQL-02120
82117	invalid OPEN or PREPARE for this connection	SQL-02122
82118	application context not found	SQL-02123
82119	connect error; can't get error text	SQL-02125
82120	precompiler/SQLLIB version mismatch.	SQL-02127
82121	FETCHed number of bytes is odd	SQL-02129
82122	EXEC TOOLS interface is not available	SQL-02130
82123	runtime context in use	SQL-02131
82124	unable to allocate runtime context	SQL-02131
82125	unable to initialize process for use with threads	SQL-02133
82126	invalid runtime context	SQL-02134



Table 8-4 (Cont.) SQLSTATE Codes

Code	Condition	Oracle Error
90000	debug events	ORA-10000 10999
99999	catch all	all others
HZ000	remote database access	

## 8.5.1 Status Variable Combinations

When MODE={ANSI | ANSI14}, the behavior of the status variables depends on the following:

- Which variables are declared.
- Declaration placement (inside or outside the Declare Section).
- The ASSUME\_SQLCODE setting.

Table 8-5 and Table 8-6 describe the resulting behavior of each status variable combination when ASSUME\_SQLCODE=NO and when ASSUME\_SQLCODE=YES, respectively.

For both Tables: when DECLARE\_SECTION=NO, any declaration of a status variable is treated as IN as far as these tables are concerned.

Do not use ASSUME\_SQLCODE=YES with DECLARE\_SECTION=NO.

Table 8-5 Status Variable Behavior with ASSUME\_SQLCODE=NO and MODE=ANSI | ANSI14 and DECLARE\_SECTION=YES

SQLCODE	SQLSTATE	SQLCA	Behavior
OUT	_	_	SQLCODE is declared and is presumed to be a status variable.
OUT	_	OUT	This status variable configuration is not supported.
OUT	_	IN	This status variable configuration is not supported.
OUT	OUT	_	SQLCODE is declared and is presumed to be a status variable, and SQLSTATE is declared but is not recognized as a status variable.
OUT	OUT	OUT	This status variable configuration is not supported.
OUT	OUT	IN	This status variable configuration is not supported.
OUT	IN	_	SQLSTATE is declared as a status variable, and SQLCODE is declared but is not recognized as a status variable.
OUT	IN	OUT	This status variable configuration is not supported.
OUT	IN	IN	This status variable configuration is not supported.
IN	_	_	SQLCODE is declared as a status variable.
IN	_	OUT	This status variable configuration is not supported.
IN	_	IN	This status variable configuration is not supported.
IN	OUT	_	SQLCODE is declared as a status variable, and SQLSTATE is declared but is not recognized as a status variable.
IN	OUT	OUT	This status variable configuration is not supported.
IN	OUT	IN	This status variable configuration is not supported.



Table 8-5 (Cont.) Status Variable Behavior with ASSUME\_SQLCODE=NO and MODE=ANSI | ANSI14 and DECLARE\_SECTION=YES

SQLCODE	SQLSTATE	SQLCA	Behavior
IN	IN	_	SQLCODE and SQLSTATE are declared as a status variables.
IN	IN	OUT	This status variable configuration is not supported.
IN	IN	IN	This status variable configuration is not supported.
_	_	_	This status variable configuration is not supported.
_	_	OUT	SQLCA is declared as a status variable.
_	_	IN	SQLCA is declared as a status host variable.
_	OUT	_	This status variable configuration is not supported.
_	OUT	OUT	SQLCA is declared as a status variable, and SQLSTATE is declared but is not recognized as a status variable.
_	OUT	IN	SQLCA is declared as a status host variable, and SQLSTATE is declared but is not recognized as a status variable.
_	IN	_	SQLSTATE is declared as a status variable.
_	IN	OUT	SQLSTATE and SQLCA are declared as status variables.
	IN	IN	SQLSTATE and SQLCA are declared as status host variables.

Table 8-6 Status Variable Behavior with ASSUME\_SQLCODE=YES and MODE=ANSI | ANSI14 and DECLARE\_SECTION=YES

SQLCODE	SQLSTATE	SQLCA	Behavior
OUT	<del>_</del>	_	SQLCODE is declared and is presumed to be a status variable.
OUT	_	OUT	This status variable configuration is not supported.
OUT	_	IN	This status variable configuration is not supported.
OUT	OUT	_	SQLCODE is declared and is presumed to be a status variable, and SQLSTATE is declared but is not recognized as a status variable.
OUT	OUT	OUT	This status variable configuration is not supported.
OUT	OUT	IN	This status variable configuration is not supported.
OUT	IN	_	SQLSTATE is declared as a status variable, and SQLCODE is declared and is presumed to be a status variable.
OUT	IN	OUT	This status variable configuration is not supported.
OUT	IN	IN	This status variable configuration is not supported.
IN		_	SQLCODE is declared as a status variable.
IN	_	OUT	This status variable configuration is not supported.
IN	_	IN	This status variable configuration is not supported.
IN	OUT	_	SQLCODE is declared as a status variable, and SQLSTATE is declared but not as a status variable.
IN	OUT	OUT	This status variable configuration is not supported.
IN	OUT	IN	This status variable configuration is not supported.
IN	IN	_	SQLCODE and SQLSTATE are declared as a status variables.
IN	IN	OUT	This status variable configuration is not supported.



Table 8-6 (Cont.) Status Variable Behavior with ASSUME\_SQLCODE=YES and MODE=ANSI | ANSI14 and DECLARE\_SECTION=YES

SQLCODE	SQLSTATE	SQLCA	Behavior
IN	IN	IN	This status variable configuration is not supported.
_	_	_	These status variable configurations are not supported. SQLCODE
_	_	OUT	must be declared when ASSUME_SQLCODE=YES.
_	_	IN	
_	OUT	_	
_	OUT	OUT	
_	OUT	IN	
_	IN	_	
_	IN	OUT	
_	IN	IN	



9

# Oracle Dynamic SQL

This chapter shows you how to use dynamic SQL, an advanced programming technique that adds flexibility and functionality to your applications. After weighing the advantages and disadvantages of dynamic SQL, you learn four methods—from simple to complex—for writing programs that accept and process SQL statements "on the fly" at run time. You learn the requirements and limitations of each method and how to choose the right method for a given job.

#### Topics are:

- Dynamic SQL
- Advantages and Disadvantages of Dynamic SQL
- When to Use Dynamic SQL
- Requirements for Dynamic SQL Statements
- How Dynamic SQL Statements Are Processed
- Methods for Using Dynamic SQL
- Using Method 1
- Sample Program 6: Dynamic SQL Method 1
- Using Method 2
- Sample Program 7: Dynamic SQL Method 2
- Using Method 3
- Sample Program 8: Dynamic SQL Method 3
- Using Oracle Method 4
- Using the DECLARE STATEMENT Statement
- Using Host Tables
- Using PL/SQL
- Dynamic SQL Statement Caching
- About Boolean Data Type Support

# 9.1 Dynamic SQL

Most database applications do a specific job. For example, a simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables. In this case, you know the makeup of the UPDATE statement at precompile time. That is, you know which tables might be changed, the constraints defined for each table and column, which columns might be updated, and the datatype of each column.

However, some applications must accept (or build) and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different SELECT statements for the various reports it generates. In this case, the statement's makeup is

unknown until run time. Such statements can, and probably will, change from execution to execution. They are aptly called *dynamic* SQL statements.

Unlike static SQL statements, dynamic SQL statements are not embedded in your source program. Instead, they are stored in character strings input to or built by the program at run time. They can be entered interactively or read from a file.

# 9.2 Advantages and Disadvantages of Dynamic SQL

Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs. Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.

For example, your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, dynamic SQL lets you write highly flexible applications.

However, some dynamic queries require complex coding, the use of special data structures, and more runtime processing. While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.

# 9.3 When to Use Dynamic SQL

In practice, static SQL will meet nearly all your programming needs. Use dynamic SQL only if you need its open-ended flexibility. Its use is suggested when one or more of the following items is unknown at precompile time:

- Text of the SQL statement (commands, clauses, and so on)
- The number of host variables
- The datatypes of host variables
- References to database objects such as columns, indexes, sequences, tables, usernames, and views

# 9.4 Requirements for Dynamic SQL Statements

To represent a dynamic SQL statement, a character string must contain the text of a valid DML or DDL SQL statement, but *not* contain the EXEC SQL clause, host-language delimiter or statement terminator.

In most cases, the character string can contain *dummy* host variables. They hold places in the SQL statement for actual host variables. Because dummy host variables are just place-holders, you do not declare them and can name them anything you like (hyphens are not allowed). For example, Oracle makes no distinction between the following two strings

```
'DELETE FROM EMP WHERE MGR = :MGRNUMBER AND JOB = :JOBTITLE'
'DELETE FROM EMP WHERE MGR = :M AND JOB = :J'
```



# 9.5 How Dynamic SQL Statements Are Processed

Typically, an application program prompts the user for the text of a SQL statement and the values of host variables used in the statement. Then Oracle *parses* the SQL statement. That is, Oracle examines the SQL statement to make sure it follows syntax rules and refers to valid database objects. Parsing also involves checking database access rights, reserving needed resources, and finding the optimal access path.

Next, Oracle *binds* the host variables to the SQL statement. That is, Oracle gets the addresses of the host variables so that it can read or write their values.

If the statement is a query, you define the SELECT variables and then Oracle FETCHes them until all rows are retrieved. The cursor is then closed.

Then Oracle *executes* the SQL statement. That is, Oracle does what the SQL statement requested, such as deleting rows from a table.

The SQL statement can be executed repeatedly using new values for the host variables.

# 9.6 Methods for Using Dynamic SQL

This section introduces the four methods you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, then offers guidelines for choosing the right method. Later sections show you how to use the methods.

The four methods are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and so on. However, each method is most useful for handling a certain kind of SQL statement, as Appropriate Method to Use shows:

Table 9-1 Appropriate Method to Use

Method	Kind of SQL Statement	
1	Non-query without input host variables.	
2	Non-query with known number of input host variables.	
3	Query with known number of select-list items and input host variables.	
4	Query with unknown number of select-list items or input host variables.	

The term select-list item includes column names and expressions.

## 9.6.1 Method 1

This method lets your program accept or build a dynamic SQL statement, then immediately execute it using the EXECUTE IMMEDIATE command. The SQL statement must not be a query (SELECT statement) and must not contain any place-holders for input host variables. For example, the following host strings qualify:

```
'DELETE FROM EMP WHERE DEPTNO = 20'
'GRANT SELECT ON EMP TO SCOTT'
```

With Method 1, the SQL statement is parsed every time it is executed (regardless of whether you have set HOLD\_CURSOR=YES).



## 9.6.2 Method 2

This method lets your program accept or build a dynamic SQL statement, then process it using the PREPARE and EXECUTE commands. The SQL statement must not be a query. The number of place-holders for input host variables and the datatypes of the input host variables must be known at precompile time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:EMPNAME, :JOBTITLE)'
'DELETE FROM EMP WHERE EMPNO = :EMPNUMBER'
```

With Method 2, the SQL statement can be parsed just once by calling PREPARE once, and executed many times with different values for the host variables. This is not true when RELEASE\_CURSOR=YES is also specified, because the statement has to be prepared again before each execution.



SQL data definition statements such as CREATE are executed once the PREPARE is completed.

## 9.6.3 Method 3

This method lets your program accept or build a dynamic query then process it using the PREPARE command with the DECLARE, OPEN, FETCH, and CLOSE cursor commands. The number of select-list items, the number of place-holders for input host variables, and the datatypes of the input host variables must be known at precompile time. For example, the following host strings qualify:

```
'SELECT DEPTNO, MIN(SAL), MAX(SAL) FROM EMP GROUP BY DEPTNO'
'SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :DEPTNUMBER'
```

## 9.6.4 Method 4

This method lets your program accept or build a dynamic SQL statement, then process it using descriptors (discussed in "Using Oracle Method 4"). The number of select-list items, the number of place-holders for input host variables, and the datatypes of the input host variables can be unknown until run time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (unknown) VALUES (unknown)'

'SELECT unknown FROM EMP WHERE DEPTNO = 20'
```

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables.



## 9.6.5 Guidelines

With all four methods, you must store the dynamic SQL statement in a character string, which must be a host variable or quoted literal. When you store the SQL statement in the string, omit the keywords EXEC SQL and the statement terminator.

With Methods 2 and 3, the number of place-holders for input host variables and the datatypes of the input host variables must be known at precompile time.

Each succeeding method imposes fewer constraints on your application, but is more difficult to code. As a rule, use the simplest method you can. However, if a dynamic SQL statement will be executed repeatedly by Method 1, use Method 2 instead to avoid re-parsing for each execution.

Method 4 provides maximum flexibility, but requires complex coding and a full understanding of dynamic SQL concepts. In general, use Method 4 only if you cannot use Methods 1, 2, or 3

The decision logic in Figure 9-1, will help you choose the correct method.

## 9.6.5.1 Avoiding Common Errors

If you use a character array to store the dynamic SQL statement, blank-pad the array before storing the SQL statement. That way, you clear extraneous characters. This is especially important when you reuse the array for different SQL statements. As a rule, always initialize (or re-initialize) the host string before storing the SQL statement.

Do not null-terminate the host string. Oracle does not recognize the null terminator as an end-of-string marker. Instead, Oracle treats it as part of the SQL statement.

If you use a VARCHAR variable to store the dynamic SQL statement, make sure the length of the VARCHAR is set (or reset) correctly before you execute the PREPARE or EXECUTE IMMEDIATE statement.

EXECUTE resets the SQLWARN warning flags in the SQLCA. So, to catch mistakes such as an unconditional update (caused by omitting a WHERE clause), check the SQLWARN flags after executing the PREPARE statement but before executing the EXECUTE statement.

Figure 9-1 shows how to choose the right method.



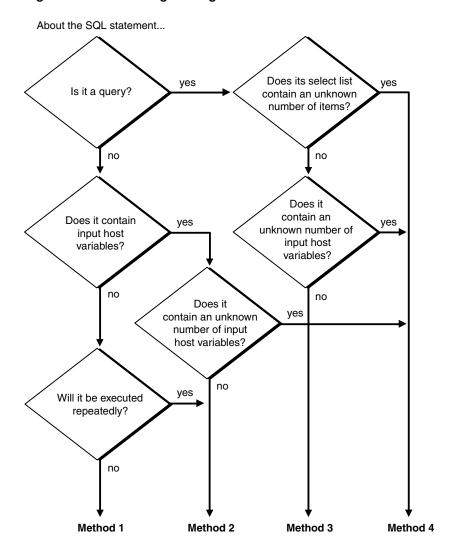


Figure 9-1 Choosing the Right Method

# 9.7 Using Method 1

The simplest kind of dynamic SQL statement results only in "success" or "failure" and uses no host variables. Some examples follow:

```
'DELETE FROM table_name WHERE column_name = constant'
'CREATE TABLE table_name ...'
'DROP INDEX index_name'
'UPDATE table_name SET column_name = constant'
'GRANT SELECT ON table_name TO username'
```

## 9.7.1 The EXECUTE IMMEDIATE Statement

Method 1 parses, then immediately executes the SQL statement using the EXECUTE IMMEDIATE command. The command is followed by a character string (host variable or literal) containing the SQL statement to be executed, which cannot be a query.

The syntax of the EXECUTE IMMEDIATE statement follows:

```
EXEC SQL EXECUTE IMMEDIATE { : HOST-STRING | STRING-LITERAL } END-EXEC.
```

In the following example, you use the host variable *SQL-STMT* to store SQL statements input by the user:

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

...

01 SQL-STMT PIC X(120);
    EXEC SQL END DECLARE SECTION END-EXEC.

...

LOOP.
    DISPLAY 'Enter SQL statement: ' WITH NO ADVANCING.
    ACCEPT SQL-STMT END-EXEC.

* -- sql_stmt now contains the text of a SQL statement
    EXEC SQL EXECUTE IMMEDIATE :SQL-STMT END-EXEC.

NEXT.
```

Because EXECUTE IMMEDIATE parses the input SQL statement before every execution, Method 1 is best for statements that are executed only once. Data definition statements usually fall into this category.

## 9.7.2 An Example

The following fragment of a program prompts the user for a search condition to be used in the WHERE clause of an UPDATE statement, then executes the statement using Method 1:

```
THE RELEASE CURSOR=YES OPTION INSTRUCTS PRO*COBOL TO
    RELEASE IMPLICIT CURSORS ASSOCIATED WITH EMBEDDED SQL
    STATEMENTS. THIS ENSURES THAT Oracle DOES NOT KEEP PARSE
    LOCKS ON TABLES, SO THAT SUBSEQUENT DATA MANIPULATION
   OPERATIONS ON THOSE TABLES DO NOT RESULT IN PARSE-LOCK
   ERRORS.
    EXEC ORACLE OPTION (RELEASE CURSOR=YES) END-EXEC.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(10) VALUE "SCOTT".
01 PASSWD PIC X(10) VALUE "TIGER".
01 DYNSTMT PIC X(80).
   EXEC SQL END DECLARE SECTION END-EXEC.
01 UPDATESTMT PIC X(40).
01 SEARCH-COND PIC X(40).
    DISPLAY "ENTER A SEARCH CONDITION FOR STATEMENT:".
   MOVE "UPDATE EMP SET COMM = 500 WHERE " TO UPDATESTMT.
    DISPLAY UPDATESTMT.
   ACCEPT SEARCH-COND.
    Concatenate SEARCH-COND to UPDATESTMT and store result
    in DYNSTMT.
    STRING UPDATESTMT DELIMITED BY SIZE
       SEARCH-COND DELIMITED BY SIZE INTO DYNSTMT.
    EXEC SQL EXECUTE IMMEDIATE : DYNSTMT END-EXEC.
```



## 9.8 Sample Program 6: Dynamic SQL Method 1

This program uses dynamic SQL Method 1 to create a table, insert a row, commit the insert, then drop the table.

IDENTIFICATION DIVISION.
PROGRAM-ID. DYNSQL1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

- INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
- \* THROUGH WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS
- \* INFORMATION AVAILABLE TO THE PROGRAM.

EXEC SQL INCLUDE SQLCA END-EXEC.

- \* INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
- THROUGH WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS
- \* INFORMATION AVAILABLE TO THE PROGRAM.

EXEC SQL INCLUDE ORACA END-EXEC.

- \* THE OPTION ORACA=YES MUST BE SPECIFIED TO ENABLE USE OF
- \* THE ORACA.

EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

- \* THE RELEASE CURSOR=YES OPTION INSTRUCTS PRO\*COBOL TO
- \* RELEASE IMPLICIT CURSORS ASSOCIATED WITH EMBEDDED SQL
- \* STATEMENTS. THIS ENSURES THAT ORACLE DOES NOT KEEP PARSE
- \* LOCKS ON TABLES, SO THAT SUBSEQUENT DATA MANIPULATION
- \* OPERATIONS ON THOSE TABLES DO NOT RESULT IN PARSE-LOCK
- \* ERRORS.

EXEC ORACLE OPTION (RELEASE CURSOR=YES) END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

- 01 USERNAME PIC X(10) VALUE "SCOTT".
- 01 PASSWD PIC X(10) VALUE "TIGER".
- 01 DYNSTMT PIC X(80) VARYING. EXEC SQL END DECLARE SECTION END-EXEC.
- \* DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
- 01 ORASLNRD PIC 9(9).

PROCEDURE DIVISION.

MAIN.

\* BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS. EXEC SQL WHENEVER SQLERROR GOTO SQLERROR END-EXEC.



```
* SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN ERROR
```

\* OCCURS.

MOVE 1 TO ORASTXTF.

\* CONNECT TO ORACLE.

EXEC SQL

CONNECT : USERNAME IDENTIFIED BY : PASSWD

END-EXEC.

DISPLAY " ".

DISPLAY "CONNECTED TO ORACLE AS USER: " WITH NO ADVANCING.

DISPLAY USERNAME.

DISPLAY " ".

- \* EXECUTE A STRING LITERAL TO CREATE THE TABLE. HERE, YOU
- \* GENERALLY USE A STRING VARIABLE INSTEAD OF A LITERAL, AS IS
- \* DONE LATER IN THIS PROGRAM. BUT, YOU CAN USE A LITERAL IF
- \* YOU WISH.

DISPLAY "CREATE TABLE DYN1 (COL1 CHAR(4))".

DISPLAY " ".

EXEC SQL EXECUTE IMMEDIATE

"CREATE TABLE DYN1 (COL1 CHAR(4))"

END-EXEC.

- \* ASSIGN A SQL STATEMENT TO THE VARYING STRING DYNSTMT.
- \* SET THE -LEN PART TO THE LENGTH OF THE -ARR PART.

  MOVE "INSERT INTO DYN1 VALUES ('TEST')" TO DYNSTMT-ARR.

  MOVE 36 TO DYNSTMT-LEN.

  DISPLAY DYNSTMT-ARR.

  DISPLAY " ".
- \* EXECUTE DYNSTMT TO INSERT A ROW. THE SQL STATEMENT IS A
- \* STRING VARIABLE WHOSE CONTENTS THE PROGRAM MAY DETERMINE
- \* AT RUN TIME.

EXEC SQL EXECUTE IMMEDIATE : DYNSTMT END-EXEC.

\* COMMIT THE INSERT.

EXEC SQL COMMIT WORK END-EXEC.

\* CHANGE DYNSTMT AND EXECUTE IT TO DROP THE TABLE.

MOVE "DROP TABLE DYN1" TO DYNSTMT-ARR.

MOVE 19 TO DYNSTMT-LEN.

DISPLAY DYNSTMT-ARR.

DISPLAY " ".

EXEC SQL EXECUTE IMMEDIATE : DYNSTMT END-EXEC.

COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.

EXEC SQL COMMIT RELEASE END-EXEC.

DISPLAY "HAVE A GOOD DAY!".

DISPLAY " ".

STOP RUN.

#### SQLERROR.

- \* ORACLE ERROR HANDLER. PRINT DIAGNOSTIC TEXT CONTAINING
- \* ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF ERROR. DISPLAY SQLERRMC.

DISPLAY "IN ", ORASTXTC.

MOVE ORASLNR TO ORASLNRD.

DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.



- \* DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
- \* SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.

  EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
- \* ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE. EXEC SQL ROLLBACK RELEASE END-EXEC. STOP RUN.

## 9.9 Using Method 2

What Method 1 does in one step, Method 2 does in two. The dynamic SQL statement, which cannot be a query, is first prepared (named and parsed), then executed.

With Method 2, the SQL statement can contain place-holders for input host variables and indicator variables. You can PREPARE the SQL statement once, then EXECUTE it repeatedly using different values of the host variables. Also, if you have not specified MODE=ANSI, you need *not* re-prepare the SQL statement after a COMMIT or ROLLBACK (unless you log off and reconnect).

The syntax of the PREPARE statement follows:

```
EXEC SQL PREPARE STATEMENT-NAME
  FROM { :HOST-STRING | STRING-LITERAL }
END-EXEC.
```

PREPARE parses the SQL statement and gives it a name.

STATEMENT-NAME is an identifier used by the precompiler, *not* a host or program variable, and should not be declared in a COBOL statement. It simply designates the prepared statement you want to EXECUTE.

The syntax of the EXECUTE statement is

```
EXEC SQL EXECUTE STATEMENT-NAME [USING HOST-VARIABLE-LIST] END-EXEC.
```

where *HOST-VARIABLE-LIST* stands for the following syntax:

```
:HOST-VAR1[:INDICATOR1] [, HOST-VAR2[:INDICATOR2], ...]
```

EXECUTE executes the parsed SQL statement, using the values supplied for each input host variable. In the following example, the input SQL statement contains the place-holder n:

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

01 EMP-NUMBER PIC S9(4) COMP VALUE ZERO.
...

01 DELETE-STMT PIC X(120) VALUE SPACES.
...

EXEC SQL END DECLARE SECTION END-EXEC.

01 WHERE-STMT PIC X(40).
01 SEARCH-COND PIC X(40).
...

MOVE 'DELETE FROM EMP WHERE EMPNO = :N AND ' TO WHERE-STMT.
DISPLAY 'Complete this statement's search condition:'.
DISPLAY WHERE-STMT.
ACCEPT SEARCH-COND.
Concatenate SEARCH-COND to WHERE-STMT and store in DELETE-STMT
```



```
STRING WHERE-STMT DELIMITED BY SIZE

SEARCH-COND DELIMITED BY SIZE INTO

DELETE-STMT.

EXEC SQL PREPARE SQLSTMT FROM :DELETE-STMT END-EXEC.

LOOP.

DISPLAY 'Enter employee number: ' WITH NO ADVANCING.

ACCEPT EMP-NUMBER.

IF EMP-NUMBER = 0

GO TO NEXT.

EXEC SQL EXECUTE SQLSTMT USING :EMP-NUMBER END-EXEC.

NEXT.
```

With Method 2, you must know the datatypes of input host variables at precompile time. In the last example, *EMP-NUMBER* was declared as type PIC S9(4) COMP. It could also have been declared as type PIC X(4) or COMP-1, because Oracle supports all these datatype conversions to the NUMBER internal datatype.

### 9.9.1 The USING Clause

When the SQL statement EXECUTE is completed, input host variables in the USING clause replace corresponding place-holders in the prepared dynamic SQL statement.

Every place-holder in the dynamic SQL statement after PREPARE must correspond to a host variable in the USING clause. So, if the same place-holder appears two or more times in the statement after PREPARE, each appearance must correspond to a host variable in the USING clause. If one of the host variables in the USING clause is an array, all must be arrays. Otherwise, only one record is then processed.

The names of the place-holders need not match the names of the host variables. However, the order of the place-holders in the dynamic SQL statement after PREPARE must match the order of corresponding host variables in the USING clause.

To specify NULLs, you can associate indicator variables with host variables in the USING clause.

#### **Related Topics**

Using Indicator Variables

## 9.10 Sample Program 7: Dynamic SQL Method 2

This program uses dynamic SQL Method 2 to insert two rows into the EMP table and then delete them.

- \* INCLUDE THE SQL COMMUNICATIONS AREA, A STRUCTURE THROUGH
- \* WHICH ORACLE MAKES RUNTIME STATUS INFORMATION (SUCH AS ERROR

- \* CODES, WARNING FLAGS, AND DIAGNOSTIC TEXT) AVAILABLE TO THE
- \* PROGRAM.

EXEC SQL INCLUDE SQLCA END-EXEC.

- \* INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE THROUGH
- \* WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS INFORMATION
- \* AVAILABLE TO THE PROGRAM. EXEC SQL INCLUDE ORACA END-EXEC.
- \* THE OPTION ORACA=YES MUST BE SPECIFIED TO ENABLE USE OF
- \* THE ORACA.

EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

- 01 USERNAME PIC X(10) VALUE "SCOTT".
- 01 PASSWD PIC X(10) VALUE "TIGER".
- 01 DYNSTMT PIC X(80) VARYING.
- 01 EMPNO PIC S9(4) COMPUTATIONAL VALUE 1234.
- 01 DEPTNO1 PIC S9(4) COMPUTATIONAL VALUE 10.
- 01 DEPTNO2 PIC S9(4) COMPUTATIONAL VALUE 20. EXEC SQL END DECLARE SECTION END-EXEC.
- \* DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
- 01 EMPNOD PIC 9(4).
- 01 DEPTNO1D PIC 9(2).
- 01 DEPTNO2D PIC 9(2).
- 01 ORASLNRD PIC 9(9).

PROCEDURE DIVISION.

MAIN.

- \* BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS. EXEC SQL WHENEVER SQLERROR GOTO SQLERROR END-EXEC.
- \* SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN ERROR
- \* OCCURS.

MOVE 1 TO ORASTXTF.

\* CONNECT TO ORACLE.

EXEC SQL

CONNECT : USERNAME IDENTIFIED BY : PASSWD

END-EXEC.

DISPLAY " ".

DISPLAY "CONNECTED TO ORACLE.".

DISPLAY " ".

- \* ASSIGN A SQL STATEMENT TO THE VARYING STRING DYNSTMT. BOTH
- \* THE ARRAY AND THE LENGTH PARTS MUST BE SET PROPERLY. NOTE
- \* THAT THE STATEMENT CONTAINS TWO HOST VARIABLE PLACEHOLDERS,
- \* V1 AND V2, FOR WHICH ACTUAL INPUT HOST VARIABLES MUST BE
- \* SUPPLIED AT EXECUTE TIME.

MOVE "INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:V1, :V2)" TO DYNSTMT-ARR.

MOVE 49 TO DYNSTMT-LEN.

- \* DISPLAY THE SQL STATEMENT AND ITS CURRENT INPUT HOST
- \* VARIABLES.

DISPLAY DYNSTMT-ARR.

MOVE EMPNO TO EMPNOD.

MOVE DEPTNO1 TO DEPTNO1D.

DISPLAY " V1 = ", EMPNOD, " V2 = ", DEPTNO1D.

- \* THE PREPARE STATEMENT ASSOCIATES A STATEMENT NAME WITH A
- \* STRING CONTAINING A SQL STATEMENT. THE STATEMENT NAME IS
- \* A SQL IDENTIFIER, NOT A HOST VARIABLE, AND THEREFORE DOES
- \* NOT APPEAR IN THE DECLARE SECTION.
- \* A SINGLE STATEMENT NAME MAY BE PREPARED MORE THAN ONCE,
- \* OPTIONALLY FROM A DIFFERENT STRING VARIABLE.

  EXEC SQL PREPARE S FROM :DYNSTMT END-EXEC.
- \* THE EXECUTE STATEMENT EXECUTES A PREPARED SQL STATEMENT
- \* USING THE SPECIFIED INPUT HOST VARIABLES, WHICH ARE
- \* SUBSTITUTED POSITIONALLY FOR PLACEHOLDERS IN THE PREPARED
- \* STATEMENT. FOR EACH OCCURRENCE OF A PLACEHOLDER IN THE
- \* STATEMENT THERE MUST BE A VARIABLE IN THE USING CLAUSE.
- \* THAT IS, IF A PLACEHOLDER OCCURS MULTIPLE TIMES IN THE
- \* STATEMENT, THE CORRESPONDING VARIABLE MUST APPEAR
- \* MULTIPLE TIMES IN THE USING CLAUSE. THE USING CLAUSE MAY
- \* BE OMITTED ONLY IF THE STATEMENT CONTAINS NO PLACEHOLDERS.
- \* A SINGLE PREPARED STATEMENT MAY BE EXECUTED MORE THAN ONCE,
- \* OPTIONALLY USING DIFFERENT INPUT HOST VARIABLES.
  EXEC SQL EXECUTE S USING :EMPNO, :DEPTNO1 END-EXEC.
- \* INCREMENT EMPNO AND DISPLAY NEW INPUT HOST VARIABLES.
  ADD 1 TO EMPNO.
  MOVE EMPNO TO EMPNOD.
  MOVE DEPTNO2 TO DEPTNO2D.
  DISPLAY " V1 = ", EMPNOD, " V2 = ", DEPTNO2D.
- \* REEXECUTE S TO INSERT THE NEW VALUE OF EMPNO AND A
- \* DIFFERENT INPUT HOST VARIABLE, DEPTNO2. A REPREPARE IS NOT
- \* NECESSARY.

  EXEC SQL EXECUTE S USING :EMPNO, :DEPTNO2 END-EXEC.
- \* ASSIGN A NEW VALUE TO DYNSTMT.

  MOVE "DELETE FROM EMP WHERE DEPTNO = :V1 OR DEPTNO = :V2"

  TO DYNSTMT-ARR.

  MOVE 50 TO DYNSTMT-LEN.
- $^\star$  DISPLAY THE NEW SQL STATEMENT AND ITS CURRENT INPUT HOST
- \* VARIABLES.

DISPLAY DYNSTMT-ARR.

DISPLAY " V1 = ", DEPTNO1D, " V2 = ", DEPTNO2D.

- \* REPREPARE S FROM THE NEW DYNSTMT.
  EXEC SQL PREPARE S FROM :DYNSTMT END-EXEC.
- \* EXECUTE THE NEW S TO DELETE THE TWO ROWS PREVIOUSLY
- \* INSERTED.

EXEC SQL EXECUTE S USING : DEPTNO1, : DEPTNO2 END-EXEC.

ROLLBACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
EXEC SQL ROLLBACK RELEASE END-EXEC.
DISPLAY " ".
DISPLAY "HAVE A GOOD DAY!".
DISPLAY " ".
STOP RUN.

#### SQLERROR.

- \* ORACLE ERROR HANDLER. PRINT DIAGNOSTIC TEXT CONTAINING
- \* ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF ERROR.

```
DISPLAY SQLERRMC.
DISPLAY "IN ", ORASTXTC.
MOVE ORASLNR TO ORASLNRD.
DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.
```

- \* DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
- \* SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH. EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
- \* ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE. EXEC SQL ROLLBACK RELEASE END-EXEC. STOP RUN.

## 9.11 Using Method 3

Method 3 is similar to Method 2 but combines the PREPARE statement with the statements needed to define and manipulate a cursor. This allows your program to accept and process queries. In fact, if the dynamic SQL statement is a query, you *must* use Method 3 or 4.

For Method 3, the number of columns in the query select list and the number of place-holders for input host variables must be known at precompile time. However, the names of database objects such as tables and columns need not be specified until run time (they cannot duplicate the names of host variables). Clauses that limit, group, and sort query results (such as WHERE, GROUP BY, and ORDER BY) can also be specified at run time.

With Method 3, you use the following sequence of embedded SQL statements:

```
EXEC SQL

PREPARE STATEMENTNAME FROM { :HOST-STRING | STRING-LITERAL }

END-EXEC.

EXEC SQL DECLARE CURSORNAME CURSOR FOR STATEMENTNAME END-EXEC.

EXEC SQL OPEN CURSORNAME [USING HOST-VARIABLE-LIST] END-EXEC.

EXEC SQL FETCH CURSORNAME INTO HOST-VARIABLE-LIST END-EXEC.

EXEC SQL CLOSE CURSORNAME END-EXEC.
```

Now let us look at what each statement does.

### 9.11.1 PREPARE

The PREPARE statement parses the dynamic SQL statement and gives it a name. In the following example, PREPARE parses the query stored in the character string SELECT-STMT and gives it the name SQLSTMT:

```
MOVE 'SELECT MGR, JOB FROM EMP WHERE SAL < :SALARY'
TO SELECT-STMT.

EXEC SQL PREPARE SQLSTMT FROM :SELECT-STMT END-EXEC.
```

Commonly, the query WHERE clause is input from a terminal at run time or is generated by the application.

The identifier *SQLSTMT* is *not* a host or program variable, but must be unique. It designates a particular dynamic *SQL* statement.

The following statement is correct also:



```
EXEC SQL PREPARE SQLSTMT FROM 'SELECT MGR, JOB FROM EMP WHERE SAL < :SALARY'END-EXEC.
```

The following PREPARE statement, which uses the '%' wildcard, is also correct:

```
MOVE "SELECT ENAME FROM TEST WHERE ENAME LIKE 'SMIT%'" TO MY-STMT. EXEC SQL
PREPARE S FROM MY-STMT
END-EXEC.
```

#### 9.11.2 DECLARE

The DECLARE statement defines a cursor by giving it a name and associating it with a specific query. The cursor declaration is local to its precompilation unit. Continuing our example, DECLARE defines a cursor named *EMPCURSOR* and associates it with *SQLSTMT*, as follows:

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR SQLSTMT END-EXEC.
```

The identifiers *SQLSTMT* and *EMPCURSOR* are *not* host or program variables, but must be unique. If you declare two cursors using the same statement name, Pro\*COBOL considers the two cursor names synonymous. For example, if you execute the statements

```
EXEC SQL PREPARE SQLSTMT FROM :SELECT-STMT END-EXEC.
EXEC SQL DECLARE EMPCURSOR FOR SQLSTMT END-EXEC.
EXEC SQL PREPARE SQLSTMT FROM :DELETE-STMT END-EXEC.
EXEC SQL DECLARE DEPCURSOR FOR SQLSTMT END-EXEC.
```

when you OPEN *EMPCURSOR*, you will process the dynamic SQL statement stored in *DELETE-STMT*, not the one stored in *SELECT-STMT*.

### 9.11.3 OPEN

The OPEN statement allocates a cursor, binds input host variables, and executes the query, identifying its active set. OPEN also positions the cursor on the first row in the active set and zeroes the rows-processed count kept by the third element of SQLERRD in the SQLCA. Input host variables in the USING clause replace corresponding place-holders in the PREPAREd dynamic SQL statement.

In our example, OPEN allocates *EMPCURSOR* and assigns the host variable *SALARY* to the WHERE clause, as follows:

```
EXEC SQL OPEN EMPCURSOR USING :SALARY END-EXEC.
```

### 9.11.4 FETCH

The FETCH statement returns a row from the active set, assigns column values in the select list to corresponding host variables in the INTO clause, and advances the cursor to the next row. When no more rows are found, FETCH returns the "no data found" error code to SQLCODE in the SQLCA.

In our example, FETCH returns a row from the active set and assigns the values of columns MGR and JOB to host variables *MGR-NUMBER* and *JOB-TITLE*, as follows:

```
EXEC SQL FETCH EMPCURSOR INTO :MGR-NUMBER,: JOB-TITLE END-EXEC.
```

Host tables can be used with Method 3.



### 9.11.5 CLOSE

The CLOSE statement disables the cursor. Once you CLOSE a cursor, you can no longer FETCH from it. In our example, the CLOSE statement disables *EMPCURSOR*, as follows:

EXEC SQL CLOSE EMPCURSOR END-EXEC.

## 9.12 Sample Program 8: Dynamic SQL Method 3

This program uses dynamic SQL Method 3 to retrieve the names of all employees in a given department from the EMP table.

```
************
* Sample Program 8: Dynamic SQL Method 3
* This program uses dynamic SQL Method 3 to retrieve the names *
* of all employees in a given department from the EMP table.
****************
IDENTIFICATION DIVISION.
PROGRAM-ID. DYNSQL3.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    INCLUDE THE SQL COMMUNICATIONS AREA, A STRUCTURE THROUGH
    WHICH ORACLE MAKES RUNTIME STATUS INFORMATION (SUCH AS ERROR
   CODES, WARNING FLAGS, AND DIAGNOSTIC TEXT) AVAILABLE TO THE
    PROGRAM.
    EXEC SQL INCLUDE SQLCA END-EXEC.
    INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE THROUGH
    WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS INFORMATION
    AVAILABLE TO THE PROGRAM.
    EXEC SOL INCLUDE ORACA END-EXEC.
    THE ORACA=YES OPTION MUST BE SPECIFIED TO ENABLE USE OF
    THE ORACA.
    EXEC ORACLE OPTION (ORACA=YES) END-EXEC.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(10) VALUE "SCOTT".
01 PASSWD PIC X(10) VALUE "TIGER".
01 DYNSTMT PIC X(80) VARYING.
01 ENAME PIC X(10).
01 DEPTNO PIC S9999 COMPUTATIONAL VALUE 10.
    EXEC SQL END DECLARE SECTION END-EXEC.
  DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
01 DEPTNOD PIC 9(2).
01 ENAMED PIC X(10).
01 SQLERRD3 PIC 9(2).
01 ORASLNRD PIC 9(4).
PROCEDURE DIVISION.
MAIN.
```



- \* BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS. EXEC SQL WHENEVER SQLERROR GO TO SQLERROR END-EXEC.
- \* SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN ERROR
- \* OCCURS.

MOVE 1 TO ORASTXTF.

\* CONNECT TO ORACLE.

EXEC SOL

CONNECT : USERNAME IDENTIFIED BY : PASSWD

END-EXEC.

DISPLAY " ".

DISPLAY "CONNECTED TO ORACLE.".

DISPLAY " ".

- \* ASSIGN A SQL QUERY TO THE VARYING STRING DYNSTMT. BOTH THE
- \* ARRAY AND THE LENGTH PARTS MUST BE SET PROPERLY. NOTE THAT
- \* THE STATEMENT CONTAINS ONE HOST VARIABLE PLACEHOLDER, V1,
- \* FOR WHICH AN ACTUAL INPUT HOST VARIABLE MUST BE SUPPLIED
- \* AT OPEN TIME.

MOVE "SELECT ENAME FROM EMP WHERE DEPTNO = :V1"
TO DYNSTMT-ARR.

MOVE 40 TO DYNSTMT-LEN.

- \* DISPLAY THE SQL STATEMENT AND ITS CURRENT INPUT HOST
- \* VARIABLE.

DISPLAY DYNSTMT-ARR.

MOVE DEPTNO TO DEPTNOD.

DISPLAY " V1 = ", DEPTNOD.

DISPLAY " ".

DISPLAY "EMPLOYEE".

DISPLAY "----".

- \* THE PREPARE STATEMENT ASSOCIATES A STATEMENT NAME WITH A
- \* STRING CONTAINING A SELECT STATEMENT. THE STATEMENT NAME,
- \* WHICH MUST BE UNIQUE, IS A SQL IDENTIFIER, NOT A HOST
- \* VARIABLE, AND SO DOES NOT APPEAR IN THE DECLARE SECTION. EXEC SQL PREPARE S FROM :DYNSTMT END-EXEC.
- \* THE DECLARE STATEMENT ASSOCIATES A CURSOR WITH A PREPARED
- \* STATEMENT. THE CURSOR NAME, LIKE THE STATEMENT NAME, DOES
- \* NOT APPEAR IN THE DECLARE SECTION.

  EXEC SQL DECLARE C CURSOR FOR S END-EXEC.
- \* THE OPEN STATEMENT EVALUATES THE ACTIVE SET OF THE PREPARED
- \* QUERY USING THE SPECIFIED INPUT HOST VARIABLES, WHICH ARE
- \* SUBSTITUTED POSITIONALLY FOR PLACEHOLDERS IN THE PREPARED
- \* QUERY. FOR EACH OCCURRENCE OF A PLACEHOLDER IN THE
- \* STATEMENT THERE MUST BE A VARIABLE IN THE USING CLAUSE.
- \* THAT IS, IF A PLACEHOLDER OCCURS MULTIPLE TIMES IN THE
- \* STATEMENT, THE CORRESPONDING VARIABLE MUST APPEAR MULTIPLE
- \* TIMES IN THE USING CLAUSE. THE USING CLAUSE MAY BE
- OMITTED ONLY IF THE STATEMENT CONTAINS NO PLACEHOLDERS.
- \* OPEN PLACES THE CURSOR AT THE FIRST ROW OF THE ACTIVE SET
- \* IN PREPARATION FOR A FETCH.
- \* A SINGLE DECLARED CURSOR MAY BE OPENED MORE THAN ONCE,
- \* OPTIONALLY USING DIFFERENT INPUT HOST VARIABLES. EXEC SQL OPEN C USING :DEPTNO END-EXEC.
- \* BRANCH TO PARAGRAPH NOTFOUND WHEN ALL ROWS HAVE BEEN



\* RETRIEVED.

EXEC SQL WHENEVER NOT FOUND GO TO NOTFOUND END-EXEC.

#### GETROWS.

- \* THE FETCH STATEMENT PLACES THE SELECT LIST OF THE CURRENT
- \* ROW INTO THE VARIABLES SPECIFIED BY THE INTO CLAUSE, THEN
- \* ADVANCES THE CURSOR TO THE NEXT ROW. IF THERE ARE MORE
- \* SELECT-LIST FIELDS THAN OUTPUT HOST VARIABLES, THE EXTRA
- \* FIELDS ARE NOT RETURNED. SPECIFYING MORE OUTPUT HOST
- \* VARIABLES THAN SELECT-LIST FIELDS RESULTS IN AN ORACLE ERROR.
  EXEC SQL FETCH C INTO :ENAME END-EXEC.
  MOVE ENAME TO ENAMED.
  DISPLAY ENAMED.
- \* LOOP UNTIL NOT FOUND CONDITION IS DETECTED. GO TO GETROWS.

```
NOTFOUND.
```

```
MOVE SQLERRD(3) TO SQLERRD3.
DISPLAY " ".
DISPLAY "QUERY RETURNED ", SQLERRD3, " ROW(S).".
```

- \* THE CLOSE STATEMENT RELEASES RESOURCES ASSOCIATED WITH THE
- \* CURSOR.

EXEC SQL CLOSE C END-EXEC.

\* COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
EXEC SQL COMMIT RELEASE END-EXEC.
DISPLAY " ".
DISPLAY "HAVE A GOOD DAY!".
DISPLAY " ".
STOP RUN.

#### ${\tt SQLERROR}$ .

- \* ORACLE ERROR HANDLER. PRINT DIAGNOSTIC TEXT CONTAINING
- \* ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF ERROR.
  DISPLAY SQLERRMC.
  DISPLAY "IN ", ORASTXTC.
  MOVE ORASLNR TO ORASLNRD.
  DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.
- \* DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
- \* SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
  EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
- \* RELEASE RESOURCES ASSOCIATED WITH THE CURSOR. EXEC SQL CLOSE C END-EXEC.
- \* ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE. EXEC SQL ROLLBACK RELEASE END-EXEC. STOP RUN.

## 9.13 Using Oracle Method 4

This section gives only an overview. For details, see Oracle Dynamic SQL: Method 4.

LOBs are not supported in Oracle Method 4. Use ANSI dynamic SQL for LOB applications and all other new applications.



There is a kind of dynamic SQL statement that your program cannot process using Method 3. When the number of select-list items or place-holders for input host variables is unknown until run time, your program must use a descriptor. A *descriptor* is an area of memory used by your program and Oracle to hold a complete description of the variables in a dynamic SQL statement.

Recall that for a multi-row query, you FETCH selected column values INTO a list of declared output host variables. If the select list is unknown, the host-variable list cannot be established at precompile time by the INTO clause. For example, you know the following query returns two column values:

```
EXEC SQL

SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :DEPT-NUMBER END-EXEC.
```

However, if you let the user define the select list, you might not know how many column values the guery will return.

## 9.13.1 Need for the SQLDA

To process this kind of dynamic query, your program must issue the DESCRIBE SELECT LIST command and declare a data structure called the SQL Descriptor Area (SQLDA). Because it holds descriptions of columns in the query select list, this structure is also called a *select descriptor*.

Likewise, if a dynamic SQL statement contains an unknown number of place-holders for input host variables, the host-variable list cannot be established at precompile time by the USING clause.

To process the dynamic SQL statement, your program must issue the DESCRIBE BIND VARIABLES command and declare another kind of SQLDA called a *bind descriptor* to hold descriptions of the place-holders for the input host variables. (Input host variables are also called *bind variables*.)

If your program has more than one active SQL statement (it might have used OPEN for two or more cursors, for example), each statement must have its own SQLDAs statement. However, non-concurrent cursors can reuse SQLDAs. There is no set limit on the number of SQLDAs in a program.

## 9.13.2 The DESCRIBE Statement

DESCRIBE initializes a descriptor to hold descriptions of select-list items or input host variables.

If you supply a select descriptor, the DESCRIBE SELECT LIST statement examines each select-list item in a prepared dynamic query to determine its name, datatype, constraints, length, scale, and precision. It then stores this information in the select descriptor.

If you supply a bind descriptor, the DESCRIBE BIND VARIABLES statement examines each place-holder in a prepared dynamic SQL statement to determine its name, length, and the datatype of its associated input host variable. It then stores this information in the bind descriptor for your use. For example, you might use place-holder names to prompt the user for the values of input host variables.



### 9.13.3 SQLDA Contents

A SQLDA is a host-program data structure that holds descriptions of select-list items or input host variables.

Though SQLDAs differ among host languages, a generic select SQLDA contains the following information about a query select list:

- Maximum number of columns that can be DESCRIBEd
- Actual number of columns found by DESCRIBE
- Addresses of buffers to store column values
- Lengths of column values
- Datatypes of column values
- Addresses of indicator-variable values
- Addresses of buffers to store column names
- Sizes of buffers to store column names
- Current lengths of column names

A generic bind SQLDA contains the following information about the input host variables in a SQL statement:

- Maximum number of place-holders that can be DESCRIBEd
- Actual number of place-holders found by DESCRIBE
- Addresses of input host variables
- Lengths of input host variables
- Datatypes of input host variables
- Addresses of indicator variables
- Addresses of buffers to store place-holder names
- Sizes of buffers to store place-holder names
- Current lengths of place-holder names
- Addresses of buffers to store indicator-variable names
- Sizes of buffers to store indicator-variable names
- Current lengths of indicator-variable names

## 9.13.4 Implementing Method 4

With Method 4, you generally use the following sequence of embedded SQL statements:

```
EXEC SQL

PREPARE STATEMENT-NAME

FROM { :HOST-STRING | STRING-LITERAL }

END-EXE

EXEC SQL

DECLARE CURSOR-NAME CURSOR FOR STATEMENT-NAME

END-EXEC.
```



```
EXEC SQL
   DESCRIBE BIND VARIABLES FOR STATEMENT-NAME
   INTO BIND-DESCRIPTOR-NAME
END-EXEC.
EXEC SOL
   OPEN CURSOR-NAME
   [USING DESCRIPTOR BIND-DESCRIPTOR-NAME]
END-EXEC.
EXEC SQL
   DESCRIBE [SELECT LIST FOR] STATEMENT-NAME
   INTO SELECT-DESCRIPTOR-NAME
END-EXEC.
EXEC SOL
   FETCH CURSOR-NAME
   USING DESCRIPTOR SELECT-DESCRIPTOR-NAME
END-EXEC.
EXEC SQL CLOSE CURSOR-NAME END-EXEC.
```

Select and bind descriptors need not work in tandem. If the number of columns in a query select list is known, but the number of place-holders for input host variables is unknown, you can use the Method 4 OPEN statement with the following Method 3 FETCH statement:

```
EXEC SQL FETCH EMPCURSOR INTO : HOST-VARIABLE-LIST END-EXEC.
```

Conversely, if the number of place-holders for input host variables is known, but the number of columns in the select list is unknown, you can use the following Method 3 OPEN statement with the Method 4 FETCH statement:

```
EXEC SQL OPEN CURSORNAME [USING HOST-VARIABLE-LIST] END-EXEC.
```

Note that EXECUTE can be used for non-queries with Method 4.

## 9.14 Using the DECLARE STATEMENT Statement

With Methods 2, 3, and 4, you might need to use the statement

```
EXEC SQL [AT dbname] DECLARE statementname STATEMENT END-EXEC.
```

where *dbname* and *statementname* are identifiers used by Pro\*COBOL, *not* host or program variables.

DECLARE STATEMENT declares the name of a dynamic SQL statement so that the statement can be referenced by PREPARE, EXECUTE, DECLARE CURSOR, and DESCRIBE. It is required if you want to execute the dynamic SQL statement at a nondefault database. An example using Method 2 follows:

```
EXEC SQL AT remotedb DECLARE sqlstmt STATEMENT END-EXEC. EXEC SQL PREPARE sqltmt FROM :sqlstring END-EXEC. EXEC SQL EXECUTE sqlstmt END-EXEC.
```

In the example, *remotedb* tells Oracle where to EXECUTE the SQL statement.

With Methods 3 and 4, DECLARE STATEMENT is also required if the DECLARE CURSOR statement precedes the PREPARE statement, as shown in the following example:

```
EXEC SQL DECLARE sqlstmt STATEMENT END-EXEC.

EXEC SQL DECLARE empcursor CURSOR FOR sqlstmt END-EXEC.

EXEC SQL PREPARE sqlstmt FROM :sqlstring END-EXEC.
```

The usual sequence of statements is



```
EXEC SQL PREPARE sqlstmt FROM :sqlstring END-EXEC. EXEC SQL DECLARE empcursor CURSOR FOR sqlstmt END-EXEC.
```

## 9.15 Using Host Tables

Usage of host tables in static and dynamic SQL is similar. For example, to use input host tables with dynamic SQL Method 2, use the syntax

```
EXEC SQL EXECUTE statementname USING : HOST-TABLE-LIST END-EXEC.
```

where *HOST-TABLE-LIST* contains one or more host tables. With Method 3, use the following syntax:

```
OPEN cursorname USING : HOST-TABLE-LIST END-EXEC.
```

To use output host tables with Method 3, use the following syntax:

```
FETCH cursorname INTO : HOST-TABLE-LIST END-EXEC.
```

With Method 4, you must use the optional FOR clause to tell Oracle the size of your input or output host table. To learn how this is done, see your host-language supplement.

## 9.16 Using PL/SQL

Pro\*COBOL treats a PL/SQL block like a single SQL statement. So, like a SQL statement, a PL/SQL block can be stored in a string host variable or literal. When you store the PL/SQL block in the string, omit the keywords EXEC SQL EXECUTE, the keyword END-EXEC, and the statement terminator.

However, there are two differences in the way Pro\*COBOL handles SQL and PL/SQL:

- All PL/SQL host variables should be treated in the same way as input host variables regardless of whether they are input or output host variables (or both).
- You cannot FETCH from a PL/SQL block because it might contain any number of SQL statements. However, you can implement similar functionality by using cursor variables.

### 9.16.1 With Method 1

If the PL/SQL block contains no host variables, you can use Method 1 to EXECUTE the PL/SQL string in the usual way.

### 9.16.2 With Method 2

If the PL/SQL block contains a known number of input and output host variables, you can use Method 2 to PREPARE and EXECUTE the PL/SQL string in the usual way.

You must put *all* host variables in the USING clause. Once the PL/SQL string EXECUTE is completed, host variables in the USING clause replace corresponding place-holders in the string after PREPARE. Though Pro\*COBOL treats all PL/SQL host variables as input host variables, values are assigned correctly. Input (program) values are assigned to input host variables, and output (column) values are assigned to output host variables.



Every place-holder in the PL/SQL string after PREPARE must correspond to a host variable in the USING clause. So, if the same place-holder appears two or more times in the PREPAREd string, each appearance must correspond to a host variable in the USING clause.

### 9.16.3 With Method 3

Methods 2 and 3 are the same except that Method 3 allows completion of a FETCH. Since you cannot FETCH from a PL/SQL block, use Method 2 instead.

### 9.16.4 With Method 4

If the PL/SQL block contains an unknown number of input or output host variables, you must use Method 4.

To use Method 4, you set up one bind descriptor for all the input and output host variables. Executing DESCRIBE BIND VARIABLES stores information about input *and* output host variables in the bind descriptor. Because you refer to all PL/SQL host variables with the methods associated with input host variables, executing DESCRIBE SELECT LIST has no effect.

The use of bind descriptors with Method 4 is detailed in your host-language supplement.

Note that in dynamic SQL Method 4, a host array cannot be bound to a PL/SQL procedure with a parameter of type "table."

### 9.16.5 Caution

Do not use ANSI-style Comments (- - ...) in a PL/SQL block that will be processed dynamically because end-of-line characters are ignored. As a result, ANSI-style Comments extend to the end of the block, not just to the end of a line. Instead, use C-style Comments (/\* ... \*/).

## 9.17 Dynamic SQL Statement Caching

Statement caching refers to the feature that provides and manages a cache of statements for each session. In the server, it means that cursors are ready to be used without the need to parse the statement again. Statement caching can be enabled in the precompiler applications, which will help in the performance improvement of all applications that rely on the dynamic SQL statements. The performance improvement is achieved by removing the overhead of parsing the dynamic statements on reuse. The precompiler application user can obtain this performance improvement using a new command line option, stmt\_cache (for the statement cache size), which will enable the statement caching of the dynamic statements. By enabling the new option, the statement cache will be created at session creation time. The caching is only applicable for the dynamic statements and the cursor cache for the static statements co-exists with the new feature.

The command line option stmt\_cache can be given any value in the range of 0 to 65535. Statement caching is disabled by default (value 0). The stmt\_cache option can be set to hold the anticipated number of distinct dynamic SQL statements in the application.

#### Example 9-1 Using the stmt\_cache Option

This example demonstrates the use of the stmt\_cache option. In this program, you insert rows into a table and select the inserted rows by using the cursor in the loop. When the



stmt\_cache option is used to precompile this program, the performance increases compared to a normal precompilation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

```
* stmtcache:
* NOTE:
* When this program is used to measure the performance with and *
* without stmt cache option, do the following changes in the
* program,
* 1. Increase ROWSCNT to high value, say 10000.
^{\star} 2. Remove all the DISPLAY statements, usually which comsumes \ ^{\star}
* significant portion of the total program execution time.
IDENTIFICATION DIVISION.
 PROGRAM-ID. stmtcache.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* EMBEDDED COBOL (file "STMTCACHE.PCO")
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 USERNAME PIC X(10) VARYING.
 01 PASSWD
                     PIC X(10) VARYING.
 01 DYNSTMT PIC X(100) VARYING.
 01 DYNSTMT2 PIC X(100) VARYING.
 01 ENAME PIC X(10).
 01 COMM PIC X(9).
     EXEC SQL END DECLARE SECTION END-EXEC.
 01 ROWSCNT PIC 9(4) COMP VALUE 10.
 01 LOOPNO PIC 9(4).
 01 STRINGFIELDS.
     02 STR PIC X(18) VARYING.
     EXEC SQL INCLUDE SQLCA END-EXEC.
 PROCEDURE DIVISION.
 BEGIN-PGM.
    EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR END-EXEC.
    PERFORM LOGON.
    MOVE "INSERT INTO BONUS (ENAME, COMM) VALUES (:A,:B)"
         TO DYNSTMT-ARR.
    MOVE 53 TO DYNSTMT-LEN.
     DISPLAY "Inserts ", ROWSCNT, " rows into BONUS table.".
     PERFORM INSDATA VARYING LOOPNO FROM 1 BY 1
            UNTIL LOOPNO > ROWSCNT.
     DISPLAY " ".
     DISPLAY "Fetches the inserted rows from BONUS.".
     DISPLAY " ENAME COMM".
```

```
MOVE "SELECT ENAME, COMM FROM BONUS WHERE COMM=:A"
        TO DYNSTMT2-ARR.
    MOVE 43 TO DYNSTMT2-LEN.
    MOVE 1 TO LOOPNO.
* Loops for preparing and fetching ROWSCNT number of times
FETCHDATA.
^{\star} Do the prepare in the loop so that the advantage of
* stmt caching is visible
    EXEC SQL PREPARE S2 FROM : DYNSTMT2 END-EXEC.
    EXEC SQL DECLARE C1 CURSOR FOR S2
    END-EXEC.
    EXEC SQL OPEN C1 USING :LOOPNO END-EXEC.
    EXEC SQL WHENEVER NOT FOUND GO TO NOTFOUND END-EXEC.
GETROWS.
* Close the cursor so that the reparsing is not required for
* stmt cache
    EXEC SQL FETCH C1 INTO :ENAME, :COMM
    END-EXEC.
    DISPLAY ENAME, COMM.
    GO TO GETROWS.
NOTFOUND.
    EXEC SQL CLOSE C1 END-EXEC.
    COMPUTE LOOPNO = LOOPNO + 1.
    IF LOOPNO <= ROWSCNT THEN
      GO TO FETCHDATA
     END-IF.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
     STOP RUN.
LOGON.
    MOVE "scott" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "tiger" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL
         CONNECT : USERNAME IDENTIFIED BY : PASSWD
     END-EXEC.
* Populates the host variable and insert into the table
INSDATA.
     EXEC SQL PREPARE S1 FROM : DYNSTMT END-EXEC.
    MOVE " " TO STR.
    STRING "EMP ", LOOPNO INTO STR
    END-STRING.
    MOVE STR TO ENAME.
    MOVE LOOPNO TO COMM.
    EXEC SQL EXECUTE S1 USING : ENAME, : COMM END-EXEC.
* HANDLES SQL ERROR CONDITIONS
SOL-ERROR.
```

```
EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
DISPLAY " ".
DISPLAY "ORACLE ERROR DETECTED:".
DISPLAY " ".
DISPLAY SQLERRMC.
EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
STOP RUN.
```

## 9.18 About Boolean Data Type Support

Using Dynamic SQL, applications can build SQL statements at runtime and take explicit control over data type comversions, With the support for boolean data type, applications can set, get bind, or define BOOLEAN column metadata using these statements:

- DESCRIBE BIND VARIABLES
- DESCRIBE SELECT LIST

#### DESCRIBE BIND VARIABLES

The DESCRIBE BIND VARIABLES statement puts descriptions of bind placeholders into a bind descriptor:

#### DESCRIBE SELECT LIST

If a dynamic SQL statement is a <code>SELECT</code> query, then a <code>DESCRIBE</code> <code>SELECT</code> <code>LIST</code> statement can be used. The <code>DESCRIBE</code> <code>SELECT</code> <code>LIST</code> statement sets the length and data type of each select-list item:



## Part II

# **Applications**

#### Part II contains the following chapters:

- ANSI Dynamic SQL
- Oracle Dynamic SQL: Method 4
- Multithreaded Applications
- Large Objects (LOBs)
- Precompiler Options
- Operating System Dependencies
- Reserved Words\_ Keywords\_ and Namespaces
- Performance Tuning
- Syntactic and Semantic Checking
- Embedded SQL Statements and Precompiler Directives
- Pro\*COBOL for Windows



10

## ANSI Dynamic SQL

This chapter describes Oracle's implementation of ANSI dynamic SQL (that is, dynamic SQL as specified by the SQL standard) which should be used for new Method 4 applications. It has enhancements over the older Oracle dynamic SQL Method 4, which is described in "Oracle Dynamic SQL: Method 4". The ANSI Method 4 supports all Oracle types, while the older Oracle Method 4 does *not* support cursor variables, tables of group items, the DML returning clause, and LOBs.

In ANSI dynamic SQL, descriptors are internally maintained by Oracle, while in the older Oracle dynamic SQL Method 4, descriptors are defined in the user's Pro\*COBOL program. In both cases, with Method 4 your Pro\*COBOL program accepts or builds SQL statements that contain a varying number of host variables.

The main sections in this chapter are:

- · Basics of ANSI Dynamic SQL
- Overview of ANSI SQL Statements
- Oracle Extensions
- ANSI Dynamic SQL Precompiler Options
- Full Syntax of the Dynamic SQL Statements
- Sample Programs: SAMPLE12.PCO

#### **Related Topics**

Oracle Dynamic SQL: Method 4

## 10.1 Basics of ANSI Dynamic SQL

#### Consider the SQL statement:

```
SELECT ename, empno FROM emp WHERE deptno = :deptno data
```

The steps you follow to use ANSI dynamic SQL are:

- Declare variables, including a string to hold the statement to be executed.
- Allocate descriptors for input and output variables.
- Prepare the statement.
- Describe input for the input descriptor.
- Set the input descriptor (in our example the one input host bind variable, deptno data).
- Declare and open a dynamic cursor.
- Set the output descriptors (in our example, the output host variables ename and empno).
- Repeatedly fetch data, using GET DESCRIPTOR to retrieve the ename and empno data fields from each row.
- Do something with the data retrieved (output it, for instance).



Close the dynamic cursor and deallocate the input and output descriptors.

## 10.1.1 Precompiler Options

Normally, if you are using ANSI dynamic SQL you will be writing to the ANSI standard for precompilers and will therefore be using the macro command line option MODE=ANSI. If you wish to use this method and do not wish to use MODE=ANSI, then the functionality is controlled by the micro command line option: DYNAMIC=ANSI.

You can either set the micro precompiler option DYNAMIC to ANSI, or set the macro option MODE to ANSI. This causes the default value of DYNAMIC to be ANSI. The other setting of DYNAMIC is <code>ORACLE</code>. For more about micro options, see "Macro and Micro Options" and "<code>DYNAMIC</code>".

In order to use ANSI type codes, set the precompiler micro option <code>TYPE\_CODE</code> to ANSI, or set the macro option <code>MODE</code> to <code>ANSI</code>. This changes the default setting of <code>TYPE\_CODE</code> to ANSI. To set <code>TYPE\_CODE</code> to <code>ANSI</code>, <code>DYNAMIC</code> must also be <code>ANSI</code>.

For a list of the ANSI SQL types see "ANSI SQL Datatypes". Use the ANSI types with precompiler option TYPE\_CODE set to ANSI if you want your application to be portable across database platforms and be as compliant to ANSI as possible.

For more details, see "MODE" and "TYPE CODE".

#### **Related Topics**

- Macro and Micro Options
- DYNAMIC
- MODE
- TYPE CODE

## 10.2 Overview of ANSI SQL Statements

Allocate a descriptor area before using it in a dynamic SQL statement.

The ALLOCATE DESCRIPTOR statement syntax is:

```
EXEC SQL ALLOCATE DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
  [WITH MAX {:occurrences | numeric_literal}]
END-EXEC.
```

A global descriptor can be used in any module in the program. A local descriptor can be accessed only in the file in which it is allocated. Local is the default.

The descriptor name, desc\_nam, is a host variable. A string literal can be used instead.

occurrences is the maximum number of bind variables or columns that the descriptor can hold, with a default of 100.

When a descriptor is no longer needed, deallocate it to conserve memory.

The deallocate statement is:

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal}
END-EXEC.
```



Use the DESCRIBE statement to obtain information on a prepared SQL statement. DESCRIBE INPUT describes bind variables for the dynamic statement that has been prepared. DESCRIBE OUTPUT (the default) can give the number, type, and length of the output columns. The simplified syntax is:

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] sql_statement
    USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

If your SQL statement has input and output values, you must allocate two descriptors: one for input and one for output values. If there are no input values, for example:

```
SELECT ename, empno FROM emp
```

then the input descriptor is not needed.

Use the SET DESCRIPTOR statement to specify input values for INSERTS, UPDATES, DELETES and the WHERE clauses of SELECT statements. Use SET DESCRIPTOR to set the number of input bind variables (stored in *COUNT*) when you have not done a DESCRIBE into your input descriptor:

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
COUNT = {:kount | numeric_literal}
END-EXEC.
```

kount can be a host variable or a numeric literal, such as 5. Use a SET DESCRIPTOR statement for each host variable, giving at least the data value of the variable:

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
    VALUE item_number DATA = :hv3
END-EXEC.
```

You can also set the type and length of the input host variable:

**Note:** If you do not set the type and length, either explicitly, through the SET DESCRIPTOR statement, or implicitly by doing a DESCRIBE OUTPUT, when TYPE\_CODE=ORACLE, the precompiler will use values for these derived from the host variable itself. When TYPE\_CODE=ANSI, you must set the type using the values in Table 10-1. You should also set the length because the ANSI default lengths may not match those of your host variable.

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   VALUE item_number TYPE = :hv1, LENGTH = :hv2, DATA = :hv3
END-EXEC.
```

We use the identifiers hv1, hv2, and hv3 to remind us that the values must be supplied by host variables. *item\_number* is the position of the input variable in the SQL statement. It can be a host variable or an integer number.

TYPE is the Type Code selected from the following table, if TYPE\_CODE is set to ANSI:

Table 10-1 ANSI SQL Datatypes

Datatype	Type Code
CHARACTER	1
CHARACTER VARYING	12
DATE	9
DECIMAL	3



Table 10-1 (Co	nt.) ANSI S	SQL Datatypes
----------------	-------------	---------------

Datatype	Type Code
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2
REAL	7
SMALLINT	5

See Table 11-2 for the Oracle type codes. Use the negative value of the Oracle code if the ANSI datatype is not in the table, and TYPE\_CODE = ANSI.

DATA is the host variable value which is input.

You can also set other input values such as indicator, precision and scale. See the more complete discussion of "SET\_DESCRIPTOR" for a list of all the possible descriptor item names.

The numeric values in the SET DESCRIPTOR statement must be declared as either PIC S9(9) COMP or PIC S9(4) COMP except for indicator and returned length values which you must declare as PIC S9(4) COMP.

In the following example, when you want to retrieve empno, set these values: VALUE=2, because <code>empno</code> is the second output host variable in the dynamic SQL statement. The host variable <code>EMPNO-TYP</code> is set to 3 (Oracle Type for integer). The length of a host integer, <code>EMPNO-LEN</code>, is set to 4, which is the size of the host variable. The DATA is equated to the host variable EMPNO-DATA which will receive the value from the database table. The code fragment is as follows:

```
...

01 DYN-STATEMENT PIC X(58)

VALUE "SELECT ename, empno FROM emp WHERE deptno =:deptno_number".

01 EMPNO-DATA PIC S9(9) COMP.

01 EMPNO-TYP PIC S9(9) COMP VALUE 3.

01 EMPNO-LEN PIC S9(9) COMP VALUE 4.

...

EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE=:EMPNO-TYP, LENGTH=:EMPNO-LEN,
DATA=:EMPNO-DATA END-EXEC.
```

After setting the input values, execute or open your statement using the input descriptor. If there are output values in your statement, set them before doing a FETCH. If you have done a DESCRIBE OUTPUT, you may have to reset the actual types and lengths of your host variables because the DESCRIBE execution will produce internal types and lengths which differ from your host variable external types and length.

After the FETCH of the output descriptor, use GET DESCRIPTOR to access the returned data. Again we show a simplified syntax with details later in this chapter:

```
EXEC SQL GET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
    VALUE item_number
    :hv1 = DATA, :hv2 = INDICATOR, :hv3 = RETURNED_LENGTH
END-EXEC.
```



desc\_nam and item\_number can be literals or host variables. A descriptor name can be a literal such as 'out'. An item number can be a numeric literal such as 2.

hv1, hv2, and hv3 are host variables. They must be host variables, not literals. Only three are shown in the example. See Table 10-4 for a list of all possible items of returned data that you can get.

Use either PIC S9(n) COMP where n is the platform-dependent upper limit, PIC S9(9) COMP or PIC S9(4) COMP for all numeric values, except for indicator and returned length variables, which must be PIC S9(4) COMP.

#### **Related Topics**

SET DESCRIPTOR

## 10.3 Sample Code

The following example demonstrates the use of ANSI Dynamic SQL. It allocates an input descriptor in and an output descriptor out to execute a SELECT statement. Input values are set through the SET DESCRIPTOR statement. The cursor is opened and fetched from and the resulting output values are retrieved through a GET DESCRIPTOR statement.

```
01 DYN-STATEMENT PIC X(58)
    VALUE "SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO =: DEPTNO-DAT".
01 EMPNO-DAT PIC S9(9) COMP.
01 EMPNO-TYP PIC S9(9) COMP
                               VALUE 3.
01 EMPNO-LEN PIC S9(9) COMP
                                VALUE 4.
01 DEPTNO-TYP PIC S9(9) COMP
                                VALUE 3.
01 DEPTNO-LEN PIC S9(9) COMP
                                VALUE 4.
01 DEPTNO-DAT PIC S9(9) COMP VALUE 10.
01 ENAME-TYP PIC S9(9) COMP VALUE 3.
01 ENAME-LEN PIC S9(9) COMP VALUE 30.
01 ENAME-DAT PIC X(30).
01 SQLCODE PIC S9(9) COMP VALUE 0.
* Place preliminary code, including connection, here
    EXEC SQL ALLOCATE DESCRIPTOR 'in' END-EXEC.
    EXEC SQL ALLOCATE DESCRIPTOR 'out' END-EXEC.
    EXEC SOL PREPARE s FROM : DYN-STATEMENT END-EXEC.
    EXEC SOL DESCRIBE INPUT S USING DESCRIPTOR 'in' END-EXEC.
    EXEC SQL SET DESCRIPTOR 'in' VALUE 1 TYPE=:DEPTNO-TYP,
       LENGTH=: DEPTNO-LEN, DATA=: DEPTNO-DAT END-EXEC.
    EXEC SQL DECLARE c CURSOR FOR s END-EXEC.
    EXEC SQL OPEN c USING DESCRIPTOR 'in' END-EXEC.
    EXEC SQL DESCRIBE OUTPUT s USING DESCRIPTOR 'out' END-EXEC.
    EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=: ENAME-TYP,
       LENGTH=: ENAME-LEN, DATA=: ENAME-DAT END-EXEC.
    EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE=: EMPNO-TYP,
       LENGTH=: EMPNO-LEN, DATA=: EMPNO-DAT END-EXEC.
    EXEC SOL WHENEVER NOT FOUND GOTO BREAK END-EXEC.
LOOP.
    IF SQLCODE NOT = 0
       GOTO BREAK.
    EXEC SQL FETCH c INTO DESCRIPTOR 'out' END-EXEC.
    EXEC SQL GET DESCRIPTOR 'OUT' VALUE 1 : ENAME-DAT = DATA END-EXEC.
    EXEC SQL GET DESCRIPTOR 'OUT' VALUE 2 :EMPNO-DAT = DATA END-EXEC.
    DISPLAY "ENAME = " WITH NO ADVANCING
```



```
DISPLAY ENAME-DAT WITH NO ADVANCING
DISPLAY "EMPNO = " WITH NO ADVANCING
DISPLAY EMPNO-DAT.
GOTO LOOP.

BREAK:
EXEC SQL CLOSE c END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR 'in' END-EXEC.
EXEC SOL DEALLOCATE DESCRIPTOR 'out' END-EXEC.
```

## 10.4 Oracle Extensions

These extensions are described next:

- Reference semantics for data items in SET statements.
- Arrays for bulk operations.
- Support for object types, NCHAR columns, and LOBs.

### 10.4.1 Reference Semantics

The ANSI standard specifies *value* semantics. To improve performance, Oracle has extended this standard to include *reference* semantics.

Value semantics makes a copy of your host variables data. Reference semantics uses the addresses of your host variables, avoiding a copy. Thus, reference semantics can provide performance improvements for large amounts of data.

To help speed up fetches, use the REF keyword before the data clauses:

```
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=:ENAME-TYP,
LENGTH=:ENAME-LEN, REF DATA=:ENAME-DAT END-EXEC.
EXEC SQL DESCRIPTOR 'out' VALUE 2 TYPE=:EMPNO-TYP,
LENGTH=:EMPNO-LEN, REF DATA=:EMPNO-DAT END-EXEC.
```

Then the host variables receive the results of the retrieves. The GET statement is not needed. The retrieved data is written directly into <code>ename\_data</code> and <code>empno\_data</code> after each FETCH.

Use of the REF keyword is allowed *only* before DATA, INDICATOR and RETURNED\_LENGTH items (which can vary with each row fetched) as in this fragment of code:

```
01 INDI PIC S9(4) COMP.
01 RETRN-LEN PIC S9(9) COMP.
...

EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=:ENAME-TYP,
LENGTH=:ENAME-LEN, REF DATA=:ENAME-DAT,
REF INDICATOR=:INDI, REF RETURNED_LENGTH =:RETRN-LEN END-EXEC.
```

After each fetch, RETRN-LEN holds the actual retrieved length of the ename field, which is useful for CHAR or VARCHAR2 data.

ENAME-LEN will not receive the returned length. It will not be changed by the FETCH statement. Use a DESCRIBE statement, followed by a GET statement to find out the maximum column width before fetching rows of data.

REF keyword is also used for other types of SQL statements than SELECT, to speed them up. Note that with reference semantics, the host variable is used rather than a



value copied into the descriptor area. The host variable data at the time of execution of the SQL statement is used, not its data at the time of the SET. Here is an example:

```
...

MOVE 1 to VAL.

EXEC SQL SET DESCRIPTOR 'value' VALUE 1 DATA = :VAL END-EXEC.

EXEC SQL SET DESCRIPTOR 'reference' VALUE 1 REF DATA = :VAL END-EXEC.

MOVE 2 to VAL.

* Will use VAL = 1

EXEC SQL EXECUTE s USING DESCRIPTOR 'value' END-EXEC.

*Will use VAL = 2

EXEC SQL EXECUTE s USING DESCRIPTOR 'reference' END-EXEC.
```

See "SET\_DESCRIPTOR" for many more details on the differences.

#### **Related Topics**

SET DESCRIPTOR

## 10.4.2 Using Tables for Bulk Operations

Oracle's implementation of ANSI dynamic SQL extends the SQL standard's dynamic SQL by providing bulk operations. To use bulk operations, use the FOR clause with an array size to specify the amount of input data or the number of rows you want to process.

The FOR clause is used in the ALLOCATE statement to give the maximum amount of data or number of rows. For example, to use a maximum array size of 100:

```
Or:

MOVE 100 TO INT-ARR-SIZE.
EXEC SQL FOR :INT-ARR-SIZE ALLOCATE DESCRIPTOR 'out' END-EXEC.
```

The FOR clause is then used in subsequent statements that access the descriptor. In an output descriptor the FETCH statement must have an array size equal to or less than the array size already used in the ALLOCATE statement:

```
EXEC SQL FOR 20 FETCH c1 USING DESCRIPTOR 'out' END-EXEC.
```

Subsequent GET statements for the same descriptor, that get DATA, INDICATOR, or RETURNED LENGTH values, must use the same array size as the FETCH statement.

```
01 VAL-DATA OCCURS 20 TIMES PIC S9(9) COMP.
01 VAL-INDI OCCURS 20 TIMES PIC S9(4) COMP.
...

EXEC SQL FOR 20 GET DESCRIPTOR 'out' VALUE 1 :VAL-DATA = DATA,
    :VAL-INDI = INDICATOR
END-EXEC.
```

However, GET statements that reference other items which do not vary from row to row, such as LENGTH, TYPE and COUNT, must *not* use the FOR clause:

```
01 CNT PIC S9(9) COMP.
01 LEN PIC S9(9) COMP.
...

EXEC SQL GET DESCRIPTOR 'out' :CNT = COUNT END-EXEC.

EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :LEN = LENGTH END-EXEC.
```



The same holds true for SET statements with reference semantics. SET statements which precede the FETCH and employ reference semantics for DATA, INDICATOR, or RETURNED LENGTH must have the same array size as the FETCH:

```
O1 REF-DATA OCCURS 20 TIMES PIC S9(9) COMP.
O1 REF-INDI OCCURS 20 TIMES PIC S9(4) COMP.

EXEC SQL FOR 20 SET DESCRIPTOR 'out' VALUE 1 REF DATA = :REF-DATA,

REF INDICATOR = :REF-INDI END-EXEC.
```

Similarly, for a descriptor that is used for input, to insert a batch of rows, for instance, the EXECUTE or OPEN statement must use an array size equal to or less than the size used in the ALLOCATE statement. The SET statement, for both value and reference semantics, that accesses DATA, INDICATOR, or RETURNED\_LENGTH must use the same array size as in the EXECUTE statement.

The FOR clause is never used on the DEALLOCATE or PREPARE statements.

The following code sample illustrates a bulk operation with no output descriptor (there is no output, only input to be inserted into the table <code>emp</code>). The value of <code>CNT</code> is 2 (there are two host variables, <code>ENAME</code> and <code>EMPNO</code>, in the <code>INSERT</code> statement). The data table <code>ENAME-TABLE</code> holds three character strings: <code>Tom</code>, <code>Dick</code> and <code>Harry</code>, in that order. Their employee numbers are in the table <code>EMPNO-TABLE</code>. The indicator table <code>ENAME-IND</code> has a value of -1 for the second element; so a <code>NULL</code> will be inserted instead of <code>Dick</code>.

```
01 DYN-STATEMENT PIC X(240) value
       "INSERT INTO EMP(ENAME, EMPNO) VALUES (:ENAME,:EMPNO)".
01 ARRAY-SIZE PIC S9(9) COMP VALUE 3.
01 ENAME-VALUES.
    05 FILLER PIC X(6) VALUE "Tom
    05 FILLER PIC X(6) VALUE "Dick ".
    05 FILLER PIC X(6) VALUE "Harry ".
01 ENAME-TABLE REDEFINES ENAME-VALUES.
    05 ENAME PIC X(6)OCCURS 3 TIMES.
01 ENAME-IND PIC S9(4) COMPOCCURS 3 TIMES.
01 ENAME-LEN PIC S9(9) COMP VALUE 6.
01 ENAME-TYP PIC S9(9) COMP VALUE 96.
01 EMPNO-VALUES.
    05 FILLER PIC S9(9) COMP VALUE 8001.
    05 FILLER PIC S9(9) COMP VALUE 8002.
    05 FILLER PIC S9(9) COMP VALUE 8003.
01 EMPNO-TABLE REDEFINES EMPNO-VALUES.
    05 EMPNO PIC S9(9) DISPLAY SIGN LEADING OCCURS 3 TIMES.
01 EMPNO-LEN PIC S9(9) COMP VALUE 4.
01 EMPNO-TYP PIC S9(9) COMP VALUE 3.
01 CNT
          PIC S9(9) COMP VALUE 2.
   EXEC SQL FOR :ARRAY-SIZE ALLOCATE DESCRIPTOR 'in' END-EXEC.
   EXEC SOL PREPARE S FROM : DYN-STATEMENT END-EXEC.
   MOVE 0 TO ENAME-IND(1).
   MOVE -1 TO ENAME-IND(2).
   MOVE 0 TO ENAME-IND(3).
   EXEC SQL SET DESCRIPTOR 'in' COUNT = :CNT END-EXEC.
   EXEC SQL SET DESCRIPTOR 'in' VALUE 1
      TYPE = :ENAME-TYP, LENGTH =:ENAME-LEN
    END-EXEC.
    EXEC SQL FOR :ARRAY-SIZE SET DESCRIPTOR 'in' VALUE 1
      DATA = :ENAME, INDICATOR = :ENAME-IND
    END-EXEC.
```



```
EXEC SQL SET DESCRIPTOR 'in' VALUE 2

TYPE = :EMPNO-TYP, LENGTH =:EMPNO-LEN

END-EXEC.

EXEC SQL FOR :ARRAY-SIZE SET DESCRIPTOR 'in' VALUE 2

DATA = :EMPNO

END-EXEC.

EXEC SQL FOR :ARRAY-SIZE EXECUTE S

USING DESCRIPTOR 'in' END-EXEC.
```

The preceding code inserts these values into the table:

```
EMPNO ENAME
8001 Tom
8002
8003 Harry
```

See the discussion in "The FOR Clause" for restrictions and cautions.

#### **Related Topics**

The FOR Clause

## 10.5 ANSI Dynamic SQL Precompiler Options

The macro option MODE (See "MODE") sets ANSI compatibility characteristics and controls a number of functions. It can have the values ANSI or ORACLE. For individual functions there are micro options that override the MODE setting.

The precompiler micro option *DYNAMIC* specifies the descriptor behavior in dynamic SQL. The precompiler micro option *TYPE\_CODE* specifies whether ANSI or Oracle datatype codes are to be used.

When the macro option MODE is set to ANSI, the micro option DYNAMIC becomes ANSI automatically. When MODE is set to ORACLE, DYNAMIC becomes ORACLE.

DYNAMIC and TYPE\_CODE cannot be used inline.

The following table describes how the DYNAMIC setting affects various functionality:

Table 10-2 DYNAMIC Option Settings

Function	DYNAMIC=ANSI	DYNAMIC=ORACLE
Descriptor creation.	Must use ALLOCATE statement.	Must use an Oracle format descriptor.
Descriptor destruction.	May use DEALLOCATE statement.	N/A
Retrieving data.	May use both FETCH and GET statements.	Must use only FETCH statement.
Setting input data.	May use DESCRIBE INPUT statement. Must use SET statement.	Must set descriptor values in code. Must use DESCRIBE BIND VARIABLES statement.
Descriptor representation.	Single quoted literal or host identifier which contains the descriptor name.	Host variable, a pointer to SQLDA.

Table 10-2 (Cont.) DYNAMIC Option Settings

Function	DYNAMIC=ANSI	DYNAMIC=ORACLE
Data types available.	All ANSI types except BIT and all Oracle types.	Oracle types except objects, LOBs, and cursor variables.

The micro option TYPE\_CODE is set by the precompiler to the same setting as the macro option MODE. TYPE\_CODE can only equal ANSI if DYNAMIC equals ANSI.

The following table shows the functionality corresponding to the TYPE\_CODE settings.

Table 10-3 TYPE\_CODE Option Settings

Function	TYPE_CODE=ANSI	TYPE_CODE=ORACLE
Data type code numbers input and returned in dynamic SQL.	Use ANSI code numbers when ANSI type exists. Otherwise, use the negative of the Oracle code number.	Use Oracle code numbers.  May be used regardless of the setting of DYNAMIC.
	Only valid when DYNAMIC=ANSI.	

#### **Related Topics**

MODE

## 10.6 Full Syntax of the Dynamic SQL Statements

For more details on all these statements, see the alphabetical listing in the appendix "Embedded SQL Statements and Precompiler Directives".

#### **Related Topics**

Embedded SQL Statements and Precompiler Directives

### 10.6.1 ALLOCATE DESCRIPTOR

This statement is used only for ANSI dynamic SQL.

#### **Purpose**

Use this statement to allocate a SQL descriptor area. Supply a descriptor, the maximum number of occurrences of host bind items, and an array size.

#### **Syntax**

```
EXEC SQL [FOR [:]array_size] ALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
    {:desc_nam | string_literal} [WITH MAX occurrences]
END-EXEC.
```



#### 10.6.1.1 Variables

A number of variables can be used with the ALLOCATE descriptor. These include: array\_size, desc\_nam, and occurrences.

#### array\_size

The optional array\_size clause (an Oracle extension) supports table processing. It tells the precompiler that the descriptor is usable for table processing.

```
GLOBAL | LOCAL
```

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

#### desc nam

The desc\_nam variable defines the local descriptors that must be unique in the module. A runtime error is generated if the descriptor has been previously allocated, but not deallocated. A global descriptor must be unique for the application or a runtime error results.

#### occurrences

The optional occurrences clause is the maximum number of host variables possible in the descriptor. The occurrences variable must be an integer constant between 0 and 64K, or an error is returned. The default is 100. A precompiler error results if it does not conform to these rules.

### 10.6.1.2 Examples

```
EXEC SQL ALLOCATE DESCRIPTOR 'SELDES' WITH MAX 50 END-EXEC.

EXEC SQL FOR :BATCH ALLOCATE DESCRIPTOR GLOBAL :BINDDES WITH MAX 25 END-EXEC.
```

### 10.6.2 DEALLOCATE DESCRIPTOR

#### **Purpose**

To free memory, use the deallocate statement. This statement deallocates a previously allocated SQL descriptor area.

#### **Syntax**

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
{:desc_nam | string_literal}
END-EXEC.
```

#### **Variables**

#### desc nam

The only variable available with the deallocate descriptor is desc\_nam (for descriptor name.) A runtime error results when a descriptor with the same name and scope has not been allocated, or has already been allocated and deallocated.



#### **Examples**

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES' END-EXEC.

EXEC SQL DEALLOCATE DESCRIPTOR :BINDDES END-EXEC.
```

### 10.6.3 GET DESCRIPTOR

#### **Purpose**

Use to obtain information from a SQL descriptor area.

#### **Syntax**

```
EXEC SQL [FOR [:]array_size] GET DESCRIPTOR [GLOBAL | LOCAL]
{:desc_nam | string_literal}
{:hv0 = COUNT | VALUE item_number :hv1 = item_name1
[{, :hvN = item_nameN }]}
END-EXEC.
```

#### **Variables**

#### array size

The FOR array\_size variable is an optional Oracle extension. FOR array\_size has to be equal to the array size field in the FETCH statement.

#### desc\_nam

The descriptor name.

#### **GLOBAL | LOCAL**

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

#### COUNT

The total number of bind variables.

#### **VALUE** item number

The position of the item in the SQL statement. item\_number can be a variable or a constant. If item\_number is greater than COUNT, the "no data found" condition is returned. item\_number must be greater than 0.

#### hv1 .. hvN

These are host variables to which values are transferred.

#### item name1 .. item nameN

The descriptor item names corresponding to the host variables. The possible ANSI descriptor item names are listed in the following table.

Table 10-4 Definitions of Descriptor Item Names

Descriptor Item Name	Meaning
TYPE	See Table 10-1 for the ANSI type codes. See Table 11-2 for the Oracle type codes. Use the negative value of the Oracle code if
	the ANSI datatype is not in the table, and TYPE_CODE = ANSI.



Table 10-4 (Cont.) Definitions of Descriptor Item Names

Descriptor Item Name	Meaning
LENGTH	Length of data in the column. In characters for NCHAR, and otherwise in bytes. Set by the DESCRIBE OUTPUT.
OCTET_LENGTH	Length of data in bytes.
RETURNED_LENGTH	The actual data length after a FETCH. It is undefined for fixed-length character types.
RETURNED_OCTET_LENGTH	Length of the returned data in bytes.
PRECISION	The number of digits.
SCALE	For exact numeric types, the number of digits to the right of the decimal point.
NULLABLE	If 1, the column can have NULL values. If 0, the column cannot have NULL values.
INDICATOR	The associated indicator value.
DATA	The data value.
NAME	Column name.
CHARACTER_SET_NAME	Column's character set.

The following table lists the Oracle extensions to the descriptor item names.

Table 10-5 Oracle Extensions to Definitions of Descriptor Item Names

Descriptor Item Name	Meaning
NATIONAL_CHARACTER	If 2, NCHAR or NVARCHAR2. If 1, character. If 0, non-character.
INTERNAL_LENGTH	The internal length, in bytes.

#### **Usage Notes**

Use the FOR clause in GET DESCRIPTOR statements which contain DATA, INDICATOR, and RETURNED\_LENGTH items only.

The internal type is provided by the DESCRIBE OUTPUT statement. For both input and output, you must set the type to be the external type of your host variable. TYPE is the Oracle or ANSI code in Table 10-1. You will receive the negative value of the Oracle type code if the ANSI type is not in the table.

LENGTH contains the column length in characters for fields that have fixed-width National Language character sets. It is in bytes for other character columns. It is set in DESCRIBE OUTPUT.

RETURNED\_LENGTH is the actual data length set by the FETCH statement. It is in bytes or characters as described for LENGTH. The fields OCTET\_LENGTH and RETURNED OCTET LENGTH are the lengths in bytes.

NULLABLE = 1 means that the column can have NULLS; NULLABLE = 0 means it cannot.

CHARACTER\_SET\_NAME only has meaning for character columns. For other types, it is undefined. The DESCRIBE OUTPUT statement obtains the value.



DATA and INDICATOR are the data value and the indicator status for that column. If data = NULL, but the indicator was not requested, an error is generated at runtime ("DATA EXCEPTION, NULL VALUE, NO INDICATOR PARAMETER").

#### **Oracle-specific Descriptor Item Names**

NATIONAL\_CHARACTER = 2 if the column is an NCHAR or NVARCHAR2 column. If the column is a character (but not National Character) column, this item is set to 1. If a non-character column, this item becomes 0 after DESCRIBE OUTPUT is executed.

INTERNAL\_LENGTH is for compatibility with Oracle dynamic Method 4. It has the same value as the length member of the Oracle descriptor area. See "Oracle Dynamic SQL: Method 4".

#### **Examples**

```
EXEC SQL GET DESCRIPTOR :BINDDES :COUNT = COUNT END-EXEC.

EXEC SQL GET DESCRIPTOR 'SELDES' VALUE 1 :TYP = TYPE, :LEN = LENGTH END-EXEC.

EXEC SQL FOR :BATCH GET DESCRIPTOR LOCAL 'SELDES' VALUE :SEL-ITEM-NO :IND = INDICATOR, :DAT = DATA END-EXEC.
```

#### **Related Topics**

Oracle Dynamic SQL: Method 4

## 10.6.4 SET DESCRIPTOR

#### **Purpose**

Use this statement to set information in the descriptor area from host variables. The SET DESCRIPTOR statement supports *only* host variables for the item names.

#### **Syntax**

```
EXEC SQL [FOR [:]array_size] SET DESCRIPTOR [GLOBAL | LOCAL]
  {:desc_nam | string_literal}
  {COUNT = :hv0 | VALUE item_number
  [REF] item_name1 = :hv1
  [{, [REF] item_nameN = :hvN}]}
END-EXEC.
```

#### **Variables**

#### array size

This optional Oracle clause permits using arrays when setting the descriptor items DATA, INDICATOR, and RETURNED\_LENGTH only. You cannot use other items in a SET DESCRIPTOR that contains the FOR clause. All host variable table sizes must match. Use the same array size for the SET statement that you use for the FETCH statement.

#### desc nam

The descriptor name. It follows the rules in ALLOCATE DESCRIPTOR.

#### COUNT

The number of bind (input) or define (output) variables.



#### **VALUE** item number

Position in the dynamic SQL statement of a host variable.

#### hv1 .. hvN

The host variables (not constants) that you set.

#### item name1 .. item nameN

In a similar way to the GET DESCRIPTOR syntax (see "GET DESCRIPTOR"), item\_name can take on these values:

Table 10-6 Descriptor Item Names for SET DESCRIPTOR

Descriptor Item Name	Meaning
TYPE	See Table 10-1 for the ANSI type codes. See Table 11-2 for the Oracle type codes. Use the negative value of the Oracle type code if the Oracle type is not in the table, and TYPE_CODE = ANSI.
LENGTH	Maximum length of data in the column.
PRECISION	The number of digits.
SCALE	For exact numeric types, the number of bytes to the right of the decimal point.
INDICATOR	The associated indicator value. Set for reference semantics.
DATA	Value of the data to be set. Set for reference semantics.
CHARACTER_SET_NAME	Column's character set.

The Oracle extensions to the descriptor item names are listed in the following table.

Table 10-7 Extensions to Descriptor Item Names for SET DESCRIPTOR

Descriptor Item Name	Meaning
RETURNED_LENGTH	Length returned after a FETCH. Set if reference semantics is being used.
NATIONAL_CHARACTER	Set to 2 when the input host variable is an NCHAR or NVARCHAR2 type.

#### **Usage Notes**

Reference semantics is another optional Oracle extension that speeds performance. Use the keyword REF before these descriptor items names only: DATA, INDICATOR, RETURNED\_LENGTH. When you use the REF keyword you do not need to use a GET statement. Complex data types and DML returning clauses require the REF form of SET DESCRIPTOR. See "DML Returning Clause".

When REF is used the associated host variable itself is used in the SET. The GET is not needed in this case. The RETURNED\_LENGTH can only be set when you use the REF semantics, not the value semantics.

Use the same array size for the SET or GET statements that you use in the FETCH.

Set the NATIONAL CHAR field to 2 for NCHAR host input values.

When setting an object type's characteristics, you must set USER\_DEFINED\_TYPE\_NAME and USER\_DEFINED\_TYPE\_NAME\_LENGTH.



If omitted, USER\_DEFINED\_TYPE\_SCHEMA and USER\_DEFINED\_TYPE\_SCHEMA\_LENGTH default to the current connection.

#### **Related Topics**

- DML Returning Clause
- GET DESCRIPTOR

### 10.6.4.1 Example

Bulk table examples are found in "Using Tables for Bulk Operations".

```
O1 BINDNO PIC S9(9) COMP VALUE 2.

01 INDI PIC S9(4) COMP VALUE -1.

01 DATA PIC X(6) COMP VALUE "ignore".

01 BATCH PIC S9(9) COMP VALUE 1.

...

EXEC SQL FOR :batch ALLOCATE DESCRIPTOR :BINDDES END-EXEC.
EXEC SQL SET DESCRIPTOR GLOBAL :BINDDES COUNT = 3 END-EXEC.
EXEC SQL FOR :batch SET DESCRIPTOR :BINDDES

VALUE :BINDNO INDICATOR = :INDI, DATA = :DATA END-EXEC.
```

### 10.6.5 Use of PREPARE

#### **Purpose**

The PREPARE statement used in this method is the same as the PREPARE statement used in the Oracle dynamic SQL methods. An Oracle extension allows a quoted string for the SQL statement, as well as a variable.

#### **Syntax**

```
EXEC SQL PREPARE statement id FROM :sql statement END-EXEC.
```

#### **Variables**

#### statement\_id

This must not be declared; it is an undeclared SQL identifier associated with the prepared SQL statement.

#### sql\_statement

A character string (a constant or a variable) holding the embedded SQL statement.

#### **Examples**

```
...
01 STATEMENT PIC X(255)

VALUE "SELECT ENAME FROM emp WHERE deptno = :d".

...

EXEC SQL PREPARE S1 FROM :STATEMENT END-EXEC.
```



### 10.6.6 DESCRIBE INPUT

### **Purpose**

This statement returns information about the input bind variables.

### **Syntax**

```
EXEC SQL DESCRIBE INPUT statement_id USING [SQL] DESCRIPTOR
  [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

#### **Variables**

### statement id

The same as used in PREPARE and DESCRIBE OUTPUT. This must not be declared; it is a SQL identifier.

### GLOBAL | LOCAL

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

#### desc nam

The descriptor name.

### **Usage Notes**

Only COUNT and NAME are implemented for bind variables in this version.

### **Examples**

```
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR GLOBAL :BINDDES END-EXEC. EXEC SQL DESCRIBE INPUT S2 USING DESCRIPTOR 'input' END-EXEC.
```

### 10.6.7 DESCRIBE OUTPUT

### **Purpose**

The DESCRIBE INPUT statement is used to obtain information about the columns in a PREPARE statement. The ANSI syntax differs from the older syntax. The information which is stored in the SQL descriptor area is the number of values returned and associated information such as type, length, and name.

### **Syntax**

```
EXEC SQL DESCRIBE [OUTPUT] statement_id USING [SQL] DESCRIPTOR
  [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

### **Variables**

### statement id

The statement id is a SQL identifier. It must not be declared.



### GLOBAL | LOCAL

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

### desc nam

The descriptor name. Either a host variable preceded by a ':', or a single-quoted string. OUTPUT is the default and can be omitted.

### **Examples**

```
...
01 DESNAME PIC X(10) VALUE "SELDES".
...
EXEC SQL DESCRIBE S1 USING SQL DESCRIPTOR 'SELDES' END-EXEC.
* Or:
EXEC SQL DESCRIBE OUTPUT S1 USING DESCRIPTOR :DESNAME END-EXEC.
```

### 10.6.8 FXFCUTE

### **Purpose**

EXECUTE matches input and output variables in a prepared SQL statement and then executes the statement. This ANSI version of EXECUTE differs from the older EXECUTE statement by allowing two descriptors in one statement to support DML RETURNING.

### **Syntax**

```
EXEC SQL [FOR [:]array_size] EXECUTE statement_id
   [USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}]
   [INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}]
END-EXEC.
```

#### **Variables**

### array\_size

The number of rows the statement will process.

### statement id

The same as used in PREPARE. This must not be declared; it is a SQL identifier. It can be a literal.

### **GLOBAL | LOCAL**

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

#### desc nam

The descriptor name. Either a host variable preceded by a ':', or a single-quoted string.

### **Usage Notes**

The INTO clause implements the RETURNING clause for INSERT, UPDATE and DELETE (See "Inserting Rows" and succeeding pages).



### **Examples**

```
EXEC SQL EXECUTE S1 USING SQL DESCRIPTOR GLOBAL :BINDDES END-EXEC.

EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES' END-EXEC.
```

### **Related Topics**

Inserting Rows

### 10.6.9 Use of EXECUTE IMMEDIATE

### **Purpose**

The EXECUTE IMMEDIATE statement executes a literal or host variable character string containing the SQL statement. The ANSI SQL form of this statement is the same as in the older dynamic SQL:

### **Syntax**

```
EXEC SQL EXECUTE IMMEDIATE [:] sql statement END-EXEC.
```

### **Variables**

Only one variable is available with the EXECUTE IMMEDIATE statement.

### sql\_statement

The sql\_statement variable is the SQL statement or PL/SQL block in a character string. It can be a host variable or a literal.

### **Examples**

EXEC SQL EXECUTE IMMEDIATE : statement END-EXEC.

## 10.6.10 Use of DYNAMIC DECLARE CURSOR

### **Purpose**

The DYNAMIC DECLARE CURSOR statement declares a cursor that is associated with a statement which is a query. This is a form of the generic Declare Cursor statement.

#### **Syntax**

EXEC SQL DECLARE cursor name CURSOR FOR statement id END-EXEC.

### **Variables**

#### cursor name

A cursor variable (a SQL identifier, not a host variable).

### statement id

An undeclared SQL identifier (the same as the one used in the PREPARE statement).

### **Examples**

EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.



### 10.6.11 OPEN Cursor

### **Purpose**

The OPEN statement associates input parameters with a cursor and then opens the cursor.

### **Syntax**

```
EXEC SQL [FOR [:]array_size] OPEN dyn_cursor
  [[USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] desc_nam1]
  [INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] desc_nam2] ]
END-EXEC.
```

### **Variables**

### array\_size

This limit is less than or equal to number specified when the descriptor was allocated.

### **GLOBAL | LOCAL**

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

### dyn\_cursor

The cursor variable.

### desc\_nam1, desc\_nam2

The optional descriptor names.

### **Usage Notes**

If the prepared statement associated with the cursor contains colons or question marks, a USING clause must be specified, or an error results at runtime. The INTO clause supports DML RETURNING (See "Inserting Rows" and succeeding sections on DELETE and UPDATE).

#### **Examples**

```
EXEC SQL OPEN C1 USING SQL DESCRIPTOR :BINDDES END-EXEC.

EXEC SQL FOR :LIMIT OPEN C2 USING DESCRIPTOR :B1, :B2
   INTO SQL DESCRIPTOR :SELDES
END-EXEC.
```

### **Related Topics**

Inserting Rows

### 10.6.12 FETCH

### **Purpose**

The FETCH statement fetches a row for a cursor declared with a dynamic DECLARE statement.



### **Syntax**

```
EXEC SQL [FOR [:]array_size] FETCH cursor INTO [SQL] DESCRIPTOR
  [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

### **Variables**

### array size

The number of rows the statement will process.

#### cursor

The dynamic cursor that was previously declared.

### **GLOBAL | LOCAL**

GLOBAL means that the descriptor name is known to all program files. LOCAL means that it is known only in the file in which it is allocated. LOCAL is the default.

### desc nam

Descriptor name.

### **Usage Notes**

The optional <code>array\_size</code> in the FOR clause must be less than or equal to the number specified in the ALLOCATE DESCRIPTOR statement.

RETURNED\_LENGTH is undefined for fixed-length character types.

### **Examples**

```
EXEC SQL FETCH FROM C1 INTO DESCRIPTOR 'SELDES' END-EXEC.

EXEC SQL FOR :ARSZ FETCH C2 INTO DESCRIPTOR :DESC END-EXEC.
```

# 10.6.13 CLOSE a Dynamic Cursor

### **Purpose**

The CLOSE statement closes a dynamic cursor. Its syntax is identical to the Oracle Method 4.

### **Syntax**

```
EXEC SQL CLOSE cursor END-EXEC.
```

### **Variables**

Only one variable is available with the CLOSE statement.

#### cursor

The cursor variable describes the previously declared dynamic cursor.

#### **Examples**

EXEC SQL CLOSE C1 END-EXEC.



# 10.6.14 Differences From Oracle Dynamic Method 4

The ANSI dynamic SQL interface supports all the features supported by the Oracle dynamic Method 4, with these additions:

- All datatypes, including cursor variables, and LOB types are supported by ANSI Dynamic SQL.
- The ANSI mode uses an internal SQL descriptor area which is an expansion of the external SQLDA used in Oracle older dynamic Method 4 to store its input and output information.
- New embedded SQL statements are introduced: ALLOCATE DESCRIPTOR. DEALLOCATE DESCRIPTOR, DESCRIBE, GET DESCRIPTOR, and SET DESCRIPTOR.
- The DESCRIBE statement does not return the names of indicator variables in ANSI Dynamic SQL.
- ANSI Dynamic SQL does not allow you to specify the maximum size of the returned column name or expression. The default size is set at 128.
- The descriptor name must be either an identifier in single-quotes or a host variable preceded by a colon.
- For output, the optional SELECT LIST FOR clause in the DESCRIBE is replaced by the optional keyword OUTPUT. The INTO clause is replaced by the USING DESCRIPTOR clause, which can contain the optional keyword SQL.
- For input, the optional BIND VARIABLES FOR clause of the DESCRIBE can be replaced by the keyword INPUT. The INTO clause is replaced by the USING DESCRIPTOR clause, which can contain the optional keyword SQL.
- The optional keyword SOL can come before the keyword DESCRIPTOR in the USING clause of the EXECUTE, FETCH and OPEN statements.

## 10.6.15 Restrictions

Restrictions in effect on ANSI dynamic SQL are:

- You cannot mix the two dynamic methods in the same module.
- The precompiler option DYNAMIC must be set to ANSI. The precompiler option TYPE CODE can be set to ANSI only if DYNAMIC is set to ANSI.
- The SET statement supports only host variables as item names.

# 10.7 Sample Programs: SAMPLE12.PCO

The following ANSI SQL dynamic Method 4 program, SAMPLE12.PCO, is found in the demo directory. SAMPLE12 mimics SQL\*Plus by prompting for SQL statements to be input by the user. Read the comments at the beginning for details of the program flow.

\* Sample Program 12: Dynamic SQL Method 4 using ANSI Dynamic SQL \*

- \* This program shows the basic steps required to use dynamic
- \* SQL Method 4 with ANSI Dynamic SQL. After logging on to
- \* ORACLE, the program prompts the user for a SQL statement,



```
* PREPAREs the statement, DECLAREs a cursor, checks for any
* bind variables using DESCRIBE INPUT, OPENs the cursor, and
* DESCRIBEs any select-list variables. If the input SQL
^{\star} statement is a query, the program FETCHes each row of data,
* then CLOSEs the cursor.
* use option dynamic=ansi when precompiling this sample.
IDENTIFICATION DIVISION.
PROGRAM-ID. ANSIDYNSQL4.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
               PIC X(20).
01 USERNAME
01 PASSWD
                  PIC X(20).
01 BDSC
                  PIC X(6) VALUE "BNDDSC".
                 PIC X(6) VALUE "SELDSC".
01 SDSC
                 PIC S9(9) COMP.
01 BNDCNT
01 SELCNT
                 PIC S9(9) COMP.
01 BNDNAME
                 PIC X(80).
01 BNDVAL
                 PIC X(80).
01 SELNAME
                 PIC X(80) VARYING.
01 SELDATA
                 PIC X(80).
01 SELTYP
                 PIC S9(4) COMP.
01 SELPREC
                 PIC S9(4) COMP.
01 SELLEN
                 PIC S9(4) COMP.
01 SELIND
                 PIC S9(4) COMP.
01 DYN-STATEMENT PIC X(80).
01 BND-INDEX PIC S9(9) COMP.
01 SEL-INDEX
               PIC S9(9) COMP.
01 VARCHAR2-TYP PIC S9(4) COMP VALUE 1.
01 VAR-COUNT
                 PIC 9(2).
               PIC 9(4).
01 ROW-COUNT
01 NO-MORE-DATA PIC X(1) VALUE "N".
01 TMPLEN
               PIC S9(9) COMP VALUE 80.
                  PIC S9(9) COMP.
01 MAX-LENGTH
    EXEC SOL INCLUDE SOLCA
                                END-EXEC.
PROCEDURE DIVISION.
START-MAIN.
    EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
    DISPLAY "USERNAME: " WITH NO ADVANCING.
    ACCEPT USERNAME.
    DISPLAY "PASSWORD: " WITH NO ADVANCING.
    ACCEPT PASSWD.
    EXEC SQL CONNECT : USERNAME IDENTIFIED BY : PASSWD END-EXEC.
    DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME.
  ALLOCATE THE BIND AND SELECT DESCRIPTORS.
    EXEC SQL ALLOCATE DESCRIPTOR :BDSC WITH MAX 20 END-EXEC.
    EXEC SQL ALLOCATE DESCRIPTOR :SDSC WITH MAX 20 END-EXEC.
    GET A SQL STATEMENT FROM THE OPERATOR.
    DISPLAY "ENTER SOL STATEMENT WITHOUT TERMINATOR:".
    DISPLAY ">" WITH NO ADVANCING.
```

```
ACCEPT DYN-STATEMENT.
    DISPLAY " ".
   PREPARE THE SQL STATEMENT AND DECLARE A CURSOR.
    EXEC SOL PREPARE S1 FROM : DYN-STATEMENT END-EXEC.
   EXEC SQL DECLARE C1 CURSOR FOR S1
                                            END-EXEC.
   DESCRIBE BIND VARIABLES.
   EXEC SQL DESCRIBE INPUT S1 USING DESCRIPTOR :BDSC END-EXEC.
    EXEC SQL GET DESCRIPTOR :BDSC :BNDCNT = COUNT END-EXEC.
    IF BNDCNT < 0
       DISPLAY "TOO MANY BIND VARIABLES."
       GO TO END-SQL
    ELSE
       DISPLAY "NUMBER OF BIND VARIABLES: " WITH NO ADVANCING
       MOVE BNDCNT TO VAR-COUNT
        DISPLAY VAR-COUNT
       EXEC SQL SET DESCRIPTOR :BDSC COUNT = :BNDCNT END-EXEC
    END-IF.
    IF BNDCNT = 0
        GO TO DESCRIBE-ITEMS.
    PERFORM SET-BND-DSC
           VARYING BND-INDEX FROM 1 BY 1
           UNTIL BND-INDEX > BNDCNT.
   OPEN THE CURSOR AND DESCRIBE THE SELECT-LIST ITEMS.
DESCRIBE-ITEMS.
   EXEC SQL OPEN C1 USING DESCRIPTOR :BDSC END-EXEC.
    EXEC SQL DESCRIBE OUTPUT S1 USING DESCRIPTOR :SDSC END-EXEC.
   EXEC SOL GET DESCRIPTOR :SDSC :SELCNT = COUNT END-EXEC.
    IF SELCNT < 0
       DISPLAY "TOO MANY SELECT-LIST ITEMS."
       GO TO END-SQL
    ELSE
       DISPLAY "NUMBER OF SELECT-LIST ITEMS: "
           WITH NO ADVANCING
       MOVE SELCNT TO VAR-COUNT
        DISPLAY VAR-COUNT
        DISPLAY " "
       EXEC SQL SET DESCRIPTOR :SDSC COUNT = :SELCNT END-EXEC
   END-IF.
   SET THE INPUT DESCRIPTOR
    IF SELCNT > 0
        PERFORM SET-SEL-DSC
           VARYING SEL-INDEX FROM 1 BY 1
           UNTIL SEL-INDEX > SELCNT
        DISPLAY " ".
   FETCH EACH ROW AND PRINT EACH SELECT-LIST VALUE.
```



```
IF SELCNT > 0
        PERFORM FETCH-ROWS UNTIL NO-MORE-DATA = "Y".
    DISPLAY " "
    DISPLAY "NUMBER OF ROWS PROCESSED: " WITH NO ADVANCING.
    MOVE SOLERRD(3) TO ROW-COUNT.
    DISPLAY ROW-COUNT.
  CLEAN UP AND TERMINATE.
    EXEC SQL CLOSE C1 END-EXEC.
    EXEC SQL DEALLOCATE DESCRIPTOR : BDSC END-EXEC.
    EXEC SQL DEALLOCATE DESCRIPTOR :SDSC END-EXEC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY!".
    DISPLAY " ".
    STOP RUN.
   DISPLAY ORACLE ERROR MESSAGE AND CODE.
SQL-ERROR.
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
    PERFORMED SUBROUTINES BEGIN HERE:
    SET A BIND-LIST ELEMENT'S ATTRIBUTE
    LET THE USER FILL IN THE BIND VARIABLES AND
    REPLACE THE OS DESCRIBED INTO THE DATATYPE FIELDS OF THE
    BIND DESCRIPTOR WITH 1S TO AVOID AN "INVALID DATATYPE"
    ORACLE ERROR
SET-BND-DSC.
        EXEC SOL GET DESCRIPTOR :BDSC VALUE
               :BND-INDEX :BNDNAME = NAME END-EXEC.
        DISPLAY "ENTER VALUE FOR ", BNDNAME.
        ACCEPT BNDVAL.
        EXEC SQL SET DESCRIPTOR :BDSC VALUE :BND-INDEX
          TYPE = : VARCHAR2-TYP, LENGTH = : MAX-LENGTH,
          DATA = :BNDVAL END-EXEC.
* SET A SELECT-LIST ELEMENT'S ATTRIBUTES
SET-SEL-DSC.
    MOVE SPACES TO SELNAME-ARR.
    EXEC SQL GET DESCRIPTOR :SDSC VALUE :SEL-INDEX
          :SELNAME = NAME, :SELTYP = TYPE,
          :SELPREC = PRECISION, :SELLEN = LENGTH END-EXEC.
    IF DATATYPE IS DATE, LENGTHEN TO 9 CHARACTERS.
    IF SELTYP = 12
        MOVE 9 TO SELLEN.
    IF DATATYPE IS NUMBER, SET LENGTH TO PRECISION.
    MOVE 0 TO TMPLEN.
    IF SELTYP = 2 AND SELPREC = 0
        MOVE 40 TO TMPLEN.
    IF SELTYP = 2 AND SELPREC > 0
```

```
ADD 2 TO SELPREC
        MOVE SELPREC TO TMPLEN.
    IF SELTYP = 2
        IF TMPLEN > MAX-LENGTH
            DISPLAY "COLUMN VALUE TOO LARGE FOR DATA BUFFER."
            GO TO END-SQL
        ELSE
            MOVE TMPLEN TO SELLEN.
* COERCE DATATYPES TO VARCHAR2.
    MOVE 1 TO SELTYP.
    DISPLAY COLUMN HEADING.
    DISPLAY " ", SELNAME-ARR(1:SELLEN) WITH NO ADVANCING.
    EXEC SQL SET DESCRIPTOR :SDSC VALUE :SEL-INDEX
       TYPE = :SELTYP, LENGTH = :SELLEN END-EXEC.
* FETCH A ROW AND PRINT THE SELECT-LIST VALUE.
FETCH-ROWS.
    EXEC SQL FETCH C1 INTO DESCRIPTOR :SDSC END-EXEC.
    IF SQLCODE NOT = 0
        MOVE "Y" TO NO-MORE-DATA.
    IF SQLCODE = 0
        PERFORM PRINT-COLUMN-VALUES
            VARYING SEL-INDEX FROM 1 BY 1
            UNTIL SEL-INDEX > SELCNT
        DISPLAY " ".
* PRINT A SELECT-LIST VALUE.
PRINT-COLUMN-VALUES.
    MOVE SPACES TO SELDATA.
    returned length is not set for blank padded types
    IF SELTYP EQUALS 1
       EXEC SQL GET DESCRIPTOR :SDSC VALUE :SEL-INDEX
         :SELDATA = DATA, :SELIND = INDICATOR,
         :SELLEN = LENGTH END-EXEC
    ELSE
       EXEC SQL GET DESCRIPTOR :SDSC VALUE :SEL-INDEX
          :SELDATA = DATA, :SELIND = INDICATOR,
          :SELLEN = RETURNED LENGTH END-EXEC.
    IF (SELIND = -1)
       move " NULL" to SELDATA.
    DISPLAY SELDATA(1:SELLEN), " "
          WITH NO ADVANCING.
```

11

# Oracle Dynamic SQL: Method 4

This chapter shows you how to implement Oracle dynamic SQL Method 4, which lets your program accept or build dynamic SQL statements that contain a varying number of host variables.

New applications should be developed using the newer ANSI SQL Method 4 described in "ANSI Dynamic SQL". The ANSI Method 4 supports all Oracle types, while the older Oracle Method 4 does *not* support cursor variables, tables of group items, the DML returning clause, and LOBs.

Subjects discussed include the following:

- Meeting the Special Requirements of Method 4
- Understanding the SQL Descriptor Area (SQLDA)
- The SQLDA Variables
- Prerequisite Knowledge
- The Basic Steps
- A Closer Look at Each Step
- Using Host Tables with Method 4
- Sample Program 10: Dynamic SQL Method 4



For a discussion of dynamic SQL Methods 1, 2, and 3, and an overview of Oracle Method 4, see "Oracle Dynamic SQL".

### **Related Topics**

- ANSI Dynamic SQL
- Oracle Dynamic SQL

# 11.1 Meeting the Special Requirements of Method 4

Before looking into the requirements of Method 4, you should be familiar with the terms select-list item and place-holder. Select-list items are the columns or expressions following the keyword SELECT in a query. For example, the following dynamic query contains three select-list items:

```
SELECT ENAME, JOB, SAL + COMM FROM EMP WHERE DEPTNO = 20
```

Place-holders are dummy bind (input) variables that hold places in a SQL statement for actual bind variables. You do not declare place-holders and can name them anything you like. Place-holders for bind variables are most often used in the SET, VALUES, and WHERE

clauses. For example, the following dynamic SQL statements each contain two place-holders.

```
INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:E, :D)
DELETE FROM DEPT WHERE DEPTNO = :DNUM AND LOC = :DLOC
```

Place-holders cannot reference table or column names.

## 11.1.1 Advantages of Method 4

Unlike Methods 1, 2, and 3, dynamic SQL Method 4 lets your program:

- Accept or build dynamic SQL statements that contain an unknown number of select-list items or place-holders
- Take explicit control over datatype conversion between Oracle and COBOL types

To add this flexibility to your program, you must give the runtime library additional information.

### 11.1.2 Information the Database Needs

Pro\*COBOL generates calls to Oracle for all executable dynamic SQL statements. If a dynamic SQL statement contains no select-list items or place-holders, the database needs no additional information to execute the statement. The following DELETE statement falls into this category:

```
* Dynamic SQL statement...

MOVE 'DELETE FROM EMP WHERE DEPTNO = 30' TO STMT.
```

However, most dynamic SQL statements contain select-list items or place-holders for bind variables, as shown in the following UPDATE statement:

```
* Dynamic SQL statement with place-holders...
MOVE 'UPDATE EMP SET COMM = :C WHERE EMPNO = :E' TO STMT.
```

To execute a dynamic SQL statement that contains select-list items or place-holders for bind variables, or both, the database needs information about the program variables that will hold output or input values. Specifically, the database needs the following information:

- The number of select-list items and the number of bind variables
- The length of each select-list item and bind variable
- The datatype of each select-list item and bind variable
- The memory address of each output variable that will store the value of a selectlist item, and the address of each bind variable

For example, to write the value of a select-list item, the database needs the address of the corresponding output variable.

## 11.1.3 Where the Information is Stored

All the information the database needs about select-list items or place-holders for bind variables, except their values, is stored in a program data structure called the SQL Descriptor Area (SQLDA).



Descriptions of select-list items are stored in a *select SQLDA*, and descriptions of place-holders for bind variables are stored in a *bind SQLDA*.

The values of select-list items are stored in output buffers; the values of bind variables are stored in input buffers. You use the library routine SQLADR to store the addresses of these data buffers in a select or bind SQLDA, so that the database knows where to write output values and read input values.

How do values get stored in these data variables? A FETCH generates output values using a cursor, and input values are filled in by your program, typically from information entered interactively by the user.

### 11.1.4 How Information is Obtained

You use the DESCRIBE statement to help get the information the database needs. The DESCRIBE SELECT LIST statement examines each select-list item to determine its name, datatype, constraints, length, scale, and precision, then stores this information in the select SQLDA for your use. For example, you might use select-list names as column headings in a printout. DESCRIBE also stores the total number of select-list items in the SQLDA.

The DESCRIBE BIND VARIABLES statement examines each place-holder to determine its name and length, then stores this information in an input buffer and bind SQLDA for your use. For example, you might use place-holder names to prompt the user for the values of bind variables.

# 11.2 Understanding the SQL Descriptor Area (SQLDA)

This section describes the SQLDA data structure in detail. You learn how to declare it, what variables it contains, how to initialize them, and how to use them in your program.

# 11.2.1 Purpose of the SQLDA

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or place-holders for bind variables. To process this kind of dynamic SQL statement, your program must explicitly declare SQLDAs, also called *descriptors*. Each descriptor corresponds to a group item in your program.

A *select descriptor* stores descriptions of select-list items and the addresses of output buffers that hold the names and values of select-list items.



The name of a select-list item can be a column name, a column alias, or the text of an expression such as SAL + COMM.

A *bind descriptor* stores descriptions of bind variables and indicator variables and the addresses of input buffers where the names and values of bind variables and indicator variables are stored.

Remember, some descriptor variables contain addresses, not values. Therefore, you must declare data buffers to hold the values. You decide the sizes of the required input and output buffers. Because COBOL does not support pointers, you must use the library subroutine



SQLADR to get the addresses of input and output buffers. You learn how to call SQLADR in the section "Using SQLADR".

### **Related Topics**

Using SQLADR

# 11.2.2 Multiple SQLDAs

If your program has more than one active dynamic SQL statement, each statement must have its own SQLDA. You can declare any number of SQLDAs with different names. For example, you might declare three select SQLDAs named SELDSC1, SELDSC2, and SELDSC3, so that you can FETCH from three concurrently open cursors. However, non-concurrent cursors can reuse SQLDAs.

# 11.2.3 Declaring a SQLDA

To declare select and bind SQLDAs, you can code them into your program using the sample select and bind SQLDAs shown in Figure 11-1. You can modify the table dimensions to suit your needs.



For byte-swapped platforms, use COMP5 instead of COMP when declaring a SQLDA.



Figure 11-1 Sample Pro\*COBOL SQLDA Descriptors and Data Buffers (64 bit)

```
01 BNDDSC.
     02 SQLDNUM
                            PIC S9(9) COMP-5 VALUE 20.
     02 SQLDFND
                            PIC S9(9) COMP-5.
        BNDDVAR
                            OCCURS 20 TIMES.
         03 BNDDV
                            PIC S9(18) COMP-5.
         03 BNDDFMT
                            PIC S9(18) COMP-5.
         03 BNDDVLN
                            PIC S9(9) COMP-5.
         03 BNDDFMTL
                            PIC 59(4) COMP-5.
                            PIC S9(4) COMP-5.
         03 BNDDVTYP
         03 BNDDI
                            PIC 59(18) COMP-5.
         03 BNDDH-VNAME
                            PIC S9(18) COMP-5.
         03 BNDDH-MAX-VNAMEL PIC S9(4) COMP-5.
         03 BNDDH-CUR-VNAMEL PIC S9(4) COMP-5.
         03 BNDDFILL1
                            PIC 59(9) COMP-5.
         03 BNDDI-VNAME
                            PIC S9(18) COMP-5.
         03 BNDDI-MAX-VNAMEL PIC S9(4) COMP-5.
         03 BNDDI-CUR-VNAMEL PIC S9(4) COMP-5.
         03 BNDDFILL2 PIC S9(9) COMP-5.
         03 BNDDFCLP
                            PIC S9(18) COMP-5.
        03 BNDDFCRCP PIC S9(18) COMP-5.
SQLBNDCP OCCURS 20 TIMES PI
     02 SQLBNDCP
                           OCCURS 20 TIMES PIC S9(9) COMP-5.
 01 XBNDDI.
     03 BND-DI
                            OCCURS 20 TIMES PIC S9(4) COMP-5.
 01 SELDSC.
     02 SQLDNUM
                           PIC S9(9) COMP-5 VALUE 20.
     02 SQLDFND
                           PIC 59(9) COMP-5.
     02 SELDVAR
                           OCCURS 20 TIMES.
         03 SELDV
                            PIC S9(18) COMP-5.
         03 SELDFMT
                            PIC S9(18) COMP-5.
         03 SELDVLN
                            PIC S9(9), COMP-5.
         03 SELDFMTL
                            PIC 59(4) COMP-5.
         03 SELDVTYP
                            PIC S9(4) COMP-5.
                            PIC S9(18) COMP-5.
         03 SELDI
         03 SELDH-VNAME
                            PIC S9(18) COMP-5.
         03 SELDH-MAX-VNAMEL PIC S9(4) COMP-5.
         03 SELDH-CUR-VNAMEL PIC S9(4) COMP-5.
         03 SELDETLL1
                            PIC 59(9) COMP-5.
         03 SELDI-VNAME
                            PIC S9(18) COMP-5.
         03 SELDI-MAX-VNAMEL PIC S9(4) COMP-5.
         03 SELDI-CUR-VNAMEL PIC S9(4) COMP-5.
         03 SELDFILL2 PIC S9(9) COMP-5.
         03 SELDFCLP
                            PIC 59(18) COMP-5.
         03 SELDFCRCP
                           PIC S9(18) COMP-5.
     02 SQLDSCCP
                           OCCURS 20 TIMES PIC S9(18) COMP-5.
    XSELDI.
     03 SEL-DI
                             OCCURS 20 TIMES PIC S9(4) COMP-5.
    PRECISION
                              PIC S9(9) COMP-5.
                              PIC 59(9) COMP-5.
 01 SCALE
    NULLS-ALLOWED
                              PIC S9(9) COMP-5.
 01
 01 DISPLAY-LENGTH
                              PIC 59(9) COMP-5.
                              PIC S9(9) COMP-5 VALUE 80.
 01 MAX-LENGTH
```

With SQL Domain metadata support, the SQLDA structure is extended to store domain name and domain length. The SQLDSCCP element is extended to provide SQLCOLHASDOMAIN feature.

### Sample Pro\*COBOL SQLDA Descriptors with SQL Domain metadata

```
01 SELDSC.
   02 SQLDNUM
                                           PIC S9(9) COMP VALUE 20.
   02 SOLDFND
                                           PIC S9(9) COMP.
   02 SELDVAR
                                           OCCURS 20 TIMES.
       03 SELDV
                                           PIC S9(18) COMP.
       03 SELDFMT
                                           PIC S9(18) COMP.
       03 SELDVLN
                                           PIC S9(9) COMP.
       03 SELDFMTL
                                           PIC S9(4)
                                                      COMP.
       03 SELDVTYP
                                           PIC S9(4)
                                                      COMP.
       03 SELDI
                                           PIC S9(18) COMP.
       03 SELDH-VNAME
                                           PIC S9(18) COMP.
```



```
03 SELDH-MAX-VNAMEL
                                            PIC S9(4) COMP.
    03 SELDH-CUR-VNAMEL
                                            PIC S9(4) COMP.
    03 SELDFILL1
                                           PIC S9(9) COMP.
                                           PIC S9(18) COMP.
    03 SELDI-VNAME
    03 SELDI-VNAMEL
03 SELDI-CUR-VNAMEL
                                           PIC S9(4) COMP.
                                            PIC S9(4) COMP.
    03 SELDFILL2
                                            PIC S9(9) COMP.
03 SELDFCLP PIC S9(18) COMP.
03 SELDFCRCP PIC S9(18) COMP.
02 SQLDSCCP OCCURS 20 TIMES PIC S9(18) COMP.
02 SQLDSCD-VNAME OCCURS 20 TIMES PIC S9(18) COMP.
02 SQLDSCD-MAX-VNAMEL OCCURS 20 TIMES PIC S9(4) COMP.
02 SOLDSCD-CUR-VNAMEL OCCURS 20 TIMES PIC S9(4) COMP.
```

The SQLDSCD-VNAME table contains the addresses of data buffers that store select-list domain names as they appear in dynamic SQL statements. You must set the elements of SQLDSCD-VNAME using SQLADR before issuing the DESCRIBE command.

The SQLDSCD-MAX-VNAMEL table contains the maximum lengths of the data buffers that store select-list domain names. The buffers are addressed by the elements of SQLDSCD-VNAME.

The SQLDSCD-CUR-VNAMEL table contains the actual lengths of the names of the select-list domain names. The DESCRIBE command sets the table of actual lengths to the number of characters in each select-list.

You can store the SQLDAs in files (named SELDSC and BNDDSC, for example), and then copy the files into your program with the INCLUDE statement as follows:

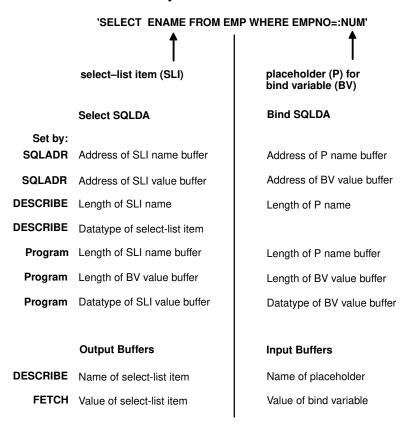
```
EXEC SQL INCLUDE SELDSC END-EXEC. EXEC SQL INCLUDE BNDDSC END-EXEC.
```

Figure 11-2 shows whether variables are set by SQLADR calls, DESCRIBE commands, FETCH commands, or program assignments.



Figure 11-2 How Variables Are Set

### **Dynamic SQL Statement**



# 11.3 The SQLDA Variables

This section explains the purpose and use of each variable in the SQLDA.

### **SQLDNUM**

This variable specifies the maximum number of select-list items or place-holders that can be included in DESCRIBE. Thus, SQLDNUM determines the number of elements in the descriptor tables.

Before issuing a DESCRIBE command, you must set this variable to the dimension of the descriptor tables. After the DESCRIBE, you must reset it to the actual number of variables in the DESCRIBE, which is stored in SQLDFND.

### **SQLDFND**

The SQLDFND variable is the actual number of select-list items or place-holders found by the DESCRIBE command.

SQLDFND is set by DESCRIBE. If SQLDFND is negative, the DESCRIBE command found too many select-list items or place-holders for the size of the descriptor. For example, if you set SQLDNUM to 10 but DESCRIBE finds 11 select-list items or place-holders, SQLDFND is set to -11. If this happens, you cannot process the SQL statement without reallocating the descriptor.



After the DESCRIBE, you must set SQLDNUM equal to SQLDFND.

### SELDV | BNDDV

The **SELDV** | **BNDDV** table contains the addresses of data buffers that store select-list or bind-variable values.

You must set the elements of SELDV or BNDDV using SQLADR.

### **Select Descriptors**

The following statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

directs the database to store FETCHed select-list values in the data buffers addressed by SELDV(1) through SELDV(SQLDNUM). Thus, the database stores the Jth select-list value in SEL-DV(J).

### **Bind Descriptors**

You must set the bind descriptors before issuing the OPEN command. The following statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

directs Oracle to execute the dynamic SQL statement using the bind-variable values addressed by BNDDV(1) through BNDDV(SQLDNUM). (Typically, the values are entered by the user.) The database finds the Jth bind-variable value in BND-DV(J).

### SELDFMT | BNDDFMT

The **SELDFMT** | **BNDDFMT** table contains the addresses of data buffers that store select-list or bind-variable conversion format strings. You can currently use it only for COBOL packed decimals. The format for the conversion string is PP.+SS or PP.-SS where PP is the precision and SS is the scale. For definitions of precision and scale, see "Extracting Precision and Scale" in Coercing Datatypes.

The use of format strings is optional. If you want a conversion format for the Jth select-list item or bind variable, set SELDFMT(J) or BNDDFMT(J) using SQLADR, then store the packed-decimal format (07.+02 for example) in SEL-DFMT or BND-DFMT. Otherwise, set SELDFMT(J) or BNDDFMT(J) to zero.

### SELDVLN | BNDDVLN

The **SELDVLN | BNDDVLN** table contains the lengths of select-list variables or bind-variable values stored in the data buffers.

### **Select Descriptors**

DESCRIBE SELECT LIST sets the table of lengths to the maximum expected for each select-list item. However, you might want to reset some lengths before issuing a FETCH command. FETCH returns at most *n* characters, where *n* is the value of SELDVLN(J) before the FETCH command.

The format of the length differs among datatypes. For CHAR select-list items, DESCRIBE SELECT LIST sets SELDVLN(J) to the maximum length in bytes of the select-list item. For NUMBER select-list items, scale and precision are returned respectively in the low and next-higher bytes of the variable. You can use the library



routine SQLPRC to extract precision and scale values from SELDVLN. See the section "Extracting Precision and Scale".

You must reset SELDVLN(J) to the required length of the data buffer before the FETCH. For example, when coercing a NUMBER to a COBOL character string, set SELDVLN(J) to the precision of the number plus two for the sign and decimal point. When coercing a NUMBER to a COBOL floating point number, set SELDVLN(J) to the length of the appropriate floating point type on your system.

For more information about the lengths of coerced datatypes, see the section "Converting Data".

### **Bind Descriptors**

You must set the Bind Descriptor lengths before issuing the OPEN command. For example, you can use the following statements to set the lengths of bind-variable character strings entered by the user:

```
PROCEDURE DIVISION.

...

PERFORM GET-INPUT-VAR

VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.

...

GET-INPUT-VAR.

DISPLAY "Enter value of ", BND-DH-VNAME(J).

ACCEPT INPUT-STRING.

UNSTRING INPUT-STRING DELIMITED BY " "

INTO BND-DV(J) COUNT IN BNDDVLN(J).
```

Because Oracle accesses a data buffer indirectly, using the address in SELDV(J) or BNDDV(J), it does not know the length of the value in that buffer. If you want to change the length Oracle uses for the Jth select-list or bind-variable value, reset SELDVLN(J) or BNDDVLN(J) to the length you need. Each input or output buffer can have a different length.

### SELDFMTL | BNDDFMTL

This is a table containing the lengths of select-list or bind-variable conversion format strings. Currently, you can use it only for COBOL packed decimal.

The use of format strings is optional. If you want a conversion format for the Jth select-list item or bind variable, set SELDFMTL(J) before the FETCH or BNDDFMTL(J) before the OPEN to the length of the packed-decimal format stored in SEL-DFMT or BND-DFMT. Otherwise, set SELDFMTL(J) or BNDDFMTL(J) to zero.

If the value of SELDFMTL(J) or BNDDFMTL(J) is zero, SELDFMT(J) or BNDDFMT(J) are not used.

### SELDVTYP | BNDDVTYP

The **SELDVTYP** | **BNDDVTYP** table contains the datatype codes of select-list or bind-variable values. These codes determine how Oracle data is converted when stored in the data buffers addressed by elements of SELDV. The datatype descriptor table is further described in "Converting Data".

### **Select Descriptors**

DESCRIBE SELECT LIST sets the table of datatype codes to the *internal* datatype (for example, VARCHAR2, CHAR, NUMBER, or DATE) of the items in the select list.



Before a FETCH is executed, you might want to reset some datatypes because the internal format of datatypes can be difficult to handle. For display purposes, it is usually a good idea to coerce the datatype of select-list values to VARCHAR2. For calculations, you might want to coerce numbers from Oracle to COBOL format. See "Coercing Datatypes".

The high bit of SELDVTYP(J) is set to indicate the NULL/not NULL status of the Jth select-list column. You must always clear this bit before issuing an OPEN or FETCH command. Use the library routine SQLNUL to retrieve the datatype code and clear the NULL/not NULL bit. For more information, see: "Handling NULL/Not NULL Datatypes".

It is best to change the NUMBER internal datatype to an external datatype compatible with that of the COBOL data buffer addressed by SELDV(J).

### **Bind Descriptors**

DESCRIBE BIND VARIABLES sets the table of datatype codes to zeros. You must reset the table of datatypes before issuing the OPEN command. The code represents the external (COBOL) datatype of the buffer addressed by BNDDV(J). Often, bind-variable values are stored in character strings, so the datatype table elements are set to 1 (the VARCHAR2 datatype code).

To change the datatype of the Jth select-list or bind-variable value, reset SELDVTYP(J) or BNDDVTYP(J) to the datatype you want.

### SELDI | BNDDI

The **SELDI | BNDDI** table contains the addresses of data buffers that store indicator-variable values. You must set the elements of SELDI or BNDDI using SQLADR.

### **Select Descriptors**

You must set this table before issuing the FETCH command. When Oracle executes the statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

if the Jth returned select-list value is NULL, the buffer addressed by SELDI(J) is set to -1. Otherwise, it is set to zero (the value is not NULL) or a positive integer (the value was truncated).

### **Bind Descriptors**

You must initialize this table and set the associated indicator variables before issuing the OPEN command. When Oracle executes the statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

the buffer addressed by BNDDI(J) determines whether the Jth bind variable is NULL. If the value of an indicator variable is -1, its associated bind variable is NULL.

### SELDH-VNAME | BNDDH-VNAME

The **SELDH-VNAME** | **BNDDH-VNAME** table contains the addresses of data buffers that store select-list or place-holder names as they appear in dynamic SQL statements. You must set the elements of SELDH-VNAME or BNDDH-VNAME using SQLADR before issuing the DESCRIBE command.



DESCRIBE directs Oracle to store the name of the Jth select-list item or place-holder in the data buffer addressed by SELDH-VNAME(J) or BNDDH-VNAME(J). Thus, Oracle stores the Jth select-list or place-holder name in SEL-DH-VNAME(J) or BND-DH-VNAME(J).



The **SELDH-VNAME** | **BNDDH-VNAME** table contains only the name of the column, and *not* the table-qualifier.column name, even if you provide it in your SQL statement. If, for example, you were to do a describe of select-list in the SQL statement select a.owner from all\_tables the software will return not a.owner, but instead, owner. If necessary, you should use column aliases to correctly identify a column in the select list.

### SELDH-MAX-VNAMEL | BNDDH-MAX-VNAMEL

The **SELDH-MAX-VNAMEL** | **BNDDH-MAX-VNAMEL** table contains the maximum lengths of the data buffers that store select-list or place-holder names. The buffers are addressed by the elements of SELDH-VNAME or BNDDH-VNAME.

You must set the elements of SELDH-MAX-VNAMEL or BNDDH-MAX-VNAMEL before issuing the DESCRIBE command. Each select-list or place-holder name buffer can have a different length.

### SELDH-CUR-VNAMEL | BNDDH-CUR-VNAMEL

The **SELDH-CUR-VNAMEL** | **BNDDH-CUR-VNAMEL** table contains the actual lengths of the names of the select-list or place-holder. DESCRIBE sets the table of actual lengths to the number of characters in each select-list or place-holder name.

### SELDI-VNAME | BNDDI-VNAME

The **SELDI-VNAME** | **BNDDI-VNAME** table contains the addresses of data buffers that store indicator-variable names.

You can associate indicator-variable values with select-list items and bind variables. However, you can associate indicator-variable names only with bind variables. You can use this table only with bind descriptors. You must set the elements of BNDDI-VNAME using SQLADR before issuing the DESCRIBE command.

DESCRIBE BIND VARIABLES directs Oracle to store any indicator-variable names in the data buffers addressed by BNDDI-VNAME(1) through BNDDI-VNAME(SQLDNUM). Thus, Oracle stores the Jth indicator-variable name in BND-DI-VNAME(J).

### SELDI-MAX-VNAMEL | BNDDI-MAX-VNAMEL

The **SELDI-MAX-VNAMEL | BNDDI-MAX-VNAMEL** table contains the maximum lengths of the data buffers that store indicator-variable names. The buffers are addressed by the elements of SELDI-VNAME or BNDDI-VNAME.

You can associate indicator-variable names only with bind variables. You can use this table only with bind descriptors.

You must set the elements BNDDI-MAX-VNAMEL(1) through BNDDI-MAX-VNAMEL(SQLDNUM) before issuing the DESCRIBE command. Each indicator-variable name buffer can have a different length.



### SELDI-CUR-VNAMEL | BNDDI-CUR-VNAMEL

The **SELDI-CUR-VNAMEL** | **BNDDI-CUR-VNAMEL** table contains the actual lengths of the names of the indicator variables. You can associate indicator-variable names only with bind variables. You can use this table only with bind descriptors.

DESCRIBE BIND VARIABLES sets the table of actual lengths to the number of characters in each indicator-variable name.

### SELDFCLP | BNDDFCLP

The **SELDFCLP** | **BNDDFCLP** table is reserved for future use. It must be present because Oracle expects the group item SELDSC or BNDDSC to be a certain size. You must currently set the elements of SELDFCLP and BNDDFCLP to zero.

### SELDFCRCP | BNDDFCRCP

The **SELDFCRCP** | **BNDDFCRCP** table is reserved for future use. It must be present because Oracle expects the group item SELDSC or BNDDSC to be a certain size. You must set the elements of SELDFCRCP and BNDDFCRCP to zero.

### **Related Topics**

- Converting Data
- Coercing Datatypes
- Handling NULL/Not NULL Datatypes

# 11.4 Prerequisite Knowledge

You need a working knowledge of the following subjects to implement dynamic SQL Method 4:

- Using the library routine SQLADR
- converting data
- coercing datatypes
- handling NULL/not NULL datatypes

# 11.4.1 Using SQLADR

You must call the library subroutine SQLADR to get the addresses of data buffers that store input and output values. You store the addresses in a bind or select SQLDA so that Oracle knows where to read bind-variable values or write select-list values.

### Call SQLADR using the syntax

CALL "SQLADR" USING BUFFER, ADDRESS.

where:

#### **BUFFER**

Is a data buffer that stores the value or name of a select-list item, bind variable, or indicator variable.



#### **ADDRESS**

Is an integer variable that returns the address of the data buffer.

A call to SQLADR stores the address of BUFFER in ADDRESS. The next example uses SQLADR to initialize the select descriptor tables SELDV, SELDH-VNAME, and SELDI. Their elements address data buffers for select-list values, select-list names, and indicator values.

```
PROCEDURE DIVISION.

...

PERFORM INIT-SELDSC

VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.

...

INIT-SELDSC.

CALL "SQLADR" USING SEL-DV(J), SELDV(J).

CALL "SQLADR" USING SEL-DH-VNAME(J), SELDH-VNAME(J).

CALL "SQLADR" USING SEL-DI(J), SELDI(J).
```

## 11.4.2 Converting Data

This section provides more detail about the datatype descriptor table. In host programs that use neither datatype equivalencing nor dynamic SQL Method 4, the conversion between internal and external datatypes is determined at precompile time. By default, Pro\*COBOL assigns a specific external datatype to each host variable. For example, Pro\*COBOL assigns the INTEGER external datatype to host variables of type PIC S9(n) COMP.

However, Method 4 lets you control data conversion and formatting. You specify conversions by setting datatype codes in the datatype descriptor table.

### **Internal Datatypes**

Internal datatypes specify the formats used by Oracle to store column values in database tables to represent pseudocolumn values.

When you issue a DESCRIBE SELECT LIST command, Oracle returns the internal datatype code for each select-list item to the SELDVTYP (datatype) descriptor table. For example, the datatype code for the Jth select-list item is returned to SELDVTYP(J).

Table 11-1 shows the internal datatypes and their codes:

Table 11-1 Internal Datatypes and Related Codes

Internal Datatype	Code
VARCHAR2	1
NUMBER	2
LONG	8
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHAR	96

### **External Datatypes**

External datatypes specify the formats used to store values in input and output host variables.

The DESCRIBE BIND VARIABLES command sets the BNDDVTYP table of datatype codes to zeros. Therefore, you must reset the codes *before* issuing the OPEN command. The codes tell Oracle which external datatypes to expect for the various bind variables. For the Jth bind variable, reset BNDDVTYP(J) to the external datatype you want.

The following table shows the external datatypes and their codes, as well as the corresponding COBOL datatypes:

Table 11-2 Oracle External and Related COBOL Datatypes

4 bit
PARATE
PARATE

### Notes:

- 1. For use in an EXEC SQL VAR statement only.
- 2. Include the *n*-byte length field.



For more information about the datatypes and their formats, see "The Oracle Database Datatypes".

### **PL/SQL Datatypes**

PL/SQL provides a variety of predefined scalar and composite datatypes. A *scalar* type has no internal components. A *composite* type has internal components that can be manipulated individually. "Initialize the Descriptors" shows the predefined PL/SQL scalar datatypes and their internal datatype equivalence

Table 11-3 PL/SQL Datatype Equivalences with Internal Datatypes

PL/SQL Datatype	Oracle Internal Datatype
VARCHAR	VARCHAR2
VARCHAR2	
BINARY_INTEGER	NUMBER
DEC	
DECIMAL	
DOUBLE PRECISION	
FLOAT	
INT	
INTEGER	
NATURAL	
NUMBER	
NUMERIC	
POSITIVE	
REAL SMALLINT	
	1010
LONG	LONG
ROWID	ROWID
DATE	DATE
RAW	RAW
LONG RAW	LONG RAW
CHAR	CHAR
CHARACTER	
STRING	

### **Related Topics**

- The Oracle Database Datatypes
- Initialize the Descriptors

# 11.4.3 Coercing Datatypes

For a select descriptor, DESCRIBE SELECT LIST can return any of the internal datatypes. Often, as in the case of character data, the internal datatype corresponds exactly to the external datatype you want to use. However, a few internal datatypes map to external datatypes that can be difficult to handle. Thus, you might want to reset some elements in the SELDVTYP descriptor table.



For example, you might want to reset NUMBER values to FLOAT values, which correspond to PIC S9(n)V9(n) COMP-1 values in COBOL. Oracle does any necessary conversion between internal and external datatypes at FETCH time. Be sure to reset the datatypes *after* the DESCRIBE SELECT LIST but *before* the FETCH.

For a bind descriptor, DESCRIBE BIND VARIABLES does *not* return the datatypes of bind variables, only their number and names. Therefore, you must explicitly set the BNDDVTYP table of datatype codes to tell Oracle the external datatype of each bind variable. Oracle does any necessary conversion between external and internal datatypes at OPEN time.

When you reset datatype codes in the SELDVTYP or BNDDVTYP descriptor table, you are "coercing datatypes." For example, to coerce the Jth select-list value to VARCHAR2, use the following statement:

```
* Coerce select-list value to VARCHAR2.
MOVE 1 TO SELDVTYP(J).
```

When coercing a NUMBER select-list value to VARCHAR2 for display purposes, you must also extract the precision and scale bytes of the value and use them to compute a maximum display length. Then, before the FETCH, you must reset the appropriate element of the SELDVLN (length) descriptor table to tell Oracle the buffer length to use. To specify the length of the Jth select-list value, set SELDVLN(J) to the length you need.

For example, if DESCRIBE SELECT LIST finds that the Jth select-list item is of type NUMBER, and you want to store the returned value in a COBOL variable declared as PIC S9(n)V9(n) COMP-1, simply set SELDVTYP(J) to 4 and SELDVLN(J) to the length of COMP-1 numbers on your system.

### **Exceptions**

In some cases, the internal datatypes that DESCRIBE SELECT LIST returns might not suit your purposes. Two examples of this are DATE and NUMBER. When you DESCRIBE a DATE select-list item, Oracle returns the datatype code 12 to the SELDVTYP table. Unless you reset the code before the FETCH, the date value is returned in its 7-byte internal format. To get the date in its default character format, you must change the datatype code from 12 to 1 (VARCHAR2) and increase the SELDVLN value from 7 to 9.

Similarly, when you DESCRIBE a NUMBER select-list item, Oracle returns the datatype code 2 to the SELDVTYP table. Unless you reset the code before the FETCH, the numeric value is returned in its internal format, which is probably not desired. Therefore, change the code from 2 to 1 (VARCHAR2), 3 (INTEGER), 4 (FLOAT), or some other appropriate datatype.

### **Extracting Precision and Scale**

The library subroutine SQLPRC extracts precision and scale. Normally, it is used after the DESCRIBE SELECT LIST, and its first parameter is SELDVLN(J). To call SQLPRC, use the following syntax

```
CALL "SQLPRC" USING LENGTH, PRECISION, SCALE.
```

where:



Syntax	Description	
LENGTH	Is an integer variable that stores the length of a NUMBER value. The scale and precision of the value are stored in the low and next-higher bytes, respectively.	
PRECISION	Is an integer variable that returns the <i>precision</i> of the NUMBER value. Precision is the number of significant digits. It is set to zero if the select-list item refers to a NUMBER of unspecified size. In this case, because the size is unspecified, assume the maximum precision, 38.	
SCALE	Is an integer variable that returns the <i>scale</i> of the NUMBER value. Scale specifies where rounding will occur. For example, a scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46); a scale of -3 means that the number is rounded to the nearest thousand (3.456 becomes 3000).	

The following example shows how SQLPRC is used to compute maximum display lengths for NUMBER values that will be coerced to VARCHAR2:

```
WORKING-STORAGE SECTION.
01 PRECISION PIC S9(9) COMP.
01 SCALE
                 PIC S9(9) COMP.
01 DISPLAY-LENGTH PIC S9(9) COMP.
01 MAX-LENGTH PIC S9(9) COMP VALUE 80.
PROCEDURE DIVISION.
    PERFORM ADJUST-LENGTH
       VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
ADJUST-LENGTH.
    If datatype is NUMBER, extract precision and scale.
    IF SELDVTYP(J) = 2
       CALL "SQLPRC" USING SELDVLN(J), PRECISION, SCALE.
    MOVE 0 TO DISPLAY-LENGTH.
    Precision is set to zero if the select-list item
   refers to a NUMBER of unspecified size. We allow for
    a maximum precision of 10.
    IF SELDVTYP(J) = 2 AND PRECISION = 0
       MOVE 10 TO DISPLAY-LENGTH.
   Allow for possible decimal point and sign.
    IF SELDVTYP(J) = 2 AND PRECISION > 0
       ADD 2 TO PRECISION
       MOVE PRECISION TO DISPLAY-LENGTH.
```

Notice that the first parameter in the subroutine call is the Jth element in the table of select-list lengths.

The SQLPRC procedure, defined in the SQLLIB runtime library, returns zero as the precision and scale values for certain SQL datatypes. The SQLPR2 procedure is similar to SQLPRC in that it has the same syntax and returns the same binary values, except for the datatypes shown in this table:

Table 11-4 Datatype Exceptions to the SQLPR2 Procedure

SQL Datatype	Binary Precision	Binary Scale
FLOAT	126	-127



Table 11-4 (Cont.) Datatype Exceptions to the SQLPR2 Procedure

SQL Datatype	Binary Precision	Binary Scale
FLOAT(n)	n (range is 1 126)	-127
REAL	63	-127
DOUBLE PRECISION	126	-127

# 11.4.4 Handling NULL/Not NULL Datatypes

For every select-list column (not expression), DESCRIBE SELECT LIST returns a NULL/not NULL indication in the datatype table of the select descriptor. If the Jth select-list column is constrained to be not NULL, the high-order bit of SELDVTYP(J) datatype variable is clear; otherwise, it is set.

Before using the datatype in an OPEN or FETCH statement, if the NULL status bit is set, you must clear it. Never set the bit.

You can use the library routine SQLNUL to find out if a column allows NULL datatypes and to clear the datatype's NULL status bit. You call SQLNUL using the syntax

```
CALL "SQLNUL" USING VALUE-TYPE, TYPE-CODE, NULL-STATUS.
```

#### where:

Syntax	Description	
VALUE-TYPE	Is a 2-byte integer variable that stores the datatype code of a select-list column.	
TYPE-CODE	Is a 2-byte integer variable that returns the datatype code of the select-list column with the high-order bit cleared.	
NULL-STATUS	Is an integer variable that returns the NULL status of the select-list column. 1 means that the column allows NULLs; 0 means that it does not.	

### The following example shows how to use SQLNUL:

```
WORKING-STORAGE SECTION.

Declare variable for subroutine call.
01 NULL-STATUS PIC S9(9) COMP.

PROCEDURE DIVISION.

MAIN.

EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.

PERFORM HANDLE-NULLS

VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.

HANDLE-NULLS.

* Find out if column is NOT NULL, and clear high-order bit.
CALL "SQLNUL" USING SELDVTYP(J), SELDVTYP(J), NULL-STATUS.

* If NULL-STATUS = 1, NULLs are allowed.
```



Notice that the first and second parameters in the subroutine call are the same. Respectively, they are the datatype variable before and after its NULL status bit is cleared.

# 11.5 The Basic Steps

Method 4 can be used to process *any* dynamic SQL statement. In the example in "Using Host Tables with Method 4", a query is processed so that you can see how both input and output host variables are handled.

To process the dynamic query, our example program takes the following steps:

- Declare a host string to hold the query text.
- 2. Declare select and bind descriptors.
- 3. Set the maximum number of select-list items and place-holders that can be DESCRIBEd.
- 4. Initialize the select and bind descriptors.
- Store the query text in the host string.
- PREPARE the query from the host string.
- 7. DECLARE a cursor FOR the query.
- 8. DESCRIBE the bind variables INTO the bind descriptor.
- 9. Reset the number of place-holders to the number actually found by DESCRIBE.
- 10. Get values for the bind variables found by DESCRIBE.
- 11. OPEN the cursor USING the bind descriptor.
- 12. DESCRIBE the select list INTO the select descriptor.
- 13. Reset the number of select-list items to the number actually found by DESCRIBE.
- 14. Reset the length and datatype of each select-list item for display purposes.
- 15. FETCH a row from the database INTO data buffers using the select descriptor.
- 16. Process the select-list values returned by FETCH.
- 17. CLOSE the cursor when there are no more rows to FETCH.



If the dynamic SQL statement is *not* a query or contains a known number of select-list items or place-holders, then some of the preceding steps are unnecessary.

### **Related Topics**

Using Host Tables with Method 4

# 11.6 A Closer Look at Each Step

This section discusses each step in more detail. A full-length example program illustrating Method 4 is seen at the end of this chapter. With Method 4, you use the following sequence of embedded SQL statements:



```
EXEC SQL
   PREPARE <statement name>
   FROM {:<host string> | <string literal>}
EXEC SOL
   DECLARE <cursor name> CURSOR FOR <statement name>
END-EXEC.
EXEC SOL
   DESCRIBE BIND VARIABLES FOR <statement name>
   INTO <bind descriptor name>
END-EXEC.
EXEC SQL
   OPEN <cursor name>
    [USING DESCRIPTOR <bind descriptor name>]
EXEC SQL
   DESCRIBE [SELECT LIST FOR] <statement name>
   INTO <select descriptor name>
END-EXEC.
EXEC SQL
   FETCH <cursor name> USING DESCRIPTOR <select descriptor name>
END-EXEC.
EXEC SQL
   CLOSE <cursor name>
END-EXEC.
```

If the number of select-list items in a dynamic query is known, you can omit DESCRIBE SELECT LIST and use the following Method 3 FETCH statement:

```
EXEC SQL FETCH <cursor_name> INTO <host_variable_list> END-EXEC.
```

Alternatively, if the number of place-holders for bind variables in a dynamic SQL statement is known, you can omit DESCRIBE BIND VARIABLES and use the following Method 3 OPEN statement:

```
EXEC SQL OPEN <cursor name> [USING <host variable list>] END-EXEC.
```

The next section illustrates how these statements allow your host program to accept and process a dynamic SQL statement using descriptors.

### Note:

Several figures accompany the following discussion. To avoid cluttering the figures, it was necessary to confine descriptor tables to 3 elements and to limit the maximum length of names and values to 5 and 10 characters, respectively.

# 11.6.1 Declare a Host String

Your program needs a host variable to store the text of the dynamic SQL statement. The host variable (SELECTSTMT in our example) must be declared as a character string:

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



```
01 SELECTSTMT PIC X(120).
EXEC SQL END DECLARE SECTION END-EXEC.
```

# 11.6.2 Declare the SQLDAs

Because the query in our example might contain an unknown number of select-list items or place-holders, you must declare select and bind descriptors. Instead of hard-coding the SQLDAs, you use INCLUDE to copy them into your program, as follows:

```
EXEC SQL INCLUDE SELDSC END-EXEC. EXEC SQL INCLUDE BNDDSC END-EXEC.
```

For reference, the INCLUDEd declaration of SELDSC follows:

```
WORKING-STORAGE SECTION.
01 SELDSC.
              SQLDNUM PIC S9(9) COMP.
SQLDFND PIC S9(9) COMP.
SELDVAR OCCURS 3 TIMES.
10 SELDV PIC S9(9) COMP.
10 SELDFMT PIC S9(9) COMP.
10 SELDVLN PIC S9(9) COMP.
10 SELDFMTL PIC S9(4) COMP.
10 SELDFMTL PIC S9(4) COMP.
10 SELDVTYP PIC S9(4) COMP.
10 SELDI PIC S9(9) COMP.
          05 SOLDNUM
          05 SOLDFND
          05 SELDVAR
                                        PIC S9(9) COMP.
               10 SELDI
               10 SELDH-VNAME
                                        PIC S9(9) COMP.
               10 SELDH-MAX-VNAMEL PIC S9(4) COMP.
               10 SELDH-CUR-VNAMEL PIC S9(4) COMP.
               10 SELDI-VNAME PIC S9(9) COMP.
               10 SELDI-MAX-VNAMEL PIC S9(4) COMP.
               10 SELDI-CUR-VNAMEL PIC S9(4) COMP.
               10 SELDFCLP PIC S9(9) COMP.
10 SELDFCRCP PIC S9(9) COMP.
01 XSELDI.
    05 SEL-DI
                         OCCURS 3 TIMES PIC S9(9) COMP.
01 XSELDIVNAME.
        05 SEL-DI-VNAME OCCURS 3 TIMES PIC X(5).
01 XSELDV.
         05 SEL-DV
                            OCCURS 3 TIMES PIC X(10).
01 XSELDHVNAME.
          05 SEL-DH-VNAME OCCURS 3 TIMES PIC X(5).
```

## 11.6.3 Set the Maximum Number to DESCRIBE

You next set the maximum number of select-list items or place-holders that can be described, as follows:

```
MOVE 3 TO SQLDNUM IN SELDSC. MOVE 3 TO SQLDNUM IN BNDDSC.
```

## 11.6.4 Initialize the Descriptors

You must initialize several descriptor variables. Some require the library subroutine SQLADR.

In our example, you store the maximum lengths of name buffers in the SELDH-MAX-VNAMEL, BNDDH-MAX-VNAMEL, and BNDDI-MAX-VNAMEL tables, and use SQLADR to

store the addresses of value and name buffers in the SELDV, SELDI, BNDDV, BNDDI, SELDH-VNAME, BNDDH-VNAME, and BNDDI-VNAME tables.

```
PROCEDURE DIVISION.
    PERFORM INIT-SELDSC
       VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
    PERFORM INIT-BNDDSC
       VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
INIT-SELDSC.
   MOVE SPACES TO SEL-DV(J).
   MOVE SPACES TO SEL-DH-VNAME (J) .
   MOVE 5 TO SELDH-MAX-VNAMEL(J).
    CALL "SQLADR" USING SEL-DV(J), SELDV(J).
    CALL "SQLADR" USING SEL-DH-VNAME(J), SELDH-VNAME(J).
   CALL "SQLADR" USING SEL-DI(J), SELDI(J).
    . . .
INIT-BNDDSC.
   MOVE SPACES TO BND-DV(J).
   MOVE SPACES TO BND-DH-VNAME(J).
   MOVE SPACES TO BND-DI-VNAME(J).
   MOVE 5 TO BNDDH-MAX-VNAMEL(J).
   MOVE 5 TO BNDDI-MAX-VNAMEL(J).
    CALL "SQLADR" USING BND-DV(J), BNDDV(J).
    CALL "SQLADR" USING BND-DH-VNAME(J), BNDDH-VNAME(J).
    CALL "SQLADR" USING BND-DI(J), BNDDI(J).
    CALL "SQLADR" USING BND-DI-VNAME(J), BNDDI-VNAME(J).
```

Figure 11-3 and Figure 11-4 represent the resulting descriptors.

Figure 11-3 Initialized Select Descriptor

SQLDNUM	3	, Data Buffers
SQLDFND		
SELDV	address of SEL-DV(1) address of SEL-DV(2) address of SEL-DV(3)	For values of select-list items:
SELDVLN	1 2 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 2 3 4 5 6 7 8 9 10
SELDTYP	3	
SELDI	address of SEL-DI(1) address of SEL-DI(2) address 0f SEL-DI(3)	For values of indicators:
SELDH_VNAME	address of SEL-DH-VNAME(1) address of SEL-DH-VNAME(2) address of SEL-DH-VNAME(3)	
SELDH_MAX_VNAMEL	1 5 2 5 3 5	1 2 3 4 5
SELDH_CUR_VNAMEL	1 2 3	



SQLDNUM **Data Buffers** 3 SQLDFND For values of bind variables: 1 address of BND-DV(1) BNDDV 2 address of BND-DV(2) 3 address of BND-DV(3) 1 2 3 4 5 6 2 **BNDDVLN** 3 2 **BNDDVTYP** 3 For values of indicators: address of BND-DI(1) BNDDI 2 address of BND-DI(2) 2 3 address of BND-DI(3) 3 address of BND-DI-VNAME(1) For names of placeholders: address of BND-DI-VNAME(2) 2 **BNDDH-VNAME** address of BND-DI-VNAME(3) 3 2 3 1 2 3 4 5 5 BNDDH-MAX-VNAMEL 3 5 BNDDH-CUR-VNAMEL 3 address of BND-DI-VNAME(1) For names of placeholders: 2 BNDDH-VNAME address of BND-DI-VNAME(2) 3 address of BND-DI-VNAME(3) 2 3 2 1 2 3 4 5 5 BNDDH-MAX-VNAMEL 3 5 BNDDH-CUR-VNAMEL 2 3

Figure 11-4 Initialized Bind Descriptor

# 11.6.5 Store the Query Text in the Host String

Next, you prompt the user for a SQL statement, then store the input string in SELECTSTMT as follows:

DISPLAY "Enter a SELECT statement: " WITH NO ADVANCING. ACCEPT SELECTSTMT.

We assume the user entered the following string:

SELECT ENAME, EMPNO, COMM FROM EMP WHERE COMM < : BONUS

# 11.6.6 PREPARE the Query from the Host String

PREPARE parses the SQL statement and gives it a name. In our example, PREPARE parses the host string SELECTSTMT and gives it the name SQLSTMT, as follows:

EXEC SQL PREPARE SQLSTMT FROM : SELECTSTMT END-EXEC.

### 11.6.7 DECLARE a Cursor

DECLARE CURSOR defines a cursor by giving it a name and associating it with a specific SELECT statement.

To declare a cursor for static queries, use the following syntax:

```
EXEC SQL DECLARE cursor name CURSOR FOR SELECT ...
```

To declare a cursor for dynamic queries, the statement name given to the dynamic query by PREPARE replaces the static query. In our example, DECLARE CURSOR defines a cursor named EMPCURSOR and associates it with SOLSTMT, as follows:

EXEC SQL DECLARE EMPCURSOR CURSOR FOR SQLSTMT END-EXEC.



You must declare a cursor for all dynamic SQL statements, not just queries. With non-queries, OPENing the cursor executes the dynamic SQL statement.

### 11.6.8 DESCRIBE the Bind Variables

DESCRIBE BIND VARIABLES puts descriptions of bind variables into a bind descriptor. In our example, DESCRIBE readies BNDDSC as follows:

```
EXEC SQL

DESCRIBE BIND VARIABLES FOR SQLSTMT

INTO BNDDSC

END-EXEC.
```

Note that BNDDSC must *not* be prefixed with a colon.

The DESCRIBE BIND VARIABLES statement must follow the PREPARE statement but precede the OPEN statement.

Figure 11-5 shows the bind descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SQLDFND to the actual number of place-holders found in the processed SQL statement.

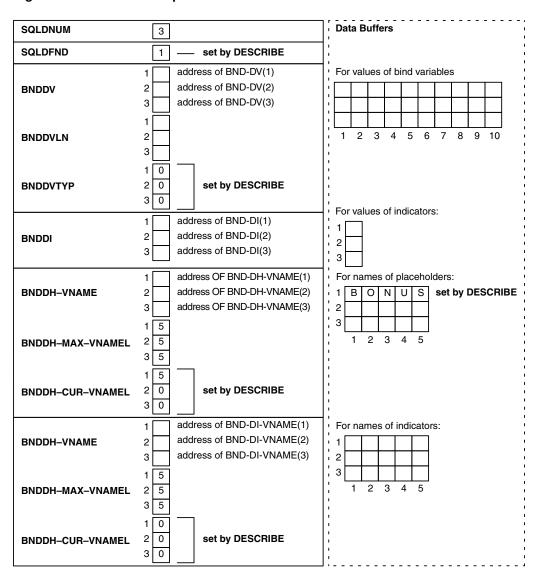


Figure 11-5 Bind Descriptor after the DESCRIBE

### 11.6.9 Reset Number of Place-Holders

Next, you must reset the maximum number of place-holders to the number actually found by DESCRIBE, as follows:

```
IF SQLDFND IN BNDDSC < 0
   DISPLAY "Too many bind variables"
   GOTO ROLL-BACK
ELSE
   MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC
END-IF.</pre>
```



### 11.6.10 Get Values for Bind Variables

Your program must get values for the bind variables in the SQL statement. How the program gets the values is up to you. For example, they can be hard-coded, read from a file, or entered interactively.

In our example, a value must be assigned to the bind variable that replaces the place-holder BONUS in the query WHERE clause. Prompt the user for the value, then process it, as follows:

```
PROCEDURE DIVISION.
...

PERFORM GET-INPUT-VAR

VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
...

GET-INPUT-VAR.
...

* Replace the 0 DESCRIBEd into the datatype table
* with a 1 to avoid an "invalid datatype" Oracle error.
MOVE 1 TO BNDDVTYP(J).

* Get value of bind variable.
DISPLAY "Enter value of ", BND-DH-VNAME(J).
ACCEPT INPUT-STRING.
UNSTRING INPUT-STRING DELIMITED BY " "
INTO BND-DV(J) COUNT IN BNDDVLN(J).
```

Assuming that the user supplied a value of 625 for BONUS, the next table shows the resulting bind descriptor.



**Data Buffers SQLDNUM** 1 reset by program **SQLDFND** 1 address of BND-DV(1) For values of bind variables: 2 address of BND-DV(2) **BNDDV** 6 2 5 address of BND-DV(3) 3 1 3 set by program **BNDDVLN** 2 1 2 3 4 5 6 7 8 9 10 3 1 reset by program 2 0 **BNDDVTYP** 3 0 For values of indicators: address of BND-DI(1) 1 1 0 — set by program address of BND-DI(2) **BNDDI** 2 2 3 address of BND-DI(3) 3 address of BND-DH-VNAME(1) 1 For names of placeholders: address of BND-DH-VNAME(2) 2 BONUS **BNDDH-VNAME** address of BND-DH-VNAME(3) 3 2 3 1 5 1 2 3 4 5 5 BNDDH-MAX-VNAMEL 2 3 5 5 1 **BNDDH-CUR-VNAMEL** 2 0 3 [ 0 address of BND-DI-VNAME(1) For names of indicators: 1 address of BND-DI-VNAME(2) BNDDH-VNAME 2 address of BND-DI-VNAME(3) 3 2 3 1 **[** 5 2 3 4 BNDDH-MAX-VNAMEL 2 5 3 5 0 1 **BNDDH-CUR-VNAMEL** 2 0 3 0

Figure 11-6 Bind Descriptor after Assigning Values

### 11.6.11 OPEN the Cursor

The OPEN statement for dynamic queries is similar to the one for static queries, except the cursor is associated with a bind descriptor. Values determined at run time and stored in buffers addressed by elements of the bind descriptor tables are used to evaluate the SQL statement. With queries, the values are also used to identify the active set.

In our example, OPEN associates EMPCURSOR with BNDDSC as follows:

EXEC SQL
OPEN EMPCUR USING DESCRIPTOR BNDDSC
END-EXEC.



Remember, BNDDSC must *not* be prefixed with a colon.

Then, OPEN executes the SQL statement. With queries, OPEN also identifies the active set and positions the cursor at the first row.

### 11.6.12 DESCRIBE the Select List

If the dynamic SQL statement is a query, the DESCRIBE SELECT LIST statement must follow the OPEN statement but precede the FETCH statement.

DESCRIBE SELECT LIST puts descriptions of select-list items into a select descriptor. In our example, DESCRIBE readies SELDSC as follows:

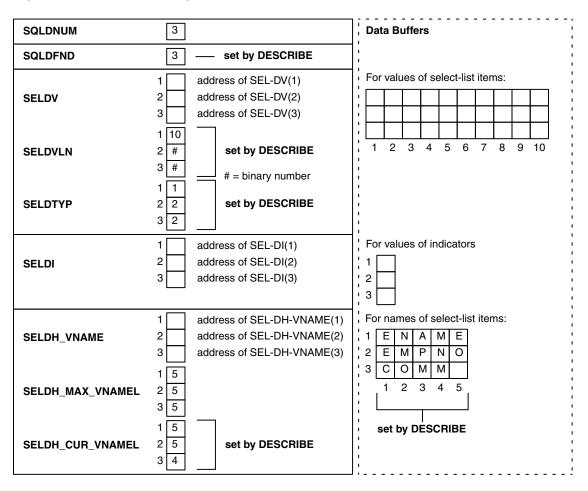
```
EXEC SQL

DESCRIBE SELECT LIST FOR SQLSTMT INTO SELDSC END-EXEC.
```

Accessing the data dictionary, DESCRIBE sets the length and datatype of each select-list value.

Figure 11-7 shows the select descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SQLDFND to the actual number of items found in the query select list. If the SQL statement is not a query, SQLDFND is set to zero. Also notice that the NUMBER lengths are not usable yet. For columns defined as NUMBER, you must use the library subroutine SQLPRC to extract precision and scale. See the section "Coercing Datatypes".

Figure 11-7 Select Descriptor after the DESCR





#### **Related Topics**

Coercing Datatypes

### 11.6.13 Reset Number of Select-List Items

Next, you must reset the maximum number of select-list items to the number actually found by DESCRIBE, as follows:

```
MOVE SQLDFND IN SELDSC TO SQLDNUM IN SELDSC.
```

### 11.6.14 Reset Length/Datatype of Each Select-List Item

Before fetching the select-list values, the example resets some elements in the length and datatype tables for display purposes.

```
PROCEDURE DIVISION.
    PERFORM COERCE-COLUMN-TYPE
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
COERCE-COLUMN-TYPE.
    Clear NULL bit.
    CALL "SQLNUL" USING SELDVTYP(J), SELDVTYP(J), NULL-STATUS.
    If datatype is DATE, lengthen to 9 characters.
    IF SELDVTYP(J) = 12
       MOVE 9 TO SELDVLN(J).
    If datatype is NUMBER, extract precision and scale.
    MOVE 0 TO DISPLAY-LENGTH.
    IF SELDVTYP(J) = 2 AND PRECISION = 0
       MOVE 10 TO DISPLAY-LENGTH.
    IF SELDVTYP(J) = 2 AND PRECISION > 0
       ADD 2 TO PRECISION
       MOVE PRECISION TO DISPLAY-LENGTH.
    IF SELDVTYP(J) = 2
        IF DISPLAY-LENGTH > MAX-LENGTH
            DISPLAY "Column value too large for data buffer."
            GO TO END-PROGRAM
        ELSE
            MOVE DISPLAY-LENGTH TO SELDVLN(J).
    Coerce datatypes to VARCHAR2.
    MOVE 1 TO SELDVTYP(J).
```

Figure 11-8 shows the resulting select descriptor. Notice that the NUMBER lengths are now usable and that all the datatypes are VARCHAR2. The lengths in SELDVLN(2) and SELDVLN(3) are 6 and 9 because we increased the DESCRIBEd lengths of 4 and 7 by 2 to allow for a possible sign and decimal point.

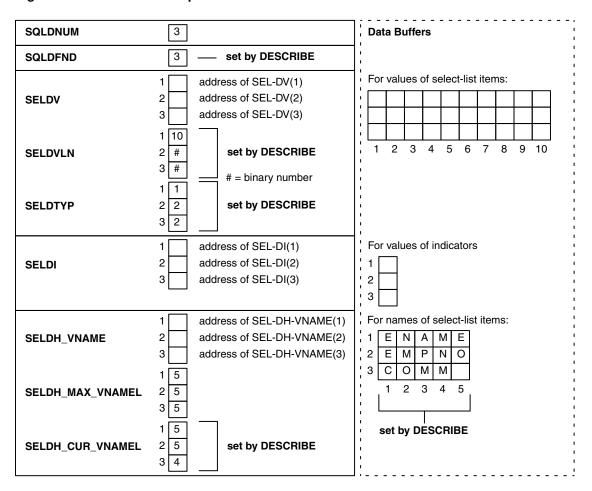


Figure 11-8 Select Descriptor before the FETCH

### 11.6.15 FETCH Rows from the Active Set

FETCH returns a row from the active set, stores select-list values in the data buffers, and advances the cursor to the next row in the active set. If there are no more rows, FETCH sets SQLCODE in the SQLCA, the SQLCODE variable, or the SQLSTATE variable to the "no data found" error code. In the following example, FETCH returns the values of columns ENAME, EMPNO, and COMM to SELDSC:

```
EXEC SQL FETCH EMPCURSOR USING DESCRIPTOR SELDSC END-EXEC.
```

"Select Descriptor after the FETCH" shows the select descriptor in our example after the FETCH. Notice that Oracle has stored the select-list and indicator values in the data buffers addressed by the elements of SELDV and SELDI.

For output buffers of datatype 1, Oracle, using the lengths stored in SELDVLN, left-justifies CHAR or VARCHAR2 data, and right-justifies NUMBER data.

The value MARTIN was retrieved from a VARCHAR2(10) column in the EMP table. Using the length in SELDVLN(1), Oracle left-justifies the value in a 10-byte field, filling the buffer.

The value 7654 was retrieved from a NUMBER(4) column and coerced to 7654. However, the length in SELDVLN(2) was increased by two to allow for a possible sign and decimal point, so Oracle right-justifies the value in a 6-byte field.

The value 482.50 was retrieved from a NUMBER(7,2) column and coerced to 482.50. Again, the length in SELDVLN(3) was increased by two, so Oracle right-justifies the value in a 9-byte field.

### 11.6.16 Get and Process Select-List Values

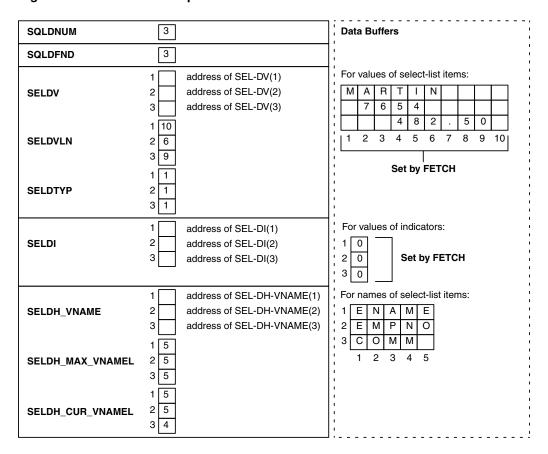
After the FETCH, your program can process the select-list values returned by FETCH. In our example, values for columns ENAME, EMPNO, and COMM are processed.

### 11.6.17 CLOSE the Cursor

CLOSE disables the cursor. In our example, CLOSE disables EMPCURSOR as follows:

EXEC SQL CLOSE EMPCURSOR END-EXEC

Figure 11-9 Select Descriptor after the FETCH





# 11.7 About Using Domain Metadata

An application can get define metadata for a dynamic SQL statement by using the DESCRIBE SELECT LIST command, which places metadata information into the SQL descriptor area.

```
EXEC SQL DESCRIBE SELECT LIST FOR statement_name INTO
    select_descriptor_name;
```

The SQLDA structure is extended to store domain name and domain length.



Declaring a SQLDA

The SQLDSCCP element provides the SQLCOLHASDOMAIN function to check if a column has a domain.

```
EXEC SQL INCLUDE bndsel END-EXEC.

MOVE "SELECT empId, empEmail FROM empTable" to select_stmt.

EXEC SQL DESCRIBE SELECT LIST FOR select_stmt INTO sel_des END-EXEC.

MOVE "empEMail" to colName).

CALL " SQLCOLHASDOMAIN" USING

NULL

SQLDSCCP(1)

hasDomain.

IF hasDomain = 1

DISPLAY colName, " has domain Name property".
```

# 11.8 Using Host Tables with Method 4

To use input or output host tables with Method 4, you must use the optional FOR clause to tell Oracle the size of your host table. For more information about the FOR clause, see "Host Tables".

Set descriptor entries for the Jth select-list item or bind variable, but instead of addressing a single data buffer, SELDVLN(J) or BNDDVLN(J) addresses a table of data buffers. Then use a FOR clause in the EXECUTE or FETCH statement, as appropriate, to tell Oracle the number of table elements you want to process.

This procedure is necessary because Oracle has no other way of knowing the size of your host table.

In the following example (32 bit only), two input host tables are used to insert 8 pairs of values of EMPNO and DEPTNO into the table EMP. Note that EXECUTE can be used for non-queries with Method 4.

```
IDENTIFICATION DIVISION.

PROGRAM-ID. DYN4INS.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 BNDDSC.

02 SQLDNUM PIC S9(9) COMP VALUE 2.

02 SQLDFND PIC S9(9) COMP.
```



```
02 BNDDVAR
                          OCCURS 2 TIMES.
       03 BNDDV
                          PIC S9(9) COMP.
       03 BNDDFMT
                          PIC S9(9) COMP.
       03 BNDDVLN
                          PIC S9(9) COMP.
       03 BNDDFMTL
                          PIC S9(4) COMP.
       03 BNDDVTYP
                          PIC S9(4) COMP.
       03 BNDDI
                          PIC S9(9) COMP.
       03 BNDDH-VNAME PIC S9(9) COMP.
       03 BNDDH-MAX-VNAMEL PIC S9(4) COMP.
       03 BNDDH-CUR-VNAMEL PIC S9(4) COMP.
       03 BNDDI-VNAME PIC S9(9) COMP.
       03 BNDDI-MAX-VNAMEL PIC S9(4) COMP.
       03 BNDDI-CUR-VNAMEL PIC S9(4) COMP.
       03 BNDDFCLP
                           PIC S9(9) COMP.
       03 BNDDFCRCP
                           PIC S9(9) COMP.
01 XBNDDI.
    03 BND-DI
                           OCCURS 2 TIMES PIC S9(4) COMP.
01 XBNDDIVNAME.
    03 BND-DI-VNAME
                           OCCURS 2 TIMES PIC X(80).
01 XBNDDV.
   Since you know what the SQL statement will be, you can set
   up a two-dimensional table with a maximum of 2 columns and
   8 rows. Each element can be up to 10 characters long. (You
   can alter these values according to your needs.)
   03 BND-COLUMN OCCURS 2 TIMES.
05 BND-ELEMENT OCCURS 8 TIMES PIC X(10).
01 XBNDDHVNAME.
                        OCCURS 2 TIMES PIC X(80).
   03 BND-DH-VNAME
01 COLUMN-INDEX
                          PIC 999.
01 ROW-INDEX
                          PIC 999.
01 DUMMY-INTEGER PIC 9999.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01 USERNAME PIC X(20).
       01 PASSWD
                           PIC X(20).
       01 DYN-STATEMENT PIC X(80).
       01 NUMBER-OF-ROWS PIC S9(4) COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
PROCEDURE DIVISION.
START-MAIN.
    EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
    MOVE "SCOTT" TO USERNAME.
    MOVE "TIGER" TO PASSWD.
    EXEC SQL
       CONNECT : USERNAME IDENTIFIED BY : PASSWD
    END-EXEC.
    DISPLAY "Connected to Oracle".
    Initialize bind and select descriptors.
    PERFORM INIT-BNDDSC THRU INIT-BNDDSC-EXIT
       VARYING COLUMN-INDEX FROM 1 BY 1
       UNTIL COLUMN-INDEX > 2.
    Set up the SQL statement.
    MOVE SPACES TO DYN-STATEMENT.
    MOVE "INSERT INTO EMP(EMPNO, DEPTNO) VALUES(:EMPNO,:DEPTNO)"
       TO DYN-STATEMENT.
```

DISPLAY DYN-STATEMENT.

```
Prepare the SQL statement.
    EXEC SQL
        PREPARE S1 FROM : DYN-STATEMENT
    END-EXEC.
    Describe the bind variables.
    EXEC SQL
        DESCRIBE BIND VARIABLES FOR S1 INTO BNDDSC
    END-EXEC.
    PERFORM Z-BIND-TYPE THRU Z-BIND-TYPE-EXIT
        VARYING COLUMN-INDEX FROM 1 BY 1
        UNTIL COLUMN-INDEX > 2.
    IF SQLDFND IN BNDDSC < 0
        DISPLAY "TOO MANY BIND VARIABLES."
        GO TO SQL-ERROR
    ELSE
        DISPLAY "BIND VARS = " WITH NO ADVANCING
        MOVE SQLDFND IN BNDDSC TO DUMMY-INTEGER
        DISPLAY DUMMY-INTEGER
        MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC.
        MOVE 8 TO NUMBER-OF-ROWS.
        PERFORM GET-ALL-VALUES THRU GET-ALL-VALUES-EXIT
            VARYING ROW-INDEX FROM 1 BY 1
            UNTIL ROW-INDEX > NUMBER-OF-ROWS.
    Execute the SQL statement.
    EXEC SQL FOR : NUMBER-OF-ROWS
        EXECUTE S1 USING DESCRIPTOR BNDDSC
    END-EXEC.
    DISPLAY "INSERTED " WITH NO ADVANCING.
    MOVE SQLERRD(3) TO DUMMY-INTEGER.
    DISPLAY DUMMY-INTEGER WITH NO ADVANCING.
    DISPLAY " ROWS.".
    GO TO END-SOL.
SQL-ERROR.
    Display any SQL error message and code.
    DISPLAY SQLERRMC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
END-SQL.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL COMMIT WORK RELEASE END-EXEC.
    STOP RUN.
INIT-BNDDSC.
    Start of COBOL PERFORM procedures, initialize the bind
    descriptor.
    MOVE 80 TO BNDDH-MAX-VNAMEL(COLUMN-INDEX).
    CALL "SQLADR" USING
        BND-DH-VNAME (COLUMN-INDEX)
        BNDDH-VNAME (COLUMN-INDEX) .
    MOVE 80 TO BNDDI-MAX-VNAMEL(COLUMN-INDEX).
    CALL "SOLADR" USING
```

```
BND-DI-VNAME (COLUMN-INDEX)
        BNDDI-VNAME (COLUMN-INDEX).
    MOVE 10 TO BNDDVLN(COLUMN-INDEX).
    CALL "SQLADR" USING
        BND-ELEMENT (COLUMN-INDEX, 1)
        BNDDV (COLUMN-INDEX).
    MOVE ZERO TO BNDDI (COLUMN-INDEX) .
    CALL "SQLADR" USING
        BND-DI (COLUMN-INDEX)
        BNDDI (COLUMN-INDEX) .
    MOVE ZERO TO BNDDFMT (COLUMN-INDEX) .
    MOVE ZERO TO BNDDFMTL (COLUMN-INDEX) .
    MOVE ZERO TO BNDDFCLP(COLUMN-INDEX).
    MOVE ZERO TO BNDDFCRCP(COLUMN-INDEX).
INIT-BNDDSC-EXIT.
    EXIT.
Z-BIND-TYPE.
    Replace the Os DESCRIBEd into the datatype table with 1s to
    avoid an "invalid datatype" Oracle error.
    MOVE 1 TO BNDDVTYP (COLUMN-INDEX).
Z-BIND-TYPE-EXIT.
    EXIT.
GET-ALL-VALUES.
    Get the bind variables for each row.
    DISPLAY "ENTER VALUES FOR ROW NUMBER ", ROW-INDEX.
    PERFORM GET-BIND-VARS
        VARYING COLUMN-INDEX FROM 1 BY 1
        UNTIL COLUMN-INDEX > SQLDFND IN BNDDSC.
GET-ALL-VALUES-EXIT.
    EXIT.
GET-BIND-VARS.
    Get the value of each bind variable.
    DISPLAY " ENTER VALUE FOR ", BND-DH-VNAME (COLUMN-INDEX)
        WITH NO ADVANCING.
    ACCEPT BND-ELEMENT (COLUMN-INDEX, ROW-INDEX) .
GET-BIND-VARS-EXIT.
    EXIT.
```

#### **Related Topics**

Host Tables

# 11.9 Sample Program 10: Dynamic SQL Method 4

This program shows the basic steps required to use dynamic SQL Method 4.

#### Note:

The following example is 32 bit only. To avoid problems, please use the example from \$ORACLE HOME/precomp/demo/procob2/sample10.pco and include the correct version of the variable declaration file.

bndsel.cob 64 bit version in precomp/public (64 bit computer)

bndsel.cob 32 32 bit version in precomp/public32 (64 bit computer)

After logging on, the program prompts the user for a SQL statement, prepares statement, declares a cursor, checks for any bind variables using DESCRIBE BIND, opens the cursor, and describes any select-list variables. If the input SQL statement is a query, the program fetches each row of data, then closes the cursor.

```
****************
* Sample Program 10: Dynamic SQL Method 4
^{\star} This program shows the basic steps required to use dynamic ^{\star}
* SQL Method 4. After logging on to ORACLE, the program
* prompts the user for a SQL statement, PREPAREs the
* statement, DECLAREs a cursor, checks for any bind variables *
* using DESCRIBE BIND, OPENs the cursor, and DESCRIBEs any
* select-list variables. If the input SQL statement is a
* query, the program FETCHes each row of data, then CLOSEs
* the cursor.
******************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. DYNSQL4.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 BNDDSC.
         SQLDNUM PIC S9(9) COMP VALUE 20.

SQLDFND PIC S9(9) COMP.

BNDDVAR OCCURS 20 TIMES.

03 BNDDV PIC S9(9) COMP.

03 BNDDFMT PIC S9(9) COMP.

03 BNDDVLN PIC S9(9) COMP.

03 BNDDFMTL PIC S9(4) COMP.

03 BNDDVTYP PIC S9(4) COMP.

03 BNDDI PIC S9(9) COMP
     02 SQLDNUM
02 SQLDFND
     02 BNDDVAR
          03 BNDDH-VNAME PIC S9(9) COMP.
          03 BNDDH-MAX-VNAMEL PIC S9(4) COMP.
          03 BNDDH-CUR-VNAMEL PIC S9(4) COMP.
          03 BNDDI-VNAME PIC S9(9) COMP.
          03 BNDDI-MAX-VNAMEL PIC S9(4) COMP.
          03 BNDDI-CUR-VNAMEL PIC S9(4) COMP.
         03 BNDDFCLP PIC S9(9) COMP.
03 BNDDFCRCP PIC S9(9) COMP.
 01 XBNDDI.
     03 BND-DI
```

OCCURS 20 TIMES PIC S9(4) COMP.

```
01 XBNDDIVNAME.
   03 BND-DI-VNAME
                            OCCURS 20 TIMES PIC X(80).
01 XBNDDV.
   03 BND-DV
                           OCCURS 20 TIMES PIC X(80).
01 XBNDDHVNAME.
                           OCCURS 20 TIMES PIC X(80).
   03 BND-DH-VNAME
01 SELDSC.
   02 SQLDNUM
                            PIC S9(9) COMP VALUE 20.
   02 SQLDFND
                           PIC S9(9) COMP.
   02 SELDVAR
                           OCCURS 20 TIMES.
       03 SELDV
                           PIC S9(9) COMP.
       03 SELDFMT
                           PIC S9(9) COMP.
       03 SELDVLN
                            PIC S9(9) COMP.
       03 SELDFMTL
                            PIC S9(4) COMP.
       03 SELDVTYP
                           PIC S9(4) COMP.
       03 SELDI
                           PIC S9(9) COMP.
       03 SELDH-VNAME PIC S9(9) COMP.
       03 SELDH-MAX-VNAMEL PIC S9(4) COMP.
       03 SELDH-CUR-VNAMEL PIC S9(4) COMP.
       03 SELDI-VNAME
                       PIC S9(9) COMP.
       03 SELDI-MAX-VNAMEL PIC S9(4) COMP.
       03 SELDI-CUR-VNAMEL PIC S9(4) COMP.
       03 SELDFCLP PIC S9(9) COMP.
       03 SELDFCRCP
                           PIC S9(9) COMP.
01 XSELDI.
   03 SEL-DI
                           OCCURS 20 TIMES PIC S9(4) COMP.
01 XSELDIVNAME.
   03 SEL-DI-VNAME
                           OCCURS 20 TIMES PIC X(80).
01 XSELDV.
   03 SEL-DV
                           OCCURS 20 TIMES PIC X(80).
01 XSELDHVNAME.
   03 SEL-DH-VNAME
                           OCCURS 20 TIMES PIC X(80).
01 TABLE-INDEX
                           PIC 9(3).
01 VAR-COUNT
                           PIC 9(2).
01 ROW-COUNT
                           PIC 9(4).
01 NO-MORE-DATA
                           PIC X(1) VALUE "N".
01 NULLS-ALLOWED
                           PIC S9(9) COMP.
01 PRECISION
                            PIC S9(9) COMP.
01 SCALE
                            PIC S9(9) COMP.
01 DISPLAY-LENGTH
                           PIC S9(9) COMP.
                           PIC S9(9) COMP VALUE 80.
01 MAX-LENGTH
01 COLUMN-NAME
                           PIC X(30).
01 NULL-VAL
                           PIC X(80) VALUE SPACES.
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERNAME PIC X(20).
01 PASSWD
                 PIC X(20).
01 DYN-STATEMENT PIC X(80).
   EXEC SQL END DECLARE SECTION END-EXEC.
   EXEC SQL INCLUDE SQLCA
                                END-EXEC.
```

PROCEDURE DIVISION.

START-MAIN.

EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.

DISPLAY "USERNAME: " WITH NO ADVANCING.

ACCEPT USERNAME.

DISPLAY "PASSWORD: " WITH NO ADVANCING.

ACCEPT PASSWD.

EXEC SQL CONNECT : USERNAME IDENTIFIED BY : PASSWD END-EXEC. DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME.

\* INITIALIZE THE BIND AND SELECT DESCRIPTORS.

PERFORM INIT-BNDDSC

VARYING TABLE-INDEX FROM 1 BY 1

UNTIL TABLE-INDEX > 20.

PERFORM INIT-SELDSC

VARYING TABLE-INDEX FROM 1 BY 1

UNTIL TABLE-INDEX > 20.

\* GET A SQL STATEMENT FROM THE OPERATOR.

DISPLAY "ENTER SQL STATEMENT WITHOUT TERMINATOR:". DISPLAY ">" WITH NO ADVANCING.

ACCEPT DYN-STATEMENT.

DISPLAY " ".

\* PREPARE THE SQL STATEMENT AND DECLARE A CURSOR.

EXEC SQL PREPARE S1 FROM :DYN-STATEMENT END-EXEC. EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.

\* DESCRIBE ANY BIND VARIABLES.

EXEC SQL DESCRIBE BIND VARIABLES FOR S1 INTO BNDDSC  $\mathtt{END-EXEC.}$ 

IF SQLDFND IN BNDDSC < 0
DISPLAY "TOO MANY BIND VARIABLES."
GO TO END-SQL

ELSE
DISPLAY "NUMBER OF BIND VARIABLES: " WITH NO ADVANCING
MOVE SQLDFND IN BNDDSC TO VAR-COUNT
DISPLAY VAR-COUNT
MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC

REPLACE THE 0S DESCRIBED INTO THE DATATYPE FIELDS OF THE

\* BIND DESCRIPTOR WITH 1S TO AVOID AN "INVALID DATATYPE"

\* ORACLE ERROR

END-IF.

MOVE 1 TO TABLE-INDEX.

FIX-BIND-TYPE.

MOVE 1 TO BNDDVTYP(TABLE-INDEX)

ADD 1 TO TABLE-INDEX

IF TABLE-INDEX <= 20

GO TO FIX-BIND-TYPE. LET THE USER FILL IN THE BIND VARIABLES. IF SOLDFND IN BNDDSC = 0 GO TO DESCRIBE-ITEMS. MOVE 1 TO TABLE-INDEX. GET-BIND-VAR. DISPLAY "ENTER VALUE FOR ", BND-DH-VNAME (TABLE-INDEX) . ACCEPT BND-DV (TABLE-INDEX) . ADD 1 TO TABLE-INDEX IF TABLE-INDEX <= SQLDFND IN BNDDSC GO TO GET-BIND-VAR. OPEN THE CURSOR AND DESCRIBE THE SELECT-LIST ITEMS. DESCRIBE-ITEMS. EXEC SQL OPEN C1 USING DESCRIPTOR BNDDSC END-EXEC. EXEC SQL DESCRIBE SELECT LIST FOR S1 INTO SELDSC END-EXEC. IF SQLDFND IN SELDSC < 0 DISPLAY "TOO MANY SELECT-LIST ITEMS." GO TO END-SQL ELSE DISPLAY "NUMBER OF SELECT-LIST ITEMS: " WITH NO ADVANCING MOVE SQLDFND IN SELDSC TO VAR-COUNT DISPLAY VAR-COUNT DISPLAY " " MOVE SQLDFND IN SELDSC TO SQLDNUM IN SELDSC END-IF. COERCE THE DATATYPE OF ALL SELECT-LIST ITEMS TO VARCHAR2. IF SOLDNUM IN SELDSC > 0 PERFORM COERCE-COLUMN-TYPE VARYING TABLE-INDEX FROM 1 BY 1 UNTIL TABLE-INDEX > SOLDNUM IN SELDSC DISPLAY " ". FETCH EACH ROW AND PRINT EACH SELECT-LIST VALUE. IF SQLDNUM IN SELDSC > 0 PERFORM FETCH-ROWS UNTIL NO-MORE-DATA = "Y". DISPLAY " " DISPLAY "NUMBER OF ROWS PROCESSED: " WITH NO ADVANCING. MOVE SOLERRD(3) TO ROW-COUNT. DISPLAY ROW-COUNT. CLEAN UP AND TERMINATE. EXEC SQL CLOSE C1 END-EXEC. EXEC SQL COMMIT WORK RELEASE END-EXEC. DISPLAY " ". DISPLAY "HAVE A GOOD DAY!". DISPLAY " ".

STOP RUN.

```
DISPLAY ORACLE ERROR MESSAGE AND CODE.
SQL-ERROR.
    DISPLAY " ".
    DISPLAY SQLERRMC.
END-SOL.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
  PERFORMED SUBROUTINES BEGIN HERE:
   INIT-BNDDSC: INITIALIZE THE BIND DESCRIPTOR.
INIT-BNDDSC.
   MOVE SPACES TO BND-DH-VNAME (TABLE-INDEX) .
   MOVE 80 TO BNDDH-MAX-VNAMEL (TABLE-INDEX).
    CALL "SQLADR" USING
        BND-DH-VNAME (TABLE-INDEX)
        BNDDH-VNAME (TABLE-INDEX) .
    MOVE SPACES TO BND-DI-VNAME (TABLE-INDEX).
    MOVE 80 TO BNDDI-MAX-VNAMEL (TABLE-INDEX).
    CALL "SQLADR" USING
        BND-DI-VNAME (TABLE-INDEX)
        BNDDI-VNAME (TABLE-INDEX).
    MOVE SPACES TO BND-DV(TABLE-INDEX).
    MOVE 80 TO BNDDVLN(TABLE-INDEX).
    CALL "SQLADR" USING
        BND-DV (TABLE-INDEX)
        BNDDV (TABLE-INDEX) .
    MOVE ZERO TO BND-DI (TABLE-INDEX) .
    CALL "SQLADR" USING
        BND-DI (TABLE-INDEX)
        BNDDI (TABLE-INDEX) .
    MOVE ZERO TO BNDDFMT (TABLE-INDEX) .
    MOVE ZERO TO BNDDFMTL(TABLE-INDEX).
    MOVE ZERO TO BNDDFCLP(TABLE-INDEX).
   MOVE ZERO TO BNDDFCRCP (TABLE-INDEX) .
  INIT-SELDSC: INITIALIZE THE SELECT DESCRIPTOR.
INIT-SELDSC.
   MOVE SPACES TO SEL-DH-VNAME (TABLE-INDEX).
    MOVE 80 TO SELDH-MAX-VNAMEL (TABLE-INDEX).
    CALL "SQLADR" USING
        SEL-DH-VNAME (TABLE-INDEX)
        SELDH-VNAME (TABLE-INDEX) .
    MOVE SPACES TO SEL-DI-VNAME (TABLE-INDEX).
    MOVE 80 TO SELDI-MAX-VNAMEL (TABLE-INDEX).
    CALL "SQLADR" USING
        SEL-DI-VNAME (TABLE-INDEX)
        SELDI-VNAME (TABLE-INDEX).
    MOVE SPACES TO SEL-DV(TABLE-INDEX).
```

```
MOVE 80 TO SELDVLN (TABLE-INDEX).
    CALL "SQLADR" USING
         SEL-DV (TABLE-INDEX)
         SELDV (TABLE-INDEX) .
    MOVE ZERO TO SEL-DI (TABLE-INDEX) .
    CALL "SQLADR" USING
         SEL-DI (TABLE-INDEX)
         SELDI (TABLE-INDEX) .
    MOVE ZERO TO SELDFMT (TABLE-INDEX) .
    MOVE ZERO TO SELDFMTL (TABLE-INDEX) .
    MOVE ZERO TO SELDFCLP (TABLE-INDEX) .
    MOVE ZERO TO SELDFCRCP (TABLE-INDEX) .
    COERCE SELECT-LIST DATATYPES TO VARCHAR2.
COERCE-COLUMN-TYPE.
    CALL "SQLNUL" USING
        SELDVTYP (TABLE-INDEX)
         SELDVTYP (TABLE-INDEX)
         NULLS-ALLOWED.
    IF DATATYPE IS DATE, LENGTHEN TO 9 CHARACTERS.
    IF SELDVTYP (TABLE-INDEX) = 12
        MOVE 9 TO SELDVLN (TABLE-INDEX).
    IF DATATYPE IS NUMBER, SET LENGTH TO PRECISION.
    IF SELDVTYP(TABLE-INDEX) = 2
         CALL "SQLPRC" USING
             SELDVLN (TABLE-INDEX)
             PRECISION
             SCALE.
    MOVE 0 TO DISPLAY-LENGTH.
    IF SELDVTYP (TABLE-INDEX) = 2 AND PRECISION = 0
         MOVE 40 TO DISPLAY-LENGTH.
     IF SELDVTYP (TABLE-INDEX) = 2 AND PRECISION > 0
         ADD 2 TO PRECISION
        MOVE PRECISION TO DISPLAY-LENGTH.
    IF SELDVTYP(TABLE-INDEX) = 2
         IF DISPLAY-LENGTH > MAX-LENGTH
            DISPLAY "COLUMN VALUE TOO LARGE FOR DATA BUFFER."
             GO TO END-SQL
         ELSE
             MOVE DISPLAY-LENGTH TO SELDVLN (TABLE-INDEX) .
    COERCE DATATYPES TO VARCHAR2.
    MOVE 1 TO SELDVTYP (TABLE-INDEX).
    DISPLAY COLUMN HEADING.
    MOVE SEL-DH-VNAME (TABLE-INDEX) TO COLUMN-NAME.
    DISPLAY COLUMN-NAME(1:SELDVLN(TABLE-INDEX)), " "
         WITH NO ADVANCING.
*FETCH A ROW AND PRINT THE SELECT-LIST VALUE.
FETCH-ROWS.
    EXEC SQL FETCH C1 USING DESCRIPTOR SELDSC END-EXEC.
    IF SOLCODE NOT = 0
        MOVE "Y" TO NO-MORE-DATA.
```

```
IF SQLCODE = 0
    PERFORM PRINT-COLUMN-VALUES
        VARYING TABLE-INDEX FROM 1 BY 1
        UNTIL TABLE-INDEX > SQLDNUM IN SELDSC
        DISPLAY " ".

*PRINT A SELECT-LIST VALUE.

PRINT-COLUMN-VALUES.
    IF SEL-DI(TABLE-INDEX) = -1
        DISPLAY NULL-VAL(1:SELDVLN(TABLE-INDEX)), " "
        WITH NO ADVANCING
    ELSE
        DISPLAY SEL-DV(TABLE-INDEX)(1:SELDVLN(TABLE-INDEX)), " "
        WITH NO ADVANCING
    END-IF.
```



12

# **Multithreaded Applications**

If your development platform does not support threads, you can ignore this chapter.

The sections of this chapter are:

- Introduction to Threads
- Runtime Contexts in Pro\*COBOL
- Runtime Context Usage Models
- User Interface Features for Multithreaded Applications
- Multithreaded Example

### 12.1 Introduction to Threads

Multithreaded applications have multiple *threads* executing in a shared address space. Threads are "lightweight" subprocesses that execute within a process. They share code and data segments, but have their own program counters, machine registers, and stack. Variables declared without the thread-local attribute in working storage (as opposed to local-storage or thread-local storage) are common to all threads, and a mutual exclusivity mechanism is often required to manage access to these variables from multiple threads within an application. *Mutexes* are the synchronization mechanism to insure that data integrity is preserved.

For further discussion of mutexes, see texts on multithreading. For more detailed information about multithreaded applications, see the documentation of your threads functions.

Pro\*COBOL supports development of multithreaded Oracle Server applications (on platforms that support multithreaded applications) using the following:

- A command-line option to generate thread-safe code.
- Embedded SQL statements and directives to support multithreading.
- Thread-safe SQLLIB and other client-side Oracle libraries.



While your platform may support a particular thread package, see your platformspecific Oracle documentation to determine whether Oracle supports it.

The chapter's topics discuss how to use the preceding features to develop multithreaded Pro\*COBOL applications:

- Runtime contexts for multithreaded applications.
- Two models for using runtime contexts.
- User-interface features for multithreaded applications.
- Programming considerations for writing multithreaded applications with Pro\*COBOL.

Sample multithreaded Pro\*COBOL applications.

### 12.2 Runtime Contexts in Pro\*COBOL

To loosely couple a thread and a connection, in Pro\*COBOL we introduce the concept of a *runtime context*. The runtime context includes the following resources and their current states:

- Zero or more connections to one or more Oracle servers.
- Zero or more cursors used for the server connections.
- Inline options, such as MODE, HOLD\_CURSOR, RELEASE\_CURSOR, and SELECT\_ERROR.

Rather than simply supporting a loose coupling between threads and connections, Pro\*COBOL enables you to loosely couple threads with runtime contexts. Pro\*COBOL enables your application to define a handle to a runtime context, and pass that handle from one thread to another.

For example, an interactive application spawns a thread, T1, to execute a query and return the first 10 rows to the application. T1 then terminates. After obtaining the necessary user input, another thread, T2, is spawned (or an existing thread is used) and the runtime context for T1 is passed to T2 so it can fetch the next 10 rows by processing the same cursor. This is shown in Figure 12-1:



**Main Program** ENABLE THREADS ALLOCATE :ctx Connect... FREE :ctx Execution Time T1 Thread Shared runtime context is passed from USE :ctx one thread to Fetch.. the next T2 Thread USE :ctx Fetch. Thread USE :ctx Fetch... Note: The syntax used in this and subsequent figures is for structural use only, for correct syntax, see the section titled, "User-interface Features for Multi-threaded Applications." Server

Figure 12-1 Loosely Coupling Connections and Threads

**Application** 

# 12.3 Runtime Context Usage Models

Two possible models for using runtime contexts in multithreaded applications are shown here:

- Multiple threads sharing a single runtime context.
- Multiple threads using separate runtime contexts.

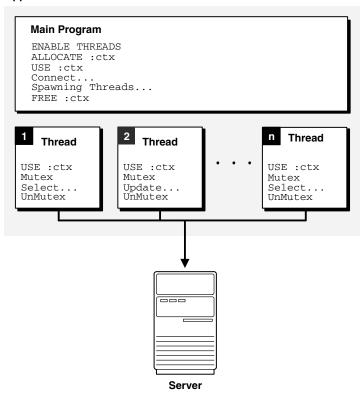
Regardless of the model you use for runtime contexts, you *cannot* share a runtime context between multiple threads *at the same time*. If two or more threads attempt to use the same runtime context simultaneously, a runtime error occurs

## 12.3.1 Multiple Threads Sharing a Single Runtime Context

"Context Sharing Among Threads" shows an application running in a multithreaded environment. The various threads share a single runtime context to process one or more SQL statements. Again, runtime contexts cannot be shared by multiple threads at the same time. The mutexes in "Context Sharing Among Threads" show how to prevent concurrent usage.

Figure 12-2 Context Sharing Among Threads

#### **Application**

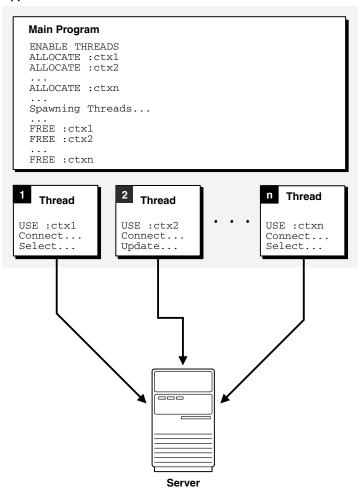


# 12.3.2 Multiple Threads Sharing Multiple Runtime Contexts

Figure 12-3 shows an application that executes multiple threads using multiple runtime contexts. In this situation, the application does not require mutexes, because each thread has a dedicated runtime context.

Figure 12-3 No Context Sharing Among Threads

#### Application



# 12.4 User Interface Features for Multithreaded Applications

Pro\*COBOL provides the following user-interface features to support multithreaded applications:

- Host variables can be declared in the LOCAL-STORAGE and the THREAD-LOCAL-STORAGE sections.
- The command-line option THREADS=YES | NO.
- Embedded SQL statements and directives.
- Thread-safe SQLLIB public functions.

### 12.4.1 THREADS Option

With THREADS=YES specified on the command line, Pro\*COBOL ensures that the generated code is thread-safe, given that you follow the guidelines described in "Multithreading Programming Considerations". With THREADS=YES specified, Pro\*COBOL

verifies that all SQL statements execute within the scope of a user-defined runtime context. If your program does not meet this requirement, a precompiler error is returned.

#### **Related Topics**

Multithreading Programming Considerations

# 12.4.2 Embedded SQL Statements and Directives for Runtime Contexts

The following embedded SQL statements and directives support the definition and usage of runtime contexts and threads:

- EXEC SQL ENABLE THREADS END-EXEC.
- EXEC SQL CONTEXT ALLOCATE :context\_var END-EXEC.
- EXEC SQL CONTEXT USE { :context\_var | DEFAULT} END-EXEC.
- EXEC SQL CONTEXT FREE :context\_var END-EXEC.

For these EXEC SQL statements, *context\_var* is the handle to the runtime context and must be declared of type SQL-CONTEXT as follows:

```
01 SQL-CONTEXT context var END-EXEC.
```

Using DEFAULT means that the default (global) runtime context will be used in all embedded SQL statements that lexically follow until another CONTEXT USE statement overrides it.

Examples illustrating the various uses of context statements are shown.

### 12.4.2.1 Host Tables of SQL-CONTEXT Are Not Allowed

You cannot declare host tables of SQL-CONTEXT. Instead, declare a host table of S9(9) COMP variables and then pass them to the subprogram one at a time after redeclaring them in the subprogram as SQL-CONTEXT.

### 12.4.2.2 EXEC SQL ENABLE THREADS

This executable SQL statement initializes a process that supports multiple threads. This must be the first executable SQL statement in a program that contains a multithreaded application. There can only be one ENABLE THREADS statement in all files of an application, or an error results. For more detailed information, see "ENABLE THREADS (Executable Embedded SQL Extension)".

#### **Related Topics**

ENABLE THREADS (Executable Embedded SQL Extension)

### 12.4.2.3 EXEC SQL CONTEXT ALLOCATE

This executable SQL statement allocates and initializes memory for the specified runtime context; the runtime-context variable must be declared of type SQL\_CONTEXT. For more detailed information, see "CONTEXT ALLOCATE (Executable Embedded SQL Extension)".



#### **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension)

### 12.4.2.4 EXEC SQL CONTEXT USE

The EXEC SQL CONTEXT USE directive instructs the precompiler to use the specified runtime context for subsequent executable SQL statements. The runtime context specified must be previously allocated using an EXEC SQL CONTEXT ALLOCATE statement.

The EXEC SQL CONTEXT USE directive works similarly to the EXEC SQL WHENEVER directive in that it affects all executable SQL statements which positionally follow it in a given source file without regard to standard COBOL scope rules.

For more detailed information, see "CONTEXT USE (Oracle Embedded SQL Directive)", and "CONTEXT ALLOCATE (Executable Embedded SQL Extension)".

#### **Related Topics**

- CONTEXT USE (Oracle Embedded SQL Directive)
- CONTEXT ALLOCATE (Executable Embedded SQL Extension)

### 12.4.2.5 EXEC SQL CONTEXT FREE

The EXEC SQL CONTEXT FREE executable SQL statement frees the memory associated with the specified runtime context and places a null pointer in the host program variable. For more detailed information, see "CONTEXT FREE (Executable Embedded SQL Extension)".

#### **Related Topics**

CONTEXT FREE (Executable Embedded SQL Extension)

### 12.4.3 Communication with Pro\*C/C++ Programs

Runtime contexts can be passed using arguments defined in the Linkage Section. Multithreaded Pro\*C/C++ programs can call Pro\*COBOL subprograms and Pro\*COBOL programs can call subprograms written in Pro\*C/C++.

### 12.4.4 Multithreading Programming Considerations

While Oracle ensures that the SQLLIB code is thread-safe, you are responsible for ensuring that your source code is designed to work properly with threads. For example, carefully consider the scope of the variables you use.

In addition, multithreading requires design decisions regarding the following:

- Including one SQLCA for each runtime context.
- Declaring the SQLDA as a thread-safe group item, like the SQLCA, typically an auto variable, one for each runtime context.
- Declaring host variables in a thread-safe fashion, in other words, carefully consider your use of static and global host variables.
- Avoiding simultaneous use of a runtime context in multiple threads.
- Whether or not to use default database connections or to explicitly define them using the AT clause.



No more than one executable embedded SQL statement, for example, EXEC SQL UPDATE, may be outstanding on a runtime context at a given time.

Existing requirements for precompiled applications also apply. For example, all references to a given cursor must appear in the same source file.

### 12.4.4.1 Restrictions on Multithreading

The following restrictions be in effect when using threads:

- You cannot use an array of datatype SQL-CONTEXT.
- Concurrent threads should each have its own SQLCA.
- Concurrent threads should each have its own context areas.

# 12.4.5 Multiple Context Examples

The code fragments in this section show how to use multiple contexts, and demonstrate the scope of the context use statement.

### 12.4.5.1 Example 1

In the first example, the precompiler option setting THREADS=YES is not needed, because we are not generating threads:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
* declare a context area
01 CTX1 SQL-CONTEXT.
01 UID1 PIC X(11) VALUE "SCOTT/TIGER".
01 UID2 PIC X(10) VALUE "MARY/LION"
PROCEDURE DIVISION.
* allocate context area
    EXEC SQL CONTEXT ALLOCATE : CTX1 END-EXEC.
    EXEC SOL CONTEXT USE :CTX1 END-EXEC.
* all statements until the next context use will use CTX1
    EXEC SQL CONNECT : UID1 END-EXEC.
    EXEC SQL SELECT ....
    EXEC SQL CONTEXT USE DEFAULT END-EXEC.
* all statements physically after the preceding lines will use the default
    EXEC SQL CONNECT : UID2 END-EXEC.
    EXEC SOL INSERT ...
```

### 12.4.5.2 Example 2

This next example shows multiple contexts. One context is used by the generated thread while the other is used by the main program. The started thread, SUBPRGM1, will use context CTX1, which is passed to it through the LINKAGE SECTION. This example also demonstrates the scope of the CONTEXT USE statement.



You must precompile the main program file, and the main program of every subsequent example in this section, with the option THREADS=YES.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
* declare two context areas
01 CTX1 SQL-CONTEXT.
01 CTX2 SQL-CONTEXT.
PROCEDURE DIVISION.
* enable threading
     EXEC SQL ENABLE THREADS END-EXEC.
* allocate context areas
     EXEC SQL CONTEXT ALLOCATE : CTX1 END-EXEC.
     EXEC SQL CONTEXT ALLOCATE : CTX2 END-EXEC.
* include your code to start thread "SUBPGM1" using CTX1 here.
     EXEC SQL CONTEXT USE :CTX2 END-EXEC.
* all statement physically after the preceding lines will use CTX2
     EXEC SQL CONNECT : USERID END-EXEC.
     EXEC SQL INSERT ....
The thread SUBPRGM1 is in a another file:
PROGRAM-ID. SUBPRGM1.
01 USERID PIC X(11) VALUE "SCOTT/TIGER".
LINKAGE SECTION.
01 CTX1 SOL-CONTEXT.
 PROCEDURE DIVISION USING CTX1.
     EXEC SQL CONTEXT USE :CTX1 END-EXEC.
     EXEC SQL CONNECT : USERID END-EXEC.
```

### 12.4.5.3 Example 3

The following example uses multiple threads. Each thread has its own context. If the threads are to be executed concurrently, each thread *must* have its own context. Contexts are passed to the thread with the USING CLAUSE of the START statement and are declared in the LINKAGE SECTION of the threaded subprogram.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
...
DATA DIVISION.

01 CTX1 SQL-CONTEXT.
```

EXEC SQL SELECT ...



```
O1 CTX2 SQL-CONTEXT.

PROCEDURE DIVISION.
...

EXEC SQL ENABLE THREADS END-EXEC.
EXEC SQL CONTEXT ALLOCATE :CTX1 END-EXEC.
EXEC SQL CONTEXT ALLOCATE :CTX2 END-EXEC.

* include your code to start thread "SUBPGM" using CTX1 here.
* include your code to start thread "SUBPGM" using CTX2 here.
...
```

#### The thread SUBPGM is placed in another file:

```
PROGRAM-ID. SUBPGM.
...

DATA DIVISION.
...

01 USERID PIC X(11) VALUE "SCOTT/TIGER".
...

LINKAGE SECTION.
01 CTX SQL-CONTEXT.

PROCEDURE DIVISION USING CTX.

EXEC SQL CONTEXT USE :CTX END-EXEC.
EXEC SQL CONNECT :USERID END-EXEC.
EXEC SQL SELECT ....
```

### 12.4.5.4 Example 4

The next example is based on the previous example, but does the connection in the top level program and passes the connection with the context to the threaded subprogram.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
DATA DIVISION.
01 CTX1 SQL-CONTEXT.
01 CTX2 SQL-CONTEXT.
01 USERID PIC X(11) VALUE "SCOTT/TIGER".
ROCEDURE DIVISION.
    EXEC SQL ENABLE THREADS END-EXEC.
    EXEC SQL CONTEXT ALLOCATE : CTX1 END-EXEC.
    EXEC SQL CONTEXT ALLOCATE : CTX2 END-EXEC.
    EXEC SQL CONTEXT USE :CTX1 END-EXEC.
    EXEC SQL CONNECT : USERID END-EXEC.
    EXEC SQL CONTEXT USE :CTX2 END-EXEC.
    EXEC SQL CONNECT : USERID END-EXEC.
* include your code to start thread "SUBPGM" using CTX1 here.
* include your code to start thread "SUBPGM" using CTX2 here.
```

#### The thread SUBPRGM is in another file:

```
PROGRAM-ID. SUBPGM.
```



```
LINKAGE SECTION.

01 CTX SQL-CONTEXT.

PROCEDURE DIVISION USING CTX.

EXEC SQL CONTEXT USE :CTX END-EXEC.

EXEC SQL SELECT ....
```

### 12.4.5.5 Example 5

The following example shows multiple threads which share a context. Note that in this case, the threads *must* be serialized.

```
IDENTIFICATION DIVISION.

PROGRAM-ID. MAIN.
...

DATA DIVISION.

01 CTX1 SQL-CONTEXT.

PROCEDURE DIVISION.

EXEC SQL ENABLE THREADS END-EXEC.
EXEC SQL CONTEXT ALLOCATE :CTX1 END-EXEC.

* include your code to start thread1 "SUBPGM1" using CTX1 here.
* include your code to start thread1 here.
* include your code to start thread2 "SUBPGM2" using CTX1 here.
```

There are two separate files for the two threads. First there is:

```
PROGRAM-ID. SUBPGM1.
...
DATA DIVISION.
...
01 USERID PIC X(11) VALUE "SCOTT/TIGER".
...
LINKAGE SECTION.
01 CTX SQL-CONTEXT.
PROCEDURE DIVISION USING CTX.
EXEC SQL CONTEXT USE :CTX END-EXEC.
...
EXEC SQL CONNECT :USERID END-EXEC.
```

#### Another file contains SUBPGM2:

```
PROGRAM-ID. SUBPGM2.
...
DATA DIVISION.
...
LINKAGE SECTION.
01 CTX SQL-CONTEXT.
PROCEDURE DIVISION USING CTX.
EXEC SQL CONTEXT USE :CTX END-EXEC.
EXEC SELECT ....
```



# 12.5 Multithreaded Example

This multi-file application demonstrates one way to use the SQLLIB runtime context area (SQL-CONTEXT) to support multiple threads. Precompile with THREADS=YES.

The main program, orathrd2, declares an array of S9(9) COMP variables to be used to hold the sqllib contexts. It enables threading through the

```
EXEC SQL ENABLE THREADS END-EXEC.
```

statement and then calls the subprogram oracon (in file oracon.pco) to allocate the threads. oracon also establishes a connection for each allocated context.

Next, ORTHRD2 passes the context to one of the threaded entry points, THREAD-1 or THREAD-2. THREAD-1 simply selects and displays the salary for an employee. THREAD-2 selects and updates the salary for that employee. Since THREAD-2 issues a commit, the update is visible to threads that do the SELECT after it has committed. (But not those which run concurrently with the update.) Note that the output will vary from run to run because the timing of the update and commit is non-determinant.

It is important to remember that concurrent threads must each have their own contexts. Contexts may be passed to and used by subsequent threads, but threads may not use the same context concurrently. This model could be used for connection pooling, where the maximum number of connections are created initially and passed to threads as available, to execute user's requests.

An array of S9(9) COMP variables is used because you cannot currently declare an array of SQL-CONTEXT.

**Note:** This program was developed specifically for a Sun workstation running Solaris and MicroFocus ServerExpress compiler and uses vendor-specific directives and functionality.

See your platform-specific documentation for the specific COBOL statements that support multithreading.

The main program is in file orathrd2.pco:

```
$SET REENTRANT MF
IDENTIFICATION DIVISION.
PROGRAM-ID. ORATHRD2.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
78 MAX-LOOPS VALUE 10.
01 THREAD-ID USAGE POINTER.
01 TP-1 USAGE THREAD-POINTER OCCUR
01 IDEN-4 PIC 9(4).
01 LOOP-COUNTER PIC 9(2) COMP-X EXTERNAL.
                          USAGE THREAD-POINTER OCCURS MAX-LOOPS.
01 PEMPNO
                          PIC S9(4) COMP EXTERNAL.
01 ISAL
                           PIC S9(4) COMP VALUE ZERO.
     EXEC SQL
          INCLUDE SQLCA
     END-EXEC.
THREAD-LOCAL-STORAGE SECTION.
 01 CONTEXT-AREA PIC S9(9) COMP OCCURS MAX-LOOPS.
 PROCEDURE DIVISION.
MAIN SECTION.
```



```
PERFORM INITIALISATION
    PERFORM ORACLE-CONNECTIONS VARYING LOOP-COUNTER
            FROM 1 BY 1 UNTIL LOOP-COUNTER > MAX-LOOPS
    PERFORM VARYING LOOP-COUNTER FROM 1 BY 1
       UNTIL LOOP-COUNTER > MAX-LOOPS
        PERFORM START-THREAD
    END-PERFORM
    STOP RUN.
* CHECK THAT WE ARE RUNNING UNDER A MULTI THREADED RTS.
*-----
INITIALISATION SECTION.
    CALL "CBL THREAD SELF" USING THREAD-ID ON EXCEPTION
         DISPLAY "NO THREAD SUPPORT IN THIS RTS"
         STOP RUN
    END-CALL
    IF RETURN-CODE = 1008
         DISPLAY "CANNOT RUN THIS TEST ON SINGLE THREADED RTS"
         STOP RUN
    END-IF
    DISPLAY "MULTI-THREAD RTS"
* ENABLING THREADS MUST BE DONE ONCE BEFORE ANY CONTEXT USEAGE
    EXEC SQL ENABLE THREADS END-EXEC.
    IF SOLCODE NOT = ZERO
       DISPLAY 'ERROR ENABLING ORACLE THREAD SUPPORT '
               ' - ABORTING : ' SQLERRMC
       STOP RUN
    END-IF
* SET A VALUE FOR THE EMPLOYEE NUMBER. BECAUSE THIS IS AN
  EXTERNAL VARIABLE, A COPY OF ITS VALUE IS VISIBLE TO THE
* OTHER MODULES IN THIS APPLICATION
    MOVE 7566 TO PEMPNO
    EXIT SECTION.
* CREATE THREADS AND START WITH EITHER THREAD-1 OR THREAD-2
START-THREAD SECTION.
    IF LOOP-COUNTER = 2 OR LOOP-COUNTER = 5
       START "THREAD-2 "
          USING CONTEXT-AREA (LOOP-COUNTER)
          IDENTIFIED BY TP-1 (LOOP-COUNTER)
          STATUS IS IDEN-4
          ON EXCEPTION DISPLAY "THREAD CREATE FAILED"
       END-START
       IF IDEN-4 NOT = ZERO
         DISPLAY "THREAD CREATE FAILED RETURNED " IDEN-4
       END-IF
    ELSE
       START "THREAD-1 "
          USING CONTEXT-AREA (LOOP-COUNTER)
          IDENTIFIED BY TP-1 (LOOP-COUNTER)
          STATUS IS IDEN-4
          ON EXCEPTION DISPLAY "THREAD CREATE FAILED"
       END-START
```

```
IF IDEN-4 NOT = ZERO
               DISPLAY "THREAD CREATE FAILED RETURNED " IDEN-4
          END-IF.
      START-THREAD-END.
          EXIT SECTION.
      * ALLOCATE CONTEXT AREAS ESTABLISH CONNECTION WITH EACH AREA.
      *______
      ORACLE-CONNECTIONS SECTION.
          CALL "ORACON" USING CONTEXT-AREA (LOOP-COUNTER).
      ORACLE-CONNECTIONS-END.
          EXIT SECTION.
Here is the file thread-1.pco:
      ^{\star} This is Thread 1. It selects and displays the data for
      ^{\star} the employee. The context area upon which a connection
      ^{\star} has been established is passed to the thread through the
      * linkage section. In a multi-file application, you
      ^{\star} can pass the context through the linkage section.
      * Precompile with THREADS=YES.
     $SET REENTRANT MF
      IDENTIFICATION DIVISION.
      PROGRAM-ID. THREAD-1.
      ENVIRONMENT DIVISION.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01 PEMPNO
                             PIC S9(4) COMP EXTERNAL.
      LOCAL-STORAGE SECTION.
      01 DEMPNO
                             PIC Z(4) VALUE ZERO.
                           PIC X(15) VARYING VALUE SPACES.
PIC S9(7)V99 COMP-3 VALUE ZERO.
PIC S9(4) COMP VALUE ZERO.
      01 PEMP-NAME1
      01 PSAL-VALUE1
      01 ISAL1
       01 DSAL-VALUE
                             PIC +(7).99 VALUE ZERO.
          EXEC SQL
               INCLUDE SOLCA
          END-EXEC.
      LINKAGE SECTION.
      01 CONTEXT-AREA1
                             SQL-CONTEXT.
      *_____
      * USING THE PASSED IN CONTEXT, SELECT AND DISPLAY THE
      * DATA FOR EMPLOYEE.
      PROCEDURE DIVISION USING CONTEXT-AREA1.
      MAIN SECTION.
          EXEC SQL WHENEVER SQLERROR GOTO SELECT-ERROR END-EXEC
          EXEC SQL CONTEXT USE : CONTEXT-AREA1 END-EXEC
          EXEC SQL
               SELECT ENAME, SAL
                 INTO :PEMP-NAME1, :PSAL-VALUE1:ISAL1
                 FROM EMP
```

```
WHERE EMPNO = : PEMPNO
          END-EXEC
           IF ISAL1 < ZERO
             MOVE ZERO
                         TO PSAL-VALUE1
          END-IF
                         TO DEMPNO
          MOVE PEMPNO
          MOVE PSAL-VALUE1 TO DSAL-VALUE
          DISPLAY "FOR EMP ", DEMPNO, " NAME ",
                  PEMP-NAME1-ARR (1:PEMP-NAME1-LEN),
                  " THE CURRENT SALARY IS ", DSAL-VALUE
          EXIT PROGRAM.
      * THERE HAS BEEN AN ERROR WHEN SELECTING FROM THE EMP TABLE
      SELECT-ERROR SECTION.
          EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC
          DISPLAY "HIT AN ORACLE ERROR SELECTING EMPNO 7566"
          DISPLAY "SQLCODE = ", SQLCODE
          DISPLAY "ERROR TEXT ", SQLERRMC(1:SQLERRML)
          GOBACK
          EXIT SECTION.
Here is the file thread-2.pco:
      * This is Thread 2. The program will select, then update,
      * increment, and then commit the salary. It uses the passed-in
      * context upon which a connection has previously been established.
      * Precompile with THREADS=YES.
     $SET REENTRANT MF
      IDENTIFICATION DIVISION.
      PROGRAM-ID. THREAD-2.
      ENVIRONMENT DIVISION.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01 PEMPNO
                              PIC S9(4) COMP EXTERNAL.
      LOCAL-STORAGE SECTION.
                             PIC Z(4) VALUE ZERO.
      01 DEMPNO
                            PIC X(15) VARYING VALUE SPACES.
      01 PEMP-NAME2
      01 PSAL-VALUE2 PIC S9(7)V99 COMP-3 VALUE 100.
                             PIC S9(4) COMP VALUE ZERO.
      01 ISAL2
      01 DSAL-VALUE
                            PIC +(7).99 VALUE ZERO.
          EXEC SQL
               INCLUDE SQLCA
          END-EXEC.
      LINKAGE SECTION.
      01 CONTEXT-AREA2
                              SQL-CONTEXT.
      * USING THE PASSED IN CONTEXT AREA, FIRST SELECT TO GET INITIAL
      * VALUES, INCREMENT THE SALARY, UPDATE AND COMMIT.
      PROCEDURE DIVISION USING CONTEXT-AREA2.
      MAIN SECTION.
```

EXEC SQL WHENEVER SQLERROR GOTO UPDATE-ERROR END-EXEC

```
EXEC SQL CONTEXT USE : CONTEXT-AREA2 END-EXEC
           EXEC SQL
               SELECT ENAME, SAL
                 INTO :PEMP-NAME2, :PSAL-VALUE2:ISAL2
                 FROM EMP
                WHERE EMPNO = : PEMPNO
          END-EXEC
          ADD 10 TO PSAL-VALUE2
          EXEC SQL
                  UPDATE EMP
                    SET SAL
                               = :PSAL-VALUE2
                   WHERE EMPNO = : PEMPNO
           END-EXEC
          MOVE PEMPNO
                          TO DEMPNO
          MOVE PSAL-VALUE2 TO DSAL-VALUE
           DISPLAY "FOR EMP ", DEMPNO, " NAME ",
                  PEMP-NAME2-ARR (1: PEMP-NAME2-LEN),
                   " THE UPDATED SALARY IS ", DSAL-VALUE
          THIS COMMIT IS REQUIRED, OTHERWISE THE DATABASE
          WILL BLOCK SINCE THE UPDATES ARE TO THE SAME ROW
          EXEC SQL COMMIT WORK END-EXEC
          EXIT PROGRAM.
      * THERE HAS BEEN AN ERROR WHEN UPDATING THE SAL IN THE EMP TABLE
      UPDATE-ERROR SECTION.
          EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC
           DISPLAY "HIT AN ORACLE ERROR UPDATING EMPNO 7566"
           DISPLAY "SQLCODE = ", SQLCODE
          DISPLAY "ERROR TEXT ", SQLERRMC(1:SQLERRML)
          GOBACK
          EXIT SECTION.
The file oracon.pco follows:
      * This program allocates SQLLIB runtime contexts, stores
      * a pointer to the context in the variable which was
      * passed in from the main program through the linkage section,
      * and establishes a connection on the allocated context.
      ^{\star} This program is written for Merant MicroFocus COBOL and uses
      * vendor-specific directives and functionality. Precompile
      * with THREADS=YES.
      $SET REENTRANT MF
      IDENTIFICATION DIVISION.
      PROGRAM-ID. ORACON.
      ENVIRONMENT DIVISION.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01 LOGON-STRING PIC X(40)
                                               VALUE SPACES.
          EXEC SQL
               INCLUDE SQLCA
          END-EXEC.
      LINKAGE SECTION.
       01 CONTEXT
                          SQL-CONTEXT.
       PROCEDURE DIVISION USING CONTEXT.
```

#### MAIN SECTION.

```
*-----
* ALLOCATE CONTEXT AREAS ESTABLISH CONNECTION WITH EACH AREA.
ORACLE-CONNECTION SECTION.
   MOVE "SCOTT/TIGER" TO LOGON-STRING
   EXEC SQL CONTEXT ALLOCATE : CONTEXT END-EXEC
    IF SQLCODE NOT = ZERO
      DISPLAY 'ERROR ALLOCATING CONTEXT '
             '- ABORTING : ' SQLERRMC
      GOBACK
      DISPLAY 'CONTEXT ALLOCATED'
    END-IF
    EXEC SQL CONTEXT USE : CONTEXT END-EXEC
    EXEC SQL CONNECT :LOGON-STRING END-EXEC
    IF SQLCODE NOT = ZERO
      DISPLAY 'ERROR CONNECTING SECOND THREAD TO THE DATABASE '
             '- ABORT TEST : ' SQLERRMC
      GOBACK
      DISPLAY 'CONNECTION ESTABLISHED'
    END-IF
    EXIT SECTION.
```



# Large Objects (LOBs)

This chapter describes the support provided by embedded SQL statements for the *LOB* (Large Object) datatypes. The four types of LOBs are introduced and compared to the older LONG and LONG RAW datatypes.

The embedded SQL interface in Pro\*COBOL is shown to provide similar functionality to that of the PL/SQL language.

The LOB statements and their options and host variables are presented.

Last, an example of Pro\*COBOL programming using the LOB interface.

The main sections are:

- Using LOBs
- How to Use LOBs
- Rules for LOB Statements
- LOB Statements
- LOB Sample Program: LOBDEMO1.PCO

# 13.1 Using LOBs

LOBs (large objects) are database types that are used to store large amounts of data (maximum size is 4 Gigabytes) such as ASCII text, text in National Characters, files in various graphics formats, and sound wave forms.

### 13.1.1 Internal LOBs

Internal LOBs (BLOBs, CLOBs, NCLOBs) are stored in database table spaces and have transactional support of the database server. (COMMITT, ROLLBACK, and so forth work with them.)

BLOBs (Binary LOBs) store unstructured binary (also called "raw") data, such as video clips.

*CLOBs* (Character LOBs) store large blocks of character data from the database character set.

*NCLOBs* (National Character LOBs) store large blocks of character data from the national character set.

### 13.1.2 External LOBs

External LOBs are operating system files outside the database tablespaces, that have no transactional support from the database server.

*BFILEs* (Binary Files) store data in external binary files. A BFILE can be in GIF, JPEG, MPEG, MPEG2, text, or other formats.

### 13.1.3 Security for BFILEs

The *DIRECTORY* object is used to access and use BFILEs. The DIRECTORY is a logical alias name (stored in the server) for the actual physical directory in the server file system containing the file. Users are permitted to access the file only if granted access privilege on the DIRECTORY object.

Two kinds of SQL statements can be used with BFILEs:

- The DDL (data definition language) SQL statements CREATE, REPLACE, ALTER, and DROP.
- The DML (Data Management Language) SQL statements are used to GRANT and REVOKE the READ system and object privileges on DIRECTORY objects.

A sample CREATE DIRECTORY directive is:

```
EXEC SQL CREATE OR REPLACE DIRECTORY "Mydir" AS '/usr/home/mydir' END-EXEC.
```

Other users or roles can read the directory only if you grant them permission with a DML (Data Manipulation Language) statement, such as GRANT. For example, to allow user scott to read BFILES in directory /usr/home/mydir:

```
EXEC SQL GRANT READ ON DIRECTORY "Mydir" TO scott END-EXEC.
```

Up to 10 BFILES can be opened simultaneously in one session. This default value can be changed by setting the SESSION\_MAX\_OPEN\_FILES parameter.

See *Oracle Database Advanced Application Developer's Guide* for more details on DIRECTORY objects and BFILE security, and the the GRANT command.

### 13.1.4 LOBs Compared with LONG and LONG RAW

LOBs are different from the older LONG and LONG RAW datatypes in many ways.

- The maximum size of a LOB is 4 Gigabytes versus 2 Gigabytes for LONG and LONG RAW.
- You can use random as well as sequential access methods on LOBs; you can only
  use sequential access methods on LONG and LONG RAW.
- LOBs (except NCLOBs) can be attributes of an object type that you define.
- Tables can have multiple LOB columns, but can have only one LONG or LONG RAW column.

Migration of existing LONG and LONG Raw attributes to LOBs is recommended by Oracle. Oracle plans to end support of LONG and LONG RAW in future releases. See Also: Oracle Database Upgrade Guide for more information on migration, and Oracle Database SecureFiles and Large Objects Developer's Guide for more information on LOBs.

### 13.1.5 LOB Locators

A *LOB locator* points to the actual LOB contents. The locator is returned when you retrieve the LOB, not the LOB's contents. LOB locators cannot be saved in one transaction or session and used again in a later transaction or session.



## 13.1.6 Temporary LOBs

You can create *temporary* LOBs to assist your use of database LOBs. Temporary LOBs are like local variables and are not associated with any table. They are only accessible by their creator using their locators and are deleted when a session ends.

There is no support for temporary BFILES. Temporary LOBs are only permitted to be input variables (IN values) in the WHERE clause of an INSERT statement, in the SET clause of an UPDATE, or in the WHERE clause of a DELETE statement. Temporary LOBs have no transactional support from the database server, which means that you cannot do a COMMIT or ROLLBACK on them.

Temporary LOB locators can span transactions. They are deleted when the server abnormally terminates, and when an error is returned from a database SQL operation.

## 13.1.7 LOB Buffering Subsystem

The *LBS* (LOB Buffering Subsystem) is an area of user memory provided for use as a buffer for one or more LOBs in the client's address space.

Buffering has these advantages, especially for applications on a client that does many small reads and writes to specific regions of the LOB:

- The LBS reduces round-trips to the server because you fill the buffer with multiple reads/ writes to the LOBs, and then write to the server when a FLUSH directive is executed.
- Buffering also reduces the total number of LOB updates on the server. This creates better LOB performance and saves disk space.

Oracle provides a simple buffer subsystem, not a cache. Oracle does not guarantee that the contents of a buffer are always synchronized with the server LOB value. Use the FLUSH statement to actually write updates in the server LOB.

Buffered reads/writes of a LOB are performed through its locator. A locator enabled for buffering provides a consistent read version of the LOB until you perform a write through that locator.

After being used for a buffered WRITE, a locator becomes an updated locator and provides access to the latest LOB version as seen through the buffering subsystem. All further buffered WRITEs to the LOB can be done only through this updated locator. Transactions involving buffered LOB operations cannot migrate across user sessions.

The LBS is managed by the user, who is responsible for updating server LOB values by using FLUSH statements to update them. The LBS is single-user and single-threaded. Use ROLLBACK and SAVEPOINT actions to guarantee correctness in the server LOBs. Transactional support for buffered LOB operations is not guaranteed by Oracle. To ensure transactional semantics for buffered LOB updates, you must maintain logical savepoints to perform a rollback in the event of an error.

## 13.2 How to Use LOBs

There are two methods available to access LOBs in Pro\*COBOL:

- The DBMS\_LOB package inside PL/SQL blocks.
- Embedded SQL statements.



The imbedded SQL statements are designed to give users a functional equivalent to the PL/SQL interface.

The following table compares LOB access in PL/SQL and embedded SQL statements in Pro\*COBOL. Dashes indicate missing functionality.

Table 13-1 LOB Access Methods

PL/SQL <sup>1</sup>	Pro*COBOL Embedded SQL
COMPARE()	-
INSTR()	-
SUBSTR()	-
APPEND()	APPEND
:=	ASSIGN
CLOSE()	CLOSE
COPY()	COPY
CREATETEMPORARY()	CREATE TEMPORARY
-	DISABLE BUFFERING
-	ENABLE BUFFERING
ERASE()	ERASE
GETCHUNKSIZE()	DESCRIBE
ISOPEN()	DESCRIBE
FILECLOSE()	CLOSE
FILECLOSEALL()	FILE CLOSE ALL
FILEEXISTS()	DESCRIBE
FILEGETNAME()	DESCRIBE
FILEISOPEN()	DESCRIBE
FILEOPEN()	OPEN
BFILENAME()	FILE SET <sup>2</sup>
-	FLUSH BUFFER
FREETEMPORARY()	FREE TEMPORARY
GETLENGTH()	DESCRIBE
=	-
ISTEMPORARY()	DESCRIBE
LOADFROMFILE()	LOAD FROM FILE
OPEN()	OPEN
READ()	READ
TRIM()	TRIM
WRITE()	WRITE
WRITEAPPEND()	WRITE

<sup>&</sup>lt;sup>1</sup> From dbmslob.sql. All routines are prefixed with 'DBMS\_LOB.' except BFILENAME.



<sup>&</sup>lt;sup>2</sup> The BFILENAME() built in SQL function may also be used.



You must explicitly lock the row before using any of the new statements that modify or change a LOB in any way. Operations that can modify a LOB value are APPEND, COPY, ERASE, LOAD FROM FILE, TRIM, and WRITE.

## 13.2.1 LOB Locators in Your Application

To use LOB locators in your Pro\*COBOL application use these pseudo-types:

- SQL-BLOB
- SQL-CLOB
- SQL-NCLOB
- SQL-BFILE

For example, to declare an NCLOB variable called MY-NCLOB:

```
01 MY-NCLOB SQL-NCLOB.
```

## 13.2.2 Initializing a LOB

This section discusses how to initialize different varieties of LOBs.

### 13.2.2.1 Internal LOBs

To initialize a BLOB to empty, use the *EMPTY\_BLOB()* function or use the ALLOCATE SQL statement. For CLOBs and NCLOBs, use the *EMPTY\_CLOB()* function. See *Oracle Database SQL Language Reference* for more about EMPTY\_BLOB() and EMPTY\_CLOB(). These functions are permitted only in the VALUES clause of an INSERT statement or as the source of the SET clause in an UPDATE statement.

#### For example:

```
EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
   VALUES (EMPTY_BLOB(), EMPTY_CLOB()) END-EXEC.
```

The ALLOCATE statement allocates a LOB locator and initializes it to empty, so, the following code is equivalent to the previous example:

```
O1 A-BLOB SQL-BLOB.
O1 A-CLOB SQL-CLOB.

EXEC SQL ALLOCATE :A-BLOB END-EXEC.

EXEC SQL ALLOCATE :A-CLOB END-EXEC.

EXEC SQL INSERT INTO lob_table (a_blob, a_clob)

VALUES (:A-BLOB, :A-CLOB) END-EXEC.
```

### 13.2.2.2 External LOBs

Use the LOB FILE SET statement to initialize the DIRECTORY alias of the BFILE and FILENAME this way:

```
O1 ALIAS PIC X(14) VARYING.
O1 FILENAME PIC X(14) VARYING.
O1 A-BFILE SQL-BFILE.

MOVE "lob_dir" TO ALIAS-ARR.
MOVE 7 TO ALIAS-LEN.
MOVE "image.gif" TO FILENAME-ARR
MOVE 9 TO FILENAME-LEN..
EXEC SQL ALLOCATE :A-BFILE END-EXEC.
EXEC SQL LOB FILE SET :A-BFILE

DIRECTORY = :ALIAS, FILENAME = :FILENAME END-EXEC.
EXEC SQL INSERT INTO file table (a bfile) VALUES (:A-BFILE) END-EXEC.
```

Refer to *Oracle Database Advanced Application Developer's Guide* for a complete description of DIRECTORY object naming conventions and DIRECTORY object privileges.

Alternatively, you can use the BFILENAME('directory', 'filename') function in an INSERT or UPDATE statement to initialize a BFILE column or attribute for a particular row, and give the name of the actual physical directory and filename:

```
EXEC SQL INSERT INTO file_table (a_bfile)
   VALUES (BFILENAME('lob_dir', 'image.gif'))
   RETURNING a bfile INTO :A-BFILE END-EXEC.
```



BFILENAME() does not check permissions on the directory or filename, or whether the physical directory actually exists. Subsequent file accesses that use the BFILE locator will do those checks and return an error if the file is inaccessible.

## 13.2.2.3 Temporary LOBs

A temporary LOB is initialized to empty when it is first created using the embedded SQL LOB CREATE TEMPORARY statement. The EMPTY\_BLOB() and EMPTY\_CLOB() functions cannot be used with temporary LOBs.

## 13.2.2.4 Freeing LOBs

The FREE statement is used to free the memory used by an ALLOCATE statement:

```
EXEC SQL FREE : A-BLOB END-EXEC.
```

## 13.3 Rules for LOB Statements

Here are the rules for using LOB statements:

### 13.3.1 For All LOB Statements

These general restrictions and limitations apply when manipulating LOBs with the SQL LOB statements:

- The FOR clause is not allowed in embedded SQL LOB statements. Only one LOB locator can be used in those statement. However, the ALLOCATE and FREE statements do allow FOR clauses.
- Distributed LOBs are not supported. Although you may use the AT database clause in any of the new embedded SQL LOB statements, you cannot mix LOB locators that were created or allocated using different database connections in the same SQL LOB statement.

## 13.3.2 For the LOB Buffering Subsystem

For the LBS, these rules must be followed:

- Errors in read or write accesses are reported at the next access to the server. Therefore, error recovery has to be coded by you, the user.
- When updating a LOB with buffered writes, do not update the same LOB with a method that bypasses the LOB Buffering Subsystem.
- An updated LOB locator enabled for buffering can be passed as an IN parameter to a PL/SQL procedure, but not as an IN OUT or OUT parameter. An error is returned. An error is also returned when there is an attempt to return an updated locator.
- An ASSIGN of an updated locator enabled for buffering to another locator is not allowed.
- You can append to the LOB value with buffered writes, but the starting offset must be one character after the end of the LOB. The LBS does not allow APPEND statements resulting in zero-byte fillers or spaces in LOBs in the database server.
- The character sets of the host locator bind variable and the database server CLOB must be the same.
- Only ASSIGN, READ and WRITE statements work with a locator enabled for buffering.
- The following statements result in errors when used with a locator enabled for buffering: APPEND, COPY, ERASE, DESCRIBE (LENGTH only), SELECT, and TRIM. Errors are also returned when you use these statements with a locator that is not enabled for buffering, if the LOB pointed to by the locator is being accessed in buffered mode by another locator.



The FLUSH statement must be used on a LOB enabled by the LOB Buffering Subsystem before

- Committing the transaction.
- Migrating from the current transaction to another.
- Disabling buffer operations on a LOB.

### 13.3.3 For Host Variables

Use the following rules and notes for the LOB statements:

 src and dst can refer to either internal or external LOB locators, but file refers only to external locators.



- Numeric host values (amt, src\_offset, dst\_offset, and so forth.) are declared
  as a 4-byte integer variable, PIC S9(9) COMP. The values are restricted between
  0 and 4 Gigabytes.
- The concept of NULL is part of a LOB locator. There is no need for indicator variables in the LOB statements. NULL cannot be used with numeric value variables such as amt, src offset, and so forth and result in an error.
- The offset values src offset and dst offset have default values 1.



BLOB, CLOB, and NCLOB variables need to respect the alignment requirements of your platform. Refer to you platform documentation on alignment restrictions of your particular platform.

## 13.4 LOB Statements

The statements are presented alphabetically. In all the statements where it appears, database refers to a database connection

### 13.4.1 APPEND

### **Purpose**

The APPEND statement appends a LOB value at the end of another LOB.

### **Syntax**

EXEC SQL [AT [:] database] LOB APPEND :src TO :dst END-EXEC.

### **Host Variables**

#### src (IN)

An internal LOB locator uniquely referencing the source LOB.

### dsc (IN OUT)

An internal LOB locator uniquely referencing the destination LOB.

### **Usage Notes**

The data is copied from the source LOB to the end of the destination LOB, extending the destination LOB up to a maximum of 4 Gigabytes. If the LOB is extended beyond 4 Gigabytes, an error will occur.

The source and destination LOBs must already exist and the destination LOB must be initialized.

Both the source and destination LOBs must be of the same internal LOB type. It is an error to have enabled LOB buffering for either type of locator.



### 13.4.2 ASSIGN

### **Purpose**

Assigns a LOB or BFILE locator to another.

### **Syntax**

```
EXEC SQL [AT [:] database] LOB ASSIGN : src to : dst END-EXEC.
```

### **Host Variables**

### src (IN)

LOB or BFILE locator source copied from.

### dsc (IN OUT)

LOB or BFILE locator copied to.

### **Usage Notes**

After the assignment, both locators refer to the same LOB value. The destination LOB locator must be a valid initialized (allocated) locator.

For internal LOBs, the source locator's LOB value is copied to the destination locator's LOB value only when the destination locator is stored in the table. For Pro\*COBOL, issuing a FLUSH of an object containing the destination locator will copy the LOB value.

An error is returned when a BFILE locator is assigned to an internal LOB locator and viceversa. It is also an error if the *src* and *dst* LOBs are not of the same type.

If the source locator is for an internal LOB that was enabled for buffering, and the source locator has been used to modify the LOB value through the LOB Buffering Subsystem, and the buffers have not been flushed since the WRITE, then the source locator cannot be assigned to the destination locator. This is because only one locator for each LOB can modify the LOB value through the LOB Buffering Subsystem.

## 13.4.3 CLOSE

### **Purpose**

Close an open LOB or BFILE.

### **Syntax**

```
EXEC SQL [AT [:] database] LOB CLOSE :src END-EXEC.
```

#### **Host Variables**

### src (IN OUT)

The locator of the LOB or BFILE to be closed.

### **Usage Notes**

It is an error to close the same LOB twice either with different locators or with the same locator. For external LOBs, no error is produced if the BFILE exists but has not been opened.



It is an error to COMMIT a transaction before closing all previously opened LOBs. At transaction ROLLBACK time, all LOBs that are still open will be discarded without first being closed.

### 13.4.4 COPY

### **Purpose**

Copy all or part of a LOB value into a second LOB.

### **Syntax**

```
EXEC SQL [AT [:]database] LOB COPY :amt FROM :src [AT :src_offset] TO :dst [AT :dst offset] END-EXEC.
```

#### **Host Variables**

### amt (IN)

The maximum number of bytes for BLOBs, or characters for CLOBs and NCLOBs, to copy.

### src (IN)

The locator of the source LOB.

### src\_offset (IN)

This is the number of characters for CLOB or NCLOB, and the number of bytes for a BLOB, starting from 1 at the beginning of the LOB.

### dst (IN)

The locator of the destination LOB.

### dst offset (IN)

The destination offset. Same rules as for src offset.

### **Usage Notes**

If the data already exists at the destination's offset and beyond, it is overwritten with the source data. If the destination's offset is beyond the end of the current data, zero-byte fillers (BLOBs) or spaces (CLOBs) are written into the destination LOB from the end of the current data to the beginning of the newly written data from the source.

The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is a runtime error to extend this LOB beyond 4 Gigabytes.

It is also an error to try to copy from a LOB that is not initialized.

Both the source and destination LOBs must be of the same type. LOB buffering must not be enabled for either locator.

To make a temporary LOB permanent, the COPY statement must be used to explicitly COPY the temporary LOB into a permanent one. The amt variable indicates the maximum amount to copy. If the end of the source LOB is reached before the specified amount is copied, the operation terminates without an error.



### 13.4.5 CREATE TEMPORARY

### **Purpose**

Creates a temporary LOB.

### **Syntax**

EXEC SQL [AT [:] database] LOB CREATE TEMPORARY :src END-EXEC.

### **Host Variables**

### src (IN OUT)

Before execution, when IN, src is a LOB locator previously allocated.

After execution, when OUT, src is a LOB locator that will point to a new empty temporary LOB.

### **Usage Notes**

After successful execution, the locator points to a newly created temporary LOB that resides on the database server independent of a table. The temporary LOB is empty and has zero length.

At the end of a session, all temporary LOBs are freed. Reads and Writes to temporary LOBs never go through the buffer cache.

### 13.4.6 DISABLE BUFFERING

### **Purpose**

Disables LOB buffering for the LOB locator.

#### **Syntax**

EXEC SOL [AT [:] database] LOB DISABLE BUFFERING : src END-EXEC.

### **Host Variables**

### src (IN OUT)

An internal LOB locator.

### **Usage Notes**

This statement does not support BFILEs. Subsequent reads or writes will not be done through the LBS.

**Note:** Use a FLUSH BUFFER command to make changes permanent, since the DISABLE BUFFERING statement does not implicitly flush the changes made in the LOB Buffering Subsystem.



### 13.4.7 ENABLE BUFFERING

### **Purpose**

Enables LOB buffering for the LOB locator.

### **Syntax**

```
EXEC SQL [AT [:] database] LOB ENABLE BUFFERING :src END-EXEC.
```

### **Host Variables**

#### src (IN OUT)

An internal LOB locator.

### **Usage Notes**

This statement does not support BFILEs. Subsequent reads and writes are done through the LBS.

### 13.4.8 ERASE

### **Purpose**

Erases a given amount of LOB data starting from a given offset.

### **Syntax**

```
EXEC SQL [AT [:]database] LOB ERASE :amt
FROM :src [AT :src offset] END-EXEC.
```

### **Host Variables**

### amt (IN OUT)

The input is the number of bytes or characters to erase. The returned output is the actual number erased.

### src (IN OUT)

An internal LOB locator.

### src\_offset (IN)

The offset from the beginning of the LOB, starting from 1.

### **Usage Notes**

This statement does not support BFILEs.

After execution, *amt* returns the actual number of characters/bytes that were erased. The actual number and requested number will differ if the end of the LOB value is reached before erasing the requested number of characters/bytes. If the LOB is empty, *amt* will indicate that 0 characters/bytes were erased.



For BLOBs, erasing means zero-byte fillers overwrite the existing LOB value. For CLOBs, erasing means that spaces overwrite the existing LOB value.

### 13.4.9 FILE CLOSE ALL

### **Purpose**

Closes all BFILES opened in the current session.

### **Syntax**

```
EXEC SQL [AT [:] database] LOB FILE CLOSE ALL END-EXEC.
```

### **Usage Notes**

If there are any open files in the session whose closure has not been handled properly, you can use the FILE CLOSE ALL statement to close all files opened in the session and resume file operations from the beginning.

### 13.4.10 FILE SET

### **Purpose**

Sets DIRECTORY alias and FILENAME in a BFILE locator.

### **Syntax**

```
EXEC SQL [AT [:]database] LOB FILE SET :file
DIRECTORY = :alias, FILENAME = :filename END-EXEC.
```

#### **Host Variables**

### file (IN OUT)

BFILE locator where the DIRECTORY alias and FILENAME is set.

### alias (IN)

DIRECTORY alias name to set.

#### filename (IN)

The FILENAME to set.

#### **Usage Notes**

The given BFILE locator must be first ALLOCATEd prior to its use in this statement.

Both the DIRECTORY alias name and FILENAME must be provided.

The maximum length of the DIRECTORY alias is 128 bytes. The maximum length of the FILENAME is 255 bytes.

The only external datatypes supported for use with the DIRECTORY alias name and FILENAME attributes are VARCHAR, VARCHAR2 and CHARF.

It is an error to use this statement with anything but an external LOB locator.



### 13.4.11 FLUSH BUFFER

### **Purpose**

Writes this LOB's buffers to the database server.

### **Syntax**

EXEC SQL [AT [:] database] LOB FLUSH BUFFER :src [FREE] END-EXEC.

### **Host Variables**

### src (IN OUT)

Internal LOB locator.

### **Usage Notes**

Writes the buffer data to the database LOB in the server from the LOB referenced by the input locator.

LOB buffering must have already been enabled for the input LOB locator.

The FLUSH operation, by default, does not free the buffer resources for reallocation to another buffered LOB operation. However, if you want to free the buffer explicitly, you can include the optional FREE keyword to so indicate.

## 13.4.12 FREE TEMPORARY

### **Purpose**

Free the temporary space for the LOB locator.

### **Syntax**

EXEC SQL [AT [:] database] LOB FREE TEMPORARY : src END-EXEC.

### **Host Variables**

### src (IN OUT)

The LOB locator pointing to the temporary LOB.

### **Usage Notes**

The input locator must point to a temporary LOB. The output locator is marked not initialized and can be used in subsequent LOB statements.

## 13.4.13 LOAD FROM FILE

### **Purpose**

Copy all or a part of a BFILE into an internal LOB.



### **Syntax**

```
EXEC SQL [AT [:]database] LOB LOAD :amt
FROM FILE :file [AT :src offset] INTO :dst [AT :dst offset] END-EXEC.
```

#### **Host Variables**

### amt (IN)

Maximum number of bytes to be loaded.

### file (IN OUT)

The source BFILE locator.

### src\_offset (IN)

The number of bytes offset from the beginning of the file, starting from 1.

### dst (IN OUT)

The destination LOB locator which can be BLOB, CLOB, be NCLOB.

### dst\_offset (IN)

The number of bytes (for BLOBs) or characters (CLOBs and NCLOBs) from the beginning of the destination LOB where writing will begin. It starts at 1.

### **Usage Notes**

The data is copied from the source BFILE to the destination internal LOB. No character set conversions are performed when copying the BFILE data to a CLOB or NCLOB. Therefore, the BFILE data must already be in the same character set as the CLOB or NCLOB in the database.

The source and destination LOBs must already exist. If the data already exists at the destination's start position, it is overwritten with the source data. If the destination's start position is beyond the end of the current data, zero-byte fillers (BLOBs) or spaces (CLOBs and NCLOBs) are written into the destination LOB from the end of the data to the beginning of the newly written data from the source.

The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is an error to extend this LOB beyond 4 Gigabytes.

It is also an error to copy from a BFILE that is not initialized.

The amount parameter indicates the maximum amount to load. If the end of the source BFILE is reached before the specified amount is loaded, the operation terminates without error.

### 13.4.14 OPEN

### **Purpose**

Open a LOB or BFILE for read or read/write access.



### **Syntax**

```
EXEC SQL [AT [:] database] LOB OPEN :src [ READ ONLY | READ WRITE ] END-EXEC.
```

#### **Host Variables**

### src (IN OUT)

LOB locator of the LOB or BFILE.

### **Usage Notes**

The default mode in which a LOB or BFILE can be Opened is for READ ONLY access.

For internal LOBs, being OPEN is associated with the LOB, not with the locator. Assigning an already Opened locator to another locator does not count as OPENing a new LOB. Instead, both locators refer to the same LOB. For BFILEs, being OPEN is associated with the locator.

Only 32 LOBs can be OPEN at any one time. An error will be returned when the 33rd LOB is Opened.

There is no support for writable BFILEs. Therefore, when you OPEN a BFILE in READ WRITE mode, an error is returned.

It is also an error to open a LOB in READ ONLY mode and then attempt to WRITE to the LOB.

### 13.4.15 READ

#### **Purpose**

Reads all or part of a LOB or BFILE into a buffer.

#### **Syntax**

```
EXEC SQL [AT [:] database] LOB READ :amt FROM :src [AT :src_offset] INTO :buffer [WITH LENGTH :buflen] END-EXEC.
```

#### **Host Variables**

### amt (IN OUT)

The input is the number of characters or bytes to be read. The output is the actual number of characters or bytes that were read.

If the amount of bytes to be read is larger than the buffer length it is assumed that the LOB is being READ in a polling mode. On input if this value is 0, then the data will be read in a polling mode from the input offset until the end of the LOB.

The number of bytes or characters actually read is returned in amt. If the data is read in pieces, amt will always contain the length of the last piece read.

When the end of a LOB is reached an ORA-1403: no data found error will be issued.

When reading in a polling mode, the application must invoke the LOB READ repeatedly to read more pieces of the LOB until no more data is left. Control the use of



the polling mode with the NOT FOUND condition in a WHENEVER directive to catch the ORA-1403 error.

### src (IN)

The LOB or BFILE locator.

### src\_offset (IN)

This is the absolute offset from the beginning of the LOB value from which to start reading. For character LOBs it is the number of characters from the beginning of the LOB. For binary LOBs or BFILEs it is the number of bytes. The first position is 1.

### buffer (IN OUT)

A buffer into which the LOB data will be read. The external datatype of the buffer is restricted to only a few types depending on the type of the source LOB. The maximum length of the buffer depends on the external datatype being used to store the LOB value. The following table summarizes the legal external datatypes and their corresponding maximum lengths categorized by source LOB type:

**Table 13-2** Source LOB and Precompiler Datatypes

External LOB <sup>1</sup>	Internal LOB	Precompiler External Datatype	Precompiler Maximum Length <sup>2</sup>	PL/SQL Datatype	PL/SQL Maximum Length
DEIL E	BLOB	RAW	65535	RAW	32767
BFILE		VARRAW	65533		
		LONG RAW	2147483647		
		LONG VARRAW	2147483643		
-	CLOB	VARCHAR2	65535	VARCHAR2	32767
		VARCHAR	65533		
		LONG VARCHAR	2147483643		
-	NCLOB	NVARCHAR2	4000	NVARCHAR2	4000

<sup>&</sup>lt;sup>1</sup> Any of the external datatypes shown can be used with BFILES.

#### buflen (IN)

Specifies the length of the given buffer when it cannot be determined otherwise.

### **Usage Notes**

A BFILE must already exist on the database server and must have been opened using the input locator. The database must have permission to read the file and the user must have read permission on the directory.

It is an error to try to read from an un-initialized LOB or BFILE.

The length of the buffer is determined this way:

- From buflen, when the WITH LENGTH clause is present.
- In the absence of the WITH LENGTH clause, the length is determined by treating the buffer host variable in OUT mode according to the rules in "Handling Character Data".



<sup>&</sup>lt;sup>2</sup> Lengths are measured in bytes, not characters.

### **Related Topics**

Handling Character Data

### 13.4.16 TRIM

### **Purpose**

Truncates the LOB value.

### **Syntax**

```
EXEC SQL [AT [:] database] LOB TRIM : src TO : newlen END-EXEC.
```

#### **Host Variables**

### src (IN OUT)

LOB locator for internal LOB.

### newlen (IN)

The new length of the LOB value.

### **Usage Notes**

This statement is not for BFILES. The new length cannot be greater than the current length, or an error is returned.

### 13.4.17 WRITE

### **Purpose**

Writes the contents of a buffer to a LOB.

#### **Syntax**

```
EXEC SQL [AT [:]database] LOB WRITE [APPEND] [ FIRST | NEXT | LAST | ONE ]
  :amt FROM :buffer [WITH LENGTH :buflen]
    INTO :dst [AT :dst_offset] END-EXEC.
```

### **Host Variables**

### amt (IN OUT)

The input is the number of characters or bytes to be written.

The output is the actual number of characters or bytes that is written.

When writing using a polling method, amt will return the cumulative total length written for the execution of the WRITE statement after a WRITE LAST is executed. If the WRITE statement is interrupted, amt will be undefined.

### buffer (IN)

A buffer from which the LOB data is written. See "READ" for the lengths of datatypes.



### dst (IN OUT)

The LOB locator.

### dst\_offset (IN)

The offset from the beginning of the LOB (counting from 1), in characters for CLOBs and NCLOBs, in bytes for BLOBs.

### buflen (IN)

The buffer length when it cannot be calculated in any other way.

### **Usage Notes**

If LOB data already exists, it is overwritten with the data stored in the buffer. If the offset specified is beyond the end of the data currently in the LOB, zero-byte fillers or spaces are inserted into the LOB.

Specifying the keyword APPEND in the WRITE statement causes the data to automatically be written to the end of the LOB. When APPEND is specified, the destination offset is assumed to be the end of the LOB. It is an error to specify the destination offset when using the APPEND option in the WRITE statement.

The buffer can be written to the LOB in one piece (using the ONE orientation which is the default) or it can be provided piece-wise using a standard polling method.

Polling is begun by using FIRST, then NEXT to write subsequent pieces. The LAST keyword is used to write the final piece that terminates the write.

Using this piece-wise write mode, the buffer and the length can be different in each call if the pieces are of different sizes and from different locations.

If the total amount of data passed to Oracle is less than the amount specified by the amt parameter after doing all the writes, an error results.

The same rules apply for determining the buffer length as in the READ statement. See "READ".

### **Related Topics**

READ

## **13.4.18 DESCRIBE**

#### **Purpose**

This is a statement that is equivalent to several OCI and PL/SQL statements. Use the LOB DESCRIBE SQL statement to retrieve attributes from a LOB. The LOB DESCRIBE statement has this format:

### **Syntax**

```
EXEC SQL [AT [:]database] LOB DESCRIBE :src GET attribute1 [{, attributeN}]
INTO :hv1 [[INDICATOR] :hv_ind1] [{, :hvN [[INDICATOR] :hv_indN] }]
END-EXEC.
```

where an attribute can be any of these choices:



CHUNKSIZE | DIRECTORY | FILEEXISTS | FILENAME | ISOPEN | ISTEMPORARY | LENGTH

### **Host Variables**

src (IN)

The LOB locator of an internal or external LOB.

hv1 ... hvN (OUT)

The host variables that receive the attribute values, in the order specified in the attribute name list.

hv\_ind1 ... hv\_indN (OUT)

Optional host variables that receive the indicator NULL status in the order of the attribute name list.

The following table describes the attributes, which LOB it is associated with, and the COBOL types into which they should be read:

Table 13-3 LOB Attributes

LOB Attribute	Attribute Description	Restrictions	COBOL Type
CHUNKSIZE	The optimum amount (in bytes for BLOBs and characters for CLOBs/NCLOBs) of space of a given tablespace block, which is used to store the LOB value. Chunk sizes are not fixed and the maximum possible value based on the available chunk sizes is used for storage. If you make multiple WRITE requests on the same page or set of pages, data is written to the disk only when committed and is treated as a single modification. This allows for data being written in larger chunk sizes.	only	PIC S9(9) COMP
DIRECTORY	The name of the DIRECTORY alias for the BFILE. The length, n, is between 1 and 128 bytes. Use that length in the picture.	FILE LOBs only	PIC X(n) [VARYING]
FILEEXISTS	Determines whether or not the BFILE exists on the server's operating system file system. FILEEXISTS is true when it is nonzero; false when it equals 0.	FILE LOBs only	PIC S9(9) COMP
FILENAME	The name of the BFILE. The length, n, is between 1 and 255 bytes. Use that length in the picture.	FILE LOBs only	PIC X(n) [VARYING]
ISOPEN	For BFILEs, if the input BFILE locator was never used in an OPEN statement, the BFILE is considered not to be OPENed by this locator. However, a different BFILE locator may have OPENed the BFILE. More than one OPEN can be performed on the same BFILE using different locators. For LOBs, if a different locator opened the LOB, the LOB is still considered to be OPEN by the input locator. ISOPEN is true when it is nonzero; false when it equals 0.	-	PIC S9(9) COMP
ISTEMPORARY	Determines whether or not the input LOB locator refers to a temporary LOB or not. ISTEMPORARY is true when it is nonzero; false when it equals 0.	BLOBs, CLOBs and NCLOBs only	PIC S9(9) COMP



Table 13-3 (Cont.) LOB Attributes

LOB Attribute	Attribute Description	Restrictions	COBOL Type
LENGTH	Length of BLOBs and BFILEs in bytes, CLOBs and NCLOBs in characters. For BFILEs, the length includes EOF if it exists. Empty internal LOBs have zero length. LOBs/BFILEs that are not initialized have undefined length.		PIC 9(9) COMP

### **Usage Notes**

Indicator variables should be declared as PIC S9(4) COMP. After execution has completed, SQLERRD(3) contains the number of attributes retrieved without error. If there was an execution error, the attribute at which it occurred is one more than the contents of SQLERRD(3).

### **DESCRIBE Example**

Here is a simple Pro\*COBOL example that extracts the DIRECTORY and FILENAME attributes of a given BFILE:

```
01 A-BFILE SQL-BFILE.
01 DIRECTORY PIC X(30) VARYING.
01 FILENAME PIC X(30) VARYING.
01 D-IND PIC S9(4) COMP.
01 F-IND PIC S9(4) COMP.
01 FEXISTS PIC S9(9) COMP.
01 ISOPN PIC S9(9) COMP.
```

### Finally, select a BFILE locator from some LOB table and do the DESCRIBE:

```
EXEC SQL ALLOCATE :A-BFILE END-EXEC.

EXEC SQL INSERT INTO lob_table (a_bfile) VALUES (BFILENAME ('lob.dir', 'image.gif')) END-EXEC.

EXEC SQL SELECT a_bfile INTO :A-BFILE FROM lob_table WHERE ... END-EXEC.

EXEC SQL DESCRIBE :A-BFILE GET DIRECTORY, FILENAME, FILEEXISTS, ISOPEN INTO :DIRECTORY:D-IND, :FILENAME:F-IND, FEXISTS, ISOPN ND-EXEC.
```

Indicator variables are valid for use only with the DIRECTORY and FILENAME attributes. These attributes are character strings whose values may be truncated if the host variable buffers used to hold their values are not large enough. When truncation occurs, the value of the indicator will be set to the original length of the attribute.

## 13.4.19 READ and WRITE Using the Polling Method

Here is an outline of using READ with the polling method:

Start the read polling by setting the amount to zero in the first LOB READ (or set the amount to the size of the total data to be read). The amount is first set to zero in this case outline which omits details:

```
EXEC SQL ALLOCATE :CLOB1 END-EXEC.

EXEC SOL WHENEVER NOT FOUND GOTO END-OF-CLOB END-EXEC.
```



```
EXEC SQL SELECT A_CLOB INTO :CLOB1 FROM LOB_TABLE WHERE ... END-EXEC.

MOVE 0 TO AMT.

EXEC SQL LOB READ :AMT FROM :VLOB1 AT :OFFSET INTO :BUFFER END-EXEC.

READ-LOOP.

EXEC SQL LOB READ :AMT FROM :CLOB1 INTO BUFFER $END-EXEC.

GO TO READ-LOOP.

END-OF-CLOB.

EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

EXEC SQL FREE :CLOB1 END-EXEC.
```

The following code outline writes data from a buffer into an internal CLOB. The value of AMT (16 characters) in the initial write statement should be the length of the entire data you will write. The buffer is 5 characters long.

If EOF is read in the initial read, then do the LOB WRITE ONE. If not, start polling with a LOB WRITE FIRST of the buffer. Read the data, and do a LOB WRITE NEXT of the output. No offset is needed in the LOB WRITE NEXT because data is written at the end of the last write. After EOF is read, break out of the read loop and do a LOB WRITE LAST. The amount returned must equal the initial amount value (16).

```
MOVE 16 TO AMT.
    PERFORM READ-NEXT-RECORD.
   MOVE INREC TO BUFFER-ARR.
   MOVE 5 TO BUFFER-LEN.
    IF (END-OF-FILE = "Y")
        EXEC SQL LOB WRITE ONE :AMT FROM :BUFFER INTO CLOB1
          AT :OFFSET END-EXEC.
        PERFORM DISPLAY-CLOB
    ELSE
        EXEC SOL LOB WRITE FIRST :AMT FROM :BUFFER INTO :CLOB1
          AT :OFFSET END-EXEC.
    PERFORM READ-NEXT-RECORD.
    PERFORM WRITE-TO-CLOB
      UNTIL END-OF-FILE = "Y".
    MOVE INREC TO BUFFER-ARR.
    MOVE 1 TO BUFFER-LEN.
    EXEC SQL LOB WRITE LAST :AMT FROM :BUFFER INTO :CLOB1 END-EXEC.
    PERFORM DISPLAY-CLOB.
    . . .
WRITE-TO-CLOB.
   MOVE INREC TO BUFFER-ARR.
    MOVE 5 TO BUFFER-LEN.
    EXEC SQL LOB WRITE NEXT :AMT FROM :BUFFER INTO :CLOB1 END-EXEC.
    PERFORM READ-NEXT RECORD.
READ-NEXT-RECORD.
   MOVE SPACES TO INREC.
    READ INFILE NEXT RECORD
      AT END
      MOVE "Y" TO END-OF-FILE.
```



# 13.5 LOB Sample Program: LOBDEMO1.PCO

The LOBDEMO1.PCO program illustrates several LOB embedded SQL statements. The source code is in your <code>demo</code> directory. The application uses a table named <code>license\_table</code> whose columns are social security number, name, and a CLOB containing text summarizing driving offenses. Several simplified SQL operations of a typical motor vehicle department are modeled.

The possible actions are:

- · Add new records.
- List records by social security number.
- · List information in a record, given a social security number.
- Append a new traffic violation to an existing CLOB's contents.

Here is the listing of LOBDEMO1.PCO:

```
*************
* LOB Demo 1: DMV Database
* SCENARIO:
^{\star} We consider the example of a database used to store driver's
* licenses. The licenses are stored as rows of a table containing
* three columns: the sss number of a person, his/her name and the
* text summary of the info found in his license.
* The sss number and the name are the unique social security number
* and name of an individual. The text summary is a summary of the
* information on the individual, including his driving record,
* which can be arbitrarily long and may contain comments and data
* regarding the person's driving ability.
* APPLICATION OVERVIEW:
* This example demonstrate how a Pro*COBOL client can handle the
* new LOB datatypes. Demonstrated are the mechanisms for accessing
* and storing lobs to/from tables.
  To run the demo:
  1. Execute the script, lobdemol.sql in Server Manager
  2. Precompile using Pro*COBOL
       procob lobdemo1
  3. Compile/Link (This step is platform specific)
* lobdemol.sql contains the following SQL statements:
* connect scott/tiger;
* drop table license table;
* create table license table(
* sss char(9),
* name varchar2(50),
* txt summary clob);
```

```
* insert into license table
* values('971517006', 'Dennis Kernighan',
* 'Wearing a Bright Orange Shirt');
* insert into license table
* values('555001212', 'Eight H. Number',
* 'Driving Under the Influence');
* insert into license_table
* values('010101010', 'P. Doughboy',
  'Impersonating An Oracle Employee');
* insert into license table
 values('555377012', 'Calvin N. Hobbes',
  'Driving Under the Influence');
* The main program provides the menu of actions that can be
* performed. The program stops when the number 5 (Quit) option
* is entered. Depending on the input, this main program calls
* the appropriate nested program to execute the chosen action.
****************
IDENTIFICATION DIVISION.
PROGRAM-ID. LOBDEMO1.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 USERNAME
                PIC X(5).
01 PASSWD
                 PIC X(5).
01 CHOICE
                PIC 9 VALUE 0.
01 SSS
                 PIC X(9).
01 SSSEXISTS PIC 9 VALUE ZERO.
01 LICENSE-TXT SQL-CLOB .
01 NEWCRIME PIC X(35) VARYING.
01 SSSCOUNT PIC S9(4) COMP.
01 SSSCOUNT
01 THE-STRING PIC X(200) VARYING.
01 TXT-LENGTH PIC S9(9) COMP.
 01 CRIMES.
    05 FILLER PIC X(35) VALUE "Driving Under the Influence".
    05 FILLER PIC X(35) VALUE "Grand Theft Auto".
    05 FILLER PIC X(35) VALUE "Driving Without a License".
    05 FILLER PIC X(35) VALUE
           "Impersonating an Oracle Employee".
    05 FILLER PIC X(35) VALUE "Wearing a Bright Orange Shirt".
 01 CRIMELIST REDEFINES CRIMES.
    05 CRIME PIC X(35) OCCURS 5 TIMES.
01 CRIME-INDEX PIC 9.
01 TXT-LEN
               PIC S9(9) COMP.
01 CRIME-LEN PIC S9(9) COMP.
01 NAME1
               PIC X(50) VARYING.
01 NEWNAME
                  PIC X(50).
                                 ********
    EXEC SQL INCLUDE SQLCA END-EXEC.
PROCEDURE DIVISION.
A000-CONTROL SECTION.
* A000-CONTROL
    Overall control section
A000-CNTRL.
```

```
EXEC SQL
 WHENEVER SQLERROR DO PERFORM Z900-SQLERROR
  END-EXEC.
     PERFORM B000-LOGON.
  PERFORM C000-MAIN UNTIL CHOICE = 5.
  PERFORM D000-LOGOFF.
A000-EXIT.
  STOP RUN.
B000-LOGON SECTION.
******************
* B000-LOGON
  Log on to database.
****************
B000-LGN.
   DISPLAY '*
              Welcome to the DMV Database
   MOVE "scott" TO USERNAME.
   MOVE "tiger" TO PASSWD.
   EXEC SQL
     CONNECT : USERNAME IDENTIFIED BY : PASSWD
   END-EXEC.
   DISPLAY " ".
   DISPLAY "Connecting to license database account: ",
     USERNAME, "/", PASSWD.
B000-EXIT.
   EXIT.
C000-MAIN SECTION.
              ***********
* C000-MAIN
  Display the main menu and action requests
********************
C000-MN.
   DISPLAY " ".
   DISPLAY "License Options:".
   DISPLAY "1. List available records by SSS number".
   DISPLAY "2. Get information on a particular record".
   DISPLAY "3. Add crime to a record".
   DISPLAY "4. Insert new record to database".
   DISPLAY "5. Quit".
   DISPLAY " ".
   MOVE ZERO TO CHOICE.
   PERFORM Z300-ACCEPT-CHOICE UNTIL CHOICE < 6
                     AND CHOICE > 0.
   IF (CHOICE = 1)
     PERFORM C100-LIST-RECORDS.
   IF (CHOICE = 2)
      PERFORM C200-GET-RECORD.
   IF (CHOICE = 3)
      PERFORM C300-ADD-CRIME.
   IF (CHOICE = 4)
      PERFORM C400-NEW-RECORD.
 C000-EXIT.
EXIT.
 C100-LIST-RECORDS SECTION.
************
```

```
* C100-LIST-RECORDS
  Select Social Security Numbers from LICENCSE TABLE
 and display the list
     ************
 C100-LST.
   EXEC SQL DECLARE SSS CURSOR CURSOR FOR
    SELECT SSS FROM LICENSE TABLE
   END-EXEC.
   EXEC SQL OPEN SSS CURSOR END-EXEC.
   DISPLAY "Available records:".
   PERFORM C110-DISPLAY-RECORDS UNTIL SOLCODE = 1403.
   EXEC SQL CLOSE SSS CURSOR END-EXEC.
C100-EXIT.
   EXIT.
C110-DISPLAY-RECORDS SECTION.
******************
C110-DISPLAY-RECORDS
  Fetch the next record from the cursor and display it.
******************
C110-DSPLY.
    EXEC SQL FETCH SSS CURSOR INTO :SSS END-EXEC.
    IF SQLCODE = 0 THEN
      DISPLAY SSS.
C110-EXIT.
    EXIT.
C200-GET-RECORD SECTION.
***************
* C200-GET-RECORD
  Allocates the global clob LICENSE-TXT then selects
   the name and text which corresponds to the client-supplied
   sss. It then calls Z200-PRINTCRIME to print the information and
   frees the clob.
************
C200-GTRECRD.
   PERFORM Z100-GET-SSS.
   IF (SSSEXISTS = 1)
    EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
    EXEC SQL SELECT NAME, TXT SUMMARY
        INTO :NAME1, :LICENSE-TXT FROM LICENSE TABLE
        WHERE SSS = :SSS END-EXEC
    "======="
    DISPLAY " "
    DISPLAY "NAME: ", NAME1-ARR, "SSS: ", SSS
    DISPLAY " "
    PERFORM Z200-PRINTCRIME
    DISPLAY " "
    "----"
    EXEC SQL FREE :LICENSE-TXT END-EXEC
     DISPLAY "SSS Number Not Found".
C200-EXIT.
   EXIT.
C310-GETNEWCRIME SECTION.
************
```

```
* C310-GETNEWCRIME
   Provides a list of the possible crimes to the user and
   stores the user's correct response in the variable
******************
C310-GTNWCRM.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     DISPLAY " ".
     DISPLAY "Select from the following:".
     PERFORM C311-DISPLAY-CRIME
       VARYING CRIME-INDEX FROM 1 BY 1
      UNTIL CRIME-INDEX > 5.
     MOVE ZERO TO CHOICE.
     PERFORM Z300-ACCEPT-CHOICE UNTIL CHOICE < 6
                         AND CHOICE > 0.
     MOVE CRIME (CHOICE) TO NEWCRIME-ARR.
     MOVE 35 TO NEWCRIME-LEN.
     MOVE ZERO TO CHOICE.
C310-EXIT.
     EXIT.
C311-DISPLAY-CRIME SECTION.
*****************
   C311-DISPLAY-CRIME
     Display an element of the crime table
C311-DSPLYCRM.
    DISPLAY "(", CRIME-INDEX, ") ", CRIME(CRIME-INDEX).
C311-EXIT.
     EXIT.
C320-APPENDTOCLOB SECTION.
* C320-APPENDTOCLOB
    Obtains the length of the global clob LICENSE-TXT and
    uses that in the LOB WRITE statement to append the NEWCRIME
    character buffer to the global clob LICENSE-TXT.
    The name corresponding the global SSS is then selected
    and displayed to the screen along with value of LICENSE-TXT.
    The caller to this function must allocate, select and later
    free the global clob LICENSE-TXT.
C320-PPNDTCLB.
    EXEC SQL
 WHENEVER SQLERROR DO PERFORM Z900-SQLERROR
END-EXEC.
    EXEC SQL LOB DESCRIBE : LICENSE-TXT GET LENGTH
        INTO :TXT-LEN END-EXEC.
    MOVE NEWCRIME-LEN TO CRIME-LEN.
    IF (TXT-LEN NOT = 0)
      ADD 3 TO TXT-LEN
    ELSE
      ADD 1 TO TXT-LEN.
    EXEC SQL LOB WRITE : CRIME-LEN FROM : NEWCRIME
          INTO :LICENSE-TXT AT :TXT-LEN END-EXEC.
    EXEC SQL SELECT NAME INTO :NAME1 FROM LICENSE TABLE
        WHERE SSS = :SSS END-EXEC.
```

```
DISPLAY " ".
   DISPLAY "NAME: ", NAME1-ARR, "SSS: ", SSS.
   DISPLAY " ".
   PERFORM Z200-PRINTCRIME.
   DISPLAY " ".
C320-EXIT.
  EXIT.
C300-ADD-CRIME SECTION.
* ADD-CRIME
   Obtains a sss and crime from the user and appends
   the crime to the list of crimes of the corresponding sss.
*******************
C300-DDCRM.
   EXEC SOL
     WHENEVER SQLERROR DO PERFORM Z900-SQLERROR
END-EXEC.
   PERFORM Z100-GET-SSS.
   IF (SSSEXISTS = 1)
     EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
     PERFORM C310-GETNEWCRIME
     EXEC SQL SELECT TXT SUMMARY INTO :LICENSE-TXT
         FROM LICENSE TABLE WHERE SSS = :SSS
         FOR UPDATE END-EXEC
     PERFORM C320-APPENDTOCLOB
     EXEC SQL FREE :LICENSE-TXT END-EXEC
     DISPLAY "SSS Number Not Found".
C300-EXIT.
   EXIT.
C400-NEW-RECORD SECTION.
* C400-NEW-RECORD
   Obtains the sss and name of a new record and inserts them
   along with an empty_clob() for the clob in the table.
******************
C400-NWRCRD.
   PERFORM Z100-GET-SSS.
   IF (SSSEXISTS = 1)
     DISPLAY "Record with that sss number already exists"
     DISPLAY "Name? " WITH NO ADVANCING
     ACCEPT NEWNAME
     DISPLAY " ".
     EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
     EXEC SQL INSERT INTO LICENSE TABLE
        VALUES (:SSS, :NEWNAME, EMPTY CLOB()) END-EXEC
     EXEC SQL SELECT TXT SUMMARY INTO :LICENSE-TXT
         FROM LICENSE TABLE WHERE SSS = :SSS END-EXEC
     "----"
     DISPLAY "NAME: ", NEWNAME, "SSS: ", SSS
     PERFORM Z200-PRINTCRIME
     "======""
```

```
EXEC SQL FREE :LICENSE-TXT END-EXEC.
C400-EXIT.
  EXIT.
D000-LOGOFF SECTION.
* D000-LOGOFF
  Commit the work done to the database and log off
DOOO-LGFF.
   EXEC SQL COMMIT WORK RELEASE END-EXEC.
   DISPLAY " ".
   DISPLAY "HAVE A GOOD DAY!".
   DISPLAY " ".
D000-EXIT.
   STOP RUN.
Z100-GET-SSS SECTION.
* Z100-GET-SSS
   Fills the global variable SSS with the client-supplied sss.
   Sets the global variable SSSEXISTS to 0 if the sss does not
   correspond to any entry in the database, else sets it to 1.
******************
Z100-GTSSS.
   DISPLAY "Social Security Number? " WITH NO ADVANCING.
   ACCEPT SSS.
   DISPLAY " ".
   EXEC SQL SELECT COUNT(*) INTO :SSSCOUNT FROM LICENSE TABLE
       WHERE SSS = :SSS END-EXEC.
   IF (SSSCOUNT = 0)
     MOVE 0 TO SSSEXISTS
     MOVE 1 TO SSSEXISTS.
Z100-EXIT.
  EXIT.
Z200-PRINTCRIME SECTION.
Z200-PRINTCRIME
   Obtains the length of the global clob LICENSE-TXT and
   uses that in the LOB READ statement to read the clob
   into a character buffer to display the contents of the clob.
   The caller to this function must allocate, select and later
  free the global clob LICENSE-TXT.
************
Z200-PRNTCRM.
   DISPLAY "========".
   DISPLAY " CRIME SHEET SUMMARY ".
   DISPLAY "=======".
   MOVE SPACE TO THE-STRING-ARR.
   EXEC SQL LOB DESCRIBE :LICENSE-TXT GET LENGTH
      INTO :TXT-LENGTH END-EXEC.
   IF (TXT-LENGTH = 0)
      DISPLAY "Record is clean"
    ELSE
      EXEC SQL LOB READ :TXT-LENGTH FROM :LICENSE-TXT
        INTO :THE-STRING END-EXEC
      DISPLAY THE-STRING-ARR.
```

```
Z200-EXIT.
   EXIT.
Z300-ACCEPT-CHOICE SECTION.
********************
  Z300-ACCEPT-CHOICE
    Accept a choice between 1 and 5
Z300-CCPT.
   DISPLAY "Your Selection (1-5)? " WITH NO ADVANCING.
   ACCEPT CHOICE.
   DISPLAY " ".
   IF CHOICE >5 OR CHOICE < 1 THEN
      DISPLAY "Invalid Selection"
      DISPLAY "Please Choose from the indicated list".
Z300-EXIT.
   EXIT.
Z900-SQLERROR SECTION.
* Z900-SQLERROR
   Called whenever a SQLERROR occurs.
  Display the Error, Roll Back any work done and Log Off
**********
Z900-SQLRRR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    DISPLAY " ".
   DISPLAY "ORACLE ERROR DETECTED:".
   DISPLAY " ".
   DISPLAY SQLERRMC.
   EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
Z900-EXIT.
   STOP RUN.
```



14

# **Precompiler Options**

This chapter describes the precompiler options of Pro\*COBOL. This chapter includes:

- The procob Command
- Actions During Precompilation
- About the Options
- Entering Precompiler Options
- Scope of Precompiler Options
- Quick Reference
- Using Pro\*COBOL Precompiler Options

# 14.1 The procob Command

The location of Pro\*COBOL differs from system to system. Typically, your system manager or DBA defines environment variables or aliases or uses other operating system-specific means to make the Pro\*COBOL executable accessible.

To run the Oracle Pro\*COBOL Precompiler, you issue the command

```
procob [option_name=value] [option_name=value] ...
```

The option value is always separated from the option name by an equals sign (=), with no white space around the equals sign.

For example, the INAME option specifies the source file to be precompiled. The command:

```
procob INAME=test
```

precompiles the file *test.pco* in the current directory, since Pro\*COBOL assumes that the filename extension is *.pco*.

You need not use a file extension when specifying INAME unless the extension is nonstandard.

Input and output filenames need not be accompanied by their respective option names, INAME and ONAME. When the option names are not specified, Pro\*COBOL assumes that the first filename specified on the command line is the input filename and that the second filename is the output filename.

### Thus, the command

```
procob MODE=ANSI myfile myfile.cob
```

### is equivalent to

procob MODE=ANSI INAME=myfile.pco ONAME=myfile.cob



## 14.1.1 Case-Sensitivity

In general, you can use either uppercase or lowercase for command-line option names and values. However, if your operating system is case-sensitive (as in UNIX for example) you must specify filename values, including the name of Pro\*COBOL executable, using the correct combination of upper and lowercase letters.

**Note:** Option names and option values that do not name specific operating system objects, such as filenames, are not case-sensitive. In the examples in this guide, option names are written in upper case or lower case, and option values are usually in lower case. Filenames, including the name of the Pro\*COBOL executable itself, always follow the case conventions used by the operating system on which it is executed.

With some operating systems and user shells, such as UNIX C shell, the ? may need to be preceded by an "escape" character, such as a back-slash (\). For example, instead of procob ?, you might need to use procob \? to list the Pro\*COBOL option settings.

Consult your platform-specific documentation.

# 14.2 Actions During Precompilation

During precompilation, Pro\*COBOL generates COBOL code that replaces the SQL statements embedded in your host program. The generated code includes data structures that contain the datatype, length, and address of each host variable, as well as other information required by the Oracle runtime library, SQLLIB. The generated code also contains the calls to SQLLIB routines that perform the embedded SQL operations.

Pro\*COBOL can issue warnings and error messages. These messages are described in *Oracle Database Error Messages*.

# 14.3 About the Options

Many useful options are available at precompile time. They let you control how resources are used, how errors are reported, how input and output are formatted, and how cursors are managed.

The *value* of an option is a literal, which represents text or numeric values. For example, for the option

```
... INAME=my test
```

the value is a string literal that specifies a filename.

For the option

```
... PREFETCH=100
```

the value is numeric.

Some options take Boolean values, which you can represent with the strings YES or NO, TRUE or FALSE, or with the integer literals 1 or 0, respectively. For example, the option



```
... SELECT_ERROR=YES
```

### is equivalent to

```
... SELECT_ERROR=TRUE
```

or

```
... SELECT ERROR=1
```

You leave no white space around the equals (=) sign. This is because spaces delimit individual options. For example, you might specify the option AUTO\_CONNECT on the command line as follows:

```
... AUTO CONNECT=YES
```

You can abbreviate the names of options unless the abbreviation is ambiguous. For example, you cannot use the abbreviation MAX because it might stand for MAXLITERAL or MAXOPENCURSORS.

A handy reference to the Pro\*COBOL options is available online. To see the online display, enter the Pro\*COBOL command, with no arguments, at your operating system prompt:

procob

The display gives the name, syntax, default value, and purpose of each option. Options marked with an asterisk (\*) can be specified inline as well as on the command line.

## 14.3.1 Precedence of Option Values

Option values are determined by the following, in order of increasing precedence:

- A default built in to Pro\*COBOL
- A value set in the system configuration file
- A value set in a user configuration file
- · A value entered in the command line
- A value set in an inline specification

For example, the option MAXOPENCURSORS specifies the maximum number of cached open cursors. The built-in Pro\*COBOL default value for this option is 10. However, if MAXOPENCURSORS=32 is specified in the system configuration file, the value becomes 32. The user configuration file could set it to yet another value, which then overrides the system configuration value.

If the MAXOPNCURSORS option is set on the command line, the new command-line value takes precedence. Finally, an inline specification takes precedence over all preceding defaults. For more information, see "Entering Precompiler Options".

### **Related Topics**

Entering Precompiler Options

## 14.3.2 Macro and Micro Options

Option MODE is known as a *macro* option. Some newer options, such as END\_OF\_FETCH, control only one function and are known as *micro* options. When setting a macro and a micro option, you must remember that the macro option has precedence over micro options. This is



the case if, and only if, the macro option is at a higher level of precedence than the micro option. (As described in the section "Precedence of Option Values".) This behavior is a change from releases of Pro\*COBOL prior to 8.0.

For example, the default for MODE is ORACLE, and for END\_OF\_FETCH is 1403. If you specify MODE=ANSI in the user configuration file, Pro\*COBOL will return a value of 100 at the end of fetch, overriding the default END\_OF\_FETCH value of 1403. If you specify both MODE=ANSI and END\_OF\_FETCH=1403 in the configuration file, then 1403 will be returned. If you specify END\_OF\_FETCH=1403 in your configuration file and MODE=ANSI on the command line, then 100 will be returned.

The following table lists the values of micro options set by the macro option values:

Table 14-1 How Macro Option Values Set Micro Option Values

Macro Option	Micro Option
MODE=ANSI   ISO	CLOSE_ON_COMMIT=YES
	DECLARE_SECTION=YES
	END_OF_FETCH=100
	DYNAMIC=ANSI
	TYPE_CODE=ANSI
MODE=ANSI14   ANSI13   ISO14   ISO13	CLOSE_ON_COMMIT=NO
	DECLARE_SECTION=YES
	END_OF_FETCH=100
MODE=ORACLE	CLOSE_ON_COMMIT=NO
	DECLARE_SECTION=NO
	END_OF_FETCH=1403
	DYNAMIC=ORACLE
	TYPE_CODE=ORACLE

### **Related Topics**

Precedence of Option Values

# 14.3.3 Determining Current Values

You can interactively determine the current value for one or more options by using a question mark on the command line. For example, if you issue the command

procob ?

the complete option set, along with current values, is displayed on your terminal. In this case, the values are those built into Pro\*COBOL, overridden by any values in the system configuration file. But if you issue the following command

```
procob CONFIG=my config file.cfg ?
```

and there is a file named  $my\_config\_file.cfg$  in the current directory, the options from the  $my\_config\_file.cfg$  file are listed with the other default values. Values in the user configuration file supply missing values, and they supersede values built into Pro\*COBOL and values specified in the system configuration file.

You can also determine the current value of a single option by simply specifying the option name followed by =? as in



procob MAXOPENCURSORS=?

# 14.4 Entering Precompiler Options

All Pro\*COBOL options (except CONFIG) can be entered on the command line or from a configuration file. Many options can also be entered inline. During a given run, Pro\*COBOL can accept options from all three sources.

### 14.4.1 On the Command Line

```
You enter precompiler options on the command line using ... [option_name=value] [option name=value] ...
```

Separate each option with one or more spaces. For example, you might enter the following options:

```
... ERRORS=no LTYPE=short
```

### 14.4.2 Inline

Enter options inline by coding EXEC ORACLE OPTION statements, using the following syntax:

```
EXEC ORACLE OPTION (option name=value) END-EXEC.
```

For example, you might code the following statement:

```
EXEC ORACLE OPTION (RELEASE CURSOR=YES) END-EXEC.
```

An option entered inline overrides the same option entered on the command line.

## 14.4.2.1 Advantages

The EXEC ORACLE feature is especially useful for changing option values during precompilation. For example, you might want to change the HOLD\_CURSOR and RELEASE\_CURSOR values on a statement-by-statement basis. "Performance Tuning" shows you how to use inline options to optimize runtime performance.

Specifying options inline is also helpful if your operating system limits the number of characters you can enter on the command line, and you can store inline options in configuration files. These are discussed in the next section.

### **Related Topics**

Performance Tuning

## 14.4.2.2 Scope of EXEC ORACLE

An EXEC ORACLE statement stays in effect until textually superseded by another EXEC ORACLE statement specifying the same option. In the following example, HOLD\_CURSOR=NO stays in effect until superseded by HOLD\_CURSOR=YES:

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

O1 EMP-NAME PIC X(20) VARYING.

O1 EMP-NUMBER PIC S9(4) COMP VALUE ZERO.

O1 SALARY PIC S9(5)V99 COMP-3 VALUE ZERO.
```



```
01 DEPT-NUMBER PIC S9(4) COMP VALUE ZERO.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL WHENEVER NOT FOUND GOTO NO-MORE END-EXEC.
    EXEC ORACLE OPTION (HOLD CURSOR=NO) END-EXEC.
    EXEC SQL DECLARE emp cursor CURSOR FOR
       SELECT EMPNO, DEPTNO FROM EMP
    END-EXEC.
    EXEC SQL OPEN emp_cursor END-EXEC.
    DISPLAY 'Employee Number Dept'.
    DISPLAY '----'.
    PERFORM
        EXEC SQL
         FETCH emp cursor INTO : EMP-NUMBER, : DEPT-NUMBER
       END-EXEC
        DISPLAY EMP-NUMBER, DEPT-NUMBER END-EXEC
    END-PERFORM.
NO-MORE.
        EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
    PERFORM
       DISPLAY 'Employee number? '
       ACCEPT EMP-NUMBER
        IF EMP-NUMBER IS NOT = 0
           EXEC ORACLE OPTION (HOLD CURSOR=YES) END-EXEC
            EXEC SQL SELECT ENAME, SAL
               INTO :EMP-NAME, :SALARY
               FROM EMP
               WHERE EMPNO = :EMP-NUMBER
               DISPLAY 'Salary for ', EMP-NAME, ' is ', SALARY
           END-EXEC
        END-IF
    END-PERFORM.
 NEXT-PARA.
```

## 14.4.3 Configuration Files

A configuration file is a text file that contains precompiler options. Each record (line) in the file contains one option, with its associated value or values. For example, a configuration file might contain the lines

```
FIPS=YES
MODE=ANSI
```

. . .

to set values for the FIPS and MODE options.

There is a single system configuration file for each system. The name of the system configuration file is

```
pcbcfg.cfg
```

The location of the file is operating system-specific. On most UNIX systems, the Pro\*COBOL configuration file is usually located in the *\$ORACLE\_HOME/precomp/ admin* directory, where *\$ORACLE\_HOME* is the environment variable for the database software.



Note that before release 8.0 of Pro\*COBOL, the configuration file was called *pccob.cfg*.

The Pro\*COBOL user can have one or more user configuration files. The name of the configuration file must be specified using the CONFIG command-line option. For more information, see "Determining Current Values".

**Note:** You cannot nest configuration files. This means that CONFIG is not a valid option inside a configuration file.

### **Related Topics**

Determining Current Values

## 14.4.4 Column Properties Support

Column properties are returned in an 8-byte value where each bit indicates a column property. Three column properties are supported:

You can get the column properties through dynamic statements using the new SQLDA member (sqlda->CP[]).

```
01 SELDSC.

02 SQLDNUM PIC S9(9) COMP VALUE 20.

02 SQLDFND PIC S9(9) COMP.

02 SELDVAR OCCURS 20 TIMES.

03 SELDV PIC S9(9) COMP.

03 SELDFMT PIC S9(9) COMP.

03 SELDVIN PIC S9(9) COMP.

03 SELDVIN PIC S9(9) COMP.

03 SELDFMTL PIC S9(4) COMP.

03 SELDVTYP PIC S9(4) COMP.

03 SELDH-VNAME PIC S9(9) COMP.

03 SELDH-VNAME PIC S9(9) COMP.

03 SELDH-WAX-VNAMEL PIC S9(4) COMP.

03 SELDH-CUR-VNAMEL PIC S9(4) COMP.

03 SELDI-VNAME PIC S9(4) COMP.

03 SELDI-CUR-VNAMEL PIC S9(4) COMP.

03 SELDI-CUR-VNAMEL PIC S9(4) COMP.

03 SELDFCLP PIC S9(9) COMP.

03 SELDFCLP PIC S9(9) COMP.

03 SELDFCCP PIC S9(9) COMP.

03 SELDFCCP PIC S9(9) COMP.
```

This member SQLDSCCP is updated as part of a metadata DESCRIBE.



### Note:

If the target table has more than 20 columns, you must change the SQLDSCCP OCCURS count value. For example, if the target table has 40 columns, you must enter:

```
02 SQLDSCCP OCCURS 40 TIMES PIC S9(18) COMP-5.
```

You can get the column properties through static statements using the new function <code>SOLGetColProp()</code>, which gets the column properties from the last executed statement.

SQLGetColProp() returns values determined by the column attribute colatr.

- SQL\_ATTR\_COL\_PROPERTIES: Returns an 8-byte value (colprop) containing the column properties of the named column
- SQL\_ATTR\_COL\_PROPERTY\_IS\_IDENTITY colprop is true if the named column is an identity column
- SQL\_ATTR\_COL\_PROPERTY\_IS\_GEN\_ALWAYS colprop is true if the named column always generates an auto increment value
- SQL\_ATTR\_COL\_PROPERTY\_IS\_GEN\_BY\_DEF\_ON\_NULL colprop is true if the named column generates an auto increment value in the case of a default-null column constraint.
- SQL\_ATTR\_COL\_PROPERTY\_HAS\_DOMAIN returns colprop is true if the named column has a DOMAIN constraint.

# 14.5 About Usage Domain Metadata Support

A usage domain (also called SQL domain or domain) is a database object that maintains properties and annotations associated with an existing Oracle data type. Using SQL domains, application developers can indicate intended usage of a column of a given type. Oracle Database enables applications to create a SQL domain and associate it with columns.

### See Also:

- Application Usage Domains in Oracle Database Concepts
- Application Usage Domains in Oracle Database Development Guide



Precompilers can use predefined functions to obtain domain metadata or annotation information from columns associated with a domain. Applications with references to the DOMAIN syntax must be precompiled using the common front-end parser, for example:

```
procob common_parser=yes userid=scott/tiger sample10.pco
```

The following API function gets the domain name associated with a column (in the domainName argument), and returns the length of the domain (in the domainLen argument).

```
void SQLGetDomainName(
  void *uga,
  text *coln,
  text *domainName
  ub4 *domainLen);
```

#### Parameters:

- uga (IN) runtime context
- coln (IN) column name
- domainName (OUT) domain name
- domainLen (OUT) length of the domain name

#### The following is an example:

```
EXEC SQL SELECT empId, empEmail FROM empTable END-EXEC.
Move "empEMail" to colName.
Move SQL ATTR COL PROPERTY HAS DOMAIN to cp
CALL "SQLCOLPROP" USING
                 NULL
                 ColName
                 COL-PROP-ATTR
                 COL-PROP.
CALL "SQLCOLHASDOMAIN" USING
                 NULL
                 COL-PROP
                 HAS-DOMAIN.
If hasDomain = 1
  CALL "SQLDOMAINNAME" USING
                NULL
                colName
                domainName
                domainLen.
DISPLAY "Column ", colName, " has domain Name"
         domainName(1:domainLen).
```

```
Note:
```

The CAST AS DOMAIN feature is not supported. For example:

```
EXEC SQL SELECT First_Name, CAST (Score AS domain domain_1) Int_Score into :nfname, :nscore FROM Student Score END-EXEC.
```

#### returns the following error:

ORA-11537 CAST AS DOMAIN feature is not supported.



# 14.6 Scope of Precompiler Options

A precompilation unit is a file containing COBOL code and one or more embedded SQL statements. The options specified for a given precompilation unit affect only that unit; they have no effect on other units.

For example, if you specify HOLD\_CURSOR=YES and RELEASE\_CURSOR=YES for unit A but not unit B, SQL statements in unit A run with these HOLD\_CURSOR and RELEASE\_CURSOR values, but SQL statements in unit B run with the default values. However, the MAXOPENCURSORS setting that is in effect when you connect to Oracle stays in effect for the life of that connection.

The scope of an inline option is positional, not logical. That is, an inline option affects SQL statements that follow it in the source file, not in the flow of program logic. An option setting stays in effect until the end-of-file unless you re-specify the option.

# 14.7 Quick Reference

The following table is a quick reference to the Pro\*COBOL options. Options marked with an asterisk can be entered inline.

Another handy reference is available online. To see the online display, just enter the Pro\*COBOL command without options at your operating system prompt. The display gives the name, syntax, default value, and purpose of each option.



There are some platform-specific options. For example, on byte-swapped platforms the option COMP5 governs the use of certain COMPUTATIONAL items. Check your system-specific Oracle manuals.

COMP5 is not supported on SPARC Solaris 64 bit Computers. Use COMP instead.

Added CWH\_SQL99 option for feature 25546?

Table 14-2 Option List

Syntax	Default	Specifies
ASACC={YES   NO}	NO	If YES, use ASA carriage control for listing.
ASSUME_SQLCODE={YES   NO}	NO	If YES, assume SQLCODE variable exists.
AUTO_CONNECT={YES   NO}	NO	If YES, allow automatic connect to ops\$ accounts before the first executable statement.
CLOSE_ON_COMMIT*	NO	If YES, close all cursors on COMMIT.
CONFIG=filename	(none)	Specifies name of user-defined configuration file.



Table 14-2 (Cont.) Option List

Syntax	Default	Specifies
CWH_SQL99={YES   NO}	YES	If YES, close held cursors on ROLLBACK (SQL standard behavior.)
DATE_FORMAT	LOCAL	Specifies date string format.
DBMS={NATIVE   V7   V8}	NATIVE	Version-specific behavior of Oracle at precompile time.
DECLARE_SECTION	NO	If YES, DECLARE SECTION is required.
DEFINE=symbol*	(none)	Define a symbol used in conditional precompilation.
DYNAMIC	ORACLE	Specifies Oracle or ANSI dynamic semantics in SQL Method 4.
END_OF_FETCH	1403	End-of-fetch SQLCODE value.
ERRORS={YES   NO} *	YES	If YES, display errors on the terminal.
FIPS={YES   NO}	NO	If YES, ANSI/ISO extensions are flagged.
FORMAT={ANSI   TERMINAL   VARIABLE}	ANSI	Format of input file COBOL statements.
HOLD_CURSOR={YES   NO}*	NO	If YES, hold OraCursor (do not reassign).
HOST={COBOL   COB74}	COBOL	COBOL version used in input file (COBOL 85 or COBOL 74).
[INAME=]filename	(none)	Name of input file.
INCLUDE=path*	(none)	Pathname for EXEC SQL INCLUDE files.
IRECLEN=integer	80	Record length of input file.
LITDELIM={APOST   QUOTE}	QUOTE	Delimiters for COBOL strings.
LNAME=filename	(none)	Name of listing file.
LRECLEN=integer	132	Record length of listing file.
LTYPE={LONG   SHORT   NONE} *	LONG	Type of listing.
MAXLITERAL=integer*	256	Maximum length of strings. (120 on IBM-proprietary S370 OS)
MAXOPENCURSORS=integer*	10	Maximum number of OraCursors cached (1).
MODE={ORACLE   ANSI}	ORACLE	If ANSI, follow the ANSI/ISO SQL standard.
NESTED={YES   NO}	YES	If YES, nested programs are supported.
NLS_LOCAL={YES   NO}	NO	If YES, use NCHAR semantics of previous Pro*COBOL releases.
[ONAME=]filename	iname.cob	Name of output file.
ORACA={YES   NO}*	NO	If YES, use ORACA communications area.
ORECLEN=integer	80	Record length of output file.



Table 14-2 (Cont.) Option List

Syntax	Default	Specifies
PAGELEN=integer	66	Lines for each page in listing.
PICX	CHARF	Datatype of PIC X COBOL variables.
PLAN_BASELINE={module_name   YES   NO}	NO	Specify a module name to create a SQL Plan Baseline.
PLAN_PREFIX={prefix_name  none}	none	Ensures that the plan name does not exceed 128 bytes.
PLAN_RUN={YES   NO}	NO	Executes the generated SQL file
PLAN_FIXED={YES   NO}	YES	Specifies whether or not the created plan baseline is fixed or non-fixed
PLAN_ENABLED={YES   NO}	YES	Enables the Plan Baseline that is created.
MEMFORPREFETCH=0429496729 4	none	Speed up queries by pre-fetching rows that fill the specified memory.
PREFETCH	1	Speed up queries by pre-fetching a given number of rows.
RELEASE_CURSOR={YES   NO} *	NO	If YES, release OraCursor after execute.
SELECT_ERROR={YES   NO}*	YES	If YES, generate FOUND error on SELECT.
SQLCHECK={SEMANTICS   SYNTAX}*	SYNTAX	SQL checking level.
THREADS={YES   NO}	NO	Indicates a multithreaded application.
TYPE_CODE	ORACLE	Use Oracle or ANSI type codes for dynamic SQL method 4.
UNSAFE_NULL={YES   NO}	NO	If YES, unsafe null fetches are allowed (disables the ORA-01405 message).
USERID=username/ password[@dbname]	(none)	Oracle username, password, and optional database.
VARCHAR={YES   NO}	NO	If YES, accept user-defined VARCHAR group items.
XREF={YES   NO}*	YES	If YES, generate symbol cross references in listing.

# 14.8 Using Pro\*COBOL Precompiler Options

This section is organized for easy reference. It lists the Pro\*COBOL precompiler options alphabetically and for each option gives its purpose, syntax, and default value. Usage notes that help you understand how the option works are also provided. Unless the usage notes say otherwise, the option can be entered on the command line, inline, or from a configuration file.

### 14.8.1 ASACC

#### **Purpose**

Specifies whether the listing file follows the ASA convention of using the first column in each line for carriage control.

#### **Syntax**

ASACC={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Cannot be entered inline.

## 14.8.2 ASSUME SQLCODE

#### **Purpose**

Instructs Pro\*COBOL to presume that SQLCODE is declared whether or not it is declared in the program, or of the proper type.

#### **Syntax**

ASSUME\_SQLCODE={YES | NO}

#### Default

NO

#### **Usage Notes**

Cannot be entered inline.

When DECLARE\_SECTION=YES and ASSUME\_SQLCODE=YES, SQLCODE can be declared outside a Declare Section.

When DECLARE\_SECTION=YES and ASSUME\_SQLCODE=NO, SQLCODE is recognized as the status variable if and only if at least one of the following criteria is satisfied:

- It is declared with *exactly* the right datatype.
- Pro\*COBOL finds no other status variable. If Pro\*COBOL finds a SQLSTATE declaration (of exactly the right type of course), or finds an include of a SQLCA, then it will not presume SQLCODE is declared.

When ASSUME\_SQLCODE=YES, and when SQLSTATE and/or SQLCA are declared as status variables, Pro\*COBOL presumes SQLCODE is declared whether or not it is declared or of the proper type.



## 14.8.3 AUTO\_CONNECT

#### **Purpose**

Specifies whether your program connects automatically to the default user account.

#### **Syntax**

AUTO\_CONNECT={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Cannot be entered inline.

When AUTO\_CONNECT=YES, as soon as Pro\*COBOL encounters an executable SQL statement, your program tries to log on to Oracle automatically with the userid

where refix> is the value of the Oracle initialization parameter
OS\_AUTHENT\_PREFIX (the default value is OPS\$) and <username> is your
operating system user or task name. In this case, you cannot override the default
value for MAXOPENCURORS (10), even if you specify a different value on the
command line.

When AUTO\_CONNECT=NO (the default), you must use the CONNECT statement to logon to Oracle.

## 14.8.4 CHARSET PICX

#### **Purpose**

Specifies the character set form used by PIC X variables used in select, insert, or update statements.

#### **Syntax**

CHARSET\_PICX={NCHAR\_CHARSET | DB\_CHARSET }

#### **Default**

DB\_CHARSET

#### **Usage Notes**

Can be used on the command line or in a configuration file, but not inline.

If CHARSET\_PICX = NCHAR\_CHARSET, the PIC X bind or define buffer is converted according to the server-side National Character Set. Possible performance impact might be seen when target column is CHAR. Similarly, if CHARSET\_PICX = DB\_CHARSET, the PIC X bind or define buffer is converted according to server-side Database Character Set. There can be some data loss when target column is NCHAR.



## 14.8.5 CHARSET\_PICN

#### **Purpose**

Specifies the character set form used by PIC N variables used in select, insert, or update statements.

#### **Syntax**

CHARSET\_PICN={NCHAR\_CHARSET | DB\_CHARSET }

#### **Default**

NCHAR\_CHARSET

#### **Usage Notes**

Can be used on the command line or in a configuration file, but not inline.

If CHARSET\_PICN = DB\_CHARSET, the PIC N bind or define buffer is converted according to the server-side Database Character Set. There can be some data loss when target column is NCHAR. Similarly, if CHARSET\_PICN = NCHAR\_CHARSET, the PIC N bind or define buffer is converted according to server-side National Character Set. Possible performance impact might be seen when target column is CHAR.

## 14.8.6 CLOSE\_ON\_COMMIT

#### **Purpose**

Specifies whether or not all cursors declared without the WITH HOLD clause are closed on commit.

#### **Syntax**

CLOSE\_ON\_COMMIT={YES | NO}

#### Default

NO

#### **Usage Notes**

Can be used only on the command line or in a configuration file.

This option will only have an effect when a cursor is not coded using the WITH HOLD clause in a DECLARE CURSOR statement, since that will override both the new option and the existing behavior which is associated with the MODE option. If MODE is specified at a higher level than CLOSE\_ON\_COMMIT, then MODE takes precedence. For example, the defaults are MODE=ORACLE and CLOSE\_ON\_COMMIT=NO. If the user specifies MODE=ANSI on the command line, then any cursors not using the WITH HOLD clause will be closed on commit.

When CLOSE\_ON\_COMMIT=NO (when MODE=ORACLE), issuing a COMMIT or ROLLBACK will close only cursors that are declared using the FOR UPDATE clause or are referenced in a CURRENT OF clause. Other cursors are not affected by the COMMIT or ROLLBACK statement, remain open, if they are open already. However, when



CLOSE\_ON\_COMMIT=YES (when MODE=ANSI), issuing a COMMIT or ROLLBACK closes all cursors.

For a further discussion of the precedence of this option see "Macro and Micro Options".

#### **Related Topics**

Macro and Micro Options

# 14.8.7 COMMON\_PARSER

#### **Purpose**

Specifies that the SQL99 syntax for SELECT, INSERT, DELETE, UPDATE and body of the cursor in a DECLARE CURSOR statement will be supported.

#### **Syntax**

COMMON\_PARSER={YES | NO}

**Default** 

NO

#### **Usage Notes**

Can be entered in the command line.

### 14.8.8 CONFIG

#### **Purpose**

Specifies the name of a user configuration file.

#### **Syntax**

CONFIG=filename

#### **Default**

None

#### **Usage Notes**

Can be entered only on the command line.

Pro\*COBOL can use a configuration file containing preset command-line options. However, you can specify any of several alternative files, called *user configuration files*. For more information, see "Entering Precompiler Options".

You cannot nest configuration files. Therefore, you cannot specify the option CONFIG in a configuration file.

#### **Related Topics**

Entering Precompiler Options



# 14.8.9 DATE\_FORMAT

#### **Purpose**

Species the string format in which dates are returned.

#### **Syntax**

DATE\_FORMAT={ISO | USA | EUR | JIS | LOCAL | 'fmt' (default LOCAL)

#### **Default**

**LOCAL** 

#### **Usage Notes**

Can only be entered on the command line or in a configuration file. The date strings are shown in the following table:

Table 14-3 Formats for Date Strings

Format Name	Abbreviation	Date Format
International Standards Organization	ISO	yyyy-mm-dd
USA standard	USA	mm/dd/yyyy
European standard	EUR	dd.mm.yyyy
Japanese Industrial Standard	JIS	yyyy-mm-dd
installation-defined	LOCAL	Any installation-defined form.

'fmt' is a date format model, such as "Month dd, yyyy". See "Datetime Format Elements" for the list of date format model elements.

There is one restriction on the use of the DATE\_FORMAT option: All compilation units to be linked together must use the same DATE\_FORMAT value. An error occurs when there is a mismatch in the values of DATE\_FORMAT across compilation units

## 14.8.10 DB2\_ARRAY

#### **Purpose**

Based on this option, the precompiler activates the additional array insert/select syntax.

#### **Syntax**

DB2 ARRAY={YES INO}

#### **Default**

NO

#### **Usage Notes**

If this option is set to NO, the Oracle precompiler syntax is supported, otherwise the DB2 insert/select array syntax is supported.



### 14.8.11 DBMS

#### **Purpose**

Specifies whether Oracle follows the semantic and syntactic rules of Oracle7 Database, Oracle8*i*, or the native version of Oracle (that is, the version to which your application is connected).

#### **Syntax**

DBMS={V7 | V8 | NATIVE}

#### Default

**NATIVE** 

#### **Usage Notes**

Cannot be entered inline.

With the DBMS option you control the version-specific behavior of Oracle. When DBMS=NATIVE (the default), Oracle follows the semantic and syntactic rules of the native version of Oracle.

## 14.8.12 DECLARE\_SECTION

#### **Purpose**

Specifies whether or not *only* declarations in a Declare Section are allowed as host variables.

#### **Syntax**

DECLARE\_SECTION={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Can be entered only on the command line or in a configuration file.

When MODE=ORACLE, use of the BEGIN DECLARE SECTION and END DECLARE SECTION statements are optional, starting with release 8.0 of Pro\*COBOL. The DECLARE\_SECTION option is provided for backward compatibility with previous releases. DECLARE\_SECTION is a micro option of MODE.

This option allows the user to specify MODE=ORACLE together with DECLARE\_SECTION=YES to get the same effect that previous releases provided when using MODE=ORACLE alone. (Only variables declared inside the DECLARE statements are allowed as host variables.) For a discussion of precedence of this option, see "Precedence of Option Values".

#### **Related Topics**

Precedence of Option Values



### 14.8.13 DEFINE

#### **Purpose**

Specifies a user-defined symbol that is used to include or exclude portions of source code during a conditional precompilation. For more information, see "Conditional Precompilations"

#### **Syntax**

DEFINE=symbol

#### **Default**

None

#### **Usage Notes**

If you enter DEFINE inline, the EXEC ORACLE statement takes the following form:

EXEC ORACLE DEFINE symbol END-EXEC.

#### **Related Topics**

Conditional Precompilations

### 14.8.14 DYNAMIC

#### **Purpose**

This micro option of MODE specifies the descriptor behavior in dynamic SQL Method 4.

#### **Syntax**

DYNAMIC={ORACLE | ANSI}

#### **Default**

**ORACLE** 

#### **Usage Notes**

Cannot be entered inline by use of the EXEC ORACLE OPTION statement.

See the DYNAMIC option settings in "ANSI Dynamic SQL Precompiler Options".

#### **Related Topics**

ANSI Dynamic SQL Precompiler Options

# 14.8.15 END\_OF\_FETCH

#### **Purpose**

This micro option of MODE specifies which SQLCODE value is returned when an END-OF-FETCH condition occurs after execution of a SQL statement.



#### **Syntax**

END\_OF\_FETCH={100 | 1403}

#### **Default**

1403

#### **Usage Notes**

Can be entered only on the command line or in a configuration file.

END\_OF\_FETCH is a micro option of MODE. For further discussion, see "Macro and Micro Options".

If you specify MODE=ANSI in a configuration file, Pro\*COBOL returns the SQLCODE value 100 at the END\_OF\_FETCH, overriding the default END\_OF\_FETCH=1403.

If you specify MODE=ANSI and END\_OF\_FETCH=1403 in the configuration file, then Pro\*COBOL will return the SQLCODE value 1403 at the END\_OF\_FETCH.

If you specify MODE=ANSI in the configuration file and END\_OF\_FETCH=1403 on the command line, which has a higher precedence than your configuration file, Pro\*COBOL will again return the SQLCODE value 1403 at the END\_OF\_FETCH.

#### **Related Topics**

Macro and Micro Options

## 14.8.16 ERRORS

#### **Purpose**

Specifies whether Pro\*COBOL error messages are sent to the terminal and listing file or only to the listing file.

#### **Syntax**

ERRORS={YES | NO}

#### Default

YES

#### **Usage Notes**

When ERRORS=YES, error messages are sent to the terminal and listing file.

When ERRORS=NO, error messages are sent only to the listing file.

## 14.8.17 File ID

#### **Purpose**

Denotes the unique identifier for the generated Cobol file. The generated files contain a unique number associated with the "SQLCTX" variable. The file\_id option can be used to set the value of the SQLCTX variable. The option is useful while precompiling



multiple source files, so that the user can ensure unique SQLCTX values associated with the different generated files.

#### **Syntax**

FILE\_ID=Range is 0 to 65535

#### **Default**

0

#### **Usage Notes**

The file\_id option can be used to directly assign values to "SQLCTX" variable in the generated Cobol file.

### 14.8.18 FIPS

#### **Purpose**

Specifies whether extensions to ANSI/ISO SQL are flagged (by the FIPS Flagger). An extension is any SQL element that violates ANSI/ISO format or syntax rules, except privilege enforcement rules.

#### **Syntax**

FIPS={YES | NO}

#### Default

NO

#### **Usage Notes**

When FIPS=YES, the FIPS Flagger issues warning (not error) messages if you use an Oracle extension to the Entry level of SQL-92 or use a feature of Entry level SQL-92 in a nonconforming manner.

The following extensions to ANSI/ISO SQL are flagged at precompile time:

- Array interface including the FOR clause
- SQLCA, ORACA, and SQLDA data structures
- Dynamic SQL including the DESCRIBE statement
- Embedded PL/SQL blocks
- Automatic datatype conversion
- DATE, COMP-3, NUMBER, RAW, LONG RAW, VARRAW, ROWID, and VARCHAR datatypes
- ORACLE OPTION statement for specifying runtime options
- EXEC TOOLS statements in user exits
- CONNECT statement
- TYPE and VAR datatype equivalencing statements
- AT db\_name clause



- DECLARE...DATABASE, ...STATEMENT, and ...TABLE statements
- SQLWARNING condition in WHENEVER statement
- DO and STOP actions in WHENEVER statement
- COMMENT and FORCE TRANSACTION clauses in COMMIT statement
- FORCE TRANSACTION and TO SAVEPOINT clauses in ROLLBACK statement
- RELEASE parameter in COMMIT and ROLLBACK statements
- Optional colon-prefixing of WHENEVER...DO labels and of host variables in the INTO clause

### 14.8.19 FORMAT

#### **Purpose**

Specifies the format of COBOL statements.

#### **Syntax**

FORMAT={ANSI | TERMINAL | VARIABLE}

#### **Default**

**ANSI** 

#### **Usage Notes**

Cannot be entered inline.

The format of input lines is system-dependent. Check your system-specific Oracle manuals, or your COBOL compiler.

When FORMAT=ANSI, the format of input lines conforms as much as possible to the current ANSI standard for COBOL. When FORMAT=TERMINAL, input lines can start in column 1. Example code in this book is in TERMINAL format. Use FORMAT=VARIABLE to allow Flexible B Area Length. See "Coding Areas" for a more complete description.

#### **Related Topics**

Coding Areas

# 14.8.20 HOLD\_CURSOR

#### **Purpose**

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

#### **Syntax**

HOLD\_CURSOR={YES | NO}

#### **Default**

NO



#### **Usage Notes**

You can use HOLD\_CURSOR to improve the performance of your program. For more information, see "Performance Tuning".

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. HOLD\_CURSOR controls what happens to the link between the cursor and cursor cache.

When HOLD\_CURSOR=NO, after Oracle executes the SQL statement and the cursor is closed, Pro\*COBOL marks the link as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks.

When HOLD\_CURSOR=YES, the link is maintained; Pro\*COBOL does not reuse it. This is useful for SQL statements that are executed often because it speeds up subsequent executions and there is no need to re-parse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set HOLD\_CURSOR before executing the SQL statement. For inline use with explicit cursors, set HOLD\_CURSOR before opening the cursor.

For information showing how the HOLD\_CURSOR and RELEASE\_CURSOR options interact, see "Performance Tuning".

#### **Related Topics**

Performance Tuning

### 14.8.21 HOST

#### **Purpose**

Specifies the host language to be used.

#### **Syntax**

HOST={COB74 | COBOL}

#### **Default**

COBOL

### **Usage Notes**

Cannot be entered inline.

COB74 refers to the 1974 version of ANSI-approved COBOL. COBOL refers to the 1985 version. Other values might be available on your platform.

## 14.8.22 IMPLICIT SVPT

#### **Purpose**

Controls whether an implicit savepoint is taken prior to the start of a new batched insert.



#### **Syntax**

implicit\_svpt={YES|NO}

#### **Default**

NO

#### **Usage Notes**

If yes, a savepoint is taken prior to the start of a new batch of rows. If an error occurs on the insert, an implicit "rollback to savepoint" is executed. This option exists for DB2 compatibility, the obvious downside being the extra round-trip.

If no, there is no implicit savepoint taken. If an error occurs on the buffered insert, then it is reported back to the application, but no rollback is executed.

### 14.8.23 INAME

#### **Purpose**

Specifies the name of the input file.

#### **Syntax**

INAME=filename

#### Default

None

#### **Usage Notes**

Cannot be entered inline.

All input file names must be unique at precompilation time.

When specifying the name of your input file on the command line, the keyword INAME is optional. For example, in Pro\*COBOL, you can specify *myprog.pco* instead of INAME=*myprog.pco*.

You need not use a file extension when specifying INAME unless the extension is nonstandard. On the UNIX platform, Pro\*COBOL assumes the default input file extension *pco*.

## 14.8.24 INCLUDE

#### **Purpose**

Specifies a directory path for EXEC SQL INCLUDE files. It only applies to operating systems that use directories.

#### **Syntax**

INCLUDE=path



#### **Default**

**Current directory** 

#### **Usage Notes**

Pro\*COBOL searches first in the current directory, then in the directory specified by INCLUDE, and finally in a directory for standard INCLUDE files. Hence, you need not specify a directory path for standard files.

You must still use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory. You can specify more than one path on the command line, as follows:

```
... INCLUDE=path1 INCLUDE=path2 ...
```

Pro\*COBOL searches first in the current directory, then in the directory named by *path1*, then in the directory named by *path2*, and finally in the directory for standard INCLUDE files.

**Note:** Pro\*COBOL looks for a file in the current directory first—even if you specify a directory path. Therefore, if the file you want to INCLUDE resides in another directory, make sure no file with the same name resides in the current directory.

The syntax for specifying a directory path is system-specific. Follow the conventions of your operating system.

### 14.8.25 IRECLEN

#### **Purpose**

Specifies the record length of the input file.

#### **Syntax**

IRECLEN=integer

#### Default

80

#### **Usage Notes**

Cannot be entered inline.

The value you specify for IRECLEN should not exceed the value of ORECLEN. The maximum value allowed is system-dependent.

## 14.8.26 LITDELIM

#### **Purpose**

The LITDELIM option specifies the delimiters for string constants and literals in the COBOL code generated by Pro\*COBOL.

#### **Syntax**

LITDELIM={APOST | QUOTE}



#### **Default**

QUOTE

#### **Usage Notes**

When LITDELIM=APOST, Pro\*COBOL uses apostrophes when generating COBOL code. If you specify LITDELIM=QUOTE, quotation marks are used, as in

```
CALL "SQLROL" USING SQL-TMP0.
```

In SQL statements, you must use quotation marks to delimit identifiers containing special or lowercase characters, as in

```
EXEC SQL CREATE TABLE "Emp2" END-EXEC.
```

but you must use apostrophes to delimit string constants, as in

```
EXEC SQL SELECT ENAME FROM EMP WHERE JOB = 'CLERK' END-EXEC.
```

Regardless of which delimiters are used in the Pro\*COBOL source file, Pro\*COBOL generates the delimiters specified by the LITDELIM value.

### 14.8.27 LNAME

#### **Purpose**

Specifies a nondefault name for the listing file.

#### **Syntax**

LNAME=filename

#### **Default**

Input

#### **Usage Notes**

Cannot be entered inline.

By default, the listing file is written to the current directory.

## 14.8.28 LRECLEN

#### **Purpose**

Specifies the record length of the listing file.

#### **Syntax**

LRECLEN=integer

#### **Default**

132



#### **Usage Notes**

Cannot be entered inline.

The value of LRECLEN can range from 80 through 132. If you specify a value below the range, 80 is used instead. If you specify a value above the range, an error occurs. LRECLEN should exceed IRECLEN by at least 8 to allow for the insertion of line numbers.

### 14.8.29 LTYPE

#### **Purpose**

Specifies the listing type.

#### **Syntax**

LTYPE={LONG | SHORT | NONE}

#### **Default**

LONG

#### **Usage Notes**

Cannot be entered inline.

**Table 14-4** Types of Listings

Listing Types	Description
LTYPE=LONG	input lines appear in the listing file.
LTYPE=SHORT	input lines do not appear in the listing file.
LTYPE=NONE	no listing file is created.

## 14.8.30 MAX\_ROW\_INSERT

#### **Purpose**

Controls the number of rows that need to be buffered before executing the INSERT statement.

#### **Syntax**

max\_row\_insert={0...1000}

#### **Default**

0

#### **Usage Notes**

Any number greater than zero enables buffered insert feature and buffers that many rows before executing the INSERT statement.



### 14.8.31 MAXLITERAL

#### **Purpose**

Specifies the maximum length of string literals generated by Pro\*COBOL so that compiler limits are not exceeded. For example, if your compiler cannot handle string literals longer than 132 characters, you can specify MAXLITERAL=132 on the command line.

#### **Syntax**

MAXLITERAL=integer

#### **Default**

1024

#### **Usage Notes**

The maximum value of MAXLITERAL is compiler-dependent. The default value is language-dependent, but you may have to specify a lower value. For example, some COBOL compilers cannot handle string literals longer than 132 characters, so you would specify MAXLITERAL=132.

Strings that exceed the length specified by MAXLITERAL are divided during precompilation, then recombined (concatenated) at run time.

You can enter MAXLITERAL inline but your program can set its value just once, and the EXEC ORACLE statement must precede the first EXEC SQL statement. Otherwise, Pro\*COBOL issues a warning message, ignores the extra or misplaced EXEC ORACLE statement, and continues processing.

### 14.8.32 MAXOPENCURSORS

#### **Purpose**

Specifies the number of concurrently open cursors that Pro\*COBOL tries to keep cached.

#### **Syntax**

MAXOPENCURSORS=integer

#### Default

10

#### **Usage Notes**

You can use MAXOPENCURSORS to improve the performance of your program. For more information, see "Performance Tuning".

When precompiling separately, use MAXOPENCURSORS as described in "Separate Precompilations".

MAXOPENCURSORS specifies the *initial* size of the SQLLIB cursor cache.



When an implicit statement is executed and HOLD\_CURSOR=NO, or an explicit cursor is closed, the cursor entry is marked as reusable. If this statement is issued again and the cursor entry has not been used for another statement, it is reused.

If a new cursor is needed and the number of cursors allocated is less than MAXOPENCURSORS, then the next one in the cache is allocated. Once MAXOPENCURSORS has been exceeded, Oracle first tries to reuse a previous entry. If there are no free entries, then an additional cache entry is allocated. Oracle continues to do this until the program runs out of memory or the database parameter OPEN\_CURSORS is exceeded.

During normal processing, when using HOLD\_CURSOR=NO and RELEASE\_CURSOR=NO (the default), it is advisable to set MAXOPENCURSORS to no more than 6 less than the database parameter OPEN\_CURSORS to allow for the cursors used by the data dictionary to process statements.

As your program's need for concurrently open cursors grows, you might want to re-specify MAXOPENCURSORS to match the need. A value of 45 to 50 is not uncommon, but remember that each cursor requires another private SQL area in the user process memory space. The default value of 10 is adequate for most programs.

#### **Related Topics**

- Performance Tuning
- Separate Precompilations

### 14.8.33 MODE

#### **Purpose**

This macro option specifies whether your program observes Oracle practices or complies with the current ANSI SQL standard.

#### **Syntax**

MODE={ANSI | ISO | ANSI14 | ISO14 | ANSI13 | ISO13 | ORACLE}

#### Default

**ORACLE** 

#### **Usage Notes**

Cannot be entered inline.

The following pairs of MODE values are equivalent: ANSI and ISO, ANSI14 and ISO14, ANSI13 and ISO13.

When MODE=ORACLE (the default), your embedded SQL program observes Oracle practices.

When MODE={ANSI14 | ANSI13}, your program complies closely with the current ANSI SQL standard.

When MODE=ANSI, your program complies *fully* with the ANSI standard and the following changes go into effect:

You cannot open a cursor that is already open or CLOSE a cursor that is already closed.
 (When MODE=ORACLE, you can reOPEN an open cursor to avoid re-parsing.)



 No error message is issued if Oracle assigns a truncated column value to an output host variable.

When MODE={ANSI | ANSI14}, a 4-byte integer variable named SQLCODE or a 5-byte character variable named SQLSTATE must be declared. For more information, see "Error Handling Alternatives".

#### **Related Topics**

Error Handling Alternatives

### 14.8.34 NESTED

#### **Purpose**

Indicates whether GLOBAL clauses in nested programs are to be generated. If the compiler supports nested programs, use YES as the value of NESTED.

#### **Syntax**

NESTED={YES | NO}

#### **Default**

YES

#### **Usage Notes**

Cannot be entered inline.

## 14.8.35 NLS LOCAL

#### **Purpose**

The NLS\_LOCAL option determines whether Globalization Support (formerly called NLS) character conversions are performed by the Pro\*COBOL runtime library or by the Oracle Server.

#### **Syntax**

NLS\_LOCAL={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Cannot be entered inline.

This option is for use when passing National Character Set variables to and from the server.

When NLS\_LOCAL=YES, the runtime library (SQLLIB) locally performs blank-padding and blank-stripping for host variables that have multibyte Globalization Support datatypes. Continue to use this value only for Pro\*COBOL applications written for releases before releases 8.0.



When NLS\_LOCAL=YES, because dynamic SQL statements are not processed at precompile time, this option has no effect on dynamic SQL statements.

Also, when NLS\_LOCAL=YES, columns storing multibyte Globalization Support data cannot be used in embedded data definition language (DDL) statements. This restriction cannot be enforced when precompiling, so the use of these column types within embedded DDL statements results in an execution error rather than a precompile error.

When NLS\_LOCAL=NO, blank-padding and blank-stripping operations are performed by the Oracle Server for host variables that have multibyte Globalization Support datatypes. Use for all new release 8.0, or later, applications.

The environment variable NLS\_NCHAR specifies the character set used for National Character Set data. (NCHAR, NVARCHAR2, NCLOB). If it is not specified, the character set defined or defined indirectly by NLS\_LANG will be used. See "Setting Up a Globalization Support Environment" for details.

### 14.8.36 ONAME

#### **Purpose**

Specifies the name of the output file.

#### **Syntax**

ONAME=filename

#### Default

System dependent

#### **Usage Notes**

Cannot be entered inline.

Use this option to specify the name of the output file, where the name differs from that of the input file. For example, if you issue

```
procob INAME=my test
```

the default output filename is  $my\_test.cob$ . If you want the output filename to be  $my\_test\_1.cob$ , issue the command

```
procob INAME=my test ONAME=my test 1.cob
```

Note that you should add the *.cob* extension to files specified using ONAME. There is no default extension with the ONAME option.

**Attention:** Oracle recommends that you not let the output filename default, but rather name it explicitly using ONAME.

### 14.8.37 ORACA

#### **Purpose**

Specifies whether a program can use the Oracle Communications Area (ORACA).



#### **Syntax**

ORACA={YES | NO}

#### **Default**

NO

#### **Usage Notes**

When ORACA=YES, you must place the INCLUDE ORACA statement in your program.

### 14.8.38 ORECLEN

#### **Purpose**

Specifies the record length of the output file.

#### **Syntax**

ORECLEN=integer

#### **Default**

80

#### **Usage Notes**

Cannot be entered inline.

The value you specify for ORECLEN should equal or exceed the value of IRECLEN. The maximum value allowed is system-dependent.

## 14.8.39 OUTLINE

#### **Purpose**

Indicates that the outline SQL file needs to be generated for the SQL statements.

#### **Syntax**

outline={yes | no | category\_name}

#### Default

no

#### **Usage Notes**

The outline SQL file should be in the DEFAULT category if the value is yes and the generated outline format is

```
DEFAULT <filename> <filetype> <sequence no>
```

If the category name is mentioned, then the SQL file should be generated in the category mentioned. The generated outline format for this is



```
<category_name>_<filename>_<filetype>_<sequence_no>
```

The outline SQL file is not generated if the value is no.

Semantic check should be full when this option is turned on, which means option sqlcheck=full/semantics. If sqlcheck=syntax/limited/none, then error will be generated.

### 14.8.40 OUTLNPREFIX

#### **Purpose**

Controls the generation of the outline names.

#### **Syntax**

outInprefix={none | prefix\_name}

#### **Default**

no

#### **Usage Notes**

If outInprefix=prefix\_name, then the outline format

```
<category name> <filename> <filetype>
```

is replaced with fix name> for the outline names.

If the length of the outline name exceeds 128 bytes, then this option is helpful for the user who can just specify the prefix name.

If outInprefix=none, then the outline names are generated by the system. The generated format is

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

Semantic check should be full when this option is turned on, which means option sqlcheck=full/semantics. If sqlcheck=syntax/limited/none, and/or outline=false, then error will be generated.

### 14.8.41 PAGELEN

#### **Purpose**

Specifies the number of lines for each physical page of the listing file.

#### **Syntax**

PAGELEN=integer

#### **Default**

66

#### **Usage Notes**

Cannot be entered inline. The maximum value allowed is system-dependent.

# 14.8.42 PICN\_ENDIAN

#### **Purpose**

Maintains platform endianness (little endian for Linux and Windows; big endian for Solaris) in PIC N variables.

#### **Syntax**

picn\_endian={BIG | OS}

#### **Default**

BIG

#### **Usage Notes**

If picn\_endian=big, then PIC N variables are bound with character set ID AL16UTF16.

If picn\_endian=os then PIC N variables are bound with character set ID UCS2.

The default value for this option is "big" to preserve the current behavior. This option is ignored if NLS NCHAR is not AL16UTF16.

Character set form for PIC N variables can be set by using the existing Pro\*Cobol command line option: charset\_picn={nchar\_charset | db\_charset}

### 14.8.43 PICX

#### **Purpose**

Specifies the default datatype of PIC X variables.

#### **Syntax**

PICX={CHARF | VARCHAR2}

#### **Default**

**CHARF** 

#### **Usage Notes**

Can be entered only on the command line or in a configuration file.

Starting in Pro\*COBOL 8.0, the default datatype of PIC X, N, or G variables was changed from VARCHAR2 to CHARF. PICX is provided for backward compatibility.

This new default behavior is consistent with the normal COBOL move semantics. Note that this is a change in behavior for the case where you are inserting a PIC X variable (with MODE=ORACLE) into a VARCHAR2 column. Any trailing blanks which had formerly been trimmed will be preserved. Note also, that the new default lessens the occurrence of the following anomaly: Using a PIC X bind variable initialized with trailing blanks in a WHERE clause would never match a value with the same number of trailing blanks which was stored in a char column because the bind variable's trailing blanks were stripped before the comparison.



When PICX=VARCHAR2, Oracle treats local CHAR variables in a PL/SQL block like variable-length character values. When PICX=CHARF, however, Oracle treats the CHAR variables like ANSI-compliant, fixed-length character values. See "Default for PIC X" for an extensive discussion.

#### **Related Topics**

Default for PIC X

## 14.8.44 PLAN\_BASELINE

#### **Purpose**

Creates a SQL Plan Baseline by specifying the module name.

#### **Syntax**

PLAN BASELINE={module\_name | YES | NO}

#### **Default**

NO

#### **Usage Notes**

The module name will become a part of the unique plan name generated by appending the filename, file type, and sequence number.



To generate and execute the plan baseline SQL statements, you must have the Execute privilege on the DBMS\_SPM\_INTERNAL package and the Administer SQL Management Object privilege.

The plan baseline options are valid only with SQLCHECK=FULL. If you specify these options without this, then the options are ignored and a warning is thrown informing that the option should be used with SQLCHECK=FULL only.

# 14.8.45 PLAN\_PREFIX

#### **Purpose**

Ensures that the plan name does not exceed 128 bytes.

#### **Syntax**

PLAN PREFIX={prefix name | none}

#### Default

none



#### **Usage Notes**

It is an optional command. The default is none which means that no prefix name is used, and if the plan name exceeds 128 bytes an error message is generated.

## 14.8.46 PLAN RUN

#### **Purpose**

Executes the generated SQL file.

#### **Syntax**

PLAN\_RUN={YES | NO}

**Default** 

NO

#### **Usage Notes**

If the PLAN\_RUN option is not set then the generated SQL file is not executed.

## 14.8.47 PLAN\_FIXED

#### **Purpose**

Specifies whether or not the created plan baseline is fixed or non-fixed.

#### **Syntax**

PLAN\_FIXED={ YES | NO }

Default

YES

#### **Usage Notes**

When set to NO, a non-fixed Plan Baseline is created.

## 14.8.48 PLAN ENABLED

#### **Purpose**

Enables the use of the Plan Baseline that is created.

#### **Syntax**

PLAN\_ENABLED={ YES | NO }

**Default** 

YES



#### **Usage Notes**

When set to the default, YES, the created Plan Baseline is used for plan selection. When set to NO, the Plan Baseline is created, but is not used until enabled manually.

### 14.8.49 MEMFORPREFETCH

#### **Purpose**

Use this option to speed up queries by pre-fetching the number of rows that can be accommodated in the specified memory.

#### **Syntax**

MEMFORPREFETCH=integer

#### Default

No value set.

#### **Usage Notes**

This option can be used in configuration file or on the command line. The value of the integer is used for execution of all queries using explicit cursors, subject to the rules of precedence.

When used inline it must be placed before OPEN statements with explicit cursors. Then the number of rows pre-fetched when that OPEN is done is determined by the last in-line MEMFORPREFETCH option in effect.

The MEMFORPREFETCH default is no-value-is-set. To turn off prefetching, use MEMFORPREFETCH=0 on the command line.

Prefetching is turned off when LONG or LOB columns are being accessed.

MEMFORPREFETCH is used to enhance the performance of single row fetches.

MEMFORPREFETCH values have no effect when doing array fetches, regardless of which value is assigned.

There is no single perfect prefetch memory value that can be used to assist all the fetches in an application.

Therefore, when using the MEMFORPREFETCH option, you should test different values to give a general improvement across all statements in the program. Note that if certain statements need to be tuned individually, the MEMFORPREFETCH option can be specified inline using EXEC ORACLE OPTION. Note that this will affect all fetch statements that follow the command in your program. Select the appropriate prefetch memory to enhance the performance of any particular FETCH statement. To achieve this individual prefetch count, you should use the inline prefetch option rather than from the command line.

## **14.8.50 PREFETCH**

#### **Purpose**

Use this option to speed up queries by pre-fetching a given number of rows.

#### **Syntax**

PREFETCH=integer



#### **Default**

1

#### **Usage Notes**

Can be used in a configuration file or on the command-line. The value of the integer is used for execution of all queries using explicit cursors, subject to the rules of precedence.

When used in-line it must be placed before OPEN statements with explicit cursors. Then the number of rows pre-fetched when that OPEN is done is determined by the last in-line PREFETCH option in effect.

The PREFETCH default is 1. To turn off prefetching, use PREFETCH=0 on the command line.

Prefetching is turned off when LONG or LOB columns are being accessed.

PREFETCH is used to enhance the performance of single row fetches. PREFETCH values have no effect when doing array fetches, regardless of which value is assigned.

There is no single *perfect* prefetch number that can be used to assist all the fetches in an application.

Therefore, when using the PREFETCH option, you should test different values to give a general improvement across all statements in the program. Note that if certain statements need to be tuned individually, the PREFETCH option can be specified in line using EXEC ORACLE OPTION. Note that this will affect *all* fetch statements that follow the command in your program. Select the appropriate prefetch number to enhance the performance of any particular FETCH statement. To achieve this individual prefetch count, you should use the inline prefetch option. (Rather than from the command line.)

The maximum value is 9999. See "The PREFETCH Precompiler Option" for further discussion.

#### **Related Topics**

The PREFETCH Precompiler Option

# 14.8.51 RELEASE\_CURSOR

#### **Purpose**

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

#### **Syntax**

RELEASE\_CURSOR={YES | NO}

#### **Default**

NO



#### **Usage Notes**

You can use RELEASE\_CURSOR to improve the performance of your program. For more information, see .

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. RELEASE\_CURSOR controls what happens to the link between the cursor cache and private SQL area.

When RELEASE\_CURSOR=YES, after Oracle executes the SQL statement and the cursor is closed, Pro\*COBOL immediately removes the link. This frees memory allocated to the private SQL area and releases parse locks. To make sure that associated resources are freed when you CLOSE a cursor, you must specify RELEASE\_CURSOR=YES.

When RELEASE\_CURSOR=NO, the link is maintained. Pro\*COBOL does not reuse the link unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed often because it speeds up subsequent executions. There is no need to re-parse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set RELEASE\_CURSOR before executing the SQL statement. For inline use with explicit cursors, set RELEASE\_CURSOR before opening the cursor.

Note that RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES. For information showing how these two options interact, see "Performance Tuning".

#### **Related Topics**

Precompiler Options

### 14.8.52 RUNOUTLINE

#### **Purpose**

Provides the developer with the option of executing "create outline" statements either by using precompiler or by the developer manually at a later time.

#### **Syntax**

runoutline={yes | no}

#### Default

no

#### **Usage Notes**

If runoutline=yes, then the generated 'create outline' statements are executed by the precompiler/translator at the end of a successful precompilation.

The outline option should be set to true or category\_name when runoutline is used. Semantic check should be full when this option is turned on, which means option sqlcheck=full/ semantics. If sqlcheck=syntax/limited/none, then error will be generated.



## 14.8.53 SELECT\_ERROR

#### **Purpose**

Specifies whether your program generates an error when a SELECT statement returns more than one row or more rows than a host array can accommodate.

#### **Syntax**

SELECT\_ERROR={YES | NO}

#### Default

YES

#### **Usage Notes**

When SELECT\_ERROR=YES, an error is generated if a single-row select returns too many rows or an array select returns more rows than the host array can accommodate.

When SELECT\_ERROR=NO, no error is generated when a single-row select returns too many rows or when an array select returns more rows than the host array can accommodate.

Whether you specify YES or NO, a random row is selected from the table. To ensure a specific ordering of rows, use the ORDER BY clause in your SELECT statement. When SELECT\_ERROR=NO and you use ORDER BY, Oracle returns the first row, or the first *n* rows if you are selecting into an array. When SELECT\_ERROR=YES, whether or not you use ORDER BY, an error is generated if too many rows are returned.

## 14.8.54 SQLCHECK

#### **Purpose**

Specifies the type and extent of syntactic and semantic checking...

When SQLCHECK=SEMANTICS or FULL, the SQL statements are packaged/bundled into an IDL objects using a generic grammar during parsing. The generic grammar will not try to understand the SQL syntax, it can only identify the host variables, indicator variables, and the possible SQL identifiers. During the semantic phase, the validity of the host and indicator variables are checked in the same manner it is now being done for SQL. The semantics like the table name ,column names,types will be checked in the same way it is now being handled for SQL.

The new unified parser:

- Precompiles all the existing precompiler applications.
- Supports the following SQL statements completely (all clauses of these SQL statements):
  - a. SELELCT statement
  - b. INSERT statement
  - c. DELETE statement



- d. UPDATE statement
- e. Body of the cursor in a DECLARE CURSOR statement

#### **Syntax**

SQLCHECK={SEMANTICS | FULL | SYNTAX | LIMITED}

#### **Default**

**SYNTAX** 

#### **Usage Notes**

The values SEMANTICS and FULL are equivalent, as are the values SYNTAX and LIMITED.

Pro\*COBOL can help you debug a program by checking the syntax and semantics of embedded SQL statements and PL/SQL blocks. Any errors found are reported at precompile time.

You control the level of checking by entering the SQLCHECK option inline or on the command line, or both inline and on the command line. However, the level of checking you specify inline cannot be higher than the level you specify (or accept by default) on the command line.

Pro\*COBOL generates an error when PL/SQL reserved words are used in SQL statements, even though the SQL statements are not themselves PL/SQL. If a PL/SQL reserved word must be used as an identifier, you can enclose it in double-quotes (").

When SQLCHECK=SEMANTICS, Pro\*COBOL checks the syntax and semantics of

- Data manipulation statements such as INSERT and UPDATE
- PL/SQL blocks

However, Pro\*COBOL checks only the syntax of remote data manipulation statements (those using the AT *db\_name* clause).

Pro\*COBOL gets the information for a semantic check from embedded DECLARE TABLE statements or, if you specify the option USERID, by connecting to Oracle and accessing the data dictionary. You need not connect to Oracle if every table referenced in a data manipulation statement or PL/SOL block is defined in a DECLARE TABLE statement.

If you connect to Oracle but some information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. During precompilation, a DECLARE TABLE definition overrides a data dictionary definition if they conflict.

Specify SQLCHECK=SEMANTICS when precompiling new programs. If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK=SEMANTICS and the option USERID.

When SQLCHECK=SEMANTICS or FULL, the SQL statements are first locally parsed. The verification of the host and indicator variables, and checking for the validity of SQL identifiers is done by using embedded DECLARE TABLE statement or by connecting to server when userid is specified in command line. Now the parsing is done twice when SQLCHECK = SEMANTICS or FULL once by the precompiler and once by the PL/SQL. When a new syntax is made available by SQL, the precompiler fails on the new syntax before calling PL/SQL interfaces if the local SQL grammar is not updated to accept the new syntax.



When SQLCHECK=SYNTAX, Pro\*COBOL checks the syntax of data manipulation statements. The SQL statements are locally parsed. With this command line option, the precompiler does not verify the table name or column names. When SQLCHECK=SYNTAX, the syntax of the SQL statements is checked by using the client side SQL interface.

No semantic checking is done. DECLARE TABLE statements are ignored and PL/SQL blocks are not allowed. When checking data manipulation statements, Pro\*COBOL uses Oracle syntax rules, which are downwardly compatible. Specify SQLCHECK=SYNTAX when migrating your precompiled programs.

Table 14-5 summarizes the checking done by SQLCHECK. For more information about syntactic and semantic checking, see "Syntactic and Semantic Checking".

**SOLCHECK=** SOLCHECK=S -**SEMANTICS YNTAX** Syntax Semantics Syntax Semantics DML Χ Χ Χ Remote Χ Χ **DML** PL/SQL Χ Х

Table 14-5 Checking Done by SQLCHECK

#### **Restrictions of Unified Parser**

The restrictions of unified parser for Pro\*COBOL are:

- Pro\*COBOL does not support variables without a ':' (colon).
- Define tables cannot be used. In CSF mode, plsql fails when declare table statement is passed to plsql.
- The following example fails if sqlcheck=syntax and common parser=yes is used:

```
select ename into :empname from emp@dblink;
```

In earlier versions, only local parsing is done when sqlcheck=syntax is used. But now when sqlcheck=syntax and common\_parser=yes, the statement is just bundled and sent to pcisyn(). PLSQL fails if connection is not there when dblinks are used.

The above statement can be precompiled with:

```
sqlcheck=full userid=<userid> common parser=yes
```

Error handling for insert does not follow the usual sequence. Consider the following statement where XYZ is a group item.

```
insert into emp (empno, ename) values (:XYZ)
```

PLSQL does not allow group items or structures while doing syntax check for insert statements (restriction from PLSQL.). It expects that the group item be broken into the individual elements for inserts. So the precompiler should expand the above statement to insert into emp (empno, ename) values (:b1,:b2) in the syntax phase itself.



This expansion is not possible in syntax phase. Therefore, pcisyn() is not called in syntax phase of precompiler the way it is done for other statements. The statement is deferred till semantic phase of precompiler for insert statement only. The net effect is that the error messages might not be following the usual sequence. For example:

```
EXEC SQL insert into emp (empno, ename) alues (:XYZ) END-EXEC.

^^^^^ syntax error at 'alues'

EXEC SQL select ename into :empname fro emp END-EXEC.

^^^ syntax error at 'fro'
```

Ideally error sequence should be as follows:

```
syntax error at 'alues' in INSERT
syntax error at 'fro' in SELECT
```

Because of the restrictions discussed earlier, the error sequence will be:

```
syntax error at 'fro' in SELECT
syntax error at 'alues' in INSERT
```



The line numbers are there in the reported errors. Therefore, Pro\*Cobol developers can identify the line very easily, even though it is not in the order used in the program.

#### **Related Topics**

Syntactic and Semantic Checking Basics

## 14.8.55 STMT CACHE

#### **Purpose**

Denotes the Statement cache size for the dynamic SQL statements.

#### **Syntax**

STMT\_CACHE = Range is 0 to 65535

#### Default

0

#### **Usage Notes**

The stmt\_cache option can be set to hold the anticipated number of distinct dynamic SQL statements in the application.

## 14.8.56 TYPE\_CODE

#### **Purpose**

This micro option of MODE specifies whether ANSI or Oracle datatype codes are used in ANSI dynamic SQL method 4. Its setting is the same as the setting of MODE option.

#### **Syntax**

TYPE\_CODE={ORACLE | ANSI}

#### **Default**

**ORACLE** 

#### **Usage Notes**

Cannot be entered inline.

See the possible option settings in "TYPE CODE Option Settings".

## 14.8.57 UNSAFE\_NULL

#### **Purpose**

Specifying UNSAFE\_NULL=YES prevents generation of ORA-01405 messages when fetching NULLs without using indicator variables.

#### **Syntax**

UNSAFE\_NULL={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Cannot be entered inline.

The UNSAFE\_NULL=YES is allowed only when MODE=ORACLE.

The UNSAFE\_NULL option has no effect on host variables in an embedded PL/SQL block. You *must* use indicator variables to avoid ORA-01405 errors.

When UNSAFE\_NULL=YES, no error is returned if a SELECT or FETCH statement selects a NULL, and there is no indicator variable associated with the output host variable. When UNSAFE\_NULL=NO, selecting or fetching a NULL column or expression into a host variable that has no associated indicator variable causes an error (SQLSTATE is 22002; SQLCODE is ORA-01405).

### 14.8.58 USERID

#### **Purpose**

Specifies an Oracle username and password.

#### **Syntax**

USERID=username/password[@dbname]

#### Default

None



### **Usage Notes**

Cannot be entered inline.

When SQLCHECK=SEMANTICS, if you want Pro\*COBOL to get needed information by connecting to Oracle and accessing the data dictionary, you must also specify USERID. The database alias is optional. Do not enter the brackets.

### 14.8.59 VARCHAR

### **Purpose**

The VARCHAR option instructs Pro\*COBOL to treat the COBOL group item described in "Embedded SQL" as a VARCHAR datatype.

### **Syntax**

VARCHAR={YES | NO}

### **Default**

NO

### **Usage Notes**

Cannot be entered inline.

When VARCHAR=YES, the implicit group item described in "Embedded SQL" is accepted as a VARCHAR external datatype with a length field and a string field.

When VARCHAR=NO, Pro\*COBOL does not accept the implicit group items as VARCHAR external datatypes.

### **Related Topics**

Embedded SQL

## 14.8.60 XREF

### **Purpose**

Specifies whether a cross-reference section is included in the listing file.

#### **Syntax**

XREF={YES | NO}

#### **Default**

YES

### **Usage Notes**

When XREF=YES, cross references are included for host variables, cursor names, and statement names. The cross references show where each object is defined and referenced in your program.

When XREF=NO, the cross-reference section is not included.

A

# **Operating System Dependencies**

Some details of COBOL programming vary from one system to another. This appendix is a collection of all system-specific issues regarding Pro\*COBOL. References are provided, where applicable, to other sources in your document set.

## A.1 System-Specific References in this Manual

System-specific references are described in the following section, grouped by subject area.

### A.1.1 COBOL Versions

The Pro\*COBOL Precompiler supports the standard implementation of COBOL for your operating system (usually COBOL-85 or COBOL-74). Some platforms may support both COBOL implementations. Check your Oracle system-specific documentation.

## A.1.2 Host Variables

How you declare and name host variables depends on which COBOL compiler you use. Check your COBOL user's guide for details about declaring and naming host variables.

## A.1.2.1 Declaring

Declare host variables according to COBOL rules, specifying a COBOL datatype supported by Oracle. Declaring Host Variables shows the COBOL datatypes and pseudotypes you can specify. However, your COBOL implementation might not include all of them.

## A.1.2.2 Naming

Host variable names must consist only of letters, digits, and hyphens. They must begin with a letter. They can be any length, but only the first 30 characters are significant. Your compiler might allow a different maximum length.

Due to a Pro\*COBOL limitation, when interacting with SQLLIB (C routines), some unpredictable results may occur unless boundaries for host variables are properly aligned. Refer to your COBOL documentation for specific information on defining host variable boundary alignment. Work-arounds could include:

- Manual alignment using FILLER
- FORCE the boundary by using 01 level entries
- If the data source is third party code, then use temporary variables at 77 level entries or 01 level entries, and use those as host variables.

## A.1.3 INCLUDE Statements

You can INCLUDE any file. When you precompile your Pro\*COBOL program, each EXEC SQL INCLUDE statement is replaced by a copy of the file named in the statement.

If your system uses file extensions but you do not specify one, the Pro\*COBOL Precompiler assumes the default extension for source files (usually COB). The default extension is system-dependent. Check your Oracle system-specific documentation.

If your system uses directories, you can set a directory path for included files by specifying the precompiler option INCLUDE=*path*. You must use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory. The syntax for specifying a directory path is system-specific. Check your Oracle system-specific documentation.

## A.1.4 MAXLITERAL Default

With the MAXLITERAL precompiler option you can specify the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded. The MAXLITERAL default value is 256, but you might have to specify a lower value.

For example, if your COBOL compiler cannot handle string literals longer than 132 characters, specify "MAXLITERAL=132." Check your COBOL compiler user's guide. For more information about the MAXLITERAL option, see Precompiler Options

# A.1.5 PIC N or Pic G Clause for Multi-byte Globalization Support Characters

Some COBOL compilers may not support the use of the PIC N or PIC G clause for declaring multibyte Globalization Support character variables. Check your COBOL user's guide before writing source code that uses these clauses to declare multibyte character variables.

## A.1.6 RETURN-CODE Special Register May Be Unpredictable.

The contents of the RETURN-CODE special register (for those systems that support it) are unpredictable after any SQL statement or SQLLIB function.

## A.1.7 Byte-Order of Binary Data

On some platforms, the COBOL compiler reverses the byte-ordering of binary data. See your platform-specific documentation for the COMP5 precompiler option.

Removed reference to NT - user comment 9561



B

# Reserved Words, Keywords, and Namespaces

Topics in this appendix include:

- Reserved Words and Keywords
- Reserved Namespaces

# **B.1** Reserved Words and Keywords

Some words are reserved by Oracle. That is, they have a special meaning to Oracle and cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes. To view the lists of the Oracle reserved words for SQL and PL/SQL, see the *Oracle Database SQL Language Reference* and the *Oracle Database PL/SQL Language Reference*.

Like COBOL keywords, you cannot use Pro\*COBOL keywords as variables in your program(s). Otherwise, an error will be generated. An error may result if they are used as the name of a database object such as a column. Here are the keywords used in Pro\*COBOL.

Keywords	Keywords	Keywords
all	allocate	alter
analyze	and	any
append	arraylen	as
asc	assign	at
audit	authorization	avg
begin	between	bind
both	break	buffer
buffering	by	call
cast	char	character
character_set_name	charf	charz
check	chunksize	close
collection	comment	commit
connect	constraint	constraints
context	continue	convbufsz
сору	count	create
current	currval	cursor
data	database	date
dateformat	datelang	datetime_interval_code
datetime_interval_precision	day	deallocate
dec	decimal	declare



Keywords	Keywords	Keywords
default	define	delete
deref	desc	describe
descriptor	directory	disable
display	distinct	do
double	drop	eject
else	enable	end
endif	end-exec	erase
escape	exec	execute
exists	explain	extract
fetch	file	fileexists
filename	first	float
flush	for	force
found	free	from
function	get	global
go	goto	grant
group	having	hold
host_stride_length	hour	iaf
identified	ifdef	ifndef
immediate	in	include
indicator	indicator_stride_length	input
insert	integer	internal_length
intersect	interval	into
is	isopen	istemporary
last	leading	length
level	like	list
load	lob	local
lock	long	max
message	min	minus
minute	mode	month
multiset	name	national_character
nchar	nchar_cs	next
nextval	noaudit	not
notfound	nowait	null
nullable	number	numeric
nvarchar	nvarchar2	object
ocibfilelocator	ocibloblocator	ocicloblocator
ocidate	ociextproccontext	ocinumber
ociraw	ocirowid	ocistring
octet_length	of	one



Keywords	Keywords	Keywords
only	open	option
or	oracle	order
output	overlaps	overpunch
package	partition	perform
precision	prepare	prior
procedure	put	raw
read	real	ref
reference	register	release
rename	replace	return
returned_length	returned_octet_length	returning
revoke	role	rollback
rowid	rownum	savepoint
scale	second	section
select	set	skip1
skip2	skip3	smallint
some	sql	sql-context
sql-cursor	sqlerror	sqlwarning
start	statement	stddev
stop	string	sum
sysbackup	sysdba	sysoper
sysdate		
table	temporary	threads
time	timestamp	timezone_hour
timezone_minute	to	tools
title	trailing	transaction
trigger	trim	truncate
type	uid	ulong_varchar
union	unique	unsigned
user_defined_type_name	user_defined_type_name_length	user_defined_type_schema
user_defined_type_schema_le ngth	user_defined_type_version	update
use	user	using
validate	value	values
var	varchar	varchar2
variables	variance	varnum
varraw	view	whenever
where	with	work
write	year	zone



# **B.2** Reserved Namespaces

Table B-1 contains a list of namespaces that are reserved by Oracle. The initial characters of subprogram names in Oracle libraries are restricted to the character strings in this list. Because of potential name conflicts, use subprogram names that do not begin with these characters.

For example, the Oracle Net Transparent Network Service functions all begin with the characters "NS," so avoid writing subprograms with names beginning with "NS."

Table B-1 Reserved Namespaces

Namespace	Library
XA	external functions for XA applications only
SQ	external SQLLIB functions used by Oracle Precompiler and SQL*Module applications
O, OCI	external OCI functions internal OCI functions
UPI, KP	function names from the Oracle UPI layer
NA	Oracle Net Native services product
NC	Oracle Net RPC project
ND	Oracle Net Directory
NL	Oracle Net Network Library layer
NM	Oracle Net Net Management Project
NR	Oracle Net Interchange
NS	Oracle Net Transparent Network Service
NT	Oracle Net Drivers
NZ	Oracle Net Security Service
OSN	Oracle Net V1
TTC	Oracle Net Two task
GEN, L, ORA	Core library functions
LI, LM, LX	function names from the Oracle Globalization Support layer
S	function names from system-dependent libraries



C

# Performance Tuning

This appendix shows you some simple, easy-to-apply methods for improving the performance of your applications. Using these methods, you can often reduce processing time by 25 percent or more. Topics are:

- Causes of Poor Performance
- Improving Performance
- Using Host Tables
- Using PL/SQL and Java
- Optimizing SQL Statements
- Using Indexes
- Taking Advantage of Row-Level Locking
- Eliminating Unnecessary Parsing
- · About Using Oracle Connection Manager in Traffic Director Mode

## C.1 Causes of Poor Performance

One cause of poor performance is high Oracle communication overhead. Oracle must process SQL statements one at a time. Thus, each statement results in another call to Oracle and higher overhead. In a networked environment, SQL statements must be sent over the network, adding to network traffic. Heavy network traffic can slow down your application significantly.

Another cause of poor performance is inefficient SQL statements. Because SQL is so flexible, you can get the same result using two different statements. Using one statement might be less efficient. For example, the following two SELECT statements return the same rows (the name and number of every department having at least one employee):

```
EXEC SQL SELECT DNAME, DEPTNO FROM DEPT WHERE DEPTNO IN (SELECT DEPTNO FROM EMP) END-EXEC.
```

#### Contrasted with:

```
EXEC SQL SELECT DNAME, DEPTNO
FROM DEPT
WHERE EXISTS
(SELECT DEPTNO FROM EMP WHERE DEPT.DEPTNO = EMP.DEPTNO)
END-EXEC.
```

The first statement is slower because it does a time-consuming full scan of the EMP table for every department number in the DEPT table. Even if the DEPTNO column in EMP is indexed, the index is not used because the subquery lacks a WHERE clause naming DEPTNO.

Another cause of poor performance is unnecessary parsing and binding. Recall that before executing a SQL statement, Oracle must parse and bind it. Parsing means examining the SQL statement to make sure it follows syntax rules and refers to valid database objects. Binding means associating host variables in the SQL statement with their addresses so that Oracle can read or write their values.

Many applications manage cursors poorly. This results in unnecessary parsing and binding, which adds noticeably to processing overhead.

# C.2 Improving Performance

If you are unhappy with the performance of your precompiled programs, there are several ways you can reduce overhead.

You can greatly reduce Oracle communication overhead, especially in networked environments, by

- Using host tables
- Using embedded PL/SQL

You can reduce processing overhead—sometimes dramatically—by

- Optimizing SQL statements
- SQL Statement Caching
- Using indexes
- Taking advantage of row-level locking
- Eliminating unnecessary parsing
- Avoiding unnecessary reparsing

The following sections look at each of these ways to cut overhead.

# C.3 Using Host Tables

Host tables can boost performance because they let you manipulate an entire collection of data with a single SQL statement. For example, suppose you want to insert salaries for 300 employees into the EMP table. Without tables your program must do 300 individual inserts—one for each employee. With arrays, only one INSERT is necessary. Consider the following statement:

```
EXEC SQL INSERT INTO EMP (SAL) VALUES (:SALARY) END-EXEC.
```

If *SALARY* is a simple host variable, Oracle executes the INSERT statement once, inserting a single row into the EMP table. In that row, the SAL column has the value of *SALARY*. To insert 300 rows this way, you must execute the INSERT statement 300 times.

However, if *SALARY* is a host table of size 300, Oracle inserts all 300 rows into the EMP table at once. In each row, the SAL column has the value of an element in the *SALARY* table.

For more information, see Host Tables



# C.4 Using PL/SQL and Java

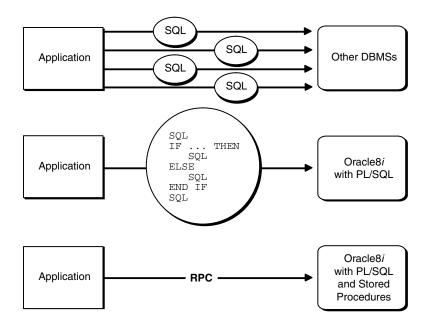
As Figure C-1 shows, if your application is database-intensive, you can use control structures to group SQL statements in a PL/SQL block, then send the entire block to Oracle. This can drastically reduce communication between your application and the database.

Also, you can use PL/SQL and Java subprograms to reduce calls from your application to the database. For example, to execute ten individual SQL statements, ten calls are required, but to execute a subprogram containing ten SQL statements, only one call is required.

Unlike anonymous blocks, PL/SQL and Java subprograms can be compiled separately and stored in a database. When called, they are passed to the PL/SQL engine immediately. Moreover, only one copy of a subprogram need be loaded into memory for execution by multiple users.

Figure C-1 PL/SQL Boosts Performance

PL/SQL Increases Performance Especially in Networked Environments



# C.5 Optimizing SQL Statements

For every SQL statement, the optimizer generates an *execution plan*, which is a series of steps that Oracle takes to execute the statement. These steps are determined by rules given in the *Oracle Database Advanced Application Developer's Guide*. Following these rules will help you write optimal SQL statements.

## C.5.1 Optimizer Hints

For every SQL statement, the optimizer generates an *execution plan*, which is a series of steps that Oracle takes to execute the statement. In some cases, you can suggest the way to

optimize a SQL statement. These suggestions, called *hints*, let you influence decisions made by the optimizer.

Hints are not directives; they merely help the optimizer do its job. Some hints limit the scope of information used to optimize a SQL statement, while others suggest overall strategies. You can use hints to specify the:

- Optimization approach for a SQL statement
- · Access path for each referenced table
- Join order for a join
- Method used to join tables

## C.5.1.1 Giving Hints

You give hints to the optimizer by placing them in a C-style Comment immediately after the verb in a SELECT, UPDATE, or DELETE statement. You can choose rule-based or cost-based optimization. With cost-based optimization, hints help maximize throughput or response time. In the following example, the ALL\_ROWS hint helps maximize query throughput:

```
EXEC SQL SELECT /*+ ALL_ROWS (cost-based) */ EMPNO, ENAME, SAL
   INTO :EMP-NUMBER, :EMP-NAME, :SALARY
   FROM EMP
   WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

The plus sign (+), which must immediately follow the Comment opener, indicates that the Comment contains one or more hints. Notice that the Comment can contain remarks as well as hints.

For more information about optimizer hints, see Performance and Scalability.

Trace Facility

You can use the SQL trace facility and the EXPLAIN PLAN statement to identify SQL statements that might be slowing down your application. The trace facility generates statistics for every SQL statement executed by Oracle. From these statistics, you can determine which statements take the most time to process. You can then concentrate your tuning efforts on those statements.

The EXPLAIN PLAN statement shows the execution plan for each SQL statement in your application. You can use the execution plan to identify inefficient SQL statements.

See Also: Tools for Performance for instructions on using trace tools and analyzing their output.

# C.6 SQL Statement Caching

Performance improvement is achieved in precompiler applications using statement caching. Any program using dynamic SQL statements, where the cursors have to be used with reparsing of the statements will have performance gain with statement caching. With this new feature, the overall execution time will be decreased.

The stmt\_cache option can be set to hold the anticipated number of distinct dynamic SQL statements in the application. The customer can set the stmt\_cache size with the

new precompiler command line option. An optimal value of stmt\_cache can not be set, as it depends on the input program behavior.

The performance can be measured with the change in the execution time (with and without statement caching).

# C.7 Using Indexes

Using rowids, an *index* associates each distinct value in a table column with the rows containing that value. An index is created with the CREATE INDEX statement. For details, see CREATE INDEX.

You can use indexes to boost the performance of queries that return less than 15% of the rows in a table. A query that returns 15% or more of the rows in a table is executed faster by a *full scan*, that is, by reading all rows sequentially. Any query that names an indexed column in its WHERE clause can use the index. For guidelines that help you choose which columns to index, see Using Indexes in Database Applications.

# C.8 Taking Advantage of Row-Level Locking

By default, Oracle locks data at the row level rather than the table level. Row-level locking allows multiple users to access different rows in the same table concurrently. The resulting performance gain is significant.

You can specify table-level locking, but it lessens the effectiveness of the transaction processing option. For more information about table locking, see "Using the LOCK TABLE Statement".

Applications that do online transaction processing benefit most from row-level locking. If your application relies on table-level locking, modify it to take advantage of row-level locking. In general, avoid explicit table-level locking.

# C.9 Eliminating Unnecessary Parsing

Eliminating unnecessary parsing requires correct handling of cursors and selective use of the following cursor management options:

- MAXOPENCURSORS
- HOLD\_CURSOR
- RELEASE CURSOR

These options affect implicit and explicit cursors, the cursor cache, and private SQL areas.

**Note:** You can use the ORACA to get cursor cache statistics. See "Using the Oracle Communications Area".

## C.9.1 Handling Explicit Cursors

Recall that there are two types of cursors: implicit and explicit (see "Errors and Warnings"). Oracle implicitly declares a cursor for all data definition and data manipulation statements. However, for queries that return more than one row, you should explicitly declare a cursor and fetch in batches rather than select into a host table. You use the DECLARE CURSOR statement to declare an explicit cursor. How you handle the opening and closing of explicit cursors affects performance.



If you need to reevaluate the active set, simply reopen the cursor. The OPEN statement will use any new host-variable values. You can save processing time if you do not close the cursor first.

Only CLOSE a cursor when you want to free the resources (memory and locks) acquired by OPENing the cursor. For example, your program should close all cursors before exiting.

**Note:** To make performance tuning easier, the precompiler lets you reopen an already open cursor. However, this is an Oracle extension to the ANSI/ISO embedded SQL standard. So, when MODE=ANSI, you must close a cursor before reopening it.

### C.9.1.1 Cursor Control

In general, there are three factors that affect the control of an explicitly declared cursor:

- Using the DECLARE, OPEN, FETCH, and CLOSE statements.
- Using the PREPARE, DECLARE, OPEN, FETCH, and CLOSE statements
- COMMIT closes the cursor when MODE=ANSI

With the first way, beware of unnecessary parsing. The OPEN statement does the parsing, but only if the parsed statement is unavailable because the cursor was closed or never opened. Your program should declare the cursor, re-open it every time the value of a host variable changes, and close it only when the SQL statement is no longer needed.

With the second way, which is used in dynamic SQL Methods 3 and 4, the PREPARE statement does the parsing, and the parsed statement is available until a CLOSE statement is executed. Your program should prepare the SQL statement and declare the cursor, re-open the cursor every time the value of a host variable changes, reprepare the SQL statement and re-open the cursor if the SQL statement changes, and close the cursor only when the SQL statement is no longer needed.

When possible, avoid placing OPEN and CLOSE statements in a loop; this is a potential cause of unnecessary re-parsing of the SQL statement. In the next example, both the OPEN and CLOSE statements are inside the outer loop. When MODE=ANSI, the CLOSE statement must be positioned as shown, because ANSI requires a cursor to be closed before being re-opened.

```
EXEC SQL DECLARE emp_cursor CURSOR FOR

SELECT ENAME, SAL FROM EMP

WHERE SAL > :SALARY

AND SAL <= :SALARY + 1000

END-EXEC.

MOVE 0 TO SALARY.

TOP.

EXEC SQL OPEN emp_cursor END-EXEC.

LOOP.

EXEC SQL FETCH emp_cursor INTO ....

IF SQLCODE = 0

GO TO LOOP

ELSE

ADD 1000 TO SALARY

END-IF.

EXEC SQL CLOSE emp cursor END-EXEC.
```



```
IF SALARY < 5000
GO TO TOP.
```

With MODE=ORACLE, however, by placing the CLOSE statement outside the outer loop, you can avoid possible re-parsing at each iteration of the OPEN statement.

```
TOP.

EXEC SQL OPEN emp_cursor END-EXEC.

LOOP.

EXEC SQL FETCH emp_cursor INTO ....

IF SQLCODE = 0

GO TO LOOP

ELSE

ADD 1000 TO SALARY

END-IF.

IF SALARY < 5000

GO TO TOP.

EXEC SQL CLOSE emp cursor END-EXEC.
```

## C.9.2 Using the Cursor Management Options

A SQL statement need be parsed only once unless you change its makeup. For example, you change the makeup of a query by adding a column to its select list or WHERE clause. The HOLD\_CURSOR, RELEASE\_CURSOR, and MAXOPENCURSORS options give you some control over how Oracle manages the parsing and re-parsing of SQL statements. Declaring an explicit cursor gives you maximum control over parsing.

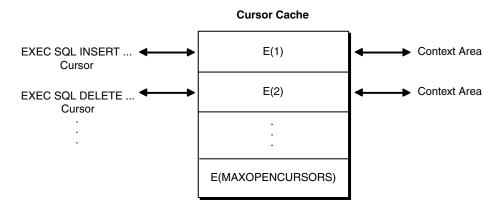
## C.9.2.1 Private SQL Areas and Cursor Cache

When any statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache is a continuously updated area of memory used for cursor management. The cursor cache entry is in turn linked to a private SQL area.

The private SQL area, a work area created dynamically at run time by Oracle, contains the parsed SQL statement, the addresses of host variables, and other information needed to process the statement. Dynamic Method 3 lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.

Figure C-2 represents the cursor cache after your program has done an insert and a delete.

Figure C-2 Cursors Linked through the Cursor Cache





### C.9.2.2 Resource Use

The maximum number of open cursors for each user session is set by the initialization parameter OPEN CURSORS.

MAXOPENCURSORS specifies the *initial* size of the cursor cache. If a new cursor is needed and there are no free cache entries, Oracle tries to reuse an entry. Its success depends on the values of HOLD\_CURSOR and RELEASE\_CURSOR and, for explicit cursors, on the status of the cursor itself.

If the value of MAXOPENCURSORS is less than the number of statements that need to be cached during the execution of the program, Oracle will search for cursor cache entries to reuse once MAXOPENCURSORS cache entries have been exhausted. For example, suppose the cache entry E(1) for an INSERT statement is marked as reusable, and the number of cache entries already equals MAXOPENCURSORS. If the program executes a new statement, cache entry E(1) and its private SQL area might be reassigned to the new statement. To reexecute the INSERT statement, Oracle would have to re-parse it and reassign another cache entry.

Oracle allocates an additional cache entry if it cannot find one to reuse. For example, if MAXOPENCURSORS=8 and all eight entries are active, a ninth is created. If necessary, Oracle keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN\_CURSORS. This dynamic allocation adds to processing overhead.

Thus, specifying a low value for MAXOPENCURSORS with HOLD\_CURSOR=NO (the default) saves memory but causes potentially expensive dynamic allocations and deallocations of new cache entries. Specifying a high value for MAXOPENCURSORS assures speedy execution but uses more memory.

## C.9.2.3 Infrequent Execution

Sometimes, the link between an *infrequently* executed SQL statement and its private SQL area should be temporary.

When HOLD\_CURSOR=NO (the default), after Oracle executes the SQL statement and the cursor is closed, the precompiler marks the link between the cursor and cursor cache as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks. However, because a prepared cursor must remain active, its link is maintained even when HOLD\_CURSOR=NO.

When RELEASE\_CURSOR=YES, after Oracle executes the SQL statement and the cursor is closed, the private SQL area is automatically freed and the parsed statement lost. This might be necessary if, for example, you wish to conserve memory.

When RELEASE\_CURSOR=YES, the link between the private SQL area and the cache entry is immediately removed and the private SQL area freed. Even if you tried to specify HOLD\_CURSOR=YES, Oracle must still reallocate memory for a private SQL area and re-parse the SQL statement before executing it. Therefore, specifying RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES.

## C.9.2.4 Frequent Execution

The links between a *frequently* executed SQL statement and its private SQL area should be maintained, because the private SQL area contains all the information



needed to execute the statement. Maintaining access to this information makes subsequent execution of the statement much faster.

When HOLD\_CURSOR=YES, the link between the cursor and cursor cache is maintained after Oracle executes the SQL statement. Thus, the parsed statement and allocated memory remain available. This is useful for SQL statements that you want to keep active because it avoids unnecessary re-parsing.

## C.9.2.5 Effect on the Shared SQL Area

Oracle caches the parsed representations of SQL statements and PL/SQL in its Shared SQL Cache. These representations are maintained until aged out by the need for the space to be used for other statements. For more information, see the *Oracle Database Concepts* manual. The behavior of the Oracle server in this respect is unaffected by the Precompiler's cursor management settings and so can have the following effects:

- When RELEASE\_CURSOR=YES and a statement is re executed, a request will be sent
  to the server to parse the statement but a full parse may not be necessary since the
  statement may still be cached.
- When using HOLD\_CURSOR=YES no locks are held on any objects referred to in the statement and so a redefinition of one of the objects in the statement will force the cached statement to become invalid and for the server to automatically reparse the statement. This may cause unexpected results.
- Nonetheless, when RELEASE\_CURSOR=YES, the re-parse might not require extra
  processing because Oracle caches the parsed representations of SQL statements and
  PL/SQL blocks in its Shared SQL Cache. Even if its cursor is closed, the parsed
  representation remains available until it is aged out of the cache.

## C.9.2.6 Embedded PL/SQL Considerations

For the purposes of cursor management, an embedded PL/SQL block is treated just like a SQL statement. When an embedded PL/SQL block is executed, a parent cursor is associated with the entire block and a link is created between the cache entry and the private SQL area in the PGA for the embedded PL/SQL block. Be aware that each SQL statement inside the embedded block also requires a private SQL area in the PGA. These SQL statements use child cursors that PL/SQL manages itself. The disposition of the child cursors is determined through its associated parent cursor. That is, the private SQL areas used by the child cursors are freed after the private SQL area for its parent cursor is freed.

#### Note:

Using the defaults, HOLD\_CURSOR=YES and RELEASE\_CURSOR=NO, after executing a SQL statement with an earlier Oracle version, its parsed representation remains available. With Oracle, under similar conditions, the parsed representation remains available only until it is aged out of the Shared SQL Cache. Normally, this is not a problem, but you might get unexpected results if the definition of a referenced object changes before the SQL statement is re-parsed.

### C.9.2.7 Parameter Interactions

HOLD\_CURSOR and RELEASE \_CURSOR Interactions shows how HOLD\_CURSOR and RELEASE\_CURSOR interact. Notice that HOLD\_CURSOR=NO overrides RELEASE\_CURSOR=NO and that RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES.



Table C-1 HOLD\_CURSOR and RELEASE \_CURSOR Interactions

HOLD_CURSOR	RELEASE_CURSOR	Links are
NO	NO	marked as reusable
YES	NO	maintained
NO	YES	removed immediately
YES	YES	removed immediately

# C.10 Avoiding Unnecessary Reparsing

When an embedded SQL statement is executed in a loop, it gets parsed only once. However, the execute phase of the SQL statement can result in errors, and statements are reparsed, with the following exceptions:

- ORA-1403 (not found)
- ORA-1405 (truncation)
- ORA-1406 (null value)

By correcting the errors, you can eliminate this unnecessary reparsing.

# C.11 About Using Oracle Connection Manager in Traffic Director Mode

Oracle Connection Manager in Traffic Director Mode is a proxy that is placed between supported database clients and database instances.



A multitenant container database is the only supported architecture in Oracle Database 21c. While the documentation is being revised, legacy terminology may persist. In most cases, "database" and "non-CDB" refer to a CDB or PDB, depending on context. In some contexts, such as upgrades, "non-CDB" refers to a non-CDB from a previous release.

Supported clients from Oracle Database 11*g* Release 2 (11.2) and later can connect to Oracle Connection Manager in Traffic Director Mode. Oracle Connection Manager in Traffic Director Mode provides improved high availability (HA) for planned and unplanned database server outages, connection multiplexing support, and load balancing. Support for Oracle Connection Manager in Traffic Director Mode is described in more detail in the following sections

- Modes of Operation
- Key Features



### **Modes of Operation**

Oracle Connection Manager in Traffic Director Mode supports the following modes of operation:

- In pooled connection mode, Oracle Connection Manager in Traffic Director Mode supports any application using the following database client releases:
  - OCI, OCCI, and Open Source Drivers (Oracle Database 11g release 2 (11.2.0.4) and later))
  - JDBC (Oracle Database 12c release 1 (12.1) and later)
  - ODP.NET (Oracle Database 12c release 2 (12.2) and later)

In addition, applications must use DRCP. That is, the application must enable DRCP in the connect string (or in the tnsnames.ora alias).

 In non-pooled connection (or dedicated) mode, Oracle Connection Manager in Traffic Director Mode supports any application using database client releases Oracle Database 11g release 2 (11.2.0.4) and later. In this mode, some capabilities, such as connection multiplexing are not available.

### **Key Features**

Oracle Connection Manager in Traffic Director Mode furnishes support for the following:

- Transparent performance enhancements and connection multiplexing, which includes:
  - Statement caching, rows prefetching, and result set caching are auto-enabled for all modes of operation.
  - Database session multiplexing (pooled mode only) using the proxy resident connection pool (PRCP), where PRCP is a proxy mode of Database Resident Connection Pooling (DRCP). Applications get transparent connect-time load balancing and run-time load balancing between Oracle Connection Manager in Traffic Director Mode and the database.
  - For multiple Oracle Connection Manager in Traffic Director Mode instances, applications get increased scalability through client-side connect time load balancing or with a load balancer (BIG-IP, NGINX, and others)
- Zero application downtime
  - Planned database maintenance or pluggable database (PDB) relocation
    - Pooled mode

Oracle Connection Manager in Traffic Director Mode responds to Oracle Notification Service (ONS) events for planned outages and redirects work. Connections are drained from the pool on Oracle Connection Manager in Traffic Director Mode when the request completes. Service relocation is supported for Oracle Database 11*g* release 2 (11.2.0.4) and later.

For PDB relocation, Oracle Connection Manager in Traffic Director Mode responds to in-band notifications when a PDB is relocated, that is even when ONS is not configured (for Oracle Database release 18c, version 18.1 and later server only)

Non-pooled or dedicated mode

When there is no request boundary information from the client, Oracle Connection Manager in Traffic Director Mode supports planned outage for many



applications (as long as only simple session state and cursor state need to be preserved across the request/transaction boundaries). This support includes:

- \* Stop service/PDB at the transaction boundary or it leverages Oracle Database release 18c continuous application availability to stop the service at the request boundary
- \* Oracle Connection Manager in Traffic Director Mode leverages
  Transparent Application Failover (TAF) failover restore to reconnect
  and restore simple states.
- Unplanned database outages for read-mostly workloads
- High Availability of Oracle Connection Manager in Traffic Director Mode to avoid a single point of failure. This is supported by:
  - Multiple instances of Oracle Connection Manager in Traffic Director Mode using a load balancer or client side load balancing/failover in the connect string
  - Rolling upgrade of Oracle Connection Manager in Traffic Director Mode instances
  - Graceful close of existing connections from client to Oracle Connection
     Manager in Traffic Director Mode for planned outages
  - In-band notifications to Oracle Database release 18c and later clients
  - For older clients, notifications are sent with the response of the current request
- For security and isolation, Oracle Connection Manager in Traffic Director Mode furnishes:
  - Database Proxy supporting transmission control protocol/transmission control protocol secure (TCP/TCPS) and protocol conversion
  - Firewall based on the IP address, service name, and secure socket layer/ transport layer security (SSL/TLS) wallets
  - Tenant isolation in a multi-tenant environment
  - Protection against denial-of-service and fuzzing attacks
  - Secure tunneling of database traffic across Oracle Database on-premises and Oracle Cloud



### See Also:

- Oracle Database Net Services Administrator's Guide for information about configuring cman.ora configuration file to set up Oracle Connection Manager in Traffic Director Mode
- Oracle Database Net Services Administrator's Guide for information about configuring databases for Oracle Connection Manager in Traffic Director Mode proxy authentication
- Oracle Database Net Services Administrator's Guide for information about configuring Oracle Connection Manager in Traffic Director Mode for unplanned down events
- Oracle Database Net Services Administrator's Guide for information about configuring Oracle Connection Manager in Traffic Director Mode for planned down events
- Oracle Database Net Services Administrator's Guide for information about configuring proxy resident connection pools for use by Oracle Connection Manager in Traffic Director Mode
- Oracle Database Net Services Administrator's Guide for information about functionality not supported for all drivers with Oracle Connection Manager in Traffic Director Mode
- Oracle Database Net Services Reference for an overview of Oracle CMAN configuration file



D

# Syntactic and Semantic Checking

By checking the syntax and semantics of embedded SQL statements and PL/SQL blocks, the Oracle Precompilers help you quickly find and fix coding mistakes. This appendix shows you how to use the SQLCHECK option to control the type and extent of checking.

### Topics are:

- Syntactic and Semantic Checking Basics
- Controlling the Type and Extent of Checking
- Specifying SQLCHECK=SEMANTICS

## D.1 Syntactic and Semantic Checking Basics

Rules of syntax specify how language elements are sequenced to form valid statements. Thus, *syntactic checking* verifies that keywords, object names, operators, delimiters, and so on are placed correctly in your SQL statement. It also applies to procedures and functions called from PL/SQL blocks. For example, the following embedded SQL statements contain syntax errors:

```
    * -- misspelled keyword WHERE
        EXEC SQL DELETE FROM EMP WERE DEPTNO = 20 END-EXEC.
    * -- missing parentheses around column names COMM and SAL EXEC SQL
        INSERT INTO EMP COMM, SAL VALUES (NULL, 1500)
        END-EXEC.
```

Rules of semantics specify how valid external references are made. Thus, *semantic checking* verifies that references to database objects and host variables are valid and that host-variable datatypes are correct. For example, the following embedded SQL statements contain semantic errors:

```
    * -- nonexistent table, EMPP
    EXEC SQL DELETE FROM EMPP WHERE DEPTNO = 20 END-EXEC.
    * -- undeclared host variable, EMP-NAME
    EXEC SQL SELECT * FROM EMP WHERE ENAME = :EMP-NAME END-EXEC.
```

The rules of SQL syntax and semantics are defined in the *Oracle Database SQL Language Reference*.

# D.2 Controlling the Type and Extent of Checking

You control the type and extent of checking by specifying the SQLCHECK option on the command line. With SQLCHECK, the type of checking can be syntactic, or both syntactic and semantic. The extent of checking can include data manipulation statements and PL/SQL blocks. However, SQLCHECK cannot check dynamic SQL statements because they are not defined fully until run time.

You can specify the following values for SQLCHECK:

- SEMANTICS | FULL
- SYNTAX | LIMITED

The values SEMANTICS and FULL are equivalent, as are the values SYNTAX and LIMITED. The default value is SYNTAX.

# D.3 Specifying SQLCHECK=SEMANTICS

When SQLCHECK=SEMANTICS, the precompiler checks the syntax and semantics of

- Data manipulation statements such as INSERT and UPDATE.
- PL/SQL blocks.

However, only syntactic checking is done on data manipulation statements that use the AT *db name* clause.

The precompiler gets the information for a semantic check from embedded DECLARE TABLE statements or, if you specify the option USERID, by connecting to the database and accessing the data dictionary.

If you connect to the database but some table information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. A DECLARE TABLE definition overrides a data dictionary definition if they conflict.

When checking data manipulation statements, the precompiler uses the Oracle set of syntax rules found in the *Oracle Database SQL Language Reference* but uses a stricter set of semantic rules. As a result, existing applications written for earlier versions of Oracle might not precompile successfully when SQLCHECK=SEMANTICS.

Specify SQLCHECK=SEMANTICS when precompiling new programs. If you embed PL/SQL blocks in a host program, you *must* specify SQLCHECK=SEMANTICS.

## D.3.1 Enabling a Semantic Check

When SQLCHECK=SEMANTICS, the precompiler can get information needed for a semantic check in either of the following ways:

- Connect to Oracle and access the data dictionary
- Use embedded DECLARE TABLE statements

## D.3.1.1 Connecting to Oracle

To do a semantic check, the precompiler can connect to the database that maintains definitions of tables and views referenced in your host program. After connecting, the precompiler accesses the data dictionary for needed information. The *data dictionary* stores table and column names, table and column constraints, column lengths, column datatypes, and so on.

If some of the needed information cannot be found in the data dictionary (because your program refers to a table not yet created, for example), you must supply the missing information using the DECLARE TABLE statement.

To connect to the database, specify the option USERID on the command line, using the syntax



```
USERID=username/password
```

where *username* and *password* comprise a valid Oracle userid. If you omit the password, you are prompted for it. If, instead of a username and password, you specify

```
USERID=/
```

the precompiler tries to connect to the database automatically with the userid

```
cprefix><username>
```

where *prefix* is the value of the initialization parameter OS\_AUTHENT\_PREFIX (the default value is OPS\$) and *username* is your operating system user or task name.

If you try connecting, but cannot (for example, if the database is unavailable), the precompiler stops processing and issues an error message. If you omit the option USERID, the precompiler must get needed information from embedded DECLARE TABLE statements.

## D.3.1.2 Using DECLARE TABLE

The precompiler can do a semantic check without connecting to the database as long as your program does not call any stored procedures or functions from an anonymous PL/SQL block. To do the check, the precompiler must get information about tables and views from embedded DECLARE TABLE directives. Thus, every table referenced in a data manipulation statement or PL/SQL block must be defined in a DECLARE TABLE statement.

The syntax of the DECLARE TABLE statement is

```
EXEC SQL DECLARE table_name TABLE
    (col_name col_datatype [DEFAULT expr] [NULL|NOT NULL], ...)
END-EXEC.
```

where *expr* is any expression that can be used as a default column value in the CREATE TABLE statement. *col\_datatype* is an Oracle column declaration. Only integers can be used, not expressions. See "DECLARE TABLE (Oracle Embedded SQL Directive)".

If you use DECLARE TABLE to define a database table that already exists, the precompiler uses your definition, ignoring the one in the data dictionary.



Е

# Embedded SQL Statements and Precompiler Directives

This appendix describes Oracle embedded SQL statements and directives. These statements and directives are prefaced in your source code with the keywords, EXEC SQL.

Note: Only statements which differ in syntax from non-embedded SQL are described in this appendix. For details of the non-embedded SQL statements, see the *Oracle Database SQL Language Reference*.

This appendix contains the following sections:

- Summary of Precompiler Directives and Embedded SQL Statements
- About the Statement Descriptions
- How to Read Syntax Diagrams
- ALLOCATE (Executable Embedded SQL Extension)
- ALLOCATE DESCRIPTOR (Executable Embedded SQL)
- CALL (Executable Embedded SQL)
- CLOSE (Executable Embedded SQL)
- COMMIT (Executable Embedded SQL)
- CONNECT (Executable Embedded SQL Extension)
- CONTEXT ALLOCATE (Executable Embedded SQL Extension)
- CONTEXT FREE (Executable Embedded SQL Extension)
- CONTEXT USE (Oracle Embedded SQL Directive)
- DECLARE CURSOR (Embedded SQL Directive)
- DECLARE DATABASE (Oracle Embedded SQL Directive)
- DECLARE STATEMENT (Embedded SQL Directive)
- DECLARE TABLE (Oracle Embedded SQL Directive)
- DELETE (Executable Embedded SQL)
- DESCRIBE (Executable Embedded SQL)
- DESCRIBE DESCRIPTOR (Executable Embedded SQL)
- ENABLE THREADS (Executable Embedded SQL Extension)
- EXECUTE ... END-EXEC (Executable Embedded SQL Extension)
- EXECUTE (Executable Embedded SQL)
- EXECUTE DESCRIPTOR (Executable Embedded SQL
- EXECUTE IMMEDIATE (Executable Embedded SQL)
- FETCH (Executable Embedded SQL)



- FETCH DESCRIPTOR (Executable Embedded SQL)
- FREE (Executable Embedded SQL Extension)
- GET DESCRIPTOR (Executable Embedded SQL)
- INSERT (Executable Embedded SQL)
- LOB APPEND (Executable Embedded SQL Extension)
- LOB ASSIGN (Executable Embedded SQL Extension)
- LOB CLOSE (Executable Embedded SQL Extension)
- LOB COPY (Executable Embedded SQL Extension)
- LOB CREATE TEMPORARY (Executable Embedded SQL Extension)
- LOB DESCRIBE (Executable Embedded SQL Extension)
- LOB DISABLE BUFFERING (Executable Embedded SQL Extension)
- LOB ENABLE BUFFERING (Executable Embedded SQL Extension)
- LOB ERASE (Executable Embedded SQL Extension)
- LOB FILE CLOSE ALL (Executable Embedded SQL Extension)
- LOB FILE SET (Executable Embedded SQL Extension)
- LOB FLUSH BUFFER (Executable Embedded SQL Extension)
- LOB FREE TEMPORARY (Executable Embedded SQL Extension)
- LOB LOAD (Executable Embedded SQL Extension)
- LOB OPEN (Executable Embedded SQL Extension)
- LOB READ (Executable Embedded SQL Extension)
- LOB TRIM (Executable Embedded SQL Extension)
- LOB WRITE (Executable Embedded SQL Extension)
- OPEN (Executable Embedded SQL)
- OPEN DESCRIPTOR (Executable Embedded SQL)
- PREPARE (Executable Embedded SQL)
- ROLLBACK (Executable Embedded SQL)
- SAVEPOINT (Executable Embedded SQL)
- SET DESCRIPTOR (Executable Embedded SQL)
- SELECT (Executable Embedded SQL)
- UPDATE (Executable Embedded SQL)
- VAR (Oracle Embedded SQL Directive)
- WHENEVER (Embedded SQL Directive)

# E.1 Summary of Precompiler Directives and Embedded SQL Statements

Embedded SQL statements place DDL, DML, and Transaction Control statements within a procedural language program. Embedded SQL is supported by the Oracle



Precompilers. Table E-2 provides a functional summary of the embedded SQL statements and directives.

The *Source/Type* column in Table E-2 is displayed in the format *source/type* where:

Table E-1 Source/Type Column Meaning

SQL Statements	Directives
source	Is either standard SQL (S) or an Oracle extension (O).
type	Is either an executable (E) statement or a directive (D).

Table E-2 Precompiler Directives and Embedded SQL Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
ALLOCATE	O/E	To allocate memory for a cursor variable, LOB locator or ROWID.
ALLOCATE DESCRIPTOR	S/E	To allocate a descriptor for ANSI dynamic SQL.
CALL	S/E	Call a stored procedure.
CLOSE	S/E	To disable a cursor.
COMMIT	S/E	To make all database changes permanent.
CONNECT	O/E	To log on to a database instance.
CONTEXT ALLOCATE	O/E	To allocate memory for a SQLLIB runtime context.
CONTEXT FREE	O/E	To free memory for a SQLLIB runtime context.
CONTEXT USE	O/E	To specify a SQLLIB runtime context.
DEALLOCATE DESCRIPTOR	S/E	To deallocate a descriptor area to free memory.
DECLARE CURSOR	S/D	To declare a cursor, associating it with a query.
DECLARE DATABASE	O/D	To declare an identifier for a nondefault database to be accessed in subsequent embedded SQL statements.
DECLARE STATEMENT	S/D	To assign a SQL variable name to a SQL statement.
DECLARE TABLE	O/D	To declare the table structure for semantic checking of embedded SQL statements by the Oracle Precompiler.
DELETE	S/E	To remove rows from a table or from a view's base table.
DESCRIBE	S/E	To initialize a descriptor, a structure holding host variable descriptions.
DESCRIBE DECRIPTOR	S/E	To obtain information about an ANSI SQL statement, and store it in a descriptor.
ENABLE THREADS	O/E	To initialize a process that supports multiple threads.
EXECUTEEND-EXEC	O/E	To execute an anonymous PL/SQL block.
EXECUTE	S/E	To execute a prepared dynamic SQL statement.
EXECUTE DESCRIPTOR	S/E	To execute a prepared statement using ANSI Dynamic SQL.
EXECUTE IMMEDIATE	S/E	To prepare and execute a SQL statement with no host variables.
FETCH	S/E	To retrieve rows selected by a query.
FETCH DESCRIPTOR	S/E	To retrieve rows selected by a query using ANSI Dynamic SQL.
FREE	S/E	To free memory used by a cursor, LOB locator, or ROWID.



Table E-2 (Cont.) Precompiler Directives and Embedded SQL Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
GET DESCRIPTOR	S/E	To move information from an ANSI SQL descriptor area into host variables.
INSERT	S/E	To add rows to a table or to a view's base table.
LOB APPEND	O/E	To append a LOB to the end of another IOB.
LOB ASSIGN	O/E	To assign a LOB or BFILE locator to another locator.
LOB CLOSE	O/E	To close an open LOB or BFILE.
LOB COPY	O/E	To copy all or part of a LOB value into another LOB.
LOB CREATE TEMPORARY	O/E	To create a temporary LOB.
LOB DESCRIBE	O/E	To retrieve attributes from a LOB.
LOB DISABLE BUFFERING	O/E	To disable LOB buffering.
LOB ENABLE BUFFERING	O/E	To enable LOB buffering.
LOB ERASE	O/E	To erase a given amount of LOB data starting from a given offset.
LOB FILE CLOSE ALL	O/E	To close all open BFILE.
LOB FILE SET	O/E	To set DIRECTORY and FILENAME in a BFILE locator.
LOB FLUSH BUFFER	O/E	To write the LOB buffers to the database server.
LOB FREE TEMPORARY	O/E	To free temporary space for the LOB locator.
LOB LOAD	O/E	To copy all or part of a BFILE into an internal LOB.
LOB OPEN	O/E	To open a LOB or BFILE to read or read/write access.
LOB READ	O/E	To read all or part of a LOB or BFILE into a buffer.
LOB TRIM	O/E	To truncate a lob value.
LOB WRITE	O/E	To write the contents of a buffer to a LOB.
OPEN	S/E	To execute the query associated with a cursor.
OPEN DESCRIPTOR	S/E	To execute the query associated with a cursor in ANSI Dynamic SQL.
PREPARE	S/E	To parse a dynamic SQL statement.
ROLLBACK	S/E	To end the current transaction and discard all changes.
SAVEPOINT	S/E	To identify a point in a transaction to which you can later roll back.
SELECT	S/E	To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.
SET DESCRIPTOR	S/E	To set information in the ANSI SQL descriptor area from host variables.
UPDATE	S/E	To change existing values in a table or in a view's base table.
VAR	O/D	To override the default datatype and assign a specific Oracle external datatype to a host variable.
WHENEVER	S/D	To specify handling for error and warning conditions.



# E.2 About the Statement Descriptions

The directives, and statements appear alphabetically. The description of each contains the following sections:

Directives	Description
Purpose	Describes the basic uses of the statement.
Prerequisites	Lists privileges you must have and steps that you must take before using the statement. Unless otherwise noted, most statements also require that the database be open by your instance.
Syntax	Shows the syntax diagram with the keywords and parameters of the statement.
Keywords and Parameters	Describes the purpose of each keyword and parameter.
Usage Notes	Discusses how and when to use the statement.
Prerequisites	Lists privileges you must have and steps that you must take before using the statement. Unless otherwise noted, most statements also require that the database be open by your instance.
Syntax	Shows the syntax diagram with the keywords and parameters of the statement.

# E.3 How to Read Syntax Diagrams

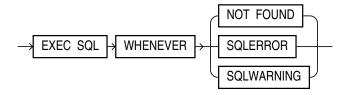
Syntax diagrams are used to illustrate embedded SQL syntax. They are drawings that depict valid syntax.

Trace each diagram from left to right, in the direction shown by the arrows.

Statements keywords appear in UPPER CASE inside rectangles. Type them exactly as shown in the rectangles. Parameters appear in lower case inside ovals. Variables are used for the parameters. Operators, delimiters, and terminators appear inside circles.

If the syntax diagram has more than one path, you can choose any path to travel.

If you have the choice of more than one keyword, operator, or parameter, your options appear in a vertical list. In the following example, you can travel down the vertical line as far as you like, then continue along any horizontal line:



According to the diagram, all of the following statements are valid:

```
EXEC SQL WHENEVER NOT FOUND ...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```



### E.3.1 Statement Terminator

In all Pro\*COBOL EXEC SQL diagrams, each statement is understood to end with the token *END-EXEC*.

## E.3.2 Required Keywords and Parameters

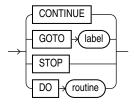
Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the main path, that is, on the horizontal line you are currently traveling. In the following example, cursor is a required parameter:



If there is a cursor named *EMPCURSOR*, then, according to the diagram, the following statement is valid:

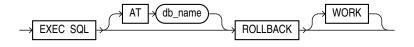
EXEC SQL CLOSE EMPCURSOR END-EXEC.

If any of the keywords or parameters in a vertical list appears on the main path, one of them is required. That is, you must choose one of the keywords or parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the four actions:



## E.3.3 Optional Keywords and Parameters

If keywords and parameters appear in a vertical list preceding the main path, they are optional. In the following example, instead of traveling down a vertical line, you can continue along the main path:

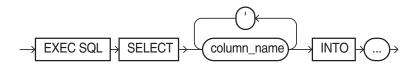


If there is a database named *oracle2*, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL ROLLBACK END-EXEC.
EXEC SQL ROLLBACK WORK END-EXEC.
EXEC SQL AT ORACLE2 ROLLBACK END-EXEC.
```

## E.3.4 Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, *column\_name* is inside a loop. So, after choosing one column name, you can go back repeatedly to choose another.

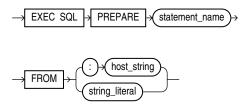


If DEBIT, CREDIT, and BALANCE are column names, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

## E.3.5 Multi-part Diagrams

Read a multi-part diagram as if all the main paths were joined end-to-end. The following example is a two-part diagram:



According to the diagram, the following statement is valid:

```
EXEC SQL PREPARE statement name FROM :host string END-EXEC.
```

## E.3.6 Oracle Names

The names of Oracle database objects, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs (\$), pound signs (#), and underscores (\_).

However, if a name is enclosed by quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle names are not case-sensitive except when enclosed by quotation marks.



# E.4 ALLOCATE (Executable Embedded SQL Extension)

### **Purpose**

To allocate a cursor variable to be referenced in a PL/SQL block, or to allocate a LOB locator, or a ROWID .

### **Prerequisites**

A cursor variable (see Embedded PL/SQL) of type SQL-CURSOR must be declared before allocating memory for the cursor variable.

### **Syntax**



### **Keywords and Parameters**

Keywords and Parameters	Description
cursor_variable	A cursor variable of type SQL-CURSOR
host_ptr	A variable of type SQL-ROWID for a ROWID, or SQL-BLOB, SQL-CLOB, or SQL-NCLOB for a LOB

### **Usage Notes**

Whereas a cursor is static, a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query.

#### **Example**

This partial example illustrates the use of the ALLOCATE statement:

```
01 EMP-CUR SQL-CURSOR.
01 EMP-REC.
...
EXEC SQL ALLOCATE :EMP-CUR END-EXEC.
```

### **Related Topics**

CLOSE (Executable Embedded SQL).

EXECUTE (Executable Embedded SQL).

FETCH (Executable Embedded SQL).

FREE (Executable Embedded SQL Extension).

# E.5 ALLOCATE DESCRIPTOR (Executable Embedded SQL)

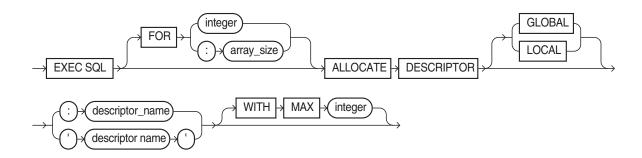
### **Purpose**

An ANSI dynamic SQL statement that allocates a descriptor.

### **Prerequisites**

None.

### **Syntax**



### **Keywords and Parameters**

Keywords and Parameters	Description
array_size	Host variable containing number of rows to be processed.
integer	Number of rows to be processed.
descriptor_name	Host variable containing number of rows to be processed.
descriptor name	Number of rows to be processed.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
WITH MAX integer	Maximum number of host variables. The default is 100.

### **Usage Notes**

Use DYNAMIC=ANSI precompiler option. For information on using this statement, see "ALLOCATE DESCRIPTOR".

### **Example**

```
EXEC SQL FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25 END-EXEC.
```

### **Related Topics**

DESCRIBE DESCRIPTOR (Executable Embedded SQL).

GET DESCRIPTOR (Executable Embedded SQL).

SET DESCRIPTOR (Executable Embedded SQL).

# E.6 CALL (Executable Embedded SQL)

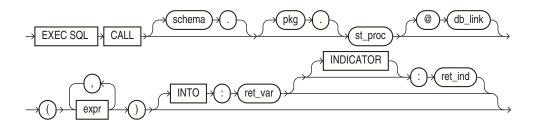
### **Purpose**

To call a stored procedure.

### **Prerequisites**

An active database connection must exist.

### **Syntax**



### **Keywords and Parameters**

Keywords and Parameters	Description
schema	Is the schema containing the procedure. If you omit schema, Oracle assumes the procedure is in your own schema.
pkg	The package where the procedure is stored.
st_proc	The stored procedure to be called.
db_link	The complete or partial name of a database link to a remote database where the procedure is located. For information on referring to database links, see References to Objects in Remote Databases.
expr	The list of expressions that are the parameters of the procedure.
ret_var	The host variable that receives the returned value of a function.
ret_ind	The indicator variable for ret_var.

### **Usage Notes**

For more about this statement, see Calling a Stored PL/SQL or Java Subprogram.

For a complete discussion of stored procedures, see Developing Applications with Multiple Programming Languages.

### **Example**

```
05 EMP-NAME PIC X(10) VARYING.
05 EMP-NUMBER PIC S9(4) COMP VALUE ZERO.
05 SALARY PIC S9(5)V99 COMP-3 VALUE ZERO.
05 D-EMP-NUMBER PIC 9(4).
```



```
ACCEPT D-EMP-NUMBER.

EXEC SQL

CALL mypkge.getsal(:EMP-NUMBER, :D-EMP-NUMBER, :EMP-NAME) INTO :SALARY END-EXEC.
```

### **Related Topics**

None

# E.7 CLOSE (Executable Embedded SQL)

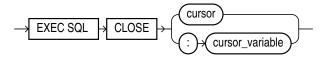
### **Purpose**

To disable a cursor, freeing the resources acquired by opening the cursor, and releasing parse locks.

### **Prerequisites**

The cursor or cursor variable must be open and MODE=ANSI.

### **Syntax**



### **Keywords and Parameters**

Keywords and Parameters	Description
cursor	The cursor to be closed
cursor_variable	The cursor variable to be closed.

### **Usage Notes**

Rows cannot be fetched from a closed cursor. A cursor need not be closed to be reopened. The HOLD\_CURSOR and RELEASE\_CURSOR precompiler options alter the effect of the CLOSE statement. For information on these options, see <u>Precompiler Options</u>.

### **Example**

This example illustrates the use of the CLOSE statement:

```
EXEC SQL CLOSE EMP-CUR END-EXEC.
```

### **Related Topics**

DECLARE CURSOR (Embedded SQL Directive).

OPEN (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).



# E.8 COMMIT (Executable Embedded SQL)

### **Purpose**

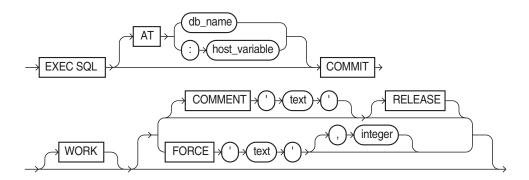
To end your current transaction, making permanent all its changes to the database and optionally freeing all resources and disconnecting from the database server.

### **Prerequisites**

To commit your current transaction, no privileges are necessary.

To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

### **Syntax**



### **Keyword and Parameters**

Keywords and Parameters	Description
AT	Identifies the database to which the COMMIT statement is issued. The database can be identified by either:
db_name	A database identifier declared in a previous DECLARE DATABASE statement or used in a CONNECT statement.
host_variable	If you omit this clause, Oracle issues the statement to your default database.
WORK	Is supported only for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.
COMMENT	Specifies a comment to be associated with the current transaction. The 'text' is a quoted literal of up to 50 characters that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt.
RELEASE	Frees all resources and disconnects the application from the Oracle Server.



<b>Keywords and Parameters</b>	Description
FORCE	Manually commits an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can also use the optional integer to explicitly assign the transaction a system change number (SCN). If you omit the integer, the transaction is committed using the current SCN.

### **Usage Notes**

Always explicitly commit or rollback the last transaction in your program by using the COMMIT or ROLLBACK statement and the RELEASE option. Oracle automatically rolls back changes if the program terminates abnormally.

The COMMIT statement has no effect on host variables or on the flow of control in the program. For more information on this statement, see "Using the COMMIT Statement".

### **Example**

This example illustrates the use of the embedded SQL COMMIT statement:

```
EXEC SQL AT SALESDB COMMIT RELEASE END-EXEC.
```

### **Related Topics**

ROLLBACK (Executable Embedded SQL) .

SAVEPOINT (Executable Embedded SQL) .

# E.9 CONNECT (Executable Embedded SQL Extension)

### **Purpose**

To logon to an Oracle database.

### **Prerequisites**

You must have CREATE SESSION system privilege in the specified database.

### **Syntax**

```
EXEC SQL
CONNECT { :user IDENTIFIED BY :oldpswd | :usr_psw }
[[AT { dbname | :host_variable }] USING :connect_string ]
[ {ALTER AUTHORIZATION :newpswd | IN {SYSDBA | SYSOPER | SYSBACKUP} MODE} ]
END-EXEC.
```



# **Keyword and Parameters**

Keywords and Parameters	Description
user	Specifies your username and password separately.
password	
user_password	Is a single host variable containing the connect string username/password[@dbname].
	To allow Oracle to verify your connection through your operating system, specify "/" as the :user_password value.
AT	Identifies the database to which the connection is made. The database can be identified by either:
db_name	A database identifier declared in a previous DECLARE DATABASE statement.
host_variable	A host variable whose value is a previously declared db_name.
USING	Specifies the Oracle Net Services database specification string used to connect to a nondefault database. If you omit this clause, you are connected to your default database.
ALTER AUTHORIZATION	Change password to the following string.
new_password	New password string.
IN SYSDBA MODE IN SYSOPER MODE IN SYSBACKUP MODE	Connect with SYSDBA, SYSOPER, or SYSBACKUP system privileges. Not allowed when ALTER AUTHORIZATION is used, or precompiler option AUTO_CONNECT is set to YES.

#### **Usage Notes**

A program can have multiple connections, but can only connect once to your default database. For more information on this statement, see: "Concurrent Logons".

# **Example**

The following example illustrate the use of CONNECT:

```
EXEC SQL CONNECT :USERNAME
IDENTIFIED BY :PASSWORD
END-EXEC.
```

You can also use this statement in which the value of :userid is the value of :username and :password separated by a "/" such as 'SCOTT/TIGER':

```
EXEC SQL CONNECT : USERID END-EXEC.
```

# **Related Topics**

COMMIT (Executable Embedded SQL).

DECLARE DATABASE (Oracle Embedded SQL Directive).

ROLLBACK (Executable Embedded SQL).



# E.10 CONTEXT ALLOCATE (Executable Embedded SQL Extension)

#### **Purpose**

To initialize a SQLLIB runtime context that is referenced in an EXEC SQL CONTEXT USE statement

# **Prerequisites**

The runtime context must be declared of type SQL-CONTEXT.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
context	The SQLLIB runtime context for which memory is to be allocated.

# **Usage Notes**

For more information on this statement, see "Embedded SQL Statements and Directives for Runtime Contexts".

# **Example**

This example illustrates the use of a CONTEXT ALLOCATE statement in a Pro\*COBOL embedded SQL program:

EXEC SQL CONTEXT ALLOCATE :ctx1 END-EXEC.

# **Related Topics**

CONTEXT FREE (Executable Embedded SQL Extension).

CONTEXT USE (Oracle Embedded SQL Directive).

# E.11 CONTEXT FREE (Executable Embedded SQL Extension)

#### **Purpose**

To free all memory associated with a runtime context and place a null pointer in the host program variable.



# **Prerequisites**

The CONTEXT ALLOCATE statement must be used to allocate memory for the specified runtime context before the CONTEXT FREE statement can free the memory allocated for it.

### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
context	The allocated runtime context for which the memory is to be deallocated.

### **Usage Notes**

For more information on this statement, see "Embedded SQL Statements and Directives for Runtime Contexts".

# **Example**

This example illustrates the use of a CONTEXT FREE statement in a Pro\*COBOL embedded SQL program:

EXEC SQL CONTEXT FREE :ctx1 END-EXEC.

#### **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension).

CONTEXT USE (Oracle Embedded SQL Directive).

# E.12 CONTEXT USE (Oracle Embedded SQL Directive)

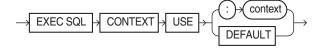
# **Purpose**

To instruct the precompiler to use the specified SQLLIB runtime context on subsequent executable SQL statements

#### **Prerequisites**

The runtime context specified by the CONTEXT USE directive must be previously declared.

# **Syntax**





### **Keywords and Parameters**

Keywords and Parameters	Description
context	The allocated runtime context to use for subsequent executable SQL statements that follow it. For example, after specifying in your source code which context to use (multiple contexts can be allocated), you can connect to the Oracle Server and perform database operations within the scope of that context.
DEFAULT	Indicates that the global context is to be used.

### **Usage Notes**

This statement has no effect on declarative statements such as EXEC SQL INCLUDE or EXEC ORACLE OPTION. It works similarly to the EXEC SQL WHENEVER directive in that it affects all executable SQL statements which positionally follow it in a given source file without regard to standard C scope rules.

For more information on this statement, see "Embedded SQL Statements and Directives for Runtime Contexts".

# **Example**

This example illustrates the use of a CONTEXT USE directive in a Pro\*COBOL program:

EXEC SOL CONTEXT USE :ctx1 END-EXEC.

# **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension).

CONTEXT FREE (Executable Embedded SQL Extension).

# E.13 DEALLOCATE DESCRIPTOR (Embedded SQL Statement)

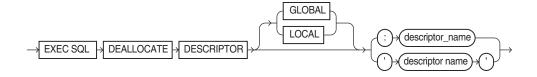
### **Purpose**

An ANSI dynamic SQL statement that deallocates a descriptor area to free memory.

# **Prerequisites**

The descriptor specified by the DEALLOCATE DESCRIPTOR statement must be previously allocated using the ALLOCATE DESCRIPTOR statement.

# **Syntax**





### **Keywords and Parameters**

Keywords and Parameters	Description
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
descriptor_name 'descriptor name'	Host variable containing the name of the allocated ANSI descriptor. Name of the allocated ANSI descriptor.

# **Usage Notes**

Use DYNAMIC=ANSI precompiler option.

For more information on this statement, see "DEALLOCATE DESCRIPTOR".

#### **Example**

EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES' END-EXEC.

# **Related Topics**

ALLOCATE DESCRIPTOR (Executable Embedded SQL).

DESCRIBE DESCRIPTOR (Executable Embedded SQL).

GET DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

SET DESCRIPTOR (Executable Embedded SQL).

# E.14 DECLARE CURSOR (Embedded SQL Directive)

# **Purpose**

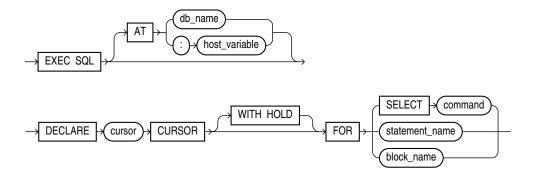
To declare a cursor, giving it a name and associating it with a SQL statement or a PL/SQL block.

# **Prerequisites**

If you associate the cursor with an identifier for a SQL statement or PL/SQL block, you must have declared this identifier in a previous DECLARE STATEMENT statement.



# **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database on which the cursor is declared. The database can be identified by either:
db_name	Database identifier declared in a previous DECLARE DATABASE statement.
host_variable	Host variable whose value is a previously declared <i>db_name</i> . If you omit this clause, Oracle declares the cursor on your default database.
cursor	Name of the cursor to be declared.
WITH HOLD	Cursor remains open after a COMMIT. The cursor must not be declared for UPDATE.
SELECT statement	Is a SELECT statement to be associated with the cursor. The following statement cannot contain an INTO clause.
statement_name	Identifies a SQL statement or PL/SQL block to be associated with the cursor. The <i>statement_name</i> or <i>block_name</i> must be previously declared in a DECLARE STATEMENT statement.

# **Usage Notes**

You must declare a cursor before referencing it in other embedded SQL statements. The scope of a cursor declaration is global within its precompilation unit and the name of each cursor must be unique in its scope. You cannot declare two cursors with the same name in a single precompilation unit.

You can reference the cursor in the WHERE clause of an UPDATE or DELETE statement using the CURRENT OF syntax, if the cursor has been opened with an OPEN statement and positioned on a row with a FETCH statement. For more information on this statement, see "WITH HOLD Clause in DECLARE CURSOR Statements".

By default, held cursors are closed after ROLLBACK to conform to SQL standard behavior. To have held cursors remain open after ROLLBACK for backward compatibility, set the command-line option, CWH\_SQL99, to NO.

New para added for feature 25546-1



# **Example**

This example illustrates the use of a DECLARE CURSOR statement:

```
EXEC SQL DECLARE EMPCURSOR CURSOR
FOR SELECT ENAME, EMPNO, JOB, SAL
FROM EMP
WHERE DEPTNO = :DEPTNO
END-EXEC.
```

# **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE DATABASE (Oracle Embedded SQL Directive).

DECLARE STATEMENT (Embedded SQL Directive) .

DELETE (Executable Embedded SQL).

FETCH (Executable Embedded SQL).

OPEN (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

SELECT (Executable Embedded SQL).

UPDATE (Executable Embedded SQL).

# E.15 DECLARE DATABASE (Oracle Embedded SQL Directive)

#### **Purpose**

To declare an identifier for a nondefault database to be accessed in subsequent embedded SQL statements.

# **Prerequisites**

You must have access to a username on the nondefault database.

#### **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
db_name	The identifier established for the nondefault database.



You declare a *db\_name* for a nondefault database so that other embedded SQL statements can refer to that database using the AT clause. Before issuing a CONNECT statement with an AT clause, you must declare a *db\_name* for the nondefault database with a DECLARE DATABASE statement.

For more information on this statement, see "Using Username/Password".

# **Example**

This example illustrates the use of a DECLARE DATABASE directive:

EXEC SOL DECLARE ORACLE3 DATABASE END-EXEC.

### **Related Topics**

COMMIT (Executable Embedded SQL)

CONNECT (Executable Embedded SQL Extension).

DECLARE CURSOR (Embedded SQL Directive).

DECLARE STATEMENT (Embedded SQL Directive) .

DELETE (Executable Embedded SQL).

EXECUTE (Executable Embedded SQL).

EXECUTE IMMEDIATE (Executable Embedded SQL).

INSERT (Executable Embedded SQL).

SELECT (Executable Embedded SQL).

UPDATE (Executable Embedded SQL).

# E.16 DECLARE STATEMENT (Embedded SQL Directive)

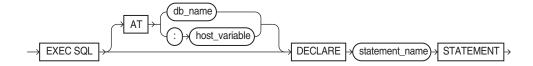
# **Purpose**

To declare an identifier for a SQL statement or PL/SQL block to be used in other embedded SQL statements.

# **Prerequisites**

None.

# **Syntax**





# **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database on which the SQL statement or PL/SQL block is declared. The database can be identified by either:
db_name	Database identifier declared in a previous DECLARE DATABASE statement.
host_variable	Host variable whose value is a previously declared <i>db_name</i> . If you omit this clause, Oracle Database declares the SQL statement or PL/SQL block on your default database.
statement_name	Is the declared identifier for the statement or PL/SQL block.

# **Usage Notes**

You must declare an identifier for a SQL statement or PL/SQL block with a DECLARE STATEMENT statement only if a DECLARE CURSOR statement referencing the identifier appears physically (not logically) in the embedded SQL program before the PREPARE statement that parses the statement or block and associates it with its identifier.

The scope of a statement declaration is global within its precompilation unit, like a cursor declaration. For more information on this statement, see "DECLARE".

#### **Example I**

This example illustrates the use of the DECLARE STATEMENT statement:

```
EXEC SQL AT REMOTEDB

DECLARE MYSTATEMENT STATEMENT
END-EXEC.

EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING
END-EXEC.

EXEC SQL EXECUTE MYSTATEMENT END-EXEC.
```

#### **Example II**

In this example, the DECLARE STATEMENT statement is required because the DECLARE CURSOR statement precedes the PREPARE statement:

```
EXEC SQL DECLARE MYSTATEMENT STATEMENT END-EXEC.

EXEC SQL DECLARE EMPCURSOR CURSOR FOR MYSTATEMENT END-EXEC.

EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC.

...
```

#### **Related Topics**

CLOSE (Executable Embedded SQL) .

DECLARE DATABASE (Oracle Embedded SQL Directive).

FETCH (Executable Embedded SQL).

OPEN (Executable Embedded SQL).



# PREPARE (Executable Embedded SQL).

# E.17 DECLARE TABLE (Oracle Embedded SQL Directive)

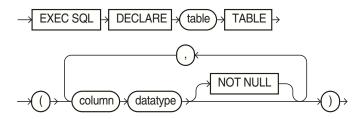
# **Purpose**

To define the structure of a table or view, including each column's datatype, default value, and NULL or NOT NULL specification for semantic checking by the precompiler when option SQLCHECK=SEMANTICS (or FULL).

#### **Prerequisites**

None.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
table	The name of the declared table.
column	A column of the table.
datatype	The datatype of a <i>column</i> . For information on Oracle datatypes, see "The Oracle Database Datatypes".
NOT NULL	Specifies that a column cannot contain nulls.

### **Usage Notes**

Datatypes can only use integers (not expressions) for length, precision, scale. For more information on using this statement, see "Specifying SQLCHECK=SEMANTICS".

## **Example**

The following statement declares the PARTS table with the PARTNO, BIN, and QTY columns:

```
EXEC SQL DECLARE PARTS TABLE

(PARTNO NUMBER NOT NULL,
BIN NUMBER,
QTY NUMBER)

END-EXEC.
```

# **Related Topics**

None.



# E.18 DELETE (Executable Embedded SQL)

# **Purpose**

To remove rows from a table or from a view's base table.

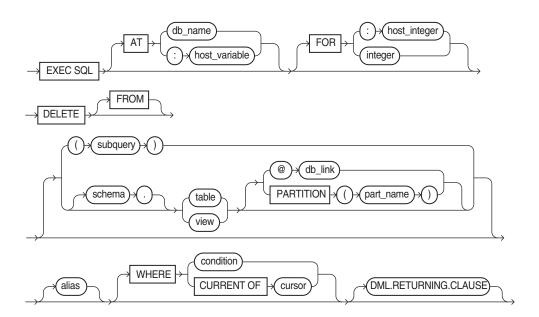
## **Prerequisites**

For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.

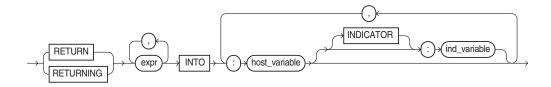
For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

The DELETE ANY TABLE system privilege also enables you to delete rows from any table or any view's base table.

#### **Syntax**



where the DML Returning clause is:





# **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database to which the DELETE statement is issued. The database can be identified by either:
	db_name: a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: a host variable whose value is a previously declared db_name.
	If you omit this clause, the DELETE statement is issued to your default database.
host_integer integer	Limits the number of times the statement is executed if the WHERE clause contains array host variables. If you omit this clause, the database executes the statement once for each component of the smallest array.
schema	The schema containing the table or view. If you omit <i>schema</i> , the database assumes the table or view is in your own schema.
table view	The name of a table from which the rows are to be deleted. If you specify <i>view</i> , the database deletes rows from the view's base table.
dblink	The complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see Basic Elements of Oracle SQL. You can only delete rows from a remote table or view if you are using the database with the distributed option.
	If you omit <i>dblink</i> , the database assumes that the table or view is located on the local database.
part_name	Name of partition in the table
alias	The alias assigned to the table. Aliases are generally used in DELETE statements with correlated queries.
WHERE	Specifies which rows are deleted. If you omit this clause entirely, all rows are deleted from the table or view.
	condition specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of TRUE, FALSE, or UNKNOWN.
	CURRENT OF specifies deletion of the record that was last fetched by the cursor.
DML returning clause	See "DML Returning Clause" for a discussion.

# **Usage Notes**

The host variables in the WHERE clause should be either all scalars or all arrays. If they are scalars, the database executes the DELETE statement only once. If they are arrays, the database executes the statement once for each set of array components. Each execution may delete zero, one, or multiple rows.

Array host variables in the WHERE clause can have different sizes. In this case, the number of times the database executes the statement is determined by the smaller of the following values:

- the size of the smallest array
- the value of the host integer in the optional FOR clause

If no rows satisfy the condition, no rows are deleted and the SQLCODE returns a NOT\_FOUND condition.



The cumulative number of rows deleted is returned through the SQLCA. If the WHERE clause contains array host variables, this value reflects the total number of rows deleted for all components of the array processed by the DELETE statement.

If no rows satisfy the condition, the database returns an error through the SQLCODE of the SQLCA. If you omit the WHERE clause, the database raises a warning flag in the fifth component of SQLWARN in the SQLCA. For more information on this statement and the SQLCA, see "Using the SQL Communications Area".

You can use comments in a DELETE statement to pass instructions, or *hints*, to the Oracle Database optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle Database Performance Tuning Guide*.

# **Example**

This example illustrates the use of the DELETE statement:

```
EXEC SQL DELETE FROM EMP

WHERE DEPTNO = :DEPTNO

AND JOB = :JOB

END-EXEC.

EXEC SQL DECLARE EMPCURSOR CURSOR

FOR SELECT EMPNO, COMM

FROM EMP

END-EXEC.

EXEC SQL OPEN EMPCURSOR END-EXEC.

EXEC SQL FETCH EMPCURSOR

INTO :EMP-NUMBER, :COMMISSION

END-EXEC.

EXEC SQL DELETE FROM EMP

WHERE CURRENT OF EMPCURSOR

END-EXEC.
```

## **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive).

DECLARE STATEMENT (Embedded SQL Directive).

# E.19 DESCRIBE (Executable Embedded SQL)

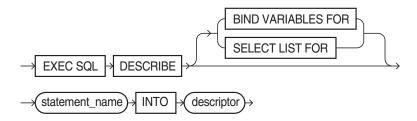
#### **Purpose**

To initialize a descriptor to hold descriptions of host variables for an Oracle dynamic SQL statement or PL/SQL block.

#### **Prerequisites**

You must have prepared the SQL statement or PL/SQL block in a previous embedded SQL PREPARE statement.

# **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
BIND VARIABLES FOR	Initializes the descriptor to hold information about the input variables for the SQL statement or PL/SQL block.
SELECT LIST FOR	Initializes the descriptor to hold information about the select list of a SELECT statement.
-	The default is SELECT LIST FOR.
statement_name	Identifies a SQL statement or PL/SQL block previously prepared with a PREPARE statement.
descriptor	The name of the descriptor to be initialized.

# **Usage Notes**

You must issue a DESCRIBE statement before manipulating the bind or select descriptor within an embedded SQL program.

You cannot describe both input variables and output variables into the same descriptor.

The number of variables found by a DESCRIBE statement is the total number of placeholders in the prepare SQL statement or PL/SQL block, rather than the total number of uniquely named placeholders. For more information on this statement, see "The DESCRIBE Statement".

# **Example**

This example illustrates the use of the DESCRIBE statement in a Pro\*COBOL embedded SQL program:

```
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC.

EXEC SQL DECLARE EMPCURSOR

FOR SELECT EMPNO, ENAME, SAL, COMM

FROM EMP

WHERE DEPTNO = :DEPT-NUMBER

END-EXEC.

EXEC SQL DESCRIBE BIND VARIABLES FOR MYSTATEMENT

INTO BINDDESCRIPTOR

END-EXEC.

EXEC SQL OPEN EMPCURSOR

USING BINDDESCRIPTOR

END-EXEC.
```



EXEC SQL DESCRIBE SELECT LIST FOR MY-STATEMENT INTO SELECTDESCRIPTOR
END-EXEC.
EXEC SQL FETCH EMPCURSOR
INTO SELECTDESCRIPTOR
END-EXEC.

# **Related Topics**

PREPARE (Executable Embedded SQL).

# E.20 DESCRIBE DESCRIPTOR (Executable Embedded SQL)

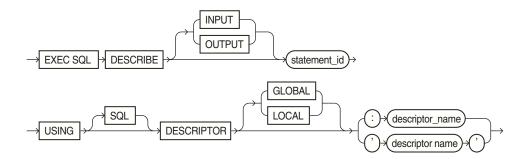
# **Purpose**

Used to obtain information about an ANSI SQL statement, and to store it in a descriptor.

# **Prerequisites**

You must have prepared the SQL statement in a previous embedded SQL PREPARE statement.

# **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
statement_id	The name of the previously prepared SQL statement or PL/SQL block. OUTPUT is the default.
desc_name	Host variable containing the name of the descriptor that will hold information about the SQL statement.
'descriptor name'	Name of the descriptor
GLOBAL   LOCAL	LOCAL is the default. It means file scope, as opposed to GLOBAL, which means application scope.



Use DYNAMIC=ANSI precompiler option. Only COUNT and NAME are implemented for the INPUT descriptor.

The number of variables found by a DESCRIBE statement is the total number of place-holders in the prepare SQL statement or PL/SQL block, rather than the total number of uniquely named place-holders. For more information on this statement, see ANSI Dynamic SQL.

#### **Example**

```
EXEC SQL PREPARE s FROM :my_stament END-EXEC.
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' END-EXEC.
```

## **Related Topics**

ALLOCATE DESCRIPTOR (Executable Embedded SQL).

DEALLOCATE DESCRIPTOR (Embedded SQL Statement).

GET DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

SET DESCRIPTOR (Executable Embedded SQL).

# E.21 ENABLE THREADS (Executable Embedded SQL Extension)

#### **Purpose**

To initialize a process that supports multiple threads.

### **Prerequisites**

You must be developing a precompiler application for and compiling it on a platform that supports multithreaded applications, and THREADS=YES must be specified on the command line.

# **Syntax**



# **Keywords and Parameters**

None.

# **Usage Notes**

The ENABLE THREADS statement must be executed once, and only once, before any other executable SQL statement and before spawning any threads. This statement does not require a host-variable specification.



# **Example**

This example illustrates the use of the ENABLE THREADS statement in a Pro\*COBOL program:

EXEC SQL ENABLE THREADS END-EXEC.

# **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension).

CONTEXT FREE (Executable Embedded SQL Extension).

CONTEXT USE (Oracle Embedded SQL Directive).

# E.22 EXECUTE ... END-EXEC (Executable Embedded SQL Extension)

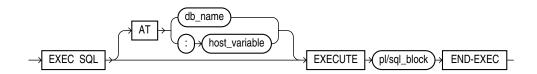
### **Purpose**

To embed an anonymous PL/SQL block into an Oracle Pro\*COBOL program.

# **Prerequisites**

None.

# **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database on which the PL/SQL block is executed. The database can be identified by either:
db_name	A database identifier declared in a previous DECLARE DATABASE statement.
host_variable	A host variable whose value is a previously declared <i>db_name</i> . If you omit this clause, the PL/SQL block is executed on your default database.
pl/sql_block	For information on PL/SQL, including how to write PL/SQL blocks, see the <i>Oracle Database PL/SQL Language Reference</i> .
END-EXEC	Must appear after the embedded PL/SQL block.



Since the Oracle Precompilers treat an embedded PL/SQL block like a single embedded SQL statement, you can embed a PL/SQL block anywhere in an Oracle Precompiler program that you can embed a SQL statement. For more information on embedding PL/SQL blocks in Oracle Precompiler programs, see Embedded PL/SQL.

## **Example**

Placing this EXECUTE statement in an Oracle Precompiler program embeds a PL/SQL block in the program:

```
EXEC SQL EXECUTE
BEGIN

SELECT ENAME, JOB, SAL

INTO :EMP-NAME:IND-NAME, :JOB-TITLE, :SALARY
FROM EMP

WHERE EMPNO = :EMP-NUMBER;

IF :EMP-NAME:IND-NAME IS NULL
THEN RAISE NAME-MISSING;
END IF;
END;
END-EXEC.
```

#### **Related Topics**

EXECUTE IMMEDIATE (Executable Embedded SQL).

# E.23 EXECUTE (Executable Embedded SQL)

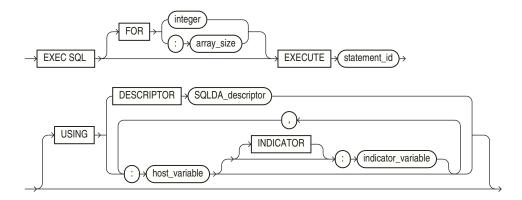
# **Purpose**

In Oracle dynamic SQL, to execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block that has been previously prepared with an embedded SQL PREPARE statement.

# **Prerequisites**

You must first prepare the SQL statement or PL/SQL block with an embedded SQL PREPARE statement.

# **Syntax**



### **Keywords and Parameters**

Keywords and Parameters	Description
FOR :array_size	Host variable containing the number of rows to be processed.
FOR integer	Number of rows to be processed.
	Limits the number of times the statement is executed when the USING clause contains array host variables If you omit this clause, Oracle executes the statement once for each component of the smallest array.
statement_id	A precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE statement to associate the precompiler identifier with the statement or PL/SQL block.
USING DESCRIPTOR	Uses an Oracle descriptor.
SQLDA_descriptor	CANNOT be used together with an ANSI descriptor (INTO clause).
USING	Specifies a list of host variables with optional indicator variables that Oracle substitutes as input variables into the statement to be executed. The host and indicator variables must be either all scalars or all arrays.
host_variable	Host variables.
indicator_variable	Indicator variables.

# **Usage Note**

For more information on this statement, see Oracle Dynamic SQL.

# **Example**

This example illustrates the use of the EXECUTE statement in a Pro\*COBOL embedded SQL program:

```
EXEC SQL PREPARE MY-STATEMENT FROM MY-STRING END-EXEC. EXEC SQL EXECUTE MY-STATEMENT USING :MY-VAR END-EXEC.
```

# **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive).

PREPARE (Executable Embedded SQL).

# E.24 EXECUTE DESCRIPTOR (Executable Embedded SQL

# **Purpose**

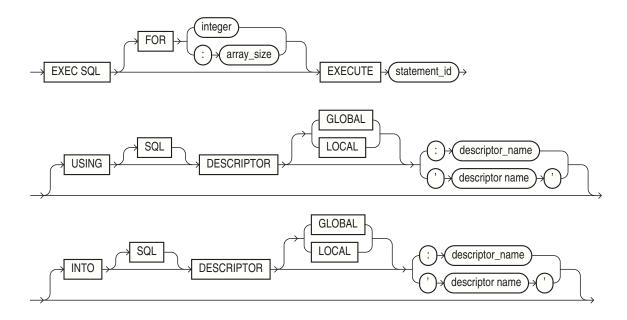
In ANSI SQL Method 4, to execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block that has been previously prepared with an embedded SQL PREPARE statement.

# **Prerequisites**

You must first prepare the SQL statement or PL/SQL block with an embedded SQL PREPARE statement.



# **Syntax**



# **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
FOR :array_size	Host variable containing the number of rows to be processed.
FOR integer	Number of rows to be processed.
	Limits the number of times the statement is executed. Oracle executes the statement once for each component of the smallest array.
statement_id	A precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE statement to associate the precompiler identifier with the statement or PL/SQL block.
USING	An ANSI input descriptor.
descriptor_name	Host variable containing name of the input descriptor.
descriptor name	Name of the input descriptor.
INTO	An ANSI output descriptor.
descriptor_name	Host variable containing the name of the output descriptor.
descriptor name	Name of the output descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.

# **Usage Notes**

For more information on this statement, see "EXECUTE".

# **Examples**

The ANSI dynamic SQL Method 4 enables DML RETURNING in a SELECT to be supported by the INTO clause in EXECUTE:

EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES' END-EXEC.

# **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive).

PREPARE (Executable Embedded SQL).

# E.25 EXECUTE IMMEDIATE (Executable Embedded SQL)

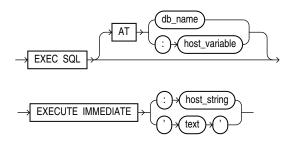
# **Purpose**

To prepare and execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block containing no host variables.

# **Prerequisites**

None.

# **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database on which the SQL statement or PL/SQL block is executed. The database can be identified by either <i>db_name</i> or <i>host_variable</i> .
	If you omit this clause, the statement or block is executed on your default database.
db_name	A database identifier declared in a previous DECLARE DATABASE statement.
host_variable	A host variable whose value is a previously declared db_name.
host_string	A host variable whose value is the SQL statement or PL/SQL block to be executed.
text	A text literal containing the SQL statement or PL/SQL block to be executed. The quotes may be omitted.
	The SQL statement can only be a DELETE, INSERT, or UPDATE statement.



When you issue an EXECUTE IMMEDIATE statement, the database parses the specified SQL statement or PL/SQL block, checking for errors, and executes it. If any errors are encountered, they are returned in the SQLCODE component of the SQLCA.

For more information on this statement, see "The EXECUTE IMMEDIATE Statement".

#### **Example**

This example illustrates the use of the EXECUTE IMMEDIATE statement:

```
EXEC SQL

EXECUTE IMMEDIATE 'DELETE FROM EMP WHERE EMPNO = 9460'
END-EXEC.
```

# **Related Topics**

PREPARE (Executable Embedded SQL).

EXECUTE (Executable Embedded SQL).

# E.26 FETCH (Executable Embedded SQL)

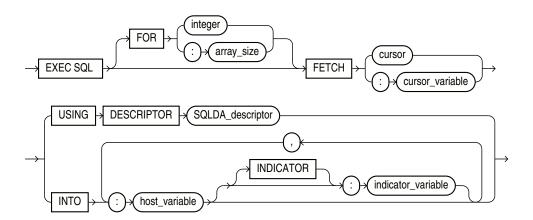
# **Purpose**

To retrieve one or more rows returned by a query, assigning the select list values to host variables. For ANSI Dynamic SQL Method 4, see "FETCH DESCRIPTOR (Executable Embedded SQL)".

# **Prerequisites**

You must first open the cursor with an the OPEN statement.

# **Syntax**





#### **Keywords and Parameters**

Keywords and Parameters	Description
FOR :array_size	Host variable containing the number of rows to be processed.
FOR integer	Number of rows to be processed.
•	Limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle fetches enough rows to fill the smallest array.
cursor	A cursor that is declared by a DECLARE CURSOR statement. The FETCH statement returns one of the rows selected by the query associated with the cursor.
cursor_variable	A cursor variable is allocated an ALLOCATE statement. The FETCH statement returns one of the rows selected by the query associated with the cursor variable.
INTO	Specifies a list of host variables and optional indicator variables into which data is fetched. These host variables and indicator variables must be declared within the program.
USING SQLDA_variable	Specifies the Oracle descriptor referenced in a previous DESCRIBE statement. Only use this clause with dynamic embedded SQL, method 4. The USING clause does not apply when a cursor variable is used.
host_variable	The host variable into which data is returned.
indicator_variable	The host indicator variable.

#### **Usage Notes**

The FETCH statement reads the rows of the active set and names the output variables which contain the results. Indicator values are set to -1 if their associated host variable is null.

The number of rows retrieved is specified by the size of the output host variables or the value specified in the FOR clause. The host variables to receive the data should be either all scalars or all arrays. If they are scalars, Oracle fetches only one row. If they are arrays, Oracle fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle fetches is determined by the smaller of the following values:

- The size of the smallest array
- The value of the *host integer* in the optional FOR clause

Of course, the number of rows fetched can be further limited by the number of rows that actually satisfy the query.

If a FETCH statement does not retrieve all rows returned by the query, the cursor is positioned on the next returned row. When the last row returned by the query has been retrieved, the next FETCH statement results in an warning code returned in the SQLCODE element of the SQLCA.

If the array is not completely filled then the warning is issued and you should check SQLERRD(3) to see how many rows were actually fetched.



Note that the FETCH statement does not contain an AT clause. You must specify the database accessed by the cursor in the DECLARE CURSOR statement.

You can only move forward through the active set with FETCH statements. If you want to revisit any of the previously fetched rows, you must reopen the cursor and fetch each row in turn. If you want to change the active set, you must assign new values to the input host variables in the cursor's query and reopen the cursor.

#### **Example**

This example illustrates the FETCH statement in a Pro\*COBOL embedded SQL program:

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
SELECT JOB, SAL FROM EMP WHERE DEPTNO = 30
END-EXEC.
...
EXEC SQL WHENEVER NOT FOUND GOTO ...
LOOP.
EXEC SQL FETCH EMPCURSOR INTO :JOB-TITLE1, :SALARY1 END-EXEC.
EXEC SQL FETCH EMPCURSOR INTO :JOB-TITLE2, :SALARY2 END-EXEC.
...
GO TO LOOP.
...
```

# **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE CURSOR (Embedded SQL Directive).

OPEN (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.27 FETCH DESCRIPTOR (Executable Embedded SQL)

#### **Purpose**

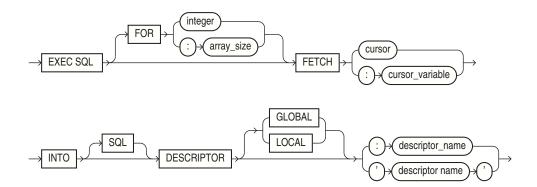
To retrieve one or more rows returned by a query, assigning the select list values to host variables. Used in ANSI Dynamic SQL Method 4.

#### **Prerequisites**

You must first open the cursor with an the OPEN statement.



# **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
FOR :array_size	Host variable containing the number of rows to be processed.
FOR integer	Number of rows to be processed.
	Limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle fetches enough rows to fill the smallest array.
cursor	A cursor that is declared by a DECLARE CURSOR statement. The FETCH statement returns one of the rows selected by the query associated with the cursor.
cursor_variable	A cursor variable is allocated an ALLOCATE statement. The FETCH statement returns one of the rows selected by the query associated with the cursor variable.
INTO	Specifies a list of host variables and optional indicator variables into which data is fetched. These host variables and indicator variables must be declared within the program.
INTO 'descriptor name'	Name of the output ANSI descriptor.
INTO :descriptor_name	Host variable containing the name of the output descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.

# **Usage Notes**

The number of rows retrieved is specified by the size of the output host variables and the value specified in the FOR clause. The host variables to receive the data should be either all scalars or all arrays. If they are scalars, Oracle fetches only one row. If they are arrays, Oracle fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle fetches is determined by the smaller of the following values:

- The size of the smallest array
- The value of the array\_size in the optional FOR clause



• Of course, the number of rows fetched can be further limited by the number of rows that actually satisfy the query.

If a FETCH statement does not retrieve all rows returned by the query, the cursor is positioned on the next returned row. When the last row returned by the query has been retrieved, the next FETCH statement results in a warning code returned in the SQLCODE element of the SQLCA.

If the array is not completely filled then the warning is issued and you should check SQLERRD(3) to see how many rows were actually fetched.

Note that the FETCH statement does not contain an AT clause. You must specify the database accessed by the cursor in the DECLARE CURSOR statement.

You can only move forward through the active set with FETCH statements. If you want to revisit any of the previously fetched rows, you must reopen the cursor and fetch each row in turn. If you want to change the active set, you must assign new values to the input host variables in the cursor's query and reopen the cursor.

Use DYNAMIC=ANSI precompiler option for the ANSI SQL Method 4 application. For more information, see "FETCH" for the ANSI SQL Method 4 application.

### **Example**

```
EXEC SQL ALLOCATE DESCRIPTOR 'output_descriptor' END-EXEC.

EXEC SQL PREPARE S FROM :dyn_statement END-EXEC.

EXEC SQL DECLARE mycursor CURSOR FOR S END-EXEC.

EXEC SQL FETCH mycursor INTO DESCRIPTOR 'output_descriptor' END-EXEC.
```

## **Related Topics**

PREPARE statement.

# E.28 FREE (Executable Embedded SQL Extension)

#### **Purpose**

To free memory used by a cursor, LOB locator, or ROWID.

# **Prerequisites**

The memory has to have been already allocated.

# **Syntax**





# **Keywords and Parameters**

Keywords and Parameters	Description
cursor_variable	A cursor variable that has previously been allocated in an ALLOCATE statement. It is of type SQL-CURSOR.
	The FETCH statement returns one of the rows selected by the query associated with the cursor variable.
host_ptr	A variable of type SQL-ROWID for a ROWID, or SQL-BLOB, SQL-CLOB, or SQL-NCLOB for a LOB.

# **Usage Notes**

See "Cursors" and "Cursor Variables".

# **Example**

```
* CURSOR VARIABLE EXAMPLE
...
01 CUR SQL-CURSOR.
...
EXEC SQL ALLOCATE :CUR END-EXEC.
...
EXEC SQL CLOSE :CUR END-EXEC.
EXEC SQL FREE :CUR END-EXEC.
```

# **Related Topics**

ALLOCATE (Executable Embedded SQL Extension).

CLOSE (Executable Embedded SQL) .

DECLARE CURSOR (Embedded SQL Directive).

# E.29 GET DESCRIPTOR (Executable Embedded SQL)

# **Purpose**

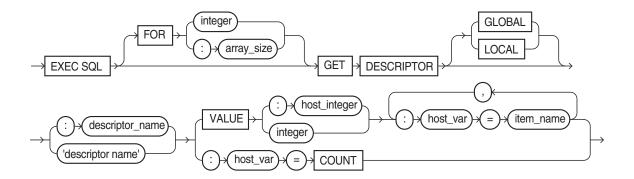
To obtain information about host variables from a SQL descriptor area.

# **Prerequisites**

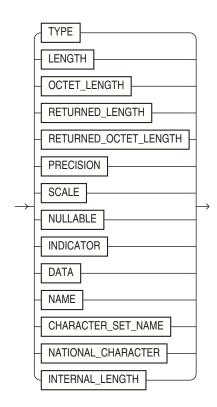
Use only with value semantics and ANSI dynamic SQL Method 4.



# **Syntax**



where item name can be one of these choices:



# **Keywords and Parameters**

Keywords and Parameters	Description
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed.
:descriptor_name	Host variable containing the name of the allocated ANSI descriptor.

Keywords and Parameters	Description
'descriptor name'	Name of the allocated ANSI descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
host_var=COUNT	Host variable containing the total number of input or output variables.
integer	Total number of input or output variables.
VALUE :host_integer	Host variable containing the position of the referenced input or output variable.
VALUE integer	The position of the referenced input or output variable.
host_var	Host variable which will receive the item's value.
item_name	The <i>item_name</i> is found in Table 10-4, and Table 10-5, under the "Descriptor Item Name" column heading.

Use DYNAMIC=ANSI precompiler option. The array size clause can be used with DATA, RETURNED\_LENGTH, and INDICATOR item names. See "GET DESCRIPTOR".

## **Example**

EXEC SQL GET DESCRIPTOR GLOBAL 'mydesc' :mydesc\_num\_vars = COUNT END-EXEC.

# **Related Topics**

ALLOCATE DESCRIPTOR (Executable Embedded SQL).

DESCRIBE DESCRIPTOR (Executable Embedded SQL).

SET DESCRIPTOR (Executable Embedded SQL).

# E.30 INSERT (Executable Embedded SQL)

#### **Purpose**

To add rows to a table or to a view's base table.

#### **Prerequisites**

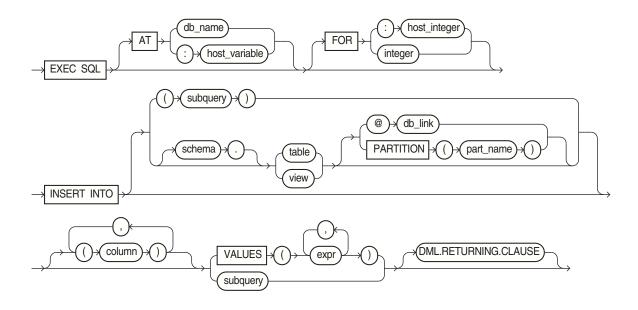
For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.

For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

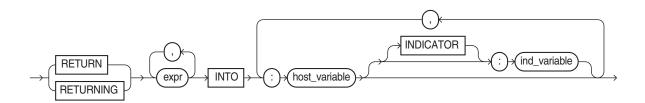
The INSERT ANY TABLE system privilege also enables you to insert rows into any table or any view's base table.



# **Syntax**



# where DML returning clause is:



# **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database on which the INSERT statement is executed. The database can be identified by either:
db_name	A database identifier declared in a previous DECLARE DATABASE statement.
host_variable	A host variable whose value is a previously declared <i>db_name</i> . If you omit this clause, the INSERT statement is executed on your default database.
FOR :host_integer	Limits the number of times the statement is executed if the VALUES clause contains array host variables. If you omit this clause, Oracle executes the statement once for each component in the smallest array.
schema	The schema containing the table or view. If you omit <i>schema</i> , Oracle assumes the table or view is in your own schema.



Keywords and Parameters	Description
table view	The name of the table into which rows are to be inserted. If you specify <i>view</i> , Oracle inserts rows into the view's base table.
db_link	A complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see Syntax for Schema Objects and Parts in SQL Statements. You can only insert rows into a remote table or view if you are using Oracle with the distributed option.
	If you omit <i>db_link</i> , Oracle assumes that the table or view is on the local database.
part_name	The name of partition in the table
column	A column of the table or view. In the inserted row, each column in this list is assigned a value from the VALUES clause or the query
	If you omit one of the table's columns from this list, the column's value for the inserted row is the column's default value as specified when the table was created. If you omit the column list altogether, the VALUES clause or query must specify values for all columns in the table.
VALUES	Specifies a row of values to be inserted into the table or view. Note that the expressions can be host variables with optional indicator variables. You must specify an expression in the VALUES clause for each column in the column list.
subquery	A subquery that returns rows that are inserted into the table. The select list of this subquery must have the same number of columns as the column list of the INSERT statement. For the syntax description of a subquery, see SELECT.
DML returning clause	See "DML Returning Clause" for a discussion.

Any host variables that appear in the WHERE clause should be either all scalars or all arrays. If they are scalars, Oracle executes the INSERT statement once. If they are arrays, Oracle executes the INSERT statement once for each set of array components, inserting one row each time.

Array host variables in the WHERE clause can have different sizes. In this case, the number of times Oracle executes the statement is determined by the smaller of the following values:

- Size of the smallest array
- The value of the *host\_integer* in the optional FOR clause.

For more information on this statement, see "The Basic SQL Statements ".

# Example I

This example illustrates the use of the embedded SQL INSERT statement:

```
EXEC SQL
INSERT INTO EMP (ENAME, EMPNO, SAL)
VALUES (:ENAME, :EMPNO, :SAL)
END-EXEC.
```



# **Example II**

This example shows an embedded SQL INSERT statement with a subquery:

```
EXEC SQL
INSERT INTO NEWEMP (ENAME, EMPNO, SAL)
SELECT ENAME, EMPNO, SAL FROM EMP
WHERE DEPTNO = :DEPTNO
END-EXEC.
```

### **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive).

# E.31 LOB APPEND (Executable Embedded SQL Extension)

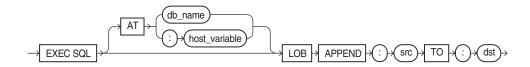
# **Purpose**

To append a LOB to the end of another LOB.

# **Prerequisites**

LOB buffering must not be enabled. The destination LOB must have been initialized.

# **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "APPEND".

#### **Related Topics**

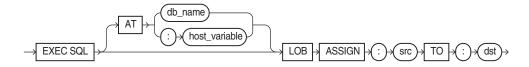
See the other LOB statements.

# E.32 LOB ASSIGN (Executable Embedded SQL Extension)

#### **Purpose**

To assign a LOB or BFILE locator to another locator.

#### **Syntax**



For usage notes as well as keywords, parameters, and examples, see "ASSIGN".

## **Related Topics**

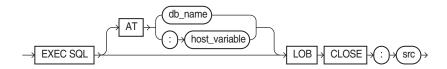
See the other LOB statements.

# E.33 LOB CLOSE (Executable Embedded SQL Extension)

# **Purpose**

To close an open LOB or BFILE.

# **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "CLOSE".

### **Related Topics**

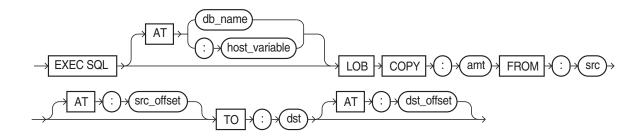
See the other LOB statements.

# E.34 LOB COPY (Executable Embedded SQL Extension)

# **Purpose**

To copy all or part of a LOB value into another LOB.

# **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "COPY".



# **Related Topics**

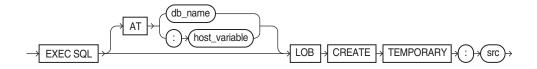
See the other LOB statements.

# E.35 LOB CREATE TEMPORARY (Executable Embedded SQL Extension)

## **Purpose**

To create a temporary LOB.

# **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "CREATE TEMPORARY".

# **Related Topics**

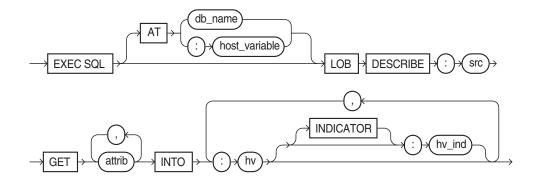
See the other LOB statements.

# E.36 LOB DESCRIBE (Executable Embedded SQL Extension)

# **Purpose**

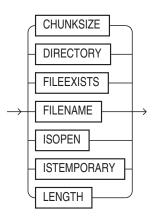
To retrieve attributes from a LOB.

### **Syntax**



where attrib is:





For usage notes as well as keywords, parameters, and examples, see "DESCRIBE".

# **Related Topics**

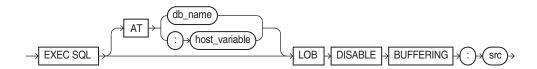
See the other LOB statements.

# E.37 LOB DISABLE BUFFERING (Executable Embedded SQL Extension)

# **Purpose**

To disable LOB buffering.

# **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "DISABLE BUFFERING".

# **Related Topics**

See the other LOB statements.

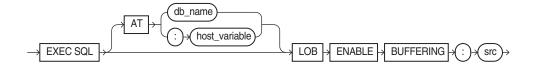


# E.38 LOB ENABLE BUFFERING (Executable Embedded SQL Extension)

#### **Purpose**

To enable LOB buffering.

### **Syntax**



## **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see ENABLE BUFFERING

# **Related Topics**

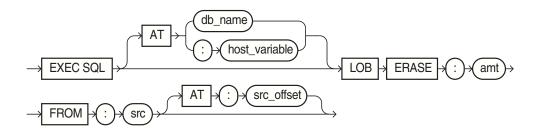
See the other LOB statements.

# E.39 LOB ERASE (Executable Embedded SQL Extension)

# **Purpose**

To erase a given amount of LOB data starting from a given offset.

# **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "ERASE".

# **Related Topics**

See the other LOB statements.

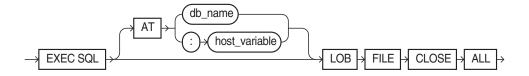


# E.40 LOB FILE CLOSE ALL (Executable Embedded SQL Extension)

#### **Purpose**

To close all open BFILEs in the current session.

#### **Syntax**



## **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "FILE CLOSE ALL".

# **Related Topics**

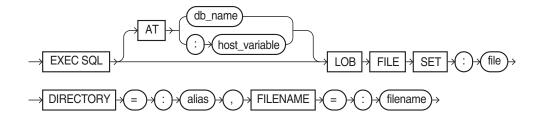
See the other LOB statements.

# E.41 LOB FILE SET (Executable Embedded SQL Extension)

#### **Purpose**

To set DIRECTORY and FILENAME in a BFILE locator.

#### **Syntax**



## **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "FILE SET".



## **Related Topics**

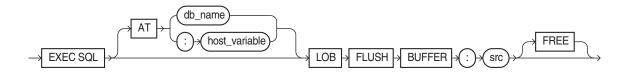
See the other LOB statements.

# E.42 LOB FLUSH BUFFER (Executable Embedded SQL Extension)

#### **Purpose**

To write the LOB buffers to the database server.

#### **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "FLUSH BUFFER".

#### **Related Topics**

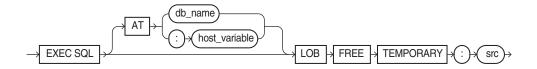
See the other LOB statements.

# E.43 LOB FREE TEMPORARY (Executable Embedded SQL Extension)

## **Purpose**

To free temporary space for the LOB locator.

#### **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "FREE TEMPORARY".



## **Related Topics**

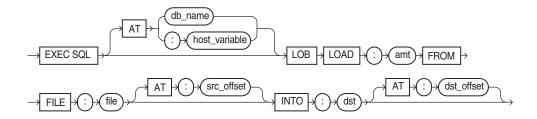
See the other LOB statements.

# E.44 LOB LOAD (Executable Embedded SQL Extension)

## **Purpose**

To copy all or part of a BFILE into an internal LOB.

## **Syntax**



## **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "LOAD FROM FILE".

## **Related Topics**

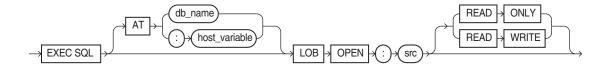
See the other LOB statements.

# E.45 LOB OPEN (Executable Embedded SQL Extension)

#### **Purpose**

To open a LOB or BFILE for read or read/write access.

## **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "OPEN".

#### **Related Topics**

See the other LOB statements.

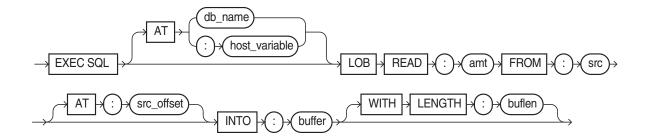


# E.46 LOB READ (Executable Embedded SQL Extension)

## **Purpose**

To read all or part of a LOB or BFILE into a buffer.

## **Syntax**



## **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "READ".

# **Related Topics**

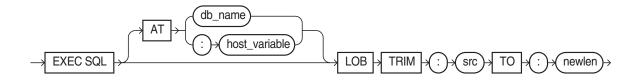
See the other LOB statements.

# E.47 LOB TRIM (Executable Embedded SQL Extension)

#### **Purpose**

To truncate a LOB value.

## **Syntax**



## **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "TRIM".

## **Related Topics**

See the other LOB statements.

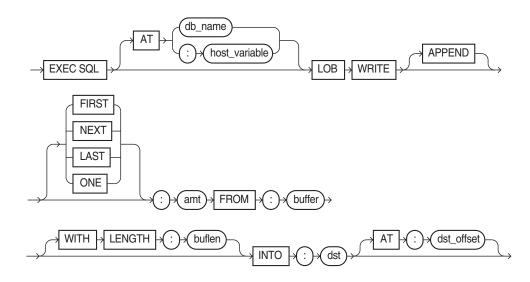


# E.48 LOB WRITE (Executable Embedded SQL Extension)

# **Purpose**

To write the contents of a buffer to a LOB.

#### **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "WRITE".

## **Related Topics**

See the other LOB statements.

# E.49 OPEN (Executable Embedded SQL)

#### **Purpose**

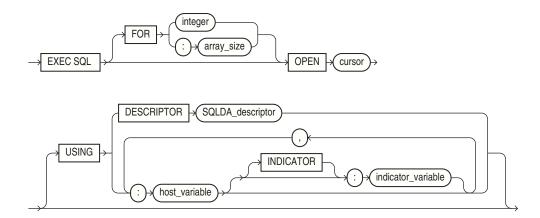
To open a cursor, evaluating the associated query and substituting the host variable names supplied by the USING clause into the WHERE clause of the query. It can be used in place of EXECUTE in dynamic SQL. For the ANSI Dynamic SQL syntax, see "OPEN DESCRIPTOR (Executable Embedded SQL)".

## **Prerequisites**

You must declare the cursor with a DECLARE CURSOR embedded SQL statement before opening it.



## **Syntax**



## **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed.
	Can only be used when the OPEN is equivalent to EXECUTE.
cursor	The (previously declared) cursor to be opened.
host_variable	Specifies a host variable with an optional indicator variable to be substituted into the statement associated with the cursor.
	CANNOT be used together with an ANSI descriptor (INTO clause).
DESCRIPTOR SQLDA_descriptor	Specifies an Oracle descriptor that describes the host variables to be substituted into the WHERE clause of the associated query. The descriptor must be initialized in a previous DESCRIBE statement. The substitution is based on position. The host variable names specified in this statement can be different from the variable names in the associated query.
	CANNOT be used together with an ANSI descriptor (INTO clause).

#### **Usage Notes**

The OPEN statement defines the active set of rows and initializes the cursor just before the first row of the active set. The values of the host variables at the time of the OPEN are substituted in the statement. This statement does not actually retrieve rows; rows are retrieved by the FETCH statement.

Once you have opened a cursor, its input host variables are not reexamined until you reopen the cursor. To change any input host variables and therefore the active set, you must reopen the cursor.

All cursors in a program are in a closed state when the program is initiated or when they have been explicitly closed using the CLOSE statement.

You can reopen a cursor without first closing it. For more information on this statement, see "Opening a Cursor".



## **Example**

This example illustrates the use of the OPEN statement in a Pro\*COBOL program:

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR SELECT ENAME, EMPNO, JOB, SAL FROM EMP
WHERE DEPTNO = :DEPTNO
END-EXEC.
EXEC SQL OPEN EMPCURSOR END-EXEC.
```

#### **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE CURSOR (Embedded SQL Directive) .

EXECUTE (Executable Embedded SQL).

FETCH (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.50 OPEN DESCRIPTOR (Executable Embedded SQL)

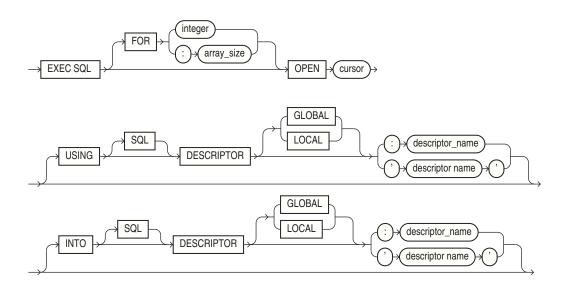
## **Purpose**

To open a cursor (for ANSI Dynamic SQL Method 4), evaluating the associated query and substituting the input host variable names supplied by the USING clause into the WHERE clause of the query. The INTO clause denotes the output descriptor. It can be used in place of EXECUTE in dynamic SQL.

# **Prerequisites**

You must declare the cursor with a DECLARE CURSOR embedded SQL statement before opening it.

#### **Syntax**





Keywords and Parameters	Description
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed.
	Used only with dynamic SQL when it is equivalent to EXECUTE.
cursor	The (previously declared) cursor to be opened.
USING DESCRIPTOR	Specifies an ANSI input descriptor with the host variable containing the name of the ANSI descriptor, or the name of the ANSI descriptor.
descriptor_name 'descriptor name'	
INTO DESCRIPTOR descriptor_name 'descriptor name'	Specifies an ANSI output descriptor with the host variable containing the name of the ANSI descriptor, or the name of the ANSI descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.

#### **Usage Notes**

Set the precompiler option DYNAMIC to ANSI.

The OPEN statement defines the active set of rows and initializes the cursor just before the first row of the active set. The values of the host variables at the time of the OPEN are substituted in the statement. This statement does not actually retrieve rows; rows are retrieved by the FETCH statement.

Once you have opened a cursor, its input host variables are not reexamined until you reopen the cursor. To change any input host variables and therefore the active set, you must reopen the cursor.

All cursors in a program are in a closed state when the program is initiated or when they have been explicitly closed using the CLOSE statement.

You can reopen a cursor without first closing it. For more information on this statement, see "Inserting Rows".

#### **Example**

```
01 DYN-STATEMENT PIC X(58) VALUE "SELECT ENAME, EMPNO FROM EMP WHERE
DEPTNO =:DEPTNO-DAT".

01 DEPTNO-DAT PIC S9(9) COMP VALUE 10.

EXEC SQL ALLOCATE DESCRIPTOR 'input-descriptor' END-EXEC.
EXEC SQL ALLOCATE DESCRIPTOR 'output-descriptor'

EXEC SQL PREPARE S FROM :DYN-STATEMENT END-EXEC.
EXEC SQL DECLARE C CURSOR FOR S END-EXEC.

EXEC SQL OPEN C USING DESCRIPTOR 'input-descriptor' END-EXEC.
```

#### **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE CURSOR (Embedded SQL Directive).



FETCH DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.51 PREPARE (Executable Embedded SQL)

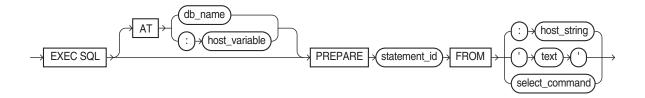
#### **Purpose**

To parse a SQL statement or PL/SQL block specified by a host variable and associate it with an identifier.

## **Prerequisites**

None.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
db_name	A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If it is omitted, or if it is an empty string, the default database connection is assumed.
host_variable	A host variable containing the name of the database connection.
array_size integer	Host variable containing the number of rows to be processed.  Number of rows to be processed.
statement_id	The identifier to be associated with the prepared SQL statement or PL/SQL block. If this identifier was previously assigned to another statement or block, the prior assignment is superseded.
host_string	A host variable whose value is the text of a SQL statement or PL/SQL block to be prepared.
text	A text literal containing the SQL statement or PL/SQL block to be executed. The quotes may be omitted.
select_command	A SELECT statement.

## **Usage Notes**

Any variables that appear in the *host\_string* or *text* are placeholders. The actual host variable names are assigned in the USING clause of the OPEN statement (input host variables) or in the INTO clause of the FETCH statement (output host variables).

A SQL statement is prepared only once, but can be executed any number of times.



## **Example**

This example illustrates the use of a PREPARE statement in a Pro\*COBOL embedded SQL program:

```
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC. EXEC SQL EXECUTE MYSTATEMENT END-EXEC.
```

#### **Related Topics**

CLOSE (Executable Embedded SQL) .

DECLARE CURSOR (Embedded SQL Directive).

FETCH (Executable Embedded SQL).

OPEN (Executable Embedded SQL).

# E.52 ROLLBACK (Executable Embedded SQL)

# **Purpose**

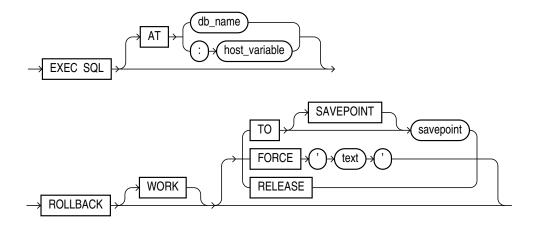
To undo work done in the current transaction. You can also use this statement to manually undo the work done by an in-doubt distributed transaction.

## **Prerequisites**

To roll back your current transaction, no privileges are necessary.

To manually roll back an in-doubt distributed transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

#### **Syntax**





Keywords and Parameters	Description
db_name	A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If it is omitted, or if it is an empty string, the default database connection is assumed.
host_variable	A host variable containing the name of the database connection. If you omit this clause, the savepoint is created on your default database.
WORK	Is optional and is provided for ANSI compatibility.
ТО	Rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.
FORCE	Manually rolls back an in-doubt distributed transaction. The transaction is identified by the <i>text</i> containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING.
	ROLLBACK statements with the FORCE clause are not supported in PL/SQL.
RELEASE	Frees all resources and disconnects the application from the database server. The RELEASE clause is not allowed with SAVEPOINT and FORCE clauses.
savepoint	The name of the savepoint to be rolled back to.

## **Usage Notes**

A transaction (or a logical unit of work) is a sequence of SQL statements that Oracle treats as a single unit. A transaction begins with the first executable SQL statement after a COMMIT, ROLLBACK or connection to the database. A transaction ends with a COMMIT statement, a ROLLBACK statement, or disconnection (intentional or unintentional) from the database. Note that Oracle issues an implicit COMMIT statement before and after processing any data definition language statement.

Using the ROLLBACK statement without the TO SAVEPOINT clause performs the following operations:

- ends the transaction
- undoes all changes in the current transaction
- erases all savepoints in the transaction
- releases the transaction's locks

Using the ROLLBACK statement with the TO SAVEPOINT clause performs the following operations:

- rolls back just the portion of the transaction after the savepoint.
- loses all savepoints created after that savepoint. Note that the named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- releases all table and row locks acquired since the savepoint. Note that other transactions that have requested access to rows locked after the savepoint must

continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the rows immediately.

It is recommended that you explicitly end transactions in application programs using either a COMMIT or ROLLBACK statement. If you do not explicitly commit the transaction and the program terminates abnormally, Oracle rolls back the last uncommitted transaction.

# Example I

The following statement rolls back your entire current transaction:

```
EXEC SOL ROLLBACK END-EXEC.
```

#### **Example II**

The following statement rolls back your current transaction to savepoint SP5:

```
EXEC SQL ROLLBACK TO SAVEPOINT SP5 END-EXEC.
```

#### **Distributed Transactions**

Oracle with the distributed option enables you to perform distributed transactions, or transactions that modify data on multiple databases. To commit or roll back a distributed transaction, you need only issue a COMMIT or ROLLBACK statement as you would any other transaction.

If there is a network failure during the commit process for a distributed transaction, the state of the transaction may be unknown, or in-doubt. After consultation with the administrators of the other databases involved in the transaction, you may decide to manually commit or roll back the transaction on your local database. You can manually roll back the transaction on your local database by issuing a ROLLBACK statement with the FORCE clause.

You cannot manually roll back an in-doubt transaction to a savepoint.

A ROLLBACK statement with a FORCE clause only rolls back the specified transaction. Such a statement does not affect your current transaction.

#### **Example III**

The following statement manually rolls back an in-doubt distributed transaction:

```
EXEC SQL ROLLBACK WORK FORCE '25.32.87' END-EXEC.
```

#### **Related Topics**

COMMIT (Executable Embedded SQL).

SAVEPOINT (Executable Embedded SQL) .

# E.53 SAVEPOINT (Executable Embedded SQL)

#### **Purpose**

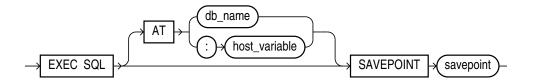
To identify a point in a transaction to which you can later roll back.

#### **Prerequisites**

None.



## **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database on which the savepoint is created. The database can be identified by either:
db_name	A database identifier declared in a previous DECLARE DATABASE statement.
host_variable	A host variable whose value is a previously declared <i>db_name</i> . If you omit this clause, the savepoint is created on your default database.
savepoint	The name of the savepoint to be created.

#### **Usage Notes**

For more information on this statement, see "Using the SAVEPOINT Statement".

#### **Example**

This example illustrates the use of the embedded SQL SAVEPOINT statement:

EXEC SQL SAVEPOINT SAVE3 END-EXEC.

#### **Related Topics**

COMMIT (Executable Embedded SQL).

ROLLBACK (Executable Embedded SQL).

# E.54 SELECT (Executable Embedded SQL)

#### **Purpose**

To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.

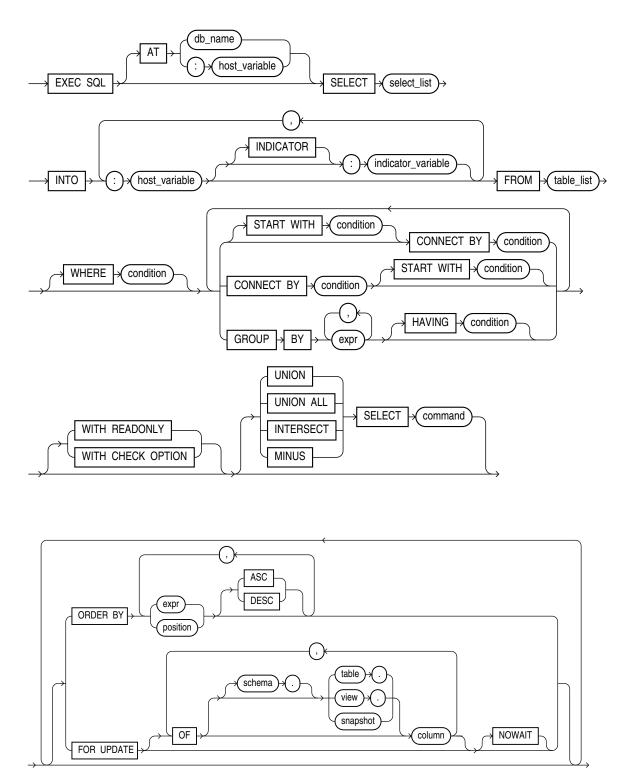
## **Prerequisites**

For you to select data from a table or snapshot, the table or snapshot must be in your own schema or you must have SELECT privilege on the table or snapshot.

For you to select rows from the base tables of a view, the owner of the schema containing the view must have SELECT privilege on the base tables. Also, if the view is in a schema other than your own, you must have SELECT privilege on the view.

The SELECT ANY TABLE system privilege also enables you to select data from any table or any snapshot or any view's base table.

# **Syntax**



Keywords and Parameters	Description
AT	Identifies the database to which the SELECT statement is issued. The database can be identified by either:
db_name	A database identifier declared in a previous DECLARE DATABASE statement.
host_variable	A host variable whose value is a previously declared db_name. If you omit this clause, the SELECT statement is issued to your default database.
select_list	Identical to the non-embedded SELECT statement except that a host variables can be used in place of literals.
INTO	Specifies output host variables and optional indicator variables to receive the data returned by the SELECT statement. Note that these variables must be either all scalars or all arrays, but arrays need not have the same size.
WHERE	Restricts the rows returned to those for which the condition is TRUE. See About SQL Conditionsfor more information. The <i>condition</i> can contain host variables, but cannot contain indicator variables. These host variables should be scalars and cannot be arrays.

All other keywords and parameters are identical to the non-embedded SQL SELECT statement.

#### **Usage Notes**

If no rows meet the WHERE clause condition, no rows are retrieved and Oracle returns an error code through the SQLCODE component of the SQLCA.

You can use comments in a SELECT statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement.

# **Example**

This example illustrates the use of the embedded SQL SELECT statement:

```
EXEC SQL SELECT ENAME, SAL + 100, JOB
INTO :ENAME, :SAL, :JOB
FROM EMP
WHERE EMPNO = :EMPNO
END-EXEC.
```

## **Related Topics**

DECLARE CURSOR (Embedded SQL Directive).

DECLARE DATABASE (Oracle Embedded SQL Directive).

EXECUTE (Executable Embedded SQL).

FETCH (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).



# E.55 SET DESCRIPTOR (Executable Embedded SQL)

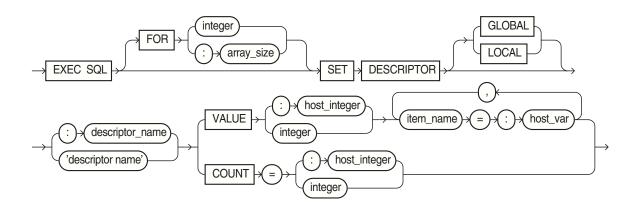
# **Purpose**

Use this ANSI dynamic SQL statement to set information in the descriptor area from host variables.

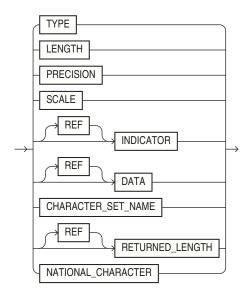
# **Prerequisites**

Use after a DESCRIBE DESCRIPTOR.

# **Syntax**



where item name can be one of these choices:



Keywords and Parameters	Description
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed. The array size clause can only be used with DATA, RETURNED_LENGTH and INDICATOR item names.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
descriptor_name	Host variable containing the name of the allocated ANSI descriptor.
'descriptor name'	Name of the allocated ANSI descriptor.
COUNT	The total number of input or output variables.
VALUE	The position of the referenced host variable in the statement.
item_name	See Table 10-6 and Table 10-7 for lists of the <i>item_names</i> , and their descriptions.
host_var	Host variable containing the total number of input or output variables.
integer	Total number of input or output variables.
host_var	The host variables used to set the item.
REF	Reference semantics are to be used. Can be used only with RETURNED_LENGTH, DATA, and INDICATOR item names.
	Must be used to set RETURNED_LENGTH.

# **Usage Notes**

Use DYNAMIC=ANSI precompiler option. Set CHARACTER\_SET\_NAME to UTF16 for client-side Unicode support. See "SET DESCRIPTOR" for complete details, including tables of descriptor item names.

## **Example**

EXEC SQL SET DESCRIPTOR GLOBAL :mydescr COUNT = 3 END-EXEC.

# **Related Topics**

ALLOCATE DESCRIPTOR (Executable Embedded SQL).

DEALLOCATE DESCRIPTOR (Embedded SQL Statement).

DESCRIBE DESCRIPTOR (Executable Embedded SQL).

GET DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.56 UPDATE (Executable Embedded SQL)

## **Purpose**

To change existing values in a table or in a view's base table.



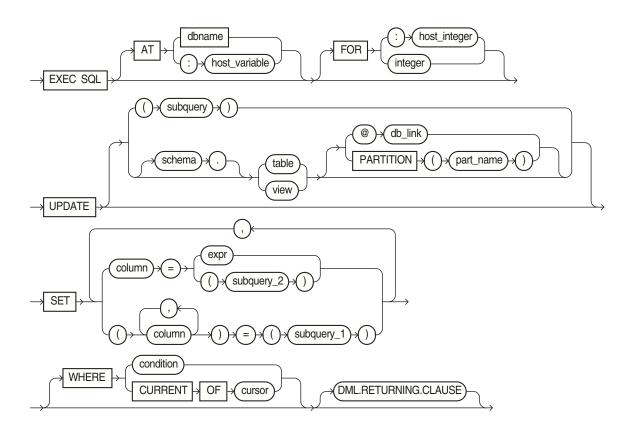
## **Prerequisites**

For you to update values in a table or snapshot, the table must be in your own schema or you must have UPDATE privilege on the table.

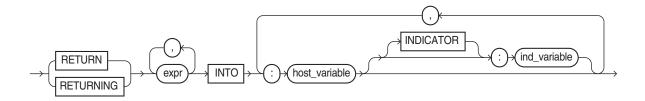
For you to update values in the base table of a view, the owner of the schema containing the view must have UPDATE privilege on the base table. Also, if the view is in a schema other than your own, you must have UPDATE privilege on the view.

The UPDATE ANY TABLE system privilege also enables you to update values in any table or any view's base table.

# **Syntax**



where DML returning clause is:





Keywords and Parameters	Description
AT	identifies the database to which the UPDATE statement is issued. The database can be identified by either dbname or host_variable.
	If you omit this clause, the UPDATE statement is issued to your default database.
dbname	A database identifier declared in a previous DECLARE DATABASE statement.
host_variable	A host variable whose value is a previously declared dbname.
FOR :host_integer	Limits the number of times the UPDATE statement is executed if the SET and WHERE clauses contain array host variables. If you omit this clause, Oracle executes the statement once for each component of the smallest array.
schema	The schema containing the table or view. If you omit schema, Oracle assumes the table or view is in your own schema.
table view	The name of the table to be updated. If you specify <i>view</i> , Oracle updates the view's base table.
dblink	A complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see References to Objects in Remote Databases. You can only use a database link to update a remote table or view if you are using Oracle with the distributed option.
part_name	Name of partition in the table
alias	A name used to reference the table, view, or subquery elsewhere in the statement.
column	The name of a column of the table or view that is to be updated. If you omit a column of the table from the SET clause, that column's value remains unchanged.
expr	The new value assigned to the corresponding column. This expression can contain host variables and optional indicator variables.
subquery_1	A subquery that returns new values that are assigned to the corresponding columns. For the syntax of a subquery, see SELECT.
subquery_2	A subquery that return a new value that is assigned to the corresponding column. For the syntax of a subquery, see SELECT.
WHERE	Specifies which rows of the table or view are updated.  If you omit this clause entirely, the database updates all rows of the table or view.  condition specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of TRUE, FALSE, or UNKNOWN.
	CURRENT OF specifies deletion of the record that was last fetched by the cursor.



Keywords and Parameters	Description
DML returning clause	See "DML Returning Clause" for a discussion.

#### **Usage Notes**

Host variables in the SET and WHERE clauses must be either all scalars or all arrays. If they are scalars, Oracle executes the UPDATE statement only once. If they are arrays, Oracle executes the statement once for each set of array components. Each execution may update zero, one, or multiple rows.

Array host variables can have different sizes. In this case, the number of times Oracle executes the statement is determined by the smaller

of the following values:

- The size of the smallest array
- The value of the host integer in the optional FOR clause

The cumulative number of rows updated is returned through the third element of the SQLERRD component of the SQLCA. When arrays are used as input host variables, this count reflects the total number of updates for all components of the array processed in the UPDATE statement. If no rows satisfy the condition, no rows are updated and Oracle returns an error message through the SQLCODE element of the SQLCA. If you omit the WHERE clause, all rows are updated and Oracle raises a warning flag in the fifth component of the SQLWARN element of the SQLCA.

You can use comments in an UPDATE statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement.

For more information on this statement, see "The Basic SQL Statements" and Database Concepts.

## **Examples**

The following examples illustrate the use of the embedded SQL UPDATE statement:

```
EXEC SQL UPDATE EMP

SET SAL = :SAL, COMM = :COMM INDICATOR :COMM-IND

WHERE ENAME = :ENAME

END-EXEC.

EXEC SQL UPDATE EMP

SET (SAL, COMM) =

(SELECT AVG(SAL)*1.1, AVG(COMM)*1.1

FROM EMP)

WHERE ENAME = 'JONES'

END-EXEC.
```

#### **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive).



# E.57 VAR (Oracle Embedded SQL Directive)

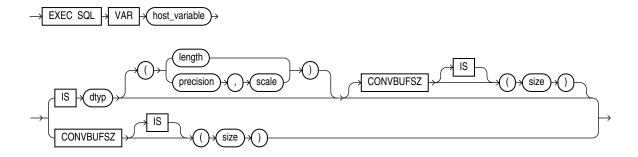
## **Purpose**

To perform *host variable equivalencing*, to assign a specific Oracle external datatype to an individual host variable, overriding the default datatype assignment. There is an optional clause, CONVBUFSZ, that specifies the size of a buffer for character set conversion.

#### **Prerequisites**

The host variable must be previously declared in the embedded SQL program.

## **Syntax**



## **Keywords and Parameters**

Keywords and Parameters	Description
host_variable	The host variable to be assigned an Oracle external datatype.
dtyp	An Oracle external datatype recognized by Pro*COBOL (not an Oracle internal datatype). The datatype may include a length, precision, or scale. This external datatype is assigned to the host_variable. For a list of external datatypes, see "External Datatypes".
size	The size in bytes of a buffer in the Oracle runtime library used to perform conversion between character sets of the <i>host_variable</i> .

# **Usage Notes**

Datatype equivalencing is useful for any of the following purposes:

- To store program data as binary data in the database
- To override default datatype conversion

For more information about Oracle datatypes, see "Sample Program 4: Datatype Equivalencing".



## **Example**

This example equivalences the host variable DEPT\_NAME to the datatype VARCHAR2 and the host variable BUFFER to the datatype RAW(200):

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 DEPT-NAME PIC X(15).

* -- default datatype is CHAR
EXEC SQL VAR DEPT-NAME IS VARCHAR2 END-EXEC.

* -- reset to STRING
...
01 BUFFER-VAR.
05 BUFFER PIC X(200).

* -- default datatype is CHAR
EXEC SQL VAR BUFFER IS RAW(200) END-EXEC.

* -- refer to RAW
...
EXEC SQL END DECLARE SECTION END-EXEC.
```

# **Related Topics**

None.

# E.58 WHENEVER (Embedded SQL Directive)

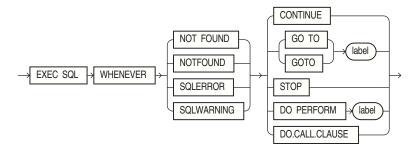
# **Purpose**

To specify the action to be taken when an error or warning results from executing an embedded SQL program.

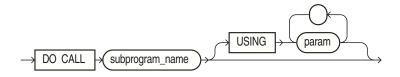
# **Prerequisites**

None.

## **Syntax**



where DO.CALL.CLAUSE is:





Keywords and Parameters	Description
NOT FOUND   NOTFOUND	Identifies any exception condition that returns an error code of +1403 to SQLCODE (or a +100 code when MODE=ANSI).
SQLERROR	Identifies a condition that results in a negative return code.
SQLWARNING	Identifies a non-fatal warning condition.
CONTINUE	Indicates that the program should progress to the next statement.
GOTO   GO TO	Indicates that the program should branch to the statement named by <i>label</i> .
STOP	Stops program execution.
DO PERFORM	Indicates that the program should perform a paragraph or section at <i>label</i> .
DO CALL	Indicates that the program should execute a subprogram.
subprogram_name	The subprogram to be executed. It may have to be inside quotes (").
USING	Indicates that the parameters of the subprogram follow.
param	A list of subprogram parameters separated by blanks.

The WHENEVER directive enables your program to take one of several possible actions in the event an embedded SQL statement results in an error or warning.

The scope of a WHENEVER statement is positional, rather than logical. A WHENEVER statement applies to all embedded SQL statements that textually follow it in the source file, not in the flow of the program logic. A WHENEVER statement remains in effect until it is superseded by another WHENEVER statement checking for the same condition.

For more information about and examples of the conditions and actions of this directive, see "WHENEVER Directive".

Do not confuse the WHENEVER embedded SQL directive with the WHENEVER SQL\*Plus command.

#### Example

The following example illustrates the use of the WHENEVER directive in a Pro\*COBOL embedded SQL program:

```
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

...

EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.

...

SQL-ERROR.

EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

DISPLAY "ORACLE ERROR DETECTED.".

EXEC SQL ROLLBACK RELEASE END-EXEC.

STOP RUN.
```

#### **Related Topics**

None



F

# **Pro\*COBOL for Windows**

This section provides an overview of building Oracle database applications with Pro\*COBOL for Windows operating systems.

This section contains these topics:

- Compiling and Linking Pro\*COBOL Applications
- Sample Programs

# F.1 Compiling and Linking Pro\*COBOL Applications

Pro\*COBOL supports the MERANT Micro Focus NetExpress version 5.0 for 32-bit Windows. This section describes how to compile and link Pro\*COBOL applications using the MERANT Micro Focus compiler.

You can build and execute a MERANT Micro Focus COBOL application in two ways:

- How to Use the IDE (using NetExpress only)
- How to Use the Animator Products (products other than NetExpress)
- The COBOL and CBLLINK Commands (all products)
- The COBSQL Command

In each of these the COBSQL utility may be used with the following advantages:

- Pro\*COBOL is run by the MERANT Micro Focus compiler and does not need to be run as a separate step.
- Animation is done using the .pco source file rather than the .cbl file produced by Pro\*COBOL.
- The MAKEYSYN directive is provided automatically and need not be specified manually.

# F.1.1 How to Use the IDE

A program generated by Pro\*COBOL can be compiled and executed from within the MERANT Micro Focus NetExpress IDE. Simply add the .cbl file generated by Pro\*COBOL to a Net Express project. To avoid potential inconsistencies when calling routines in the Oracle libraries the program should be compiled using the directive:

```
MAKESYN "COMP-5" = "COMP"
```

This directive can be specified in the build setting for the source file, the project settings or through a \$SET line at the start of the source file. When you select **Rebuild** or **Rebuild** All the IDE generates an executable ready to Run or Animate.

# F.1.2 How to Use the Animator Products

Programs can be compiled and executed from within the MERANT Micro Focus COBOL debugger, Animator V2.

To avoid potential inconsistencies when calling routines in the Oracle libraries, select the menu option Compiler Directives, and enter:

This step is required because MERANT Micro Focus COBOL stores binary numbers in Big Endian format. Oracle libraries expect binary numbers to be stored in Little Endian format (machine format).

```
MAKESYN "COMP-5" = "COMP"
```

# F.1.3 The COBOL and CBLLINK Commands

COBOL and CBLLINK can be used to build programs in two ways, depending on whether the Pro\*COBOL runtime is to be statically linked or accessed through a DLL at runtime.

For dynamic linking, the commands are:

```
COBOL sample1 /MAKESYN"COMP-5"="COMP";
CBLLINK sample1
```

For static linking, the commands are:

```
COBOL sample1 /LITLINK /MAKESYN"COMP-5"="COMP";
CBLLINK sample1 ORACLE BASE\ORACLE HOME\precomp\lib\orasq112.lib
```

The previous commands produce sample1.exe, which can be executed like any other Windows program.



MERANT Micro Focus COBOL must be installed on the same system as Pro\*COBOL to successfully execute the file.

# F.1.4 The COBSQL Command

 ${\tt COBSQL}$  can be used to simplify preprocessing and debugging. To use  ${\tt COBSQL}$ , specify the following directive to the COBOL compiler:

```
PREPROCESS (COBSQL) COBSQLTYPE=ORACLE8 ENDP
```

or the short form:

```
P(COBSQL) CSQLT=ORA8 ENDP
```

COBSQLTYPE should be set to ORACLE or ORA for versions of Pro\*COBOL prior to release 8.0. The directive may be set with a \$SET line at the start of the source file, on the COBOL command line, in program build settings or project settings for NetExpress, or with SQL compiler directives settings for Animator. At compile time, COBSQL runs



 $Pro^*COBOL$  as a background task and passes its output to the COBOL compiler together with additional information required to enable Animator to track execution using the .pco file rather than .cbl file.

When using COBSQL there is no need to deal directly with the .cbl file. Instead, add the .pco file to a NetExpress project, or open it with Animator.

# F.2 Sample Programs

Oracle provides sample programs to demonstrate the use of Pro\*COBOL with Oracle database features. See "Sample Files" for a listing of these programs.

This section describes how to use the basic precompiling, compiling, and linking commands to build the sample programs. This section also describes the preparations required for running the Pro\*COBOL sample programs.

# F.2.1 Building the Demonstration Table

To run the Pro\*COBOL sample programs, you must have a database account with the username scott and the password tiger. If this account does not exist on your database, create one before running the sample programs.

The scott account must contain the emp and dept tables. If the account does not contain these tables, use the demobld.sql script to create them.

To run the demobld.sql script:

- Start SQL\*Plus.
- 2. Connect to the database as username scott with password tiger.
- 3. Run the demobld.sql script. For example:

SQL> @ORACLE\_BASE\ORACLE\_HOME\sqlplus\demo\demobld.sql

# F.2.2 Building the Sample Programs

Pro\*COBOL supplies the makeit.bat file which is listed in the next section, for building a MERANT Micro Focus COBOL sample file:

The batch files are located in ORACLE BASE\ORACLE HOME \precomp\demo\procob2.

To build the sample programs:

1. Run the batch files with any sample file. Do not include the file extension. For example:

```
C:\ORACLE\ORA90\PRECOMP\DEMO\PROCOB2> makeit sample1
```

2. Ensure that all paths and file names reflect the configuration of your system, if you encounter errors when building a sample program.

The commands to run the sample programs assume that the following is the current working directory:

```
ORACLE_BASE\ORACLE_HOME \precomp\demo\procob2
```

You may need to modify the sample link script to reflect the configuration of your system. See "Compiling and Linking Pro\*COBOL Applications" for more information.

The makeit.bat contains the following:



```
procob iname=%1.pco ireclen=132
cobol %1 /anim /litlink makesyn "COMP-5" = "COMP";
cbllink %1 /M%1 ORACLE_BASE\ORACLE_HOME\precomp\lib\orasq112.lib
```

# F.2.3 Sample Files

The Pro\*COBOL sample files listed in Table F-1 are located in the ORACLE\_BASE\ORACLE\_HOME\precomp\demo\procob2 directory. The SQL scripts are located in the ORACLE BASE\ORACLE HOME\precomp\demo\sql directory.

Table F-1 Pro\*COBOL Sample Programs

Sample Program	Description
sample1.pco	Simple query
sample2.pco	Cursor operations
sample3.pco	Host tables
sample4.pco	Datatype equivalence
sample6.pco	Dynamic SQL Method 1
sample7.pco	Dynamic SQL Method 2
sample8.pco	Dynamic SQL Method 3
sample9.pco	Stored procedure call
calldemo.sql	Stored procedure call
sample10.pco	Dynamic SQL Method 4
sample11.pco	Cursor variable operations
sample11.sql	Cursor variable operations
sample12.pco	Dynamic SQL Method 4 using ANSI dynamic SQL
sample13.pco	Nested program
sampleco.pco	Simple query and insert
sample14.pco	Host table x
lobdemol.pco	LOB datatypes
lobdemo1.sql	LOB datatypes



# Index

A	AUTO_CONNECT option
abbreviations permitted, 2-9	instead of CONNECT statement, 3-8 AUTO_CONNECT precompiler option, 14-14
abnormal termination	automatic logons, 3-7
automatic rollback, <i>E-13</i>	Avoiding Unnecessary Reparsing, <i>C-10</i>
active set, 5-10	Avoiding Chinesessary Repaising, C 10
changing, 5-11, 5-12	D
definition, 5-10	В
when empty, 5-12	batch fetch, 7-6
ALLOCATE DESCRIPTOR statement, <i>E-9</i>	example, 7-6
ALLOCATE statement, <i>E-8</i>	number of rows returned, 7-6
use with ROWID, 4-29	BFILES
allocating	definition, 13-1
cursors, <i>E-8</i>	bind descriptor, 11-3
thread context, 12-6, E-15	bind descriptors
allocating cursor variables, 6-24	information in, 9-19
ALTER AUTHORIZATION	bind SQLDA, 11-3
changing password, 3-8	bind variables, 9-19
ANSI dynamic SQL, F-4	binding host variables, 9-3
See also dynamic SQL (ANSI), 10-1	BNDDFCLP variable (SQLDA), 11-12
ANSI format	BNDDFCRCP variable (SQLDA), 11-12
COBOL statements, 2-9	BNDDFMT variable (SQLDA), 11-8
ANSI/ISO SQL	BNDDH-CUR-VNAMEL variable (SQLDA), 11-11
extensions, 14-21	BNDDH-MAX-VNAMEL variable (SQLDA), 11-11
application development process, 2-1	BNDDH-VNAME variable (SQLDA), 11-10
ARRAYLEN statement, 6-12	BNDDI variable (SQLDA), 11-10
ASACC precompiler option, 14-13	BNDDI-CUR-VNAMEL variable (SQLDA), 11-12
ASSUME_SQLCODE precompiler option, 14-13	BNDDI-MAX-VNAMEL variable (SQLDA), 11-11
AT clause	BNDDI-VNAME variable (SQLDA), 11-11
CONNECT statement, 3-5	BNDDV variable (SQLDA), 11-8
DECLARE CURSOR statement, 3-6	BNDDVLN variable (SQLDA), 11-8
DECLARE STATEMENT statement, 3-6	BNDDVTYP variable (SQLDA), 11-9
EXECUTE IMMEDIATE statement, 3-6	
of COMMIT statement, <i>E-12</i>	С
of CONNECT statement, <i>E-14</i>	
of DECLARE CURSOR directive, <i>E-19</i>	CALL SQL statement, 6-17
of DECLARE STATEMENT directive, <i>E-22</i>	CALL statement, <i>E-10</i>
of EXECUTE IMMEDIATE statement, <i>E-34</i>	example, 6-18
of EXECUTE statement, <i>E-30</i>	case-insensitivity, 2-9
of INSERT statements, <i>E-43</i>	CHAR datatypes
of SAVEPOINT statement, <i>E-62</i>	external, 4-5
of SELECT statement, <i>E-64</i>	character host variables
of UPDATE statement, <i>E-68</i>	as output variables, 4-27
restrictions, 3-6	handling, 4-25
	server handling, 4-27



character host variables (continued)	concurrent logons, 3-3
types, 4-25	conditional precompilation, 2-20
character sets	defining symbols, 2-21
multibyte, 4-32	example, 2-21
character strings	CONFIG precompiler option, 14-15, 14-16
multibyte, 4-33	configuration file
CHARF datatype specifier, 4-42	system versus user, 14-16
using in VAR statement, 4-42	CONNECT statement, <i>E-13</i>
CHARF datatypes	AT clause, 3-5
external, 4-5	enabling a semantic check, <i>D-2</i>
CHARZ datatype	examples, <i>E-13</i>
external, 4-5	logging on to Oracle, 3-1
CLOSE command	placement, 3-2
examples, <i>E-11</i>	requirements, 3-2
CLOSE statement, <i>E-11</i>	USING clause, 3-5
example, 5-13	when not required, 3-8
in dynamic SQL method 4, 11-32	connecting to Oracle, 3-1
CLOSE ON COMMIT	automatically, 3-7
precompiler option, 3-12, 5-10	concurrently, 3-3
CLOSE ON COMMIT precompiler option, 14-15	example of, 3-2
closing	via Oracle Net Services, 3-3
cursors, <i>E-11</i>	connections
COBOL 74 A 1	default versus non-default, 3-4
COBOL 25 A 1	implicit, 3-9
COBOL-85, <i>A-1</i>	naming, 3-4
code page, 4-31	CONTEXT ALLOCATE statement, 12-6, E-15
coding area	CONTEXT FREE statement, 12-7, E-15
for paragraph names, 2-15	CONTEXT USE directive, 12-7
coding conventions, 2-9	CONTEXT USE SQL directive, <i>E-16</i>
column list, 5-7	CONTEXT USE SQL statement, 12-7
COMMENT clause	continuation lines
of COMMIT statement, <i>E-12</i>	syntax, 2-11
comments	CONTINUE action
ANSI SQL-style, 2-10	of WHENEVER directive, 8-13, E-72
C-style, 2-10	CONVBUFSZ clause in VAR statement, 4-40
embedded SQL statements, 2-10	CREATE PROCEDURE statement, 6-16
in embedded SQL, 2-10	creating
comments in dynamic PL/SQL blocks, 9-23	savepoints, <i>E-61</i>
commit, 3-10	CURRENT OF clause, 5-13, 7-5
automatic, 3-11	example, 5-13
explicit versus implicit, 3-11	mimicking with ROWID, 3-17, 7-15
COMMIT statement, 3-11, E-12	restrictions, 5-14
effects, 3-11	current row, 5-10
ending a transaction, <i>E-60</i>	CURRVAL pseudocolumn, 4-10
example, 3-11	cursor cache, 8-20
examples, <i>E-13</i>	gathering statistics about, 8-22
RELEASE option, 3-11	purpose, <i>8-19</i> , <i>C-7</i>
using in a PL/SQL block, 3-20	cursor variables, 6-23, E-8
where to place, 3-11	advantages, 6-23
committing	allocating, 6-24
transactions, <i>E-12</i>	closing, 6-27
COMP_CHARSET precompiler option, 14-16	declaring, 6-24
compilation, 2-23	fetching from, 6-26
composite types, 11-15	heap memory usage, 6-24
concurrency, 3-9	

cursor variables (continued)	datatypes (continued)
opening	need to coerce, 11-16
anonymous block, 6-26	PL/SQL equivalents, 11-15
stored procedure, 6-24	TIMESTAMP, 4-11
restrictions, 6-27	TIMESTAMP WITH LOCAL TIME ZONE,
scope, 6-24	4-11
cursors, 5-10	TIMESTAMP WITH TIME ZONE, 4-11
allocating, <i>E-8</i>	when to reset, 11-16
association with query, 5-10	DATE datatype
closing, <i>E-11</i>	converting, 4-37
declaring, 5-10	default format, 4-37
effects on performance, C-5	DATE String Format
explicit versus implicit, 5-10	explicit control over, 4-37
fetching rows from, E-35, E-37	DATE_FORMAT precompiler option, 14-17
naming, 5-10	DATE, ANSI
opening, <i>E-54</i> , <i>E-56</i>	datatype, 4-11
reopening, 5-11, 5-12	DBMS precompiler option, 14-17, 14-18
restricted scope of, 2-23	DDL, <u>14-31</u>
restrictions, 5-11	DDL (Data Definition Language), 5-1
scope, 5-11	deadlock, 3-10
using for multirow queries, 5-10	effect on transactions, 3-14
using more than one, 5-11	how broken, 3-14
when closed automatically, 5-13	DEALLOCATE DESCRIPTOR statement, <i>E-17</i>
when dioded automationly, o 10	DECIMAL datatype, 4-6
	declaration
D	cursor, 5-10
data definition language (DDL)	host variable, 2-6
data definition language (DDL)	declarative SQL statement, 2-3
description, 5-1 embedded, 14-31	using in transactions, 3-11
	declarative statements
data integrity, 3-9	also known as directives, 2-3
data lock, 3-10	DECLARE CURSOR directive, <i>E-18</i>
database links	example, 5-10
using in DELETE statement, <i>E-25</i>	examples, E-20
using in INSERT statement, <i>E-44</i>	DECLARE CURSOR statement
using in UPDATE statement, <i>E-68</i>	AT clause, 3-6
datatype	in dynamic SQL method 4, 11-25
internal versus external, 2-6	
datatype conversion	where to place, 5-11 DECLARE DATABASE directive, <i>E-20</i>
between internal and external types, 4-37	declare section
datatype equivalencing	
advantages, 4-38	defining usernames and passwords, 3-2 Declare Section
example, <i>4-41</i>	
guidelines, 4-42	allowable statements, 2-16
datatypes	example, 2-16
ANSI DATE, 4-11	purpose, 2-16
coercing NUMBER to VARCHAR2, 11-16	requirements, 2-16
conversions, 4-36	rules for defining, 2-16
dealing with Oracle internal, 11-16	using more than one, 2-17
descriptor codes, 11-16	DECLARE statement
equivalencing	using in dynamic SQL method 3, 9-15
description, 4-38	DECLARE STATEMENT directive, <i>E-21</i>
example, 4-40	examples, <i>E-21</i>
internal, <i>11-13</i>	scope of, E-22
INTERVAL DAY TO SECOND, 4-11	DECLARE STATEMENT statement
INTERVAL YEAR TO MONTH, 4-11	AT clause, 3-6

DECLARE STATEMENT statement (continued) example, 9-21	DO CALL action of WHENEVER directive, 8-14, 8-16, E-72
using in dynamic SQL, 9-21	DO PERFORM action
when required, 9-21	of WHENEVER directive, 8-14, E-72
DECLARE TABLE directive, <i>E-23</i>	DSNTIAR
examples, <i>E-23</i>	DB2 compatible feature, 8-12
DECLARE TABLE directives	DSNTIAR routine, 8-12
using with the SQLCHECK option, <i>D</i> -3	DYNAMIC option
DECLARE SECTION precompiler option, 14-18	effect on functionality, 10-9
declaring	dynamic PL/SQL, 9-22
cursor variables, 6-24	dynamic SQL
host tables, 7-2	advantages and disadvantages, 9-2
indicator variables, 4-20	boolean support, 9-26
ORACA, <i>8-20</i>	choosing the right method, 9-5
SQLCA, 8-7	guidelines, 9-5
SQLDA, 11-6	overview, 2-5, 9-1
VARCHAR variables, 4-23	restrictions, 14-31
default	using PL/SQL, 6-19
error handling, 8-12	using the AT clause, 3-6
setting of LITDELIM option, 2-12, 14-26	when useful, 9-2
setting of ORACA option, 8-20	dynamic SQL (ANSI), <i>F-4</i>
default connection, 3-4	ALLOCATE DESCRIPTOR statement, 10-10
DEFINE precompiler option, 14-19	basics, 10-1
DELETE statement, <i>E-24</i>	bulk operations, <i>10-7</i>
embedded SQL examples, <i>E-26</i>	CLOSE CURSOR statement, 10-21
example, 5-8	compared to Oracle dynamic SQL method 4,
restrictions with tables, 7-12	10-1
using host tables, 7-11	DEALLOCATE DESCRIPTOR statement,
WHERE clause, 5-8	10-11
DEPENDING ON clause, 7-2	DESCRIBE DESCRIPTOR statement, 10-17
DEPT table, 2-23	differences from Oracle dynamic SQL, 10-22
DESCRIBE BIND VARIABLES statement	EXECUTE statement, 10-18
in dynamic SQL method 4, 11-25	FETCH statement, 10-20
DESCRIBE DESCRIPTOR statement, <i>E-28</i>	GET DESCRIPTOR statement, 10-12
DESCRIBE SELECT LIST statement	OPEN statement, 10-20
in dynamic SQL method 4, 11-29	Oracle extensions, 10-6
DESCRIBE statement, <i>E-26</i>	overview, 10-2
	precompiler options, 10-2, 10-9
example, E-27	
use with PREPARE statement, <i>E-26</i>	reference semantics, 10-6 restrictions, 10-22
using in dynamic SQL method 4, 9-19	•
descriptor	sample programs, <i>10-22</i> SAMPLE12.PCO, <i>10-22</i>
naming, <i>E-27</i> descriptors	SET DESCRIPTOR statement, 10-14
•	use of DECLARE CURSOR, 10-19
bind descriptor, 11-3 purpose, 11-3	•
	use of EXECUTE IMMEDIATE statement,  10-19
select descriptor, 11-3	
SQLADR subroutine, 11-3	use of PREPARE statement, 10-16
dimension of host tables, 7-2 directives	when to use, 10-1
	dynamic SQL method 1
also known as declarative statements, 2-3	commands, 9-3
directory path	description, 9-6
INCLUDE files, 2-18	example, 9-7
DISPLAY datatype, 4-6	requirements, 9-3
distributed processing, 3-3	using EXECUTE IMMEDIATE, 9-6
DML returning clause, 5-7	using PL/SQL, 9-22

dynamic SQL method 2 commands, 9-4 description, 9-10 requirements, 9-4 using PL/SQL, 9-22 using the DECLARE STATEMENT statement, 9-21 using the EXECUTE statement, 9-10	embedded PL/SQL (continued) cursor FOR loop, 6-3 example, 6-6, 6-7 host variables, 6-1 indicator variables, 6-1 multibyte Globalization Support features, 4-33 need for SQLCHECK option, 6-6
using the PREPARE statement, 9-10	need for USERID option, 6-6
dynamic SQL method 3	overview
commands, 9-4	support for SQL, 2-5
compared to method 2, 9-14	package, 6-4
requirements, 9-4	PL/SQL table, 6-4
using PL/SQL, 9-23	requirements, 6-1
using the DECLARE statement, 9-15	subprogram, 6-3
using the DECLARE STATEMENT	user-defined record, 6-5
statement, 9-21	using %TYPE, 6-3
using the FETCH statement, 9-15	using the VARCHAR pseudotype, 6-9
using the OPEN statement, 9-15	using to improve performance, C-3
using the PREPARE statement, 9-14	VARCHAR variables, 6-1
dynamic SQL method 4	where allowed, 6-1, 6-6
CLOSE statement, 11-32	embedded SQL
DECLARE CURSOR statement, 11-25	ALLOCATE DESCRIPTOR statement, <i>E-9</i>
DESCRIBE statement, 11-25, 11-29	ALLOCATE statement, 4-29, 6-24, E-8
external datatypes, 11-13	CALL statement, 6-17, E-10
FETCH statement, 11-31	CLOSE statement, 5-13, 6-27, E-11
internal datatypes, 11-13	COMMIT statement, <i>E-12</i>
OPEN statement, 11-28	CONNECT statement, <i>E-13</i>
PREPARE statement, 11-24	CONTEXT ALLOCATE statement, 12-6, E-15
prerequisites, 11-12	CONTEXT FREE statement, 12-7, E-15
purpose of descriptors, 11-3	CONTEXT USE directive, <i>E-16</i>
requirements, 9-4, 11-2	DECLARE [CURSOR] directive, 5-10
sequence of statements used, 11-19	DEALLOCATE DESCRIPTOR statement,
SQLDA, 11-3	E-17
steps for, 11-19	DECLARE CURSOR directive, E-18
using DL/SQL 0.22	DECLARE DATABASE directive, <i>E-20</i> DECLARE STATEMENT directive, <i>E-21</i>
using PL/SQL, 9-23 using the DECLARE STATEMENT	DECLARE STATEMENT directive, E-21  DECLARE TABLE directive, E-23
statement, 9-21	DELETE statement, 5-8, E-24
using the DESCRIBE statement, 9-19	DESCRIBE DESCRIPTOR statement, <i>E-28</i>
using the FOR clause, 9-22	DESCRIBE Statement, E-26
using the SQLDA, 9-19	ENABLE THREADS statement, 12-6
when needed, 9-19	EXECUTE IMMEDIATE statement, <i>E-34</i>
dynamic SQL statement, 9-2	EXECUTE statement, <i>E-31</i>
binding of host variables, 9-3	EXECUTEEND-EXEC statement, <i>E-30</i>
how processed, 9-3	FETCH DESCRIPTOR statement, E-37
requirements, 9-2	FETCH statement, 5-12, 6-26, E-35, E-37
using host tables, 9-22	FREE statement, 6-27, <i>E</i> -39
versus static SQL statement, 9-2	GET DESCRIPTOR statement, <i>E-40</i>
,	INSERT statement, 5-7, 7-10, E-42
_	key concepts, 2-1
<u>E</u>	OPEN DESCRIPTOR statement, <i>E-56</i>
embedded DDL, 14-31	OPEN statement, 5-11, E-54, E-56
embedded PL/SQL	PREPARE statement, E-58
advantages, 6-2	ROLLBACK statement, <i>E-59</i>

embedded SQL (continued)	error reporting (continued)
SAVEPOINT statement, 3-14, E-61	rows-processed count, 8-7
SELECT statement, 5-6, 7-5, E-62	status codes, 8-7
SET DESCRIPTOR statement, E-65	warning flags, 8-7
SET TRANSACTION statement, 3-16	WHENEVER directive, <i>E-72</i>
UPDATE statement, 5-8, E-66	ERRORS precompiler option, 14-20
VAR directive, E-70	exception, PL/SQL, 6-10
versus interactive SQL, 2-4	EXEC ORACLE DEFINE statement, 2-20
when to use, 1-3	EXEC ORACLE ELSE statement, 2-20
WHENEVER directive, <i>E-71</i>	EXEC ORACLE ENDIF statement, 2-20
embedded SQL statements	EXEC ORACLE IFDEF statement, 2-20
associating paragraph names with, 2-14	EXEC ORACLE IFNDEF statement, 2-20
comments, 2-10	EXEC ORACLE statement
continuation, 2-11	scope of, <u>14-5</u>
figurative constants, 2-13	syntax for, <i>14-5</i>
mixing with host-language statements, 2-4	uses for, 14-5
referencing host tables, 7-3	using to enter options inline, 14-5
referencing host variables, 4-18	EXEC SQL clause, 2-4, 2-12
referencing indicator variables, 4-21	EXECUTE IMMEDIATE statement, <i>E-34</i>
requirements, 2-12	AT clause, 3-6
summary, <i>E-2</i>	examples, <i>E-35</i>
syntax, 2-4, 2-12	using in dynamic SQL Method 1, 9-6
terminator, 2-16	EXECUTE optional keyword of ARRAYLEN
embedding	statement, 6-13
PL/SQL blocks in Oracle7 precompiler	EXECUTE statement, <i>E-31</i>
	examples, <i>E-31</i> , <i>E-32</i>
programs, <i>E-30</i> EMP table, <i>2-23</i>	•
	using in dynamic SQL Method 2, 9-10
ENABLE THREADS SQL statement, <i>E-29</i>	EXECUTEEND-EXEC statement, <i>E-30</i>
ENABLE THREADS statement, 12-6	execution plan, C-3 EXPLAIN PLAN statement
enabling	
threads, 12-6	using to improve performance, <i>C-4</i>
encoding scheme, 4-31	explicit logon
END_OF_FETCH precompiler option, 14-19	single, 3-4
equivalencing	explicit logons, 3-4
host variable equivalencing, <i>E-70</i>	external datatypes
equivalencing datatypes, 4-38	CHAR, 4-5
error detection	CHARF, 4-5
error reporting, <i>E-72</i>	CHARZ, 4-5
error handling	DECIMAL, 4-6
alternatives, 8-1	definition, 2-6
benefits, 8-1	DISPLAY, 4-6
default, 8-12	dynamic SQL method 4, 11-13
overview, 2-7	FLOAT, 4-6
using status variables	INTEGER, 4-6
SQLCA, 8-2, 8-6	LONG, 4-6
using the ROLLBACK statement, 3-13	LONG RAW, 4-6
using the SQLGLS routine, 8-18	LONG VARCHAR, 4-7
error message text	LONG VARRAW, 4-7
SQLGLM subroutine, 8-11	parameters, 4-39
error messages	RAW, <i>4-7</i>
maximum length, 8-11	STRING, 4-8
error reporting	table of, 4-3
error message text, 8-8	UNSIGNED, 4-8
key components of, 8-7	VARCHAR, 4-8
parse error offset, 8-7	VARCHAR2, 4-8

external datatypes <i>(continued)</i> VARNUM, 4-9	Globalization Support, 4-30, 14-30 multibyte character strings, 4-32
VARRAW, 4-9	Globalization Support parameter
	NLS_LANG, <i>4-31</i>
F	GOTO action
	of WHENEVER directive, 8-14, E-72
FETCH SQL statement, <i>E-37</i>	group items
FETCH statement, 5-12, E-35	allowed as host variables, 4-18
cursor variable, 6-26	implicit VARCHAR, <i>4-24</i> guidelines
example, 5-12	datatype equivalencing, 4-42
examples, E-37	dynamic SQL, 9-5
in dynamic SQL method 4, 11-31	separate precompilation, 2-22
INTO clause, 5-12	transaction, 3-19
used after OPEN statement, E-55, E-57	transaction, 5-19
using in dynamic SQL method 3, 9-15	
fetch, batch, 7-6	Н
fetching	handar files 1 C
rows from cursors, <i>E-35</i> , <i>E-37</i>	header files, 1-6
figurative constants	heap, 8-20
embedded SQL statements, 2-13	heap memory
file extension	allocating cursor variables, 6-24
for INCLUDE files, 2-17	heap tables, 4-28
file length limit, 2-13	hint, optimizer, <i>C-4</i> hints
FIPS flagger	
warns of array usage, 7-5	in DELETE statements, <i>E-26</i>
FIPS precompiler option, 14-21	in SELECT statements, <i>E-64</i>
flags, 8-7	in UPDATE statements, <i>E-69</i>
FLOAT datatype, 4-6	HOLD_CURSOR option
FOR clause, 7-13	of ORACLE Precompilers, <i>E-11</i> using to improve performance, <i>C-9</i>
example, 7-13	
of embedded SQL EXECUTE statement,	what it affects, <i>C-5</i>
E-32, E-33	HOLD_CURSOR precompiler option, 14-22
of embedded SQL INSERT statement, <i>E-43</i> restrictions, <i>7-13</i>	host language, 2-2 HOST precompiler option, 14-23
using with host tables, 7-13	host programs, 2-2
FOR UPDATE OF clause, 3-17	host table elements
FORCE clause	maximum, 7-3
of COMMIT statement, <i>E-13</i>	host table example, 7-8
of ROLLBACK statement, <i>E-60</i>	host table example, 7-0
format mask, 4-37	advantages, 7-1
FORMAT precompiler option, 14-22	declaring, 7-2
purpose, 2-10	dimensioning, 7-2
formats of COBOL statements	multi-dimensional, 7-2
ANSI, 2-9	operations on, 2-7
TERMINAL, 2-9	referencing, 7-3
forward reference, 5-11	restrictions, 7-2, 7-7, 7-10, 7-12
FREE statement, <i>E-39</i>	restrictions on, 7-5
freeing	support for, 4-17
thread context, 12-7, E-15	using in dynamic SQL statements, 9-22
full scan, C-5	using in the DELETE statement, 7-11
idii Jodii, C J	using in the INSERT statement, 7-11
	using in the SELECT statement, 7-10
G	using in the UPDATE statement, 7-10
GET DESCRIPTOR statement, <i>E-40</i>	using in the WHERE clause, 7-14
OLI DESCRII TOR SIGIETTETH, E-40	using the FOR clause, 7-13

host tables (continued)	indicator variables, 5-2
using to improve performance, C-2	assigning values to, 4-20
variable-length, 7-2	association with host variables, 4-20
host variables, 5-1	declaring, 2-9, 4-20
assigning a value, 2-5	function, 4-20
declaring, 2-9, 2-16	function of, 4-20
declaring and naming, A-1	interpreting value, 5-3
definition, 2-13	NULLs, 6-2
host variable equivalencing, <i>E-70</i>	referencing, 4-21
in EXECUTE statement, <i>E-32</i>	required size, 4-20
in OPEN statement, <i>E-55</i>	truncated values, 6-2
initializing, 4-17	used with multibyte character strings, 4-34
length up to 30 characters, 2-6	using in PL/SQL, 6-9
naming, 2-6, 4-18, 4-19	using to detect NULLs, 4-20
overview, 2-5	using to detect truncated values, 4-20
referencing, 2-6, 4-18	using to test for NULLs, 5-5
requirements, 2-5	with PL/SQL, 6-1
restrictions, 2-14, 4-19	input host variable
using in PL/SQL, 6-6	restrictions, 5-1
where allowed, 2-5	where allowed, 5-1
with PL/SQL, 6-1	INSERT statement, <i>E-42</i>
hyphenation	column list, 5-7
of host variable names, 2-14	example, 5-7
· · · · · · · · · · · · · · · · · · ·	INTO clause, 5-7
T.	using host tables, 7-10
1	VALUES clause, 5-7
IDE, <i>F-1</i>	inserting
identifiers, ORACLE	rows into tables and views, <i>E-42</i>
how to form, <i>E-7</i>	INTEGER datatype, 4-6
implicit logons, 3-9	internal datatype
implicit VARCHAR, 4-24	definition, 2-6
IN OUT parameter mode, 6-4	internal datatypes
IN parameter mode, 6-4	dynamic SQL method 4, 11-13
in-doubt transaction, 3-19	INTERVAL DAY TO SECOND datatype, 4-11
INAME option	INTERVAL YEAR TO MONTH datatype, 4-11
when a file extension is required, 14-1	INTO clause, 5-1, 6-26
INAME precompiler option, 14-24	FETCH statement, 5-12
INCLUDE precompiler option, 14-24	INSERT statement, 5-7
INCLUDE statement	of FETCH statement, <i>E-36</i> , <i>E-38</i>
case-sensitive operating systems, <i>2-18</i>	of SELECT statement, <i>E-64</i>
declaring the ORACA, 8-20	SELECT statement, 5-6
declaring the SQLCA, 8-7	IRECLEN precompiler option, 14-25
declaring the SQLDA, 11-6	IS NULL operator
effect of, 2-17	for testing NULL values, 2-14
INCLUDE statements, <i>A-1</i>	· · · · · · · · · · · · · · · · · · ·
index	1
using to improve performance, C-5	J
index-organized tables, 4-28	Java methods
indicator table, 7-1	calling from Pro*COBOL, 6-16
indicator tables	Calling Holli Flo COBOL, 0-10
example, 7-4	L
purpose, 7-4 indicator variable	Inches and the second of the s
	language support, 1-2
using to handle NULLs, 5-3, 5-4	LEVEL pseudocolumns, 4-10
	library file, 1-6

linking, 2-23 LENGTH attribute, 13-21 LOB demo program, 13-23 purpose, 14-26 LITDELIM precompiler option, 2-12, 14-25 LNAME precompiler option, 14-26 LOB and precompiler datatypes, 13-17 LOB APPEND statement, E-45 LOB ASSIGN statement, E-45 LOB ASSIGN statement, E-45 LOB	
purpose, 14-26 locators, 13-2 rules for all statements, 13-6 rules for buffering subsystem, 13-7 LOB APPEND statement, E-45 locators, 13-2 rules for all statements, 13-7 temporary, 13-3, 13-6	
LITDELIM precompiler option, 2-12, 14-25  LNAME precompiler option, 14-26  LOB and precompiler datatypes, 13-17  LOB APPEND statement, E-45  rules for all statements, 13-6  rules for buffering subsystem, 13-7  rules for statements, 13-7  temporary, 13-3, 13-6	
LNAME precompiler option, 14-26 rules for buffering subsystem, 13-7 rules for statements, 13-7 rules for statements, 13-7 temporary, 13-3, 13-6	
LOB and precompiler datatypes, 13-17 rules for statements, 13-7 temporary, 13-3, 13-6	
LOB APPEND statement, <i>E-45</i> temporary, <i>13-3</i> , <i>13-6</i>	
· · ·	
LOB ASSIGN statement, <i>E-45</i> using polling method to read and write,	
	13-21
LOB CLOSE statement, <i>E-46</i> lock	
LOB COPY statement, <i>E-46</i> released by ROLLBACK statement, <i>E-6</i>	0
LOB CREATE statement, <i>E-47</i> LOCK TABLE statement, <i>3-18</i>	
LOB DESCRIBE statement, <i>E-47</i> example, <i>3-18</i>	
LOB DISABLE BUFFERING statement, <i>E-48</i> using the NOWAIT parameter, <i>3-18</i>	
LOB ENABLE BUFFERING statement, <i>E-49</i> locking, <i>3-10</i> , <i>3-16</i>	
LOB ERASE statement, <i>E-49</i> explicit versus implicit, <i>3-16</i>	
LOB FILE CLOSE statement, <i>E-50</i> modes, <i>3-10</i>	
LOB FILE SET statement, <i>E-50</i> overriding default, <i>3-16</i>	
LOB FLUSH BUFFER statement, <i>E-51</i> privileges needed, <i>3-20</i>	
LOB FREE TEMPORARY, <i>E-51</i> using the FOR UPDATE OF clause, <i>3-1</i>	7
LOB LOAD statement, <i>E-52</i> using the LOCK TABLE statement, <i>3-18</i>	
LOB OPEN statement, <i>E-52</i> logons	
LOB READ statement, <i>E-53</i> automatic, 3-7	
LOB statements concurrent, 3-3	
LOB APPEND, 13-8 explicit, 3-4	
LOB ASSIGN, 13-9 requirements, 3-1	
LOB CLOSE, 13-9 LONG datatype	
LOB CLOSE ALL, 13-13 external, 4-6	
LOB COPY, 13-10 LONG RAW datatype	
LOB CREATE TEMPORARY, 13-11 converting, 4-43	
LOB DISABLE BUFFERING, 13-11 LONG RAW datatypes	
LOB ENABLE BUFFERING, 13-12 external, 4-6	
LOB ERASE, 13-12 LONG VARCHAR datatype, 4-7	
LOB FILE SET, 13-13 LONG VARRAW datatype, 4-7	
LOB FLUSH BUFFER, 13-14 LRECLEN precompiler option, 14-26	
LOB FREE TEMPORARY, 13-14 LTYPE precompiler option, 14-27	
LOB LOAD FROM FILE, 13-14	
LOP OPEN 12.15	
LOB OPEN, 13-13 LOB READ, 13-16	
LOB TRIM, 13-18 MAXLITERAL, A-2	
LOB WRITE, 13-18 MAXLITERAL precompiler option, 14-28	
LOB TRIM statement, <i>E-53</i> MAXOPENCURSORS option, <i>C-5</i>	
LOB WRITE statement, <i>E-54</i> using for separate precompilation, <i>2-22</i>	
LOBS MAXOPENCURSORS precompiler option, 2	1-28
advantage of buffering, 13-3  MERANT Micro Focus COBOL	.4-20
CHUNKSIZE attribute, 13-20  IDE, F-1	
compared with LONG and LONG RAW, 13-2  NetExpress, F-1	
1.6.11	
definition, 13-1 message text, 8-8 DIRECTORY attribute, 13-20 MODE	
FILEEXISTS attribute, 13-20 MODE option	
FILENAME attribute, 13-20 effects of, 4-26	

multi-byte Globalization Support features datatypes, <i>2-14</i>	NVL function for retrieving NULL values, 2-14
multibyte character sets, 4-32 multibyte Globalization Support features with PL/SQL, 4-33	О
multithreaded applications	ONAME precompiler option, 14-31
sample program, 12-12	OPEN DESCRIPTOR statement, <i>E-56</i>
user-interface features	OPEN SQL statement, E-56
embedded SQL statements and	OPEN statement, E-54
directives, 12-6	example, 5-11
	examples, E-56
N.I.	in dynamic SQL method 4, 11-28
N	using in dynamic SQL method 3, 9-15
namacharac	opening
reserved by Oracle, <i>B-4</i>	cursors, <i>E-54</i> , <i>E-56</i>
•	
naming	opening a cursor variable, 6-24
host variables, 2-13	operators
of database objects, <i>E-7</i>	relational, 2-16
select-list items, 11-3	optimizer hint, <i>C-4</i>
naming conventions	optional division headers, <i>2-12</i>
cursor, 5-10	options
default database, 3-4	precompiler concepts, 14-2
host variable, 2-6	ORACA, 1-6, 8-2
NATIVE	declaring, 8-19, 8-20
value of DBMS option, 14-17	enabling, 8-20
NESTED precompiler option, 14-30	example, 8-23
nested programs	fields, 8-20
support for, 2-19	gathering cursor cache statistics, 8-22
NetExpress, F-1	ORACABC field, 8-20
NEXTVAL pseudocolumns, 4-10	ORACAID field, 8-20
nibbles, 4-43	ORACCHF flag, 8-21
NLS_LOCAL	ORACOC field, 8-23
precompiler option, 14-30	ORADBGF flag, 8-21
NOT FOUND condition	ORAHCHF flag, 8-21
of WHENEVER directive, 8-13, E-72	ORAHOC field, 8-22
NOWAIT parameter, 3-18	ORAMOC field, 8-23
using in LOCK TABLE statement, 3-18	ORANEX field, 8-23
NULLs	ORANOR field, 8-23
definition, 2-6	ORANPR field, 8-23
detecting, 4-20	ORASFNMC field, 8-22
handling	ORASFNML field, 8-22
in dynamic SQL method 4, 11-18	ORASLNR field, 8-22
indicator variables, 6-2	ORASTXTC field, 8-22
hardcoding, 5-3	ORASTXTF flag, 8-21
inserting, 5-3	ORASTXTL field, 8-22
meaning in SQL (NVL function), 2-14	precompiler option, 8-20
restrictions, 5-5	purpose, 8-2, 8-19
retrieving, 5-4	structure of, 8-20
SQLNUL subroutine, 11-18	ORACA precompiler option, 14-23, 14-27,
	· · · · · · · · · · · · · · · · · · ·
testing for, 5-5	14-31-14-34, 14-39
NULLs in SQL	ORACAID field, 8-20
how to detect, 2-14	ORACAID field, 8-20
NUMBER datatype	ORACCHF flag, 8-21
using the SQLPRC subroutine with, 11-16	Oracle Communications Area
	ORACA, <u>8-19</u>

Oracle Connection Manager in Traffic Director Mode, <i>C-10</i>	passwords <i>(continued)</i> hardcoding, 3-2
Oracle dynamic SQL	performance
when to use, <i>10-1</i>	causes of poor, C-1
Oracle names	improving, C-2
how to form, E-7	PIC G for Globalization Support characters, A-2
Oracle namespaces, <i>B-4</i>	PIC N for Globalization Support characters, A-2
Oracle Net	PICX precompiler option, 4-26, 14-34
using ROWID datatype, 4-8	PL/SQL
using to connect to Oracle, 3-3	advantages, <del>1-3</del>
Oracle Net Services	cursor FOR loop, 6-3
concurrent logons, 3-3	datatype equivalents, 11-15
Oracle Open Gateway	embedded, 6-1
using ROWID datatype, 4-8	exception, 6-10
ORACOC	integration with server, 6-3
in ORACA, 8-23	opening a cursor variable
ORACOC field, 8-23	anonymous block, 6-26
ORADBGF flag, 8-21	stored procedure, 6-24
ORAHCHF flag, 8-21	package, 6-4
ORAHOC field, 8-22	relationship with SQL, 1-3
ORAMOC field, 8-23	subprogram, 6-3
ORANEX	user-defined record, 6-5
in ORACA, <i>8-23</i>	PL/SQL block execution
ORANEX field, 8-23	effect on SQLCA components, 8-11
ORANOR field, 8-23	PL/SQL blocks
ORANPR field, 8-23	embedded in Oracle7 precompiler programs,
ORASFNM, in ORACA, 8-22	E-30
ORASFNMC field, 8-22	PL/SQL subprogram
ORASFNML field, 8-22	calling from Pro*COBOL, 6-16
ORASLNR	PL/SQL table, 6-4
in ORACA, <i>8-22</i>	placeholders
ORASLNR field, 8-22	duplicate, 9-23
orasql11.lib library file, 1-6	PLAN_BASELINE
ORASTXTC field, 8-22	precompiler option, 14-35
ORASTXTF flag, 8-21	PLAN_ENABLED
ORASTXTL field, 8-22	precompiler option, 14-36
ORECLEN precompiler option, 14-32	PLAN_FIXED
OUT parameter mode, 6-4	precompiler option, 14-36
output host variable, 5-1	PLAN_PREFIX
output versus input, 5-1	precompiler option, 14-35
	PLAN_RUN
P	precompiler option, 14-36
<u> </u>	plan, execution, C-3
PAGELEN precompiler option, 14-33	precompilation
paragraph names	conditional, 2-20
associating with SQL statements, 2-14	generated code, 14-2
coding area for, 2-15	separate, 2-22
parameter mode, 6-4	precompilation unit, 14-10
parse error offset, 8-7	precompiler command
parsing dynamic statements	required arguments, 14-1
PREPARE statement, <i>E-58</i>	precompiler options
passwords	abbreviating name, 14-3
changing at runtime with ALTER	ANSI Dynamic SQL, 10-9
AUTHORIZATION, 3-8	ASACC, 14-13
defining, 3-2	ASSUME_SQLCODE, 14-13



precompiler options (continued)	precompiler options (continued)
AUTO_CONNECT, 3-8, 14-14	SQLCHECK, 14-40
CLOSE_ON_COMMIT, 5-10, 5-17, 14-15	syntax for, 14-1
COMP_CHARSET, 14-16	table of how macro options set micro options,
CONFIG, <i>14-16</i>	14-4
current values, 14-4	THREADS, <i>12-5</i>
DATE_FORMAT, 14-17	TYPE_CODE, 10-9, 14-43
DBMS, 14-17, 14-18	UNSAFE_NULL, 14-44
DECLARE_SECTION, 2-17, 14-18	USERID, <u>14-44</u>
DEFINE, 14-19	VARCHAR, 14-45
displaying, 14-3	XREF, 14-45
displaying syntax, default, purpose, 14-10	precompilers
DYNAMIC, 10-9, 14-19	advantages, 1-3
END_OF_FETCH, 14-19	Globalization Support, 4-32
entering, <i>14-5</i>	language support, 1-2
entering inline, 14-5	running, 14-1
entering on the command line, 14-1	using PL/SQL, 6-6
ERRORS, 14-20	PREFETCH precompiler option, 5-15, 5-17,
FIPS, 14-21	14-37
FORMAT, 14-22	PREPARE statement, <i>E-58</i>
HOLD_CURSOR, 14-22, C-5	effect on data definition statements, 9-4
HOST, 14-23	examples, <i>E-59</i>
INAME, 14-24	in dynamic SQL method 4, 11-24
INCLUDE, 14-24	using in dynamic SQL, 9-10, 9-14
IRECLEN, 14-25	private SQL area
list, 14-10	association with cursors, 5-9
LITDELIM, 2-12, 14-26	opening, 5-9
LNAME, 14-26	purpose, C-7
LRECLEN, 14-26	Pro*COBOL
LTYPE, <i>14-27</i>	how it works, 1-1
macro and micro, 14-3	program termination, 3-15
MAXLITERAL, 14-28	programming guidelines, 2-9
MAXOPENCURSORS, 2-22, 14-28, C-5	programming language support, 1-2
MODE, 4-26, 8-3, 10-9, 14-29	pseudocolumns, 4-9
name of the system configuration file, 14-6	CURRVAL, 4-10
NESTED, <i>14-30</i>	LEVEL, <i>4-10</i>
NLS_LOCAL, <i>14-30</i>	NEXTVAL, 4-10
ONAME, <i>14-31</i>	ROWNUM, <i>4-10</i>
ORACA, 8-20, 14-23, 14-27, 14-31-14-34,	
14-39	Q
ORECLEN, 14-32	<u> </u>
PAGELEN, <i>14-33</i>	queries
PICX, 4-26, 14-34	association with cursor, 5-10
PLAN_BASELINE, 14-35	single-row versus multirow, 5-6
PLAN_ENABLED, 14-36	query, 5-5
PLAN_FIXED, 14-36	4,
PLAN PREFIX, 14-35	D
PLAN RUN, <i>14-36</i>	R
precedence, 14-3	RAW datatype
PREFETCH, 5-15, 14-37	· · · · · · · · · · · · · · · · · · ·
RELEASE_CURSOR, 14-38, C-5	converting, 4-43
respecifying, 14-10	external, 4-7
scope of, 14-10	RAWTOHEX function, 4-43
SELECT_ERROR, <i>14-40</i>	read consistency, 3-10
specifying, 14-1	READ ONLY parameter
,,	using in SET TRANSACTION, 3-16

read-only transaction, 3-16	roll back
ending, 3-16	to a savepoint, <i>E-61</i>
example, 3-16	to the same savepoint multiple times, <i>E-60</i>
record, user-defined, 6-5	rollback
REDEFINES clause	automatic, 3-13
purpose, <i>2-15</i>	purpose, <i>3-10</i>
restrictions, 2-15	statement-level, 3-13
reference	rollback segments, 3-10
host variable, 2-6	ROLLBACK statement, 3-12, E-59
reference cursor, 6-23	effects, 3-12
referencing	ending a transaction, <i>E-60</i>
host tables, 7-3	example, <i>3-13</i>
host variables, 4-18	examples, <i>E-61</i>
indicator variables, 4-21	RELEASE option, 3-13
VARCHAR variables, 4-24	TO SAVEPOINT clause, 3-12
relational operators	using in a PL/SQL block, 3-20
COBOL versus SQL, 2-16	using in error-handling routines, 3-13
RELEASE option, <i>3-11</i> , <i>3-15</i>	where to place, 3-13
COMMIT statement, 3-11	rolling back
omitting, 3-16	transactions, <i>E-59</i>
restrictions, 3-15	row lock
ROLLBACK statement, 3-13	acquiring with FOR UPDATE OF, 3-17
RELEASE_CURSOR option, C-5	using to improve performance, <i>C-5</i>
of ORACLE Precompilers, <i>E-11</i>	when acquired, 3-17
RELEASE_CURSOR precompiler option, 14-38	when released, 3-17
remote database	ROWID datatype
declaration of, <i>E-20</i>	heap tables versus index-organized tables,
restrictions	4-28
AT clause, 3-6	Universal, 4-28
CURRENT OF clause, 5-14	use of, 4-28
cursor declaration, 5-11	use of ALLOCATE, 4-29
cursor variables, 6-27	ROWID pseudocolumns
dynamic SQL, 14-31	retrieving with SQLROWIDGET, 4-29
FOR clause, 7-13	universal ROWID, 4-28
host tables, 7-2, 7-7, 7-10, 7-12	using to mimic CURRENT OF, 3-17, 7-15
host tables, 7-2, 7-7, 7-10, 7-12 host variables, 4-19	ROWNUM pseudocolumns, <i>4-10</i>
naming, 2-13	rows
referencing, 4-19	fetching from cursors, <i>E-35</i> , <i>E-37</i>
input host variable, 5-1	inserting into tables and views, <i>E-42</i>
on host tables, 7-5	updating, E-66
REDEFINES clause, 2-15	rows-processed count, 8-7
RELEASE option, 3-15	RR diagrams
separate precompilation, 2-22	see syntax diagrams, <i>E-5</i>
SET TRANSACTION statement, 3-16	See Symax diagrams, L-3
SQLGLM subroutine, 8-12	S
SQLIEM subroutine, 8-12	
TO SAVEPOINT clause, 3-15	sample database table
use of CURRENT OF clause, 7-5	DEPT table, 2-23
retrieving rows from a table	EMP table, 2-23
embedded SQL, <i>E-62</i>	sample programs
RETURN-CODE special register is	calling a stored procedure, 6-19
unpredictable, A-2	cursor operations, 5-23
returning clause, 5-7	cursor variable use, 6-27
in INSERT, 5-7	cursor variables
	PL/SQL source, 6-27

sample programs (continued)	SELDI-MAX-VNAMEL variable (SQLDA), 11-11
datatype equivalencing, 4-44	SELDI-VNAME variable (SQLDA), 11-11
dynamic SQL Method 1, 9-8	SELDV variable (SQLDA), 11-8
dynamic SQL method 2, 9-11	SELDVLN variable (SQLDA), 11-8
dynamic SQL Method 3, 9-16	SELDVTYP variable (SQLDA), 11-9
dynamic SQL method 4, 11-37	select descriptor, 11-3
EXEC ORACLE scope, 14-5	select descriptors
fetching in batches, 7-8, 7-17	information in, 9-19
LOBDEMO1.PCO, 13-23	select list, 5-6
location of, 1-5	select SQLDA
PL/SQL examples, 6-6	purpose of, 11-3
SAMPLE1.PCO, 2-24	SELECT statement, <i>E-62</i>
SAMPLE10.PCO, <i>11-36</i>	available clauses, 5-7
SAMPLE11.PCO, 6-27	embedded SQL examples, <i>E-64</i>
SAMPLE12.PCO, 10-22	example, 5-6
SAMPLE13.PCO, 2-20	INTO clause, 5-6
SAMPLE14.PCO, 7-17	using host tables, 7-5
SAMPLE2.PCO, 5-23	SELECT_ERROR option, 5-6
SAMPLE3.PCO, 7-8	SELECT_ERROR precompiler option, 14-40
SAMPLE4.PCO, 4-44	select-list items
SAMPLE6.PCO, 9-8	naming, 11-3
SAMPLE7.PCO, 9-11	semantic checking, <i>D-1</i>
SAMPLE8.PCO, 9-16	enabling, D-2
SAMPLE9.PCO, 6-19	using the SQLCHECK option, <i>D-1</i>
simple query, 2-24	separate precompilation
tables of group items, 7-17	· · · · · · · · · · · · · · · · · · ·
<del>-</del> ,	guidelines, 2-22
WHENEVERDO CALL example, 8-16	restrictions, 2-22
savepoint, 3-14	session, 3-9
SAVEPOINT statement, 3-14, E-61	Sessions
example, 3-14	beginning, <i>E-13</i>
examples, <i>E-62</i>	SET clause, 5-8
savepoints	using a subquery, 5-8
creating, E-61	SET DESCRIPTOR statement, <i>E-65</i>
when erased, 3-15	SET TRANSACTION statement
scalar types, 11-15	example, 3-16
scale	READ ONLY parameter, 3-16
definition of, 4-39	restrictions, 3-16
using SQLPRC to extract, 4-39	snapshot, 3-10
when negative, 4-39	SQL
scope	summary of statements, <i>E-2</i>
cursor variables, 6-24	SQL codes
of DECLARE STATEMENT directive, <i>E-22</i>	returned by SQLGLS routine, 8-19
of precompiler options, 14-10	SQL Communications Area, 2-8
of the EXEC ORACLE statement, 14-5	SQL Descriptor Area, 9-19, 11-3
WHENEVER directive, 8-17	SQL directives
search condition, 5-8	CONTEXT USE, 12-7
using in the WHERE clause, 5-8	DECLARE CURSOR, <i>E-18</i>
SELDFCLP variable (SQLDA), 11-12	DECLARE DATABASE, <i>E-20</i>
SELDFCRCP variable (SQLDA), 11-12	DECLARE STATEMENT, <i>E-21</i>
SELDFMT variable (SQLDA), 11-8	DECLARE TABLE, <i>E-23</i>
SELDH-CUR-VNAMEL variable (SQLDA), 11-11	VAR, <i>E-70</i>
SELDH-MAX-VNAMEL variable (SQLDA), 11-11	WHENEVER, E-71
SELDH-VNAME variable (SQLDA), 11-10	SQL directives CONTEXT USE, <i>E-16</i>
SELDI variable (SQLDA), 11-10	SQL statements
SELDI-CUR-VNAMEL variable (SQLDA), 11-12	ALLOCATE, <i>E-8</i>

SQL statements (continued)	SQL-CONTEXT, 12-6
ALLOCATE DESCRIPTOR, E-9	host tables not allowed, 12-6
CALL, <i>E-10</i>	variable declaration, 4-14
CLOSE, <i>E-11</i>	SQL*Plus, 1-3
COMMIT, E-12	SQLADR subroutine
CONNECT, E-13	example, <u>11-22</u>
CONTEXT ALLOCATE, <i>E-15</i>	parameters, <u>11-12</u>
CONTEXT FREE, E-15	storing buffer addresses, 11-3
controlling transactions, 3-10	syntax, <i>11-12</i>
DEALLOCATE DESCRIPTOR, E-17	SQLCA, 1-6, 8-2
DELETE, <i>E-24</i>	components set for a PL/SQL block, 8-11
DESCRIBE, E-26	declaring EXTERNAL, 2-22
DESCRIBE DESCRIPTOR, E-28	fields, 8-8
ENABLE THREADS, <i>E-29</i>	interaction with Oracle, 2-8
EXECUTE, E-31	overview, 2-7
EXECUTE DESCRIPTOR, E-32	
	sharing, <i>2-22</i> SQLCABC field, <i>8-8</i>
EXECUTE IMMEDIATE, E-34	
EXECUTEEND-EXEC, E-30	SQLCAID field, 8-8
FETCH, <i>E-35</i> , <i>E-37</i>	SQLCODE field, 8-9
FETCH DESCRIPTOR, E-37	SQLERRD(3) field, 8-9
FREE, <i>E-39</i>	SQLERRD(5) field, <i>8-10</i>
GET DESCRIPTOR, E-40	SQLERRMC field, 8-9
INSERT, <i>E-42</i>	SQLERRML field, 8-9
LOB APPEND, <i>E-45</i>	SQLWARN(4) flag, 8-10
LOB ASSIGN, <i>E-45</i>	using with Oracle Net Services, 8-6
LOB CLOSE, <i>E-46</i>	SQLCA status variable
LOB COPY, <i>E-46</i>	data structure, 8-6
LOB CREATE, <i>E-47</i>	declaring, 8-7
LOB DESCRIBE, <i>E-47</i>	effect of MODE option, 8-3
LOB DISABLE BUFFERING, <i>E-48</i>	explicit versus implicit checking, 8-2
LOB ENABLE BUFFERING, <i>E-49</i>	purpose, 8-6
LOB ERASE, <i>E-49</i>	SQLCABC field, 8-8
LOB FILE CLOSE, <i>E-50</i>	SQLCAID field, 8-8
LOB FILE SET, <i>E-50</i>	SQLCHECK option
LOB FLUSH BUFFER, <i>E-51</i>	using the DECLARE TABLE statement, D-3
LOB FREE TEMPORARY, <i>E-51</i>	using to check syntax/semantics, D-1
LOB LOAD, <i>E-52</i>	SQLCHECK precompiler option, 14-40
LOB OPEN, E-52	SQLCODE field, 8-9
LOB READ, E-53	interpreting its value, 8-9
LOB TRIM, E-53	SQLCODE status variable
LOB WRITE, E-54	effect of MODE option, 8-3
OPEN, <i>E-54</i> , <i>E-56</i>	usage, <i>8-3</i>
OPEN DESCRIPTOR, E-56	SQLDA, 1-6, 9-19, 9-20
optimizing to improve performance, C-3	bind versus select, 9-20
PREPARE, <i>E-58</i>	BNDDFCLP variable, 11-12
ROLLBACK, <i>E-59</i>	BNDDFCRCP variable, 11-12
SAVEPOINT, <i>E-61</i>	BNDDFMT variable, 11-8
SELECT, E-62	BNDDH-CUR-VNAMEL variable, 11-11
•	BNDDH-MAX-VNAMEL variable, 11-11 BNDDH-MAX-VNAMEL variable, 11-11
SET DESCRIPTOR, E-65	BNDDH-WAME variable, 11-11 BNDDH-VNAME variable, 11-10
static versus dynamic, 2-5	
summary of, <i>E-2</i>	BNDDI variable, 11-10
UPDATE, E-66	BNDDI-CUR-VNAMEL variable, 11-12
using to control a cursor, 5-6, 5-10	BNDDI-MAX-VNAMEL variable, 11-11
SQL_CURSOR, <i>E-8</i>	BNDDI-VNAME variable, 11-11
	BNDDV variable, 11-8

SQLDA (continued)	SQLNUL subroutine (continued)
BNDDVLN variable, 11-8	purpose, <i>11-18</i>
BNDDVTYP variable, 11-9	syntax, <i>11-18</i>
declaring, 11-6	SQLPRC subroutine
example, <u>11-6</u>	example, <i>11-17</i>
information stored in, 9-20	parameters, 11-16
purpose, 11-3	purpose, <u>11-16</u>
SELDFCLP variable, 11-12	syntax, <u>11-16</u>
SELDFCRCP variable, 11-12	SQLROWIDGET
SELDFMT variable, 11-8	retrieving ROWID of last row inserted, 4-29
SELDH-CUR-VNAMEL variable, 11-11	SQLSTATE
SELDH-MAX-VNAMEL variable, 11-11	declaring, 8-3
SELDH-VNAME variable, 11-10	SQLSTATE status variable
SELDI variable, 11-10	class code, 8-3
SELDI-CUR-VNAMEL variable, 11-12	coding scheme, 8-3
SELDI-MAX-VNAMEL variable, 11-11	effect of MODE option, 8-3
SELDI-VNAME variable, 11-11	interpreting values, 8-3
SELDV variable, 11-8	predefined classes, 8-4
SELDVLN variable, 11-8	subclass code, 8-3
SELDVTYP variable, 11-9	usage, <b>8-3</b>
SQLADR subroutine, 11-12	SQLSTM parameter, 8-18
SQLDFND variable, 11-7	SQLSTM routine, 8-18
SQLDNUM variable, 11-7	SQLWARN(4) flag, 8-10
structure, 11-7	SQLWARNING
SQLDFND variable (SQLDA), 11-7	condition WHENEVER directive, <i>E-72</i>
SQLDNUM variable (SQLDA), 11-7	SQLWARNING condition, 8-13
SQLERRD(3) field, 8-9	of WHENEVER directive, 8-13
using with batch fetch, 7-7	statement-level rollback, 3-13
SQLERRD(3) variable, 8-7	breaking deadlocks, 3-14
SQLERRD(5) field, 8-10	status codes for error reporting, 8-7
SQLERRMC field, 8-9	STMLEN parameter, 8-18
SQLERRMC variable, 8-8	STOP action
SQLERRML field, 8-9	of WHENEVER directive, 8-14, E-72
SQLERROR condition, 8-13	stored procedure
of WHENEVER directive, 8-13	opening a cursor, 6-27
WHENEVER directive, E-72	sample programs, 6-27
SQLFC parameter, 8-19	stored procedures
SQLGLM subroutine	opening a cursor, 6-24
example, <i>8-11</i>	sample programs, 6-19
parameters, 8-11	stored subprogram
provides DSNTIAR support for DB2	calling, 6-17
conversions, 8-12	creating, 6-16
purpose, <b>8-11</b>	packaged versus standalone, 6-16
restrictions, 8-12	stored versus inline, C-3
syntax, 8-11	using to improve performance, C-3
SQLGLS routine, 8-18	stored subprogram, calling, 6-16
parameters, 8-18	STRING datatype, 4-8
SQL codes returned by, 8-19	string literals
syntax, 8-18	continuing on the next line, 2-11
using to obtain SQL text, 8-18	subprogram, PL/SQL, 6-3
SQLIEM subroutine	subprogram, PL/SQL or Java, 6-16
restrictions, 8-12	subquery, 5-7
SQLNUL subroutine	example, 5-8
example, 11-18	using in the SET clause, 5-8
parameters, 11-18	using in the VALUES clause, 5-8

syntactic checking, <i>D-1</i>	transaction, 3-10
syntax	transactions
continuation lines, 2-11	committing, <i>E-12</i>
embedded SQL statements, 2-12	contents, 3-11
SQLADR subroutine, 11-12	guidelines, 3-19
SQLGLM subroutine, 8-11	how to begin, 3-11
SQLNUL subroutine, 11-18	how to end, 3-11
SQLPRC, 11-16	in-doubt, 3-19
syntax diagrams	making permanent, 3-11
description of, E-5	read-only, 3-16
how to read, <i>E-5</i>	rolling back, <i>E-59</i>
how to use, <i>E-5</i>	subdividing with savepoints, 3-14
symbols used in, <i>E-5</i>	undoing, <i>3-12</i>
syntax, embedded SQL, 2-4	undoing parts of, 3-14
SYSDATE function, 4-11	when rolled back automatically, 3-11, 3-13
SYSDBA privilege show to set, 3-9	truncated values, 6-10
SYSOPER privilege	detecting, 4-20
how to set, 3-9	indicator variables, 6-2
system failures	truncation errors
effect on transactions, 3-11	when generated, 5-5
System Global Area (SGA), 6-16	tuning, performance, <i>C-1</i>
	TYPE statements
Т	using the CHARF datatype specifier, 4-42
<u> </u>	TYPE_CODE option
table (host) elements	effect on functionality, 10-10
maximum, 7-3	TYPE_CODE precompiler option, 14-43
table lock	
acquiring with LOCK TABLE, 3-18	U
row share, <i>3-18</i>	
when released, 3-18	UID function, 4-11
tables	undo a transaction, <i>E-59</i>
elements, 7-1	universal ROWID
index-organized, 4-28	ROWID pseudocolumns, 4-28
inserting rows into, <i>E-42</i>	UNSAFE_NULL precompiler option, 14-44
updating rows in, <i>E-66</i>	UNSIGNED datatype, 4-8
tables, host, 7-1	UPDATE statement, <i>E</i> -66
TERMINAL format	embedded SQL examples, <i>E-69</i>
COBOL statements, 2-9	example, 5-8
terminator for embedded SQL statements, 2-16	SET clause, 5-8
threads, <i>E-15</i>	using host tables, 7-10
allocating context, 12-6, E-15	updating
enabling, <i>12-6</i> , <i>E-29</i>	rows in tables and views, <i>E-66</i>
freeing context, 12-7, E-15	use
use context, 12-7, E-16	thread context, 12-7
THREADS	USER function, 4-11
precompiler option, 12-5	user session, 3-9
TIMESTAMP datatype, 4-11	user-defined record, 6-5
TIMESTAMP WITH LOCAL TIME ZONE	USERID option
datatype, 4-11	using with the SQLCHECK option, <i>D-2</i>
TIMESTAMP WITH TIME ZONE datatype, 4-11	USERID precompiler option, 14-44
TO SAVEPOINT clause, 3-14	usernames
restrictions, 3-15	defining, 3-2
using in ROLLBACK statement, 3-14	hardcoding, 3-2
trace facility	USING clause
using to improve performance, C-4	CONNECT statement, 3-5

USING clause (continued)	VARYING keyword
of FETCH statement, E-36	versus VARYING phrase, 4-23
of OPEN statement, E-55	versions of COBOL supported, 2-9
using in the EXECUTE statement, 9-11	views
using indicator variables, 9-11	inserting rows into, <i>E-42</i>
using dbstring	updating rows in, <i>E-66</i>
Oracle Net Services database id	
specification, <i>E-14</i>	W
V	warning flags for error reporting, 8-7
<u> </u>	WHENEVER
V7	DO CALL example, 8-16
value of DBMS option, 14-17	WHENEVER directive, 8-12, E-71
VALUE clause	careless usage, 8-17
initializing host variables, 4-17	CONTINUE action, 8-13
VALUES clause	DO CALL action, 8-14
INSERT statement, 5-7	DO PERFORM action, 8-14
of embedded SQL INSERT statement, <i>E-44</i>	example, 8-15
of INSERT statement, <i>E-44</i>	examples, <i>E-72</i>
using a subquery, 5-8	GOTO action, 8-14
VAR directive, <i>E-70</i>	overview, 2-8
examples, <i>E-71</i>	purpose, 8-13
VAR statement	scope of, 8-17
CONVBUFSZ clause, 4-40	SQLERROR condition, 8-13
syntax for, 4-38	SQLWARNING condition, 8-13
using the CHARF datatype specifier, 4-42	STOP action, 8-14
VARCHAR datatype, 4-8	syntax, <i>8-14</i>
VARCHAR precompiler option, 14-45	using to check SQLCA automatically, 8-13
VARCHAR pseudotype	WHERE clause, 5-8
using with PL/SQL, 6-9	DELETE statement, 5-8
VARCHAR variables	of DELETE statement, <i>E-25</i>
advantages, 4-28	of UPDATE statement, <i>E-68</i>
as input variables, 4-27	search condition, 5-8
as output variables, 4-27	SELECT statement, 5-6
declaring, 4-23	UPDATE statement, 5-8
implicit group items, 4-24	using host tables, 7-14
length element, 4-23	WHERE CURRENT OF clause, 5-13
maximum length, 4-23	WITH HOLD
referencing, 4-24	clause of DECLARE CURSOR statement,
server handling, 4-27	5-11
string element, 4-23	WORK option
structure, 4-23	of COMMIT statement, <i>E-12</i>
versus fixed-length strings, 4-28	of ROLLBACK statement, <i>E-60</i>
with PL/SQL, 6-1	
VARCHAR2 datatype	X
external, 4-8	
VARNUM datatype, 4-9	XREF precompiler option, 14-45
VARRAW datatype, 4-9	

