Progress SQL-92 Guide and Reference

.

May 2001

Product Code: 4535
Item Number: 81104;9.1C

Contents

Figures

Tables

Procedures

Preface

Purpose

The manual describes the syntax and semantics of Progress SQL-92 language statements and elements.

Audience

This manual is intended for application programmers writing database applications using the Progress SQL-92 environment.

Organization of This Manual

Chapter 1, "Getting Started with SQL-92"

Introduces the application developer to the SQL-92 environment, including the interactive user interface tool SQL Explorer. Provides an overview of related documentation and resources.

Chapter 2, "SQL-92 Language Elements"

Describes language elements that are common to many SQL statements.

Chapter 3, "SQL-92 Statements"

Provides detailed reference information on each SQL statement. The statements are presented in alphabetical order.

Chapter 4, “SQL-92 Functions”

Provides detailed reference information on SQL functions. The functions are presented in alphabetical order.

Chapter 5, “Java Stored Procedures and Triggers”

Describes the Java stored procedures and triggers that SQL Server processes execute.

Chapter 6, “Java Class Reference”

Provides reference information on Java classes and methods.

Appendix A, “Progress SQL-92 Reference Information”

Provides information on these topics:

- List of Progress SQL-92 reserved words,
- Enumeration of Progress SQL-92 system limits, and
- List of Progress SQL-92 error messages.

Appendix B, “Progress SQL-92 System Catalog Tables”

Lists column names, column data types, and column lengths for system tables in the Progress SQL-92 system catalog.

Appendix C, “Data Type Compatibility Issues with Previous Versions of Progress”

Lists Progress supported data types and corresponding Progress SQL-92 data types. Describes support for the ARRAY data type.

Appendix D, “Progress SQL-92 Elements and Statements in Backus Naur Form (BNF)”

Presents SQL-92 language elements and SQL-92 syntax in Backus Naur Form.

Appendix E, “Compliance with Industry Standards”

Provides information on ANSI SQL-92 compliance and ODBC compatibility for supported Progress functions, DDL Statements, and DML Statements.

“Glossary”

Provides definition of new terms.

Typographical Conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
  - Commands or characters that the user types
  - That a word carries particular weight or emphasis
- *Italic typeface* indicates:
  - Progress variable information that the user supplies
  - New terms
  - Titles of complete publications
- `Monospaced typeface` indicates:
  - Code examples
  - System output
  - Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.

  END-ERROR, **GET**, **GO**
  ALT, **CTRL**, **SPACEBAR**, **TAB**

- When you have to press a combination of keys, they are joined by a dash. You press and hold down the first key, then press the second key.

  CTRL-X

- When you have to press and release one key, then press another key, the key names are separated with a space.

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each SQL-92 component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in an expression or a statement.

  In this example, GRANT, RESOURCE, DBA, and TO are keywords:

  SYNTAX

  ```
  GRANT { RESOURCE, DBA } TO user_name [, user_name ] ... ;
  ```

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the GRANT statement you must supply at least one *user_name*.

- Square brackets (**[ ]**) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

  In this syntax example the first *user_name* is required, and additional *user_name* specifications are optional:

  SYNTAX

  ```
  GRANT { RESOURCE, DBA } TO user_name [, user_name ] ... ;
  ```

  NOTE: The ellipsis (**. . .**) indicates repetition, as shown in a following description.

- Braces (**{ }**) around an item indicate that the item, or a choice of one of the enclosed items, is required.

  In the GRANT example, you must specify either RESOURCE or DBA or both, and at least one *user_name*. Any *user_name* specifications after the first are optional:

  SYNTAX

  ```
  GRANT { RESOURCE, DBA } TO user_name [, user_name ] ... ;
  ```

  In some cases, braces are not a syntax notation, but part of the language.

For example, this excerpt from an ODBC application invokes a stored procedure using the ODBC syntax { call *procedure_name* (*param*) }, where braces and parentheses are part of the language:

SYNTAX

```
proc1( param, "{ call proc2 (param) }", param);
```

- A vertical bar (|) indicates a choice. In the CREATE SYNONYM example you must specify a *table_name*, *view_name*, or *synonym* but you can only choose one. Note that in all SQL-92 syntax, if you specify the optional *owner_name* qualifier, there must not be a blank between the period separator and the *table_name* or *view_name* or *synonym*:

SYNTAX

```
CREATE [ PUBLIC ] SYNONYM synonym
 FOR [ owner_name.]{table_name | view_name | synonym } ;
```

In the DELETE FROM example, you must specify one of *table_name* or *view_name*:

SYNTAX

```
DELETE FROM [ owner_name.]{table_name | view_name }
  [ WHERE search_condition ] ;
```

- Ellipses (. . .) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

In the ORDER BY example, you must include one expression *(expr)* or column position *(posn)*, and you can optionally specify the sort order as ascending (ASC) or descending (DESC). You can specify additional expressions or column positions for sorting within a sorted result set. The SQL-92 engine orders the rows on the basis of the first *expr* or *posn*. If the values are the same, the second *expr* or *posn* is used in the ordering:

SYNTAX

```
ORDER BY { expr | posn } [ ASC | DESC ]
  [ , [ { expr | posn } [ ASC | DESC ] ] ... ]
```

In the GRANT example, you must include one *user_name*, but you can optionally include more. Note that each subsequent *user_name* must be preceded by a comma:

SYNTAX

```
GRANT { RESOURCE, DBA } TO user_name [, user_name ] ... ;
```

- In many examples, the syntax is too long to place in one horizontal row. In such cases, **optional** items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

  In this example, CREATE VIEW is followed by several optional items:

  SYNTAX

```
CREATE VIEW [ owner_name.]view_name
  [ ( column_name [, column_name ] ... ) ]
  AS [ ( ] query_expression [ ) ] [ WITH CHECK OPTION ] ;
```

How to Use This Manual

This book assumes that you have successfully installed the Progress database and SQL-92 development environment as described in the *Progress Installation Notes*. This book is primarily a reference guide for SQL-92 application developers. See also the *Progress Embedded SQL-92 Guide and Reference* for information on how to write an SQL application by embedding SQL statements in the C Language.

See Chapter 2, "SQL-92 Language Elements", Chapter 3, "SQL-92 Statements", and Chapter 4, "SQL-92 Functions" if you require complete syntax for SQL-92 language elements, statements and functions. You can apply this SQL-92 syntax to the Progress SQL Explorer tool, the Java ProgressTest, embedded SQL-92 applications or any SQL Client tool that generates SQL-92 compatible syntax. See Appendix E, "Compliance with Industry Standards," for a summary of Progress compliance for supported scalar functions and statements.

See Chapter 5, "Java Stored Procedures and Triggers" and Chapter 6, "Java Class Reference" for reference information on Java classes and methods, and instructions on how to create Java stored procedures.

Other Useful Documentation

This section lists Progress Software Corporation documentation that you might find useful. Unless otherwise specified, these manuals support both Windows and Character platforms and are provided in electronic documentation format on CD-ROM.

Getting Started

Progress Electronic Documentation Installation and Configuration Guide (Hard copy only)

A booklet that describes how to install the Progress EDOC viewer and collection on UNIX and Windows.

Progress Installation and Configuration Guide Version 9 for UNIX

A manual that describes how to install and set up Progress Version 9.1 for the UNIX operating system.

Progress Installation and Configuration Guide Version 9 for Windows

A manual that describes how to install and set up Progress Version 9.1 for all supported Windows and Citrix MetaFrame operating systems.

Progress Version 9 Product Update Bulletin

A guide that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Progress Master Glossary for Windows and *Progress Master Glossary for Character* (EDOC only)

Platform-specific master glossaries for the Progress documentation set. These books are in electronic format only.

Progress Master Index and Glossary for Windows and *Progress Master Index and Glossary for Character* (Hard copy only)

Platform-specific master indexes and glossaries for the Progress hard-copy documentation set.

Welcome to Progress (Hard copy only)

A booklet that explains how Progress software and media are packaged. An icon-based map groups the documentation by functionality, providing an overall view of the documentation set. *Welcome to Progress* also provides descriptions of the various services Progress Software Corporation offers.

Development Tools

Progress Basic Database Tools (Character only; information for Windows is in online help)

A guide for the Progress Database Administration tools, such as the Data Dictionary.

Database

Progress Database Design Guide

A guide that uses a sample database and the Progress Data Dictionary to illustrate the fundamental principles of relational database design. Topics include relationships, normalization, indexing, and database triggers.

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures. The procedures allow you to create and maintain your Progress databases and manage their performance.

SQL-92

Progress Embedded SQL-92 Guide and Reference

A guide to Progress Embedded SQL-92 for C, including step-by-step instructions for building ESQL-92 applications and reference information about all Embedded SQL-92 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-92 and the ANSI standards upon which it is based.

Progress JDBC Driver Guide

A guide to the Java Database Connectivity (JDBC) interface and the Progress SQL-92 JDBC driver. It describes how to set up and use the driver and details the driver's support for the JDBC interface.

Progress ODBC Driver Guide

A guide to the ODBC interface and the Progress SQL-92 ODBC driver. It describes how to set up and use the driver and details the driver's support for the ODBC interface.

Reference

Pocket Progress (Hard copy only)

A reference that lets you quickly look up information about the Progress language or programming environment.

SQL-92 Reference

These are non-Progress resources available from your technical bookseller.

A Guide to the SQL Standard

Date, C.J., with Hugh Darwen. 1997. Reading, MA: Addison Wesley.

Understanding the New SQL: A Complete Guide

Melton, Jim (Digital Equipment Corporation) and Alan R. Simon. 1993. San Francisco: Morgan Kaufmann Publishers.

1

Getting Started with SQL-92

This chapter is an introduction to the Progress SQL-92 environment.

Specifically, this chapter provides the following:

- Description of SQL concepts and Progress support for the SQL-92 standard
- Overview of considerations for developing applications for international deployment
- Description of the Progress SQL Explorer interactive tool
- References to other sources of information

1.1 Introduction to SQL

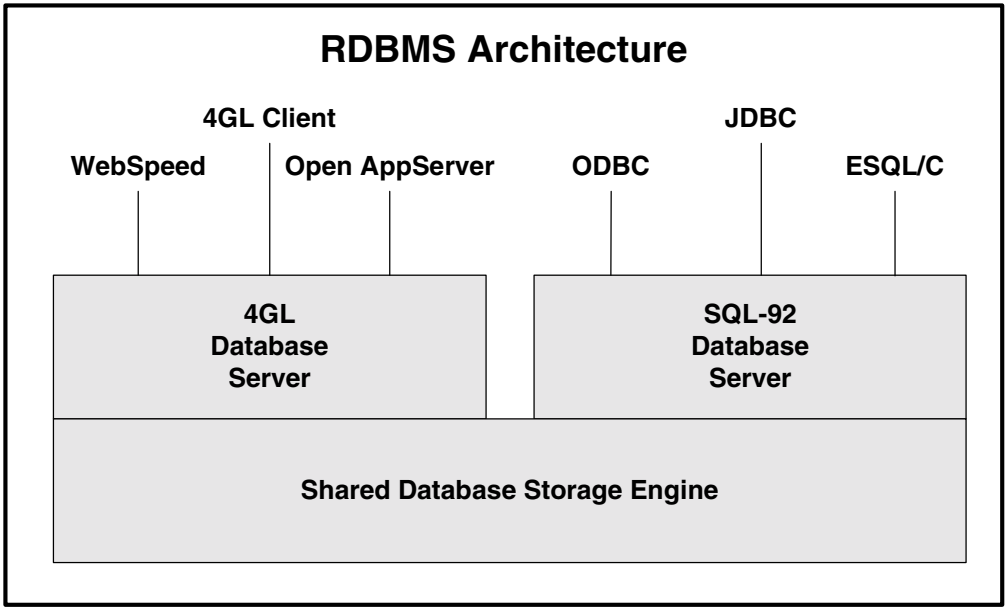
To better utilize SQL-92, it is important to understand the purpose of the language and it's capabilities.

What is SQL?

SQL is an industry-standard language specifically designed to enable people to define database structure, add and maintain data, and query data. SQL has become the standard way to represent and maintain data in a relational database.

Why use SQL?

The SQL-92 interface supports ESQL/C and open interfaces of ODBC and JDBC. Any ODBC or JDBC-complaint application will have access to the database. This provides a wide range of access to Progress data for reporting and development solutions.



SQL-92 supports a completely separate interface from the 4GL, but both interfaces use a Shared Database Storage Engine that handles low-level database functions. By sharing this Storage Engine, memory is saved and efficiency is gained regarding key database processing. This architecture also provides SQL-92 and 4GL clients concurrent access to data.

How do you use SQL?

SQL-92 must be used with other programming languages, such as COBOL, LISP, or C. Progress Software provides a precompiler for SQL statements embedded in the C Language. See the *Progress Embedded SQL-92 Guide and Reference* for a complete description of the semantics of Progress Embedded SQL-92 elements, and for information on how to use SQL in a C Language program.

The SQL-92 international standards document defines the language, but does not provide recommendations for design, or examples. See "SQL Resources," for references to comprehensive books on applying the SQL-92 standard in a development environment.

1.2 Review of Relational Database Components

A database is a collection of data and the structure of that data, called *metadata*. A relational database consists of two or more logically related data items. The following elements comprise a relational database:

Table 1–1: Components of a Relational Database

Element	Description
Table	A table is a group of related data composed of columns and rows.
Index	An index provides a means of efficiently locating a record or set of records in the database.
Row	A row is a single occurrence of data in a table.
Column	Each column characterizes one attribute of a row of data. Each column has a data type such as character, integer or decimal associated with it.

NOTE: A *View* is a subset of a database that an application can process. It may contain parts of one or more tables. Views have no independent existence. They are a way of looking at the data.

1.3 SQL Statements

The Progress SQL-92 environment uses statements to manipulate the structure of the database, maintain the data and query the database. These statements are separated into three categories.

Data Definition Language (DDL)

Maintain the structure of the data and the metadata. These statements are CREATE, ALTER, and DROP.

Data Manipulation Language (DML)

Maintain the structure of the data. These statements are INSERT, UPDATE, DELETE.

Data Control Language (DCL)

Protect the database, including the data. These statements are COMMIT, ROLLBACK, GRANT, and REVOKE.

See Chapter 3, "SQL-92 Statements," for more information on these statements.

1.3.1 SQL-92 Language Elements

The SQL-92 language has an easy to read, intuitive syntax. A language statement is a complete instruction. A statement can contain one of the following language elements found in Table 1–2.

Table 1–2: Language Elements *(1 of 2)*

Element	Description
Statement	A statement is one complete instruction to Progress.
Identifiers	An identifier is a user-supplied name for a table, view, cursor, or column.
Data types	Data types control how SQL stores information.
Expressions	An expression is a constant, column name, variable name, function or any combination of these and an operator.
Operator	An operator is an element used to manipulate, compare, or test values in an expression.
Query Expressions	Query Expressions retrieve values from tables.

Table 1–2: Language Elements *(2 of 2)*

Element	Description
Search Conditions	Search Conditions specify a condition about a given row or group of rows that is true or false.
Literals	Literals are expressions that represent a constant value.
Date-time format strings	Date-time formation strings control the output of date and time values.
Functions	A function performs a discrete task within an expression or statement, usually, returning a value.

Refer to the following chapters for more detail on this information:

- Language Elements, refer to Chapter 2, "SQL-92 Language Elements"
- Statements, refer to Chapter 3, "SQL-92 Statements"
- Functions, refer to Chapter 4, "SQL-92 Functions"

1.4 Considerations for Internationalization

The Progress SQL-92 development environment provides support for writing applications that can be deployed world-wide. Several of the functions in Chapter 2, "SQL-92 Language Elements," refer to collation and case tables. The SQL-92 server uses the `convmap` utility to manage its internal character set and related information. See the *Progress Internationalization Guide* for information on how to use these components.

When there is multi-byte character information that applies to a specific SQL-92 language element or SQL-92 statement, the considerations are noted with the element or statement.

There are some internationalization considerations that apply to all SQL-92 syntax. When the syntax requires single quotes, double quotes, parentheses, or braces, the requirement is for the single-byte ASCII encoding of these characters, and other encodings are not equivalent.

1.5 Progress SQL Explorer

SQL Explorer is a Java-based tool you can use to execute SQL-92 statements interactively.

There are two versions of SQL Explorer. A GUI version is available for the MS Windows platform only. SQL Explorer online help provides complete instructions on using the graphical SQL Explorer. A character version is available for both MS Windows and UNIX. This section describes how to accomplish the following tasks with SQL Explorer:

- Starting SQL Explorer
- Connecting to a database with SQL Explorer
- Running a single SQL-92 statement
- Running multiple SQL-92 statements
- Running SQL-92 statements from a file
- Including files containing SQL-92 statements into statement history
- Modifying SQL Explorer properties
- Running MS Windows SQL Explorer in batch mode

1.5.1 Starting SQL Explorer on Windows

To start the GUI SQL Explorer on Windows:

Choose Progress→ SQL Explorer Tool from the Windows Start menu. The SQL Explorer window appears.

The SQL Explorer window has the following features:

- A File menu containing an option for each action you can perform
- A View menu containing options for:
  - Enable Output Display
  - Toggle Table Format
  - Next Result Set
  - Previous Result Set
  - Setting Connection options, Reporting options, Logging options, and Font options

- A top pane where you enter SQL-92 statements
- Buttons that allow you to run or delete from history the statements that you have entered in the top pane, as well as buttons that allow you to scroll backward and forward through statements
- A bottom pane that displays the output or results of statements you execute
- A single-line message display area

To exit SQL Explorer choose File→ Exit.

1.5.2 Starting SQL Explorer in Character Mode

Use the following syntax to start SQL Explorer from a Windows or a UNIX command line:

SYNTAX

```
SQLEXP  [-char]
 { -S { service_name | port_number }
 [ -H host_name ] [ -db ] database_name } | { -url dbURL }

 [ -command command_string ]
 [ -infile input_file ] [ -outfile output_file ]
 [ -user username ] [ -password password ]
 [ -sqlverbose ] [ -help ]
```

`-char`

Starts the character version. The default mode for SQL Explorer is GUI on Windows. On UNIX platforms, the `-char` option is the default.

`-S` *service_name* | *port_number*

The port number or service name to use for a network connection to a database server on the same machine or on a remote host.

`-H` *host_name*

Specifies the name of the remote host on which the database server is running. The default value is “localhost.”

-db *database_name*

Specifies the physical file name of the database to which you want to connect. The -db argument is optional; you can specify the database name without the argument.

-url *dbURL*

Identifies the database to which you want to connect. The URL string has the following components:

SYNTAX

```
jdbc:jdbcprogress:T:host_name:service_name | port_number:database_name
```

jdbc:jdbcprogress:T

An identifying subprotocol string for the JDBC Driver.

host_name

The name of the system where the Progress SQL-92 JDBC data source resides.

service_name | *port_number*

Port number or service name to use for the connection.

database_name

The physical file name of the Progress database.

-command *command_string*

Specifies a SQL command to execute after startup and before displaying an interactive prompt.

-infile *input_file*

Specifies a file that contains SQL statements to be executed in batch mode after startup.

-outfile *output_file*

Specifies a file where SQL Explorer is to write SQL-92 statement output. When this option is omitted SQL-92 statement output is sent to standard output.

`-user` *username*

Specifies a *username* other than the current user login for connecting to the database.

`-password` *password*

Specifies the password for connecting to the database when password protection has been established. By default, no password is required.

`-sqlverbose`

Optionally displays statistics for the process.

`-help`

Optionally displays SQL Explorer command syntax with help text.

1.5.3 Connecting to a Database

Before you can connect SQL Explorer to a database server, the server must be configured to accept SQL connections and must be running. See the *Progress Database Administration Guide and Reference* for instructions on creating a database and starting a Progress database or database server.

Character Mode

Use the following syntax to start SQL Explorer in character mode and connect to a database:

SYNTAX

```
SQLEXP [-char] -r -url -S service_name | port_number
      -H host_name -db database_name
```

OR:

```
SQLEXP [-char] -url jdbc:jdbcprogress:T:host_name:
     service_name | port_number:database_name
```

EXAMPLE

After starting a database server for the "y2ksports" database on port 2000 of the same machine (the local host), you can start SQL Explorer in character mode and connect to the y2ksports database with the following command:

```
sqlexp -char -url jdbc:jdbcprogress:T:localhost:2000:y2ksports
```

Notice that in this example no *username* or *password* is specified. The connect request is made using the current user login as the *username*. Once you are connected, the connection URL appears in the title of the Command Prompt window and the SQL Explorer prompt is ready to accept SQL-92 statements.

Windows

Follow these steps to connect to a database from the SQL Explorer window:

1 ♦ Choose Connect from the SQL Explorer File menu. The Connect Database dialog box appears.

2 ♦ In the Connect Database dialog box, choose "basic" or "url" tab to enter the information required by the database to which you want to connect:

- To make a network connection to a database server running on the same machine leave the default host name "localhost" in the Host field, enter the service name or port number in the Service or Port field, and enter the physical filename of the database in the Database field.
- To make a network connection to a database on a remote host enter the host name in the Host field, enter the service name or port number in the Service or Port field, and enter the physical filename of the database in the Database field.

3 ♦ If password protection has been established for the database enter your User ID and Password, then click Connect. Otherwise, just click Connect.

Once you are connected the connection URL appears in the title of the SQL Explorer window.

You can change the SQL Explorer properties so the Connect Database dialog box prompts the user for the URL information. For more information, see "Modifying SQL Explorer Properties," later in this chapter.

To disconnect choose Disconnect from the File menu.

EXAMPLE

After starting a database server for the “y2ksports” database on port 2000 of the same machine (the local host), you can use SQL Explorer to connect to the y2ksports database. Use an authorized *username*, to connect to the database.

To connect to the database using the “basic tab”, enter the following information in the Connect Database dialog box, then click Connect.

- `Host: localhost`
- `Service or Port: 2000`
- `Database: y2ksports`
- `User: username`

The ’sysprogress’ *username* is the owner of the system tables, and is the initial DBA for the Progress SQL-92 sample databases. For information on authentication, see the *Progress Database Administration Guide and Reference*.

CAUTION: When a user connects to a database using the SQL Explorer, a share lock is automatically placed on the database’s schema. If multiple users connect to the same database using the SQL Explorer, and one user attempts to create a table or modify the database schema, the user will not be able to complete the modification because an exclusive lock on the schema could not be acquired.

1.5.4 Running a Single SQL-92 Statement

Once SQL Explorer is connected to a database you can execute any SQL-92 statement.

Comment Lines

SQL Explorer recognizes the pound sign (#) or double dashes (--) as comment lines in input files, as well as interactively. For a line to be treated as a comment line, one of these comment identifiers must be the first text in the line. The identifier can be in any column. SQL Explorer treats all text in the line after the comment identifier as a comment.

Character Mode

A semicolon (;) is required to indicate the end of a statement, and signal SQL Explorer to execute a single SQL-92 statement. When you execute a SQL statement the output displays to standard output. You can use the `-outfile` startup parameter to redirect the output to a file that you specify.

Use the following syntax to execute a SQL-92 statement from the SQL Explorer command prompt:

SYNTAX

```
SQLExplorer> SQL_92_statement ;
```

To query the y2ksports database for the contents of the item table with owner PUB, enter this statement at the SQL Explorer command prompt:

```
-- Selecting all rows from the item table, with owner pub

select * from pub.item;
```

Windows

When you run a single SQL-92 statement the output is displayed in table format in the bottom pane. You can use the File→ Toggle Table Format option or the **CTRL-T** accelerator key to disable the table format.

Follow these steps to execute a SQL-92 statement from the SQL Explorer window:

1 ♦ Type the SQL-92 statement in the top pane.

2 ♦ Click the Execute button. The output displays in the bottom pane.

NOTE: When executing a single statement in the Windows version the semicolon (;) is optional. Clicking the Execute button signals SQL Explorer to execute the statement.

EXAMPLE

To query the y2ksports database for all columns in the customer table with owner PUB, type the following SQL-92 statement in the top pane of the SQL Explorer window, then click Run:

```
-- Selecting all rows from the customer table with owner pub.

select * from pub.customer;
```

The output of the query appears in the bottom pane.

To execute the GRANT statement and grant DBA privileges to a specific user, type the GRANT DBA statement in the top pane, then click the Execute button.

NOTE: For SQL-92 statements that produce no output the execution status displays in the single-line message area at the bottom of the SQL Explorer window.

1.5.5 Running Multiple SQL-92 Statements

In the Windows version of SQL Explorer you can enter multiple SQL-92 statements in the top pane, then click the Execute button to execute them. With multiple statements you must end each statement with a semicolon (;). When you execute multiple statements both the statement executed and its execution results are displayed in the bottom pane.

1.5.6 Saving SQL-92 Statements to a File

In the Windows version of SQL Explorer you can save any or all of the SQL-92 statements you have entered in the top pane into a named file.

To save the current contents of the top pane to a file:

1 ♦ Choose File→ Save. The Windows Save As dialog box appears.

2 ♦ Navigate to the directory where you want to save the file, then type in the name of an `.sql` file and click OK.

1.5.7 Running SQL-92 Statements from an Input File

In both the character and Windows versions of SQL Explorer you can run SQL-92 statements from an input file. Comment lines, identified by the line beginning with the pound sign (#) or two dashes (- -) are valid in input files.

Character Mode

Executes the file in batch mode. SQL Explorer terminates after executing all the commands in the file. When you run SQL-92 statements from an input file the output displays to standard output. Use the `-outfile` startup parameter to redirect the output to a named file.

Use the following syntax to connect to a database and execute SQL-92 statements from an input file:

SYNTAX

```
SQLEXP -char -url jdbc:progress:T:host_name:
  service_name | port_number:database_name -infile input_file
```

Windows

Follow the steps to execute SQL statement from an input file:

1 ♦ Choose File→ Run File from the SQL Explorer menu. The Windows Open File dialog box appears.

2 ♦ Navigate to the directory containing the input file, select the file and click OK. The statement and its results are displayed in the bottom pane.

1.5.8 Including Files Containing SQL-92 Statements into Statement History

In the Windows version of SQL Explorer, the content of the input file is read into the statement history of the top pane. Then you can use the Prev, Next, Last, and First buttons to scroll through them. By default, the first statement displayed is the first statement loaded into history from the file. You can edit the statements, reorder them, run them or save them to another file.

To choose an input file from the SQL Explorer window:

1 ♦ Click the File button.

2 ♦ Use the Windows Open File dialog box to select the `.sql` file you want as input, then click OK. Each statement in the file is loaded into the history of the top pane.

1.5.9 Modifying SQL Explorer Properties

You can modify SQL Explorer properties in order to change how SQL Explorer behaves.

SQL Explorer property name strings do not contain blanks, and property names are case insensitive. To distinguish property names from other text, the names are presented here in mixed case. For example, the `autocommit` property is presented as `AutoCommit`. `AUTOCOMMIT`, `autocommit`, or any mixed case combination is valid.

Each property is available for both the character interface and the graphical interface, unless noted otherwise.

`AutoCommit`

Automatically commits all changes to the database when set to `true`. `False` is the default.

`ColumnWidthLimit`

Truncates the output column width when set less than the SQL column width. Default value is 25 characters. Increase the value to prevent truncating an output column that is greater than 25 characters.

`ColumnWidthMin`

Controls the minimum width of columns output to the bottom pane or standard output. Default value is 1.

`ConnectTimeout`

Allows you to specify the maximum number of seconds the SQL Explorer should wait when attempting to establish a connection, before terminating the connection request. The default value is 180 seconds.

`DisableWarnings`

Allows the SQL Explorer to send warning messages to the output screen. To enable the display of warning messages, change `DisableWarnings` from the default value of `true` to the override setting of `false`.

`Echo`

Prohibits Windows SQL Explorer from displaying output to the bottom pane when set to `false`. Default is `true`.

`FetchLimit`

Specifies the maximum number of records to be retrieved from the database when the default value of 101 is increased or decreased. To disable the `FetchLimit` completely, set the `HasFetchLimit` property to `false`.

`HasFetchLimit`

The `HasFetchLimit` property allows you to change the default value of `true` to the override setting of `false`. This property is related to the `FetchLimit` property. Setting `HasFetchLimit` to `false` allows you to disable the fetch limit completely, without requiring you to set an artificially high `FetchLimit`.

NOTE: Earlier releases of SQL Explorer enforced a fetch limit. The default for the `FetchLimit` property remains at 101 records. Setting `HasFetchLimit` to `false` allows you to disable the `FetchLimit` completely, without requiring you to set an artificially high `FetchLimit`.

`Logfile`

Allows you to change the name and/or location of the `Logfile` by setting the `Logfile` property to the full path and filename of an alternate log file. The default `Logfile`, `SQLExplorer.log` is located in the current working directory.

`Logging`

Logs all SQL statements and their output to the specified log file when set to `true`. Default value is `false`.

`Pager`

Allows you to change the default setting of `false` to the override value of `true`. When `Pager` is set to `false`, all of the output data is returned to the screen. When you choose `true`, SQL Explorer returns the number of lines specified in `PagerLimit`, which has a default value of 22 lines. You can then page up and down to review the output.

The `Pager` and `PagerLimit` properties are available in the character interface only. The graphical interface does not require these properties.

`PagerLimit`

Allows you specify how many lines SQL Explorer displays on your screen. You can set this value to the maximum available display area on your screen, or to the number of lines you choose to use for output display. A typical setting is twenty-two lines. For `PagerLimit` to take effect, you must first set the `Pager` property to `true`. If the `Pager` property is set to the default value of `false`, SQL Explorer ignores the `PagerLimit` property setting.

`ReportFormat`

Allows you to override the standard format display for output data. Setting this property to `by label` directs SQL Explorer to return data in an alternative format. A single output line in the `by label` format returns the record id, column name, and column value for one column in the result set.

`Reset` *`propertyname propertyvalue`*

Resets the property settings to their original values. Enter `@reset` to reset all property values. You can also reset one or more specified properties by supplying the property name and the new property value.

`Run` *`filename`*

Executes the specified file named. The file itself contains SQL-92 statements and SQL Explorer command statements (`@` and `!`). Different applications may have different requirements for database operations, such as transaction isolation level or automatic commit operations. You can create a custom file for each application.

`Transaction`

Sets the transaction isolation level to `COMMITTED READ`. The isolation levels specify the degree to which one transaction can modify data or database objects being used by a concurrent transaction. See the SET TRANSACTION ISOLATION LEVEL Statement and LOCK TABLE Statement in Chapter 3, "SQL-92 Statements," for complete information on isolation levels. You can set the default to any of these values:

- `UNCOMMITTED READ`
- `COMMITTED READ` (default value)
- `REPEATABLE READ`
- `SERIALIZABLE`

`useurl`

Controls the format in which the Connect dialog box prompts users for database connection information in Windows SQL Explorer.

Character and Windows

EXAMPLES

The following example displays current property settings, for both character and Windows SQL Explorer:

```
SQLExplorer> @show ;


Option values are:
     autocommit false
     columnwidthlimit 25
     columnwidthmin 1
     connecttimeout 180
     disablewarnings true
     echo true
     escapetimeout 15
     fetchlimit 101
     hasfetchlimit true
     logging false
     logfile .\SQLExplorerSession.log
     reportformat standard
     run <sqlFilename>
     transaction 3: Serializable
     useurl false;
```

This example displays the available options for setting properties, for both character and Windows SQL Explorer:

```
SQLExplorer> @help ;


Available options include:
     @help
     @autocommit {true|false}
     @columnwidthlimit <numericValue>
     @columnwidthmin <numericValue>
     @connecttimeout <numericValue>
     @disablewarnings {true|false}
     @echo {true|false}
     @escapetimeout <numericValue>
     @fetchlimit <numericValue>
     @hasfetchlimit {true|false}
     @logging {true|false}
     @logfile "<logfilename>"
     @reportformat { standard | by label }
     @reset < property_name property_value>
     @run <file_name>
     @show options
     @transaction { 0 | 1 | 2 | 3 }
          - or -
     @transaction { Uncommitted Read| Committed Read
                  | Repeatable Read | Serializable }
     @useurl {true|false};
```

This is syntax for changing the value for a specific property, both in character and Windows SQL Explorer:

SYNTAX

```
SQLExplorer> @property_name value ;
```

Windows Only

Follow these steps to change property settings for Windows SQL Explorer only:

1 ♦ Choose File→ Options. The Properties Editor dialog box appears.

2 ♦ Change any properties you want, then click OK. Any changes you make are applied immediately.

1.5.10 Running Windows SQL Explorer in Batch Mode

To run Windows SQL Explorer in batch mode set the logging property to TRUE and the echo property to FALSE.

Follow these steps to enable Windows SQL Explorer to operate in batch mode:

1 ♦ Choose File→ Log. The Log Session Options dialog box appears.

2 ♦ Type the full path and filename of the file where the session is to be logged or click Browse to open the Windows Save As dialog box and select a file.

3 ♦ Once the filename is present, click OK. SQL Explorer will now log all SQL-92 statements and their output to the specified file.

4 ♦ Choose File→ Options. The Properties Editor dialog box appears.

5 ♦ Change the Echo field to false, then click OK.

SQL Explorer logs all SQL statements and resulting output to the specified log file.

1.6 Typical Pattern of Use

This section contains the full text of four sample SQL-92 scripts. Table 1–3 lists the purpose of each script.

For complete information on creating and managing databases, see the *Progress Database Administration Guide and Reference*.

Table 1–3: Sample SQL-92 Scripts

Name of SQL-92 Script	Purpose of the Script
`Team_cre.sql`	Create a table in an existing database
`Team_upd.sql`	Populate the table
`Team_que.sql`	Execute queries against the table
`Team_xit.sql`	Drop the table from the database

You can submit these scripts sequentially to SQL Explorer. If you are using character mode, submit a script as an input file by using the `-infile` parameter. If you are using the graphical interface, choose File→ Run File from the SQL Explorer menu and select the script to run.

1.6.1 Create a Table

This section contains an SQL-92 script that creates the `team` table in an existing database.

Team_cre.sql

```
-------------------------------------------------------------------------
-- File Name: Team_cre.sql
--
-- Purpose:  Creates the team table in the y2ksports database, a copy of
--           the sports2000 database.
--
-- Requirements:
--           (1) Create the database by copying SPORTS2000.
--               Example: prodb y2ksports sports2000
--           (2) Start the database.
--               Example: proserve y2ksports -S 800
--           (3) Submit this script to SQL Explorer as an input file
--              (character mode) or as a run file (GUI mode).
--
-- Revision History:
--
-- Date                Author                    Change
-- -----              --------                  --------
--
-- 11/99               DB-DOC                    Created, V9.1A
-------------------------------------------------------------------------

create table team (
    FirstName    varchar(30) not null,
    LastName     varchar(50) not null,
    empnum       integer CONSTRAINT employee not null PRIMARY KEY,
    State        varchar(50) not null,
    Sport        char(20)
    ) ;
commit work ;
```

1.6.2 Populate the Table

This section contains an SQL-92 script that inserts and updates data in the `team` table.

Team_upd.sql

```
-------------------------------------------------------------------------
-- File Name: Team_upd.sql
-- Purpose:  Populates the team table in a copy of the
--           sports2000 database.
-- Requirements:
--           (1) Database must be running.
--           (2) Team table must exist -- see Team_cre.sql
--           (3) Submit this script to SQL Explorer as an input file
--               (character mode) or as a run file (GUI mode)
-- Revision History:
-- Date               Author                    Change
-- ----               ------                    -------
-- 11/99              DB-DOC                    Created
-------------------------------------------------------------------------

insert into team (EmpNum, Firstname, Lastname, State)
     select EmpNum, Firstname, Lastname, State from pub.employee
     where state = 'MA' ;
update team
     set sport = 'tennis1' where state = 'MA' ;
commit work ;

insert into team (EmpNum, Firstname, Lastname, State)
     select EmpNum, Firstname, Lastname, State from pub.employee
     where state = 'NY' ;
update team
     set sport = 'volleyball1' where state = 'NY' ;
commit work ;

insert into team (EmpNum, Firstname, Lastname, State)
     select EmpNum, Firstname, Lastname, State from pub.employee
     where state = 'NH' ;
update team
     set sport = 'tennis2' where state = 'NH' ;
commit work ;

insert into team (EmpNum, Firstname, Lastname, State)
     select EmpNum, Firstname, Lastname, State from pub.employee
     where state = 'TX' ;
update team
     set sport = 'volleyball2' where state = 'TX' ;
commit work ;
```

1.6.3 Execute Queries Against the Table

This section contains an SQL-92 script that issues queries against the `team` table.

Team_que.sql

```
-------------------------------------------------------------------------
-- File Name: Team_que.sql
--
-- Purpose:  Queries the team table in the y2ksports database, a copy of
--           the sports2000 database.
--
-- Requirements:
--           (1) Database must be running.
--           (2) Team table must exist -- see Team_cre.sql
--           (3) For rows to be returned you must first populate
--               the table. See Team_upd.sql.
--           (4) Submit this script to SQL Explorer as an input file
--               (character mode) or as a run file (GUI mode).
--
-- Revision History:
--
-- Date               Author                    Change
-- ----               ------                    ------
--
-- 11/99              DB-DOC                    Created, V9.1A
-------------------------------------------------------------------------

select FirstName, LastName, sport from team ;

select * from team order by LastName ;

select * from team order by sport ;

select * from team where sport like 'volleyball%' ;

commit work ;
```

1.6.4 Drop the Table

This section contains an SQL-92 script that drops the `team` table from the database.

Team_xit.sql

```
-- File Name: Team_xit.sql
--
-- Purpose: Drops the team table from the database.
--
-- Requirements:
--           (1) Database must be running.
--           (2) Team table must exist -- see Team_cre.sql
--           (3) Submit this script to SQL Explorer as an input file
--               (character mode) or as a run file (GUI mode)
--           (4) To re-create the table, run Team_cre.sql
--           (5) To re-populate the table, run Team_upd.sql
--
-- Revision History:
--
-- Date                Author                    Change
-- ----               -------                    -------
-- 11/99               DB-DOC                    Created
-------------------------------------------------------------------------

drop table team ;
commit work ;
```

1.7 Other Sources of Information

1.7.1 Progress Documentation

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures for SQL-92 and 4GL. The procedures allow you to create and maintain your Progress databases and manage their performance. Topics of interest to SQL-92 application developers include:

- Authentication
- Starting and stopping a database
- SQLDUMP utility for dumping application data from database tables to files
- SQLLOAD utility for loading application data from files to database tables
- SQLSCHEMA utility for selectively capturing database schema definitions to a file

Progress Embedded SQL-92 Guide and Reference

This is a guide to Progress Embedded SQL-92 for C, including step-by-step information on building embedded SQL applications and reference information on embedded SQL preprocessor statements. This guide provides examples and sample programs illustrating how to use static and dynamic SQL statements in a C Language program.

Progress Explorer On-Line Help

This tool assists you in the steps required to create and configure databases for the Progress SQL-92 environment.

Progress SQL Explorer On-Line Help

This tool provides complete instructions on using SQL Explorer in graphical mode.

1.7.2 SQL Resources

A Guide to The SQL Standard. Date, C.J., with Hugh Darwen. 1997. Reading, MA: Addison Wesley.

This book is a developer's guide to the SQL standard. Chris Date was one of the designers of the original relational model, upon which SQL is based.

Understanding the New SQL: A Complete Guide. Melton, Jim (Digital Equipment Corporation) and Alan R. Simon. 1993. San Francisco: Morgan Kaufmann Publishers.

This book is a comprehensive tour of the language, from basic theory to the most complex features. Jim Melton was the editor for the SQL-92 standard.

2

SQL-92 Language Elements

This chapter describes language elements that are common to many Progress SQL-92 statements. Syntax diagrams in other chapters often refer to these language elements without detailed explanations. The language elements are described in the following sections:

- Identifiers
- Data types
- Query expressions
- Search conditions
- Expressions
- Literals
- Date-time format strings

2.1 Overview

2.1.1 Definitions of Language Elements

- An *Identifier* is a user-supplied name for an element such as a table, view, cursor, or column. SQL statements use these names to refer to the elements.
- *Data types* control how SQL stores column values. CREATE TABLE statements specify data types for columns.
- *Query expressions* retrieve values from tables. Query expressions form the basis of other SQL statements and syntax elements.
- *Search conditions* specify a condition about a given row or group of rows that is true or false. Query expressions and UPDATE statements specify search conditions to restrict the number of rows in the result table.
- *Expressions* are symbols or strings of symbols used to represent or calculate a single value in a SQL statement. When SQL encounters an expression, it retrieves or calculates the value represented by the expression and uses that value when it executes the statement.
- *Literals* are a type of SQL expression that specify a constant value. Some SQL constructs allow literals but prohibit other forms of expressions.
- *Date-time format strings* control the output of date and time values. SQL interprets format strings and replaces them with formatted values.

2.1.2 Language Elements and Internationalization

When there is information that applies to a specific SQL-92 language element, the internationalization considerations are noted with the element or statement under the sub-heading “Internationalization.” See the “Considerations for Internationalization” section in Chapter 1, “Getting Started with SQL-92,” for information about using SQL-92 syntax in an application you are designing for world-wide deployment.

2.2 SQL-92 Identifiers

SQL syntax requires users to supply names for elements such as tables, views, cursors, and columns when they define them. SQL statements must use those names to refer to the table, view, or other element. In syntax diagrams, SQL identifiers are shown in lowercase type.

The maximum length for SQL identifiers is 32 characters.

There are two types of SQL identifiers:

- Conventional identifiers
- Delimited identifiers enclosed in double quotation marks

2.2.1 Conventional Identifiers

Conventional SQL identifiers must:

- Begin with an uppercase or lowercase letter.
- Contain only letters, digits, or the underscore character (_).
- Not be reserved words.
- Use ASCII characters only.

SQL does not distinguish between uppercase and lowercase letters in SQL identifiers. It converts all names specified as conventional identifiers to uppercase, but statements can refer to the names in mixed case.

EXAMPLE

The following commands show some of the characteristics of conventional identifiers:

```
-- Conventional Identifiers
-- Names are case-insensitive
CREATE TABLE ShOwCaSe   (CoLuMn1    character(5));
Update count = 0.

INSERT into SHOWCASE    (COLUMN1)   values('99');
Update count = 1.

Commit work;

SELECT * from SHOWCASE;

COLUMN1
----------------
99

-- Conventional Identifiers
-- Cannot use reserved words in identifiers
-- Examples: 'table' and 'column' cannot be used as identifiers
CREATE TABLE TABLE  (column1    character);

=== SQL Exception 1 ===
SQLState=00000
ErrorCode=-20003
Error in executeDirect:Syntax error (7587);Rolledback Xn

CREATE TABLE X      (COLUMN     varchar(10));

=== SQL Exception 1 ===
SQLState=00000
ErrorCode=-20003
Error in executeDirect:Syntax error (7587);Rolledback Xn
```

2.2.2 Delimited Identifiers

Delimited identifiers are strings of no more than 32 ASCII characters enclosed in double quotation marks (" "). Enclosing a name in double quotation marks preserves the case of the name and allows it to be a reserved word or to contain special characters. Special characters are any characters other than letters, digits, or the underscore character. Subsequent references to a delimited identifier must also use enclosing double quotation marks. To include a double quotation mark character in a delimited identifier, precede it with another double quotation mark.

NOTE: Although delimited identifiers preserve case for display purposes, case is still ignored for name matching. All of the following would refer to the same table: “Order”, “order”, Order, order. However, if the table were created using “Order”, it would display as written. With conventional identifiers, a table created as Order would display as ORDER.

NOTE: This behavior differs from the SQL-92 standard, which requires delimited identifiers to be case-sensitive.

The following SQL example shows some ways to create and refer to delimited identifiers:

```
-- Delimited Identifiers, in double quotes
-- 1st column name is: "
-- 2nd column name is: _under
-- 3rd column name is: "quotes
-- 4th column name is the word space preceded by a space
CREATE TABLE "delimited ids"
          (
          """"       char(10),
          "_under"   char(10),
          """quotes" char(10),
          " space"   char(10)
          );
Update count = 0.

INSERT INTO "delimited ids" values('string1', 'string2', 'string3',
'string4');
Update count = 1.

INSERT INTO "delimited ids" values('string5', 'string6', 'string7',
'string8');
Update count = 1.

COMMIT WORK;

SELECT * from "delimited ids";

"                _under           "quotes           space
---------------- ---------------- ---------------- ----------------
string1          string2           string3         string4
string5            string6             string7          string8
```

2.3 Data Types

SQL-92 CREATE TABLE statements specify data types for each column in the tables they define. This section describes the data types SQL supports for table columns.

There are several **categories** of SQL data types:

- Character
- Exact numeric
- Approximate numeric
- Date-time
- Bit String

NOTE: All the data types can store null values. A null value indicates that the value is not known and is distinct from all non-null values.

This is the syntax for a *data type*:

SYNTAX

```
char_data_type | exact_numeric_data_type | approx_numeric_data_type
  | date_time_data_type | bit_string_data_type
```

2.3.1 Character Data Types

This is the syntax for a *char_data_type*:

SYNTAX

```
{ CHARACTER | CHAR } [ ( length ) ]
  | { CHARACTER VARYING | CHAR VARYING | VARCHAR } [ ( length ) ]
```

```
CHARACTER [ ( length ) ]
```

CHARACTER (alias CHAR) corresponds to a null-terminated character string with the length specified. Values are padded with blanks to the specified length. The default length is 1. The maximum length is 2000 characters.

The Progress SQL-92 representation is a variable-length string. The host language representation is equivalent to a C Language character string.

```
{ CHARACTER VARYING | CHAR VARYING | VARCHAR } [ ( length ) ]
```

CHARACTER VARYING, CHAR VARYING, and VARCHAR corresponds to variable-length character string with the maximum length specified. The default length is 1 character. The maximum length is 31995 characters.

NOTES

- For data types CHARACTER(*length*) and VARCHAR(*length*) the value of *length* specifies the number of characters.
- The maximum length can be as large as 31995. The sum of all the column lengths of a table row must not exceed 31960.
- Due to index size limitations, only the narrower VARCHAR columns can be indexed.

Maximum length for VARCHAR

Specifically, the maximum length or maximum number of character for the VARCHAR data type is:

- 31995 characters for single-column tables converted from Version 8 to Version 9 and not updated in Version 9.
- 31982 characters for single-column tables originally created in Version 9 or updated in Version 9.

The maximum length of the VARCHAR data type depends on:

- The number of columns in a table — more columns in a table further limits the length of VARCHAR data
- When a table was created — tables created earlier can support longer VARCHAR data than tables created later

See the "Character-String Literals" section for details on specifying values to be stored in CHARACTER columns.

National Language Support (NLS)

The VARCHAR data type has NLS. The choice of character set affects the available character count or maximum length of the data column. The limits established above assume a single byte character set. Using a multiple byte character set lowers the maximum character count proportionally. For example, if all the characters in a character set take 3 bytes per character, in Version 9 the practical maximum is 10660 (31982 divided by 3). If, however, you are using

a variable width character set, you will be able to hold between 10660 and 31982 characters depending on the actual mix of characters you use.

2.3.2 Exact Numeric Data Types

See the "Numeric Literals" section for information on specifying values to be stored in numeric columns.

This is the syntax for an *exact_numeric_data_type:*

SYNTAX

```
TINYINT | SMALLINT | INTEGER
  | NUMERIC | NUMBER [ ( precision [ , scale ] ) ]
  | DECIMAL [ ( precision , scale ) ]
```

TINYINT

Corresponds to an integer value in the range -128 to +127 inclusive.

SMALLINT

Corresponds to an integer value in the range of -32768 to +32767 inclusive.

INTEGER

Corresponds to an integer value in the range of -2 ** 31 to 2 ** 31-1 inclusive.

NUMERIC | NUMBER [(*precision* [, *scale*])]

Corresponds to a number with the given precision (maximum number of digits) and scale (the number of digits to the right of the decimal point). By default, NUMERIC columns have a precision of 32 and a scale of 0. If NUMERIC columns omit the scale, the default scale is 0.

The range of values for a NUMERIC type column is *-n* to *+n* where *n* is the largest number that can be represented with the specified precision and scale. If a value exceeds the precision of a NUMERIC column, SQL generates an overflow error. If a value exceeds the scale of a NUMERIC column, SQL rounds the value.

NUMERIC type columns cannot specify a negative scale or specify a scale larger than the precision.

EXAMPLE

The following example shows what values can be inserted into a column created with a precision of 3 and scale of 2:

```
-- Illustrate bounds for precision 3 and scale 2
create table bounds (col1   numeric(3,2));
Update count = 0.

insert into bounds values(33.33);

ErrorCode=-20052
Error in executeDirect:Overflow error;Rolledback Xn


insert into bounds values(33.9);

ErrorCode=-20052
Error in executeDirect:Overflow error;Rolledback Xn


insert into bounds values(3.3);
Update count = 1.

insert into bounds values(33);

ErrorCode=-20052
Error in executeDirect:Overflow error;Rolledback Xn


insert into bounds values(3.33);
Update count = 1.

insert into bounds values(3.3333);
Update count = 1.

insert into bounds values(3.3555);
Update count = 1.

select * from bounds;

            COL1
----------------
            3.33
            3.36
             3.3
            3.33
```

DECIMAL **[** (*precision* , *scale*) **]**

Equivalent to type NUMERIC.

2.3.3 Approximate Numeric Data Types

See the "Numeric Literals" section for details on specifying values to be stored in numeric columns.

This is the syntax for an *approximate_numeric_data_type*:

SYNTAX

```
{ REAL | DOUBLE PRECISION | FLOAT [ ( precision ) ] }
```

`REAL`

Corresponds to a single precision floating-point number equivalent to the C Language float type.

`DOUBLE PRECISION`

Corresponds to a double precision floating-point number equivalent to the C Language double type.

`FLOAT [ ( precision ) ]`

Corresponds to a double precision floating-point number of the given precision, in bytes. By default, `FLOAT` columns have a precision of 8. The `REAL` data type is same as a `FLOAT(4)` and double-precision is the same as a `FLOAT(8)`.

2.3.4 Date-time Data Types

See the "Date-time Literals" section for details on specifying values to be stored in date-time columns. See "Date-format Strings" section for details on using format strings to specify the output format of date-time columns.

This is the syntax for a *date_time_data_type:*

SYNTAX

```
DATE | TIME | TIMESTAMP
```

`DATE`

Stores a date value as three parts: year, month, and day. The ranges for the parts are:

- Year: 1 to 9999

- Month: 1 to 12
- Day: Lower limit is 1; the upper limit depends on the month and the year.

`TIME`

Stores a time value as four parts: hours, minutes, seconds, and milliseconds. The range for the parts is:

- Hours: 0 to 23
- Minutes: 0 to 59
- Seconds: 0 to 59
- Milliseconds: 0 to 999

`TIMESTAMP`

Combines the parts of `DATE` and `TIME`.

2.3.5 Bit String Data Types

This is the syntax for a *bit_string_data_type*:

SYNTAX

```
BIT | BINARY | VARBINARY | LVARBINARY [ ( length ) ]
```

`BIT`

Corresponds to a single bit value of 0 or 1.

SQL statements can assign and compare values in `BIT` columns to and from columns of types `CHAR`, `VARCHAR`, `BINARY`, `VARBINARY`, `TINYINT`, `SMALLINT`, and `INTEGER`. However, in assignments from `BINARY` and `VARBINARY`, the value of the first four bits must be 0001 or 0000.

No arithmetic operations are allowed on `BIT` columns.

`BINARY [ ( length ) ]`

Corresponds to a bit field of the specified length of bytes. The default length is 1 byte. The maximum length is 2000 bytes.

In interactive SQL, INSERT statements must use a special format to store values in BINARY columns. They can specify the binary values as a bit string, hexadecimal string, or character string. INSERT statements must enclose binary values in single-quote marks, preceded by *b* for a bit string and *x* for a hexadecimal string.

Table 2–1: Specification Formats for Binary Values

Specification	Format	Example
bit string	b’ ’	b’1010110100010000’
hexadecimal string	x’ ’	x’ad10’
character string	’ ’	’ad10’

SQL interprets a character string as the character representation of a hexadecimal string.

If the data inserted into a BINARY column is less than the length specified, SQL pads it with zeros.

BINARY data can be assigned and compared to and from columns of type BIT, CHAR, and VARBINARY. Arithmetic operations are not allowed.

VARBINARY (*length*)

Corresponds to a variable-length bit field of the specified length in bytes. The default length is 1 byte. The maximum length is 31995 byte. The default length is 1.

NOTE: Due to index limitations, only the narrower VARBINARY columns can be indexed.

LVARBINARY (*length*)

Corresponds to an arbitrarily long byte array with the maximum length defined by the amount of available disk storage up to 2,000,000,000. A BLOB is an object of data type LVARBINARY.

Maximum length for VARBINARY

Specifically, the maximum length of the VARBINARY data type is:

- 31995 bytes for single-column tables converted from Version 8 to Version 9 and not updated in Version 9.
- 31982 bytes for single-column tables originally created in Version 9 or updated in Version 9.

The maximum length of the VARBINARY data type depends on:

- The number of columns in a table — more columns in a table further limits the length of VARBINARY data
- When a table was created — tables created earlier can support longer VARBINARY data than tables created later

LVARBINARY Limitations

Current limitations for LVARBINARY support are listed below:

- LVARBINARY data type will only be accessible from the SQL Engine. LVARBINARY data columns added to tables created by the 4GL are not visible to the 4GL. ESQLC does not support this data type.
- LVARBINARY data columns cannot be part of an index.
- LVARBINARY data columns cannot be used for variables or as parameters in stored procedures.
- Comparison operations are not supported on LVARBINARY columns. Comparison operations between LVARBINARY columns are not supported. Comparison operations between LVARBINARY columns and columns of other data types are not supported.
- Conversion, aggregate, and scalar functions are disallowed on this data type.
- LVARBINARY does not have National Language Support (NLS).

Language Support for LVARBINARY

This data type has normal column functionality except for the following exceptions:

- A column of data type LVARBINARY is not a valid column name in a CREATE INDEX statement.

- When issuing a CREATE TABLE statement, a valid data type for the column definitions is LVARBINARY. However, LVARBINARY do not allow the column constraints of PRIMARY KEY, FOREIGN KEY, UNIQUE, REFERENCES, and CHECK.

NOTE: When creating a table with a column of data type LVARBINARY, place the table in a new AREA.

- The VALUES option on the INSERT statement is not valid for the LVARBINARY data type.
- In a SELECT statement, a WHERE, GROUP BY, HAVING, or ORDER BY clause cannot use a column of data type LVARBINARY.
- There is no support for an UPDATE of a LVARBINARY column on a table which contains a column of data type LVARBINARY. Obtain the functionality of an UPDATE on an LVARBINARY column by using the DELETE and INSERT statements for the record.

Utility Support for LVARBINARY

Database utility support for tables including the LVARBINARY data type is listed below:

- Use BINARY DUMP/LOAD to dump and load data that contains the LVARBINARY data type.

NOTE: SQLDUMP and SQLLOAD do not support tables with LVARBINARY column data.

2.4 Query Expressions

A query expression selects the specified column values from one or more rows contained in one or more tables specified in the FROM clause. The selection of rows is restricted by a search condition in the WHERE clause. The temporary table derived through the clauses of a select statement is called a *result table*.

Query expressions form the basis of other SQL statements and syntax elements:

- SELECT statements are query expressions with optional ORDER BY and FOR UPDATE clauses.
- INSERT statements can specify a query expression to add the rows of the result table to a table.

- UPDATE statements can specify a query expression that returns a single row to modify columns of a row.
- Some search conditions can specify query expressions. In basic predicates, the result table can contain only a single value. In a *quantified_predicate* and an *in_predicate*, the result table can contain only a single column.
- The FROM clause of a query expression can specify another query expression, called a *derived table*.

This is the syntax for a *query_expression*:

SYNTAX

```
query_specification
  | query_expression set_operator query_expression
  | ( query_expression )
```

2.4.1 Query Specification

This is the syntax for a *query_specification*:

SYNTAX

```
SELECT [ ALL | DISTINCT ]
  { *
    | { table_name | alias.} * [ , { table_name. | alias.} * ] ...
    | expr [ [ AS ] [ ' ] column_title [ ' ] ]
        [, expr [ [ AS ] [ ' ] column_title [ ' ] ] ] ...
  }
FROM table_ref [, table_ref ] ... [ { NO REORDER } ]
  [ WHERE search_condition ]
  [ GROUP BY [ table.]column_name [, [ table.]column_name] ...
  [ HAVING search_condition ] ;
```

`SELECT [ ALL | DISTINCT ]`

`DISTINCT` specifies that the result table omits duplicate rows. The default, `ALL`, specifies that the result table includes all rows.

`SELECT * | { table_name. | alias. } *`

Specifies that the result table includes all columns from all tables named in the FROM clause.

`SELECT expr [ [ AS ] [ ' ] column_title [ ' ] ]`

Specifies a list of expressions, called a *select list*, whose results will form columns of the result table. Typically, the expression is a column name from a table named in the FROM clause. The expression can also be any supported mathematical expression, scalar function, or aggregate function that returns a value.

The optional '*column_title* argument specifies a new heading for the associated column in the result table. You can also use the *column_title* in an ORDER BY clause.

`FROM table_ref ...`

Specifies one or more table references. Each table reference resolves to one table (either a table stored in the database or a virtual table resulting from processing the table reference) whose rows the query expression uses to create the result table. There are three forms of table references:

- A direct reference to a table, view, or synonym.
- A derived table specified by a query expression in the FROM clause.
- A joined table that combines rows and columns from multiple tables.

The usage notes specific to each form of table reference are below.

If there are multiple table references, SQL joins the tables to form an intermediate result table that is used as the basis for evaluating all other clauses in the query expression. That intermediate result table is the *Cartesian product* of rows in the tables in the FROM clause, formed by concatenating every row of every table with all other rows in all tables.

`table_ref`

SYNTAX

```
table_name [ AS ] [ alias [ ( column_alias [ ... ] ) ] ]
  | ( query_expression ) [ AS ] alias [ (column_alias [ ... ] ) ]
  | [ ( ] joined_table [ ) ]
```

```
FROM table_name [ AS ] [ alias  [ ( column_alias  [ ... ] ) ] ]
```

Explicitly names a table. The name can be a table name, a view name, or a synonym.

An *alias* is a name you use to qualify column names in other parts of the query expression. Aliases are also called *correlation names*.

If you specify an alias, you must use it, and not the table name, to qualify column names that refer to the table. Query expressions that join a table with itself must use aliases to distinguish between references to column names.

For example, the following query expression joins the table customer with itself. It uses the aliases x and y and returns information on customers in the same city as customer SMITH:

Similar to table aliases, the *column_alias* provides an alternative name to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in *table_name*. Also, if you specify column aliases in the FROM clause, you must use **them**, and not the column names, in references to the columns.

```
FROM ( query_expression ) [ AS ] [ alias [ ( column_alias [ ... ] ) ] ]
```

Specifies a derived table through a query expression. With derived tables, you **must** specify an alias to identify the derived table.

Derived tables can also specify column aliases. Column aliases provide alternative names to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in the result table of the query expression. Also, if you specify column aliases in the FROM clause, you must use them, and not the column names, in references to the columns.

```
FROM [ ( ] joined_table [ ) ]
```

Combines data from two table references by specifying a join condition. The syntax currently allowed in the FROM clause supports only a CROSS JOIN, INNER JOIN, or LEFT OUTER JOIN.

CROSS JOIN specifies a Cartesian product of rows in the two tables. Every row in one table is joined to every row in the other table.

INNER JOIN specifies an inner join using the supplied search condition.

LEFT OUTER JOIN specifies a left outer join using the supplied search condition.

You can also specify these and other join conditions in the WHERE clause of a query expression. See the "Outer Joins" section for more information on both ways of specifying outer joins.

joined_table

SYNTAX

```
{ table_ref CROSS JOIN table_ref
  | table_ref [ INNER | LEFT [ OUTER ] ] JOIN
    table_ref ON search_condition
}
```

See the section on "Inner Joins", Cross Joins, and "Outer Joins" for more details.

WHERE *search_condition*

Specifies a *search_condition* that applies conditions to restrict the number of rows in the result table. If the query expression does not specify a WHERE clause, the result table includes all the rows of the specified table reference in the FROM clause.

The *search_condition* is applied to each row of the result table set of the FROM clause. Only rows that satisfy the conditions become part of the result table. If the result of the *search_condition* is NULL for a row, the row is not selected. Search conditions can specify different conditions for joining two or more tables. See the "Outer Joins" section for more information.

The *search_condition* is applied to each row of the result table set of the FROM clause. Only rows that satisfy the conditions become part of the result table. If the result of the *search_condition* is NULL for a row, the row is not selected.

Search conditions can specify different conditions for joining two or more tables. See the "Outer Joins" and "Search Conditions" sections for information on the different kinds of search conditions.

GROUP BY [*table_name.*]*column_name* ...

Specifies grouping of rows in the result table.

For the first column specified in the GROUP BY clause, SQL arranges rows of the result table into groups whose rows all have the same values for the specified column.

If you specify a second GROUP BY column, SQL groups rows in each main group by values of the second column.

SQL groups rows for values in additional GROUP BY columns in a similar fashion.

All columns named in the GROUP BY clause must also be in the select list of the query expression. Conversely, columns in the select list must also be in the GROUP BY clause or be part of an aggregate function.

HAVING *search_condition*

Allows you to set conditions on the groups returned by the GROUP BY clause. If the HAVING clause is used without the GROUP BY clause, the implicit group against which the search condition is evaluated is all the rows returned by the WHERE clause.

A condition of the HAVING clause can compare one aggregate function value with another aggregate function value or a constant.

2.4.2 Set Operator

This is the syntax for the *set_operator*:

SYNTAX

```
{ UNION [ ALL ] | INTERSECT | MINUS }
```

UNION [ALL]

Appends the result table from one query expression to the result table from another.

The two query expressions must have the same number of columns in their result tables, and those columns must have the same or compatible data types.

The final result table contains the rows from the second query expression appended to the rows from the first. By default, the result table does not contain any duplicate rows from the second query expression. Specify UNION ALL to include duplicate rows in the result table.

The two query expressions must have the same number of columns in their result tables, and those columns must have the same or compatible data types.

INTERSECT

Limits rows in the final result table to those that exist in the result tables from both query expressions.

The two query expressions must have the same number of columns in their result tables, and those columns must have the same or compatible data types:

MINUS

Limits rows in the final result table to those that exist in the result table from the first query expression minus those that exist in the second. In other words, the MINUS operator returns rows that exist in the result table from the first query expression but that do not exist in the second.

The two query expressions must have the same number of columns in their result tables, and those columns must have the same or compatible data types.

EXAMPLES

The following example specifies all columns in the customer table:

```
SELECT * FROM customer;
```

The *table_name.** syntax is useful when the select list refers to columns in multiple tables and you want to specify all the columns in one of those tables:

```
SELECT customer.* FROM customer;
```

```
SELECT Customer.CustNum, Customer.Name, Invoice.*
     FROM Customer, Invoice ;
```

You can qualify a column name with the name of the table it belongs to. The following examples illustrate column names qualified by table name:

```
select customer.customer_id from customer ;
```

The following example illustrates specifying a new column heading for an associated column. Enclose the new title in single or double quotation marks if it contains a space or other special characters:

```
-- Illustrate optional 'column_title' syntax
select
        FirstName as 'First Name',
        LastName as 'Last Name',
        state as 'New England State'
from employee
        where state = 'NH' or state = 'ME' or state = 'MA'
        or state = 'VT' or state = 'CT' or state = 'RI';

First Name       Last Name        New England State
---------------- ---------------- ------------------
Justine          Smith            MA
Andy             Davis            MA
Marcy            Adams            MA
Larry            Dawsen           MA
John             Burton           NH
Mark             Hall             MA
Stacey           Smith            MA
Scott            Abbott           MA
Meredith         White            NH
Heather            White             NH
```

You **must** qualify a column name if it occurs in more than one table specified in the FROM clause:

```
-- Table name qualifier required
-- Customer table has city and state columns
-- Billto table has city and state columns

select
        Customer.custnum,
        Customer.city as "Customer City",
        Customer.State as 'Customer State',
        Billto.City as "Bill City",
        Billto.State as 'Bill State'
from Customer, Billto
        where Customer.City = 'Clinton';

 CustNum Customer City    Customer State   Bill City        Bill State
-------- ---------------- ---------------- --------------------------------
1272     Clinton          MS               Montgomery       AL
1272     Clinton          MS               Atlanta          GA
1421     Clinton          SC               Montgomery       AL
1421     Clinton          SC               Atlanta          GA
1489     Clinton          OK               Montgomery       AL
1489      Clinton           OK                Atlanta            GA
```

The following example selects only the customers whose name is 'SMITH':

```
SELECT *
     FROM customer
     WHERE name = 'SMITH' ;
```

The following example finds all customers whose city is the same as the customer 'SMITH'.

```
SELECT *
     FROM customer
     WHERE city IN (
          SELECT city
          FROM customer
          WHERE name = 'SMITH') ;
```

The following example retrieves the customer and order information for customers with orders:

```
SELECT Customer.CustNum Customer.Name Orders.OrderNum Orders.OrderDate
     FROM Customer, Orders
     WHERE Customer.CustNum = Orders.CustNum ;
```

The HAVING clause in the following example compares the value of an aggregate function (COUNT (*)) to a constant (10). The query returns the customer number and number of orders for all customers who had more than 10 orders before March 31st, 1999.

```
SELECT cust_no, count(*)
     FROM orders
     WHERE order_date < TO_DATE ('3/31/1999')
     GROUP BY cust_no
     HAVING COUNT (*) > 10 ;
```

The following examples show merging a list of customers and suppliers without and with duplicate entries:

```
-- Produce a merged list of customers and suppliers
SELECT name, street, state, zip
     FROM customer
     UNION
     SELECT name, street, state, zip
     FROM supplier ;

-- Produce a list of customers and suppliers
-- with duplicate entries for those customers who are also suppliers
SELECT name, street, state, zip
     FROM customer
     UNION ALL
     SELECT name, street, state, zip
     FROM supplier ;
```

The following example illustrates the `INTERSECT` option:

```
-- Produce a list of customers who are also suppliers

SELECT name, street, state, zip FROM customer
     INTERSECT
SELECT name, street, state, zip FROM supplier ;
```

The following example illustrates the `MINUS` option:

```
-- Produce a list of suppliers who are not customers

SELECT name, street, state, zip FROM supplier ;
     MINUS
SELECT name, street, state, zip FROM customer;
```

AUTHORIZATION

- DBA privilege OR
- SELECT permission on all the tables or views referred to in the *query_expression*

SQL COMPLIANCE

SQL-92; the MINUS operator is equivalent to the SQL-92 EXCEPT operator

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications

RELATED STATEMENTS

CREATE TABLE Statement, CREATE VIEW Statement, INSERT Statement, "Search Conditions," SELECT Statement, UPDATE Statement

2.5 Joins

The following sections describe the supported joins.

2.5.1 Inner Joins

If an inner join does not specify a search condition, the result table from the join operation is the *Cartesian product* of rows in the tables, formed by concatenating every row of one table with every row of the other table. Cartesian products (also called cross products or cross joins) are not practically useful, but SQL logically processes all join operations by first forming the Cartesian products of rows from tables participating in the join.

If specified, the search condition is applied to the Cartesian product of rows from the two tables. Only rows that satisfy the search condition become part of the result table generated by the join.

A query expression can specify inner joins in either its FROM clause or its WHERE clause. For each formulation in the FROM clause, there is an equivalent syntax formulation in the WHERE clause. Currently, not all syntax specified by the SQL-92 standard is allowed in the FROM clause.

This is the syntax for a *from_clause_inner_join*:

SYNTAX

```
{ FROM table_ref CROSS JOIN table_ref
  | FROM table_ref [ INNER ] JOIN table_ref ON search_condition
}
```

`FROM` *table_ref* `CROSS JOIN` *table_ref*

Explicitly specifies that the join generates the Cartesian product of rows in the two table references. This syntax is equivalent to omitting the `WHERE` clause and a search condition.

```
FROM table_ref [ INNER ] JOIN table_ref ON search_condition
FROM table_ref, table_ref WHERE search_condition
```

Specify *search_condition* for restricting rows that will be in the result table generated by the join. In the first format, INNER is optional and has no effect. There is no difference between the WHERE form of inner joins and the JOIN ON form.

This is the syntax for a *where_clause_inner_join*:

SYNTAX

```
FROM table_ref, table_ref WHERE search_condition
```

Equi-joins

Specifies that values in one table equal the corresponding column values in the other.

Self joins

Joins a table with itself. If a WHERE clause specifies a self join, the FROM clause must use aliases to allow two different references to the same table. Also called an *auto join*.

EXAMPLE

The following queries illustrate the results of a simple CROSS JOIN operation and an equivalent formulation that does not use the CROSS JOIN syntax:

```
SELECT * FROM T1; -- Contents of T1
     C1      C2
     --      --
     10      15
     20      25
2 records selected
SELECT * FROM T2; -- Contents of T2
     C3 C4
     -- --
     10 BB
     15 DD
2 records selected
SELECT * FROM T1 CROSS JOIN T2; -- Cartesian product
     C1      C2      C3 C4
     --      --      -- --
     10      15      10 BB
     10      15      15 DD
     20      25      10 BB
     20      25      15 DD
4 records selected
SELECT * FROM T1, T2; -- Different formulation, same results
     C1      C2      C3 C4
     --      --      -- --
     10      15      10 BB
     10      15      15 DD
     20      25      10 BB
     20      25      15 DD
4 records selected
```

For customers with orders, retrieve their names and order info:

```
SELECT customer.cust_no, customer.name, orders.order_no, orders.order_date
     FROM customers, orders
     WHERE customer.cust_no = orders.cust_no ;
```

This example illustrates two ways to formulate a self-join query:

```
-- Retrieve all the customers from the same city as customer SMITH:
SELECT y.cust_no, y.name
     FROM customer AS x INNER JOIN customer AS y
     ON x.name = 'SMITH' AND y.city = x.city ;

-- Different formulation, same results:
SELECT y.cust_no, y.name
     FROM customer x, customer y
     WHERE x.name = 'SMITH' AND y.city = x.city ;
```

2.5.2 Outer Joins

An *outer join* between two tables returns more information than a corresponding inner join. An outer join returns a result table that contains all the rows from one of the tables even if there is no row in the other table that satisfies the join condition.

In a *left outer join*, the information from the table on the left is preserved: the result table contains all rows from the left table even if some rows do not have matching rows in the right table. Where there are no matching rows in the right table, SQL generates null values.

In a *right outer join*, the information from the table on the right is preserved. The result table contains all rows from the right table even if some rows do not have matching rows in the left table. Where there are no matching rows in the left table, SQL generates null values.

SQL uses two forms of syntax to support outer joins:

- In the WHERE clause of a query expression, specify the outer join operator (+) after the column name of the table for which rows will not be preserved in the result table. Both sides of an outer join search condition in a WHERE clause must be simple column references. This syntax allows both left and right outer joins.

NOTE: The Progress implementation of syntax for a *where_clause_outer_join* does not comply with the ANSI SQL-92 standard. The alternate syntax will not operate properly for clients requiring SQL-92 standard syntax.

- For left outer joins only, in the FROM clause, specify the LEFT OUTER JOIN clause between two table names, followed by a search condition. The search condition can contain only the join condition between the specified tables.

- Full (two-sided) outer joins are not supported.

- Right outer joins are only supported using the outer join operator in the WHERE clause. The keywords RIGHT OUTER JOIN are not supported currently.

This is the syntax for a *from_clause_outer_join*:

SYNTAX

```
FROM table_ref LEFT OUTER JOIN table_ref
  ON search_condition
```

This is the syntax for a *where_clause_outer_join*:

SYNTAX

```
WHERE [ table_name.] column (+) = [ table_name.]column
  | WHERE [ table_name.]column = [ table_name.]column (+)
```

EXAMPLE

The following example shows a left outer join. The query requests information about all the customers and their orders. Even if there is not a corresponding row in the orders table for each row in the customer table, NULL values are displayed for the order.order_no and order.order_date columns:

```
SELECT customer.cust_no, customer.name, orders.order_no, orders.order_date
     FROM customers, orders
     WHERE customer.cust_no = orders.cust_no (+) ;
```

The following queries illustrate outer join syntax:

```
SELECT * FROM T1; -- Contents of T1
C1  C2
--  --
10  15
20  25
2 records selected


SELECT * FROM T2; -- Contents of T2
C3  C4
--  --
10  BB
15  DD
2 records selected


-- Left outer join
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1 = T2.C3;
C1  C2  C3  C4
--  --  --  --
10  15  10  BB
20  25
2 records selected


-- Left outer join: different formulation, same results
SELECT * FROM T1, T2 WHERE T1.C1 = T2.C3 (+);
C1  C2  C3  C4
--  --  --  --
10  15  10  BB
20  25
2 records selected


-- Right outer join
SELECT * FROM T1, T2 WHERE T1.C1 (+) = T2.C3;
C1  C2  C3  C4
--  --  --  --
10  15  10  BB
        15  DD
2 records selected
```

2.6 Search Conditions

A search condition specifies a condition that is true or false about a given row or group of rows. Query expressions and UPDATE statements can specify a search condition. The search condition restricts the number of rows in the result table for the query expression or UPDATE statement.

Search conditions contain one or more predicates. Predicates that can be part of a search condition are described in the following subsections.

This is the syntax for a *search_condition*:

SYNTAX

```
[ NOT ] predicate
  [ { AND | OR } { predicate | ( search_condition ) } ]
```

predicate:

SYNTAX

```
basic_predicate
   | quantified_predicate
   | between_predicate
   | null_predicate
   | like_predicate
   | exists_predicate
   | in_predicate
   | outer_join_predicate
```

2.6.1 Logical Operators: OR, AND, NOT

Logical operators combine multiple search conditions. SQL evaluates multiple search conditions in the following order:

- Search conditions enclosed in parentheses. If there are nested search conditions in parentheses, SQL evaluates the innermost search condition first.
- Search conditions preceded by NOT.
- Search conditions combined by AND.
- Search conditions combined by OR.

EXAMPLE

The following example illustrates the use of logical operators:

```
SELECT * FROM customer
     WHERE name = 'NYQUIST' OR name = 'SMITH' ;

SELECT * FROM customer
     WHERE city = 'PRINCETON' AND state = 'NJ' ;

SELECT * FROM customer
     WHERE NOT (name = 'NYQUIST' OR name = 'SMITH') ;
```

2.6.2 Relational Operators

Relational operators specify how SQL compares expressions in basic and quantified predicates.

This is the syntax for a relational operator *(relop)*:

SYNTAX

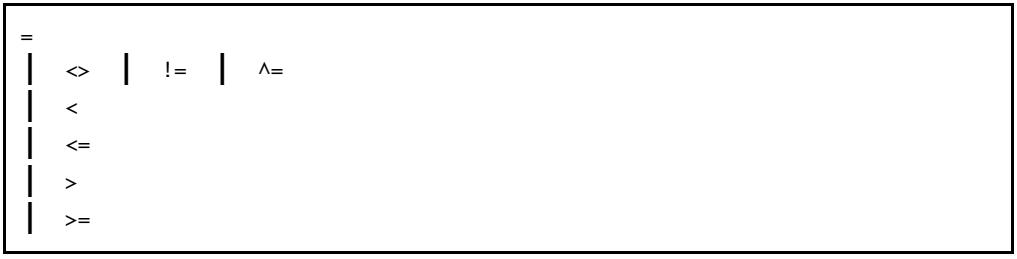


Table 2–2 lists the relational operators and the resulting predicate for each operator.

See the "Basic Predicate" section for more information.

Table 2–2: Relational Operators and Resulting Predicates *(1 of 2)*

Relational Operator	Predicate for this Relational Operator
=	True if the two expressions are equal.
<> | != | ^=	True if the two expressions are not equal. The operators != and ^= are equivalent to <>.
<	True if the first expression is less than the second expression.

Table 2–2: Relational Operators and Resulting Predicates *(2 of 2)*

Relational Operator	Predicate for this Relational Operator
<=	True if the first expression is less than or equal to the second expression.
>	True if the first expression is greater than the second expression.
>=	True if the first expression is greater than or equal to the second expression.

NOTE: For data sorting and comparison of character data, the SQL server uses a Progress collation table. The result of comparison is based on the sort weight of each character in the collation table, for these relational operators:

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

2.6.3 Basic Predicate

A basic predicate compares two values using a relational operator. See the "Relational Operators" section for information about relational operators. If a basic predicate specifies a query expression, then the query expression must return a single value. Basic predicates often specify an inner join. See the "Inner Joins" section for more information.

If the value of any expression is null or the *query_expression* does not return any value, then the result of the predicate is set to false.

This is the syntax for a *basic_predicate*:

SYNTAX

```
expression relop { expression | ( query_expression ) }
```

2.6.4 Quantified Predicate

The quantified predicate compares a value with a collection of values using a relational operator (see the "Relational Operators" section). A quantified predicate has the same form as a basic predicate with the *query_expression* being preceded by the ALL, ANY, or SOME keyword. The result table returned by *query_expression* can contain only a single column.

When you specify ALL, the predicate evaluates to true if the *query_expression* returns no values or the specified relationship is true for all the values returned.

When you specify SOME or ANY, the predicate evaluates to true if the specified relationship is true for at least one value returned by the *query_expression*. There is no difference between the SOME and ANY keywords. The predicate evaluates to false if the *query_expression* returns no values or if the specified relationship is false for all the values returned.

This is the syntax for a *quantified_predicate*:

SYNTAX

```
expression relop { ALL | ANY | SOME } ( query_expression )
```

EXAMPLE

```
10 < ANY ( SELECT COUNT(*)
         FROM order_tbl
         GROUP BY custid ;
          )
```

2.6.5 BETWEEN Predicate

The BETWEEN predicate can be used to determine if a value is within a specified value range or not. The first expression specifies the lower bound of the range and the second expression specifies the upper bound of the range.

The predicate evaluates to true if the value is greater than or equal to the lower bound of the range and less than or equal to the upper bound of the range.

This is the syntax for a *between_predicate*:

SYNTAX

```
expression [ NOT ] BETWEEN expression AND expression
```

EXAMPLE

```
salary BETWEEN 20000.00 AND 100000.00
```

2.6.6 NULL Predicate

The NULL predicate can be used for testing null values of database table columns.

This is the syntax for a *null_predicate*.

SYNTAX

```
column_name IS [ NOT ] NULL
```

EXAMPLE

```
contact_name IS NOT NULL
```

2.6.7 LIKE Predicate

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified after the LIKE keyword in a string constant. The pattern can be specified by a string in which the underscore (_) and percent sign (%) characters have special semantics.

Use the ESCAPE clause to disable the special semantics given to the characters (_) and (%). The escape character specified must precede the special characters in order to disable their special semantics.

This is the syntax for a *like_predicate*:

SYNTAX

```
column_name [ NOT ] LIKE string_constant [ ESCAPE escape_character ]
```

NOTES

- The column name specified in the LIKE predicate must refer to a character string column.
- A percent sign (%) in the pattern matches zero or more characters of the column string.
- A underscore symbol (_) in the pattern matches any single character of the column string.

- The LIKE predicate is multi-byte enabled. The *string_constant* and the *escape_character* may contain multi-byte characters, and the *escape_character* can be a multi-byte character. A percent sign (%) or an underscore (_) in the *string_constant* can represent a multi-byte character. However, the percent sign or underscore itself must be the single-byte ASCII encoding.

EXAMPLE

This example illustrates three ways to use the LIKE predicate:

```
cust_name LIKE '%Computer%'

cust_name LIKE '___'

item_name LIKE '%\_%' ESCAPE '\'
```

In the first LIKE clause, for all strings with the substring 'Computer' the predicate evaluates to true. In the second LIKE clause, for all strings which are exactly three characters long the predicate evaluates to true. In the third LIKE clause the backslash character (\ ) is specified as the escape character, which means that the special interpretation given to the underscore character (_) is disabled. The pattern evaluates to TRUE if the item_name column has embedded underscore characters.

2.6.8 EXISTS Predicate

The EXISTS predicate can be used to check for the existence of specific rows. The *query_expression* returns rows rather than values. The predicate evaluates to true if the number of rows returned by the *query_expression* is nonzero.

This is the syntax for an *exists_predicate*:

SYNTAX

```
EXISTS (query_expression)
```

EXAMPLE

In this example, the predicate evaluates to true if the specified customer has any orders:

```
EXISTS (SELECT * FROM order_tbl
     WHERE order_tbl.custid = '888' ;)
```

2.6.9 IN Predicate

The IN predicate can be used to compare a value with a set of values. If an IN predicate specifies a query expression, then the result table it returns can contain only a single column.

This is the syntax for an *in_predicate*:

SYNTAX

```
expression [ NOT ] IN
  { (query_expression) | (constant , constant [ , ... ] ) }
```

EXAMPLE

```
address.state IN ('MA', 'NH')
```

2.6.10 OUTER JOIN Predicate

An outer join predicate specifies two tables and returns a result table that contains all the rows from one of the tables, even if there is no matching row in the other table. See the "Outer Joins" section for more information.

This is the syntax for an *outer_join_predicate*:

SYNTAX

```
[ table_name.]column = [ table_name.]column (+)
 | [ table_name.]column (+) = [ table_name.]column
```

2.7 Expressions

An expression is a symbol or string of symbols used to represent or calculate a single value in a SQL statement. When you specify an expression in a statement, SQL retrieves or calculates the value represented by the expression and uses that value when it executes the statement. Expressions are also called scalar expressions or value expressions.

This is the syntax for an expression (*expression*):

SYNTAX

```
[ table_name. | alias. ] column_name
  | character_literal
  | numeric_literal
  | date_time_literal
  | aggregate_function
  | scalar_function
  | numeric_arith_expr
  | date_arith_expr
  | conditional_expr
  | (expression)
```

`[ table_name. | alias. ] column_name`

Specifies a column in a table. You can qualify a column name with the name of the table or the alias it belongs to.

`character_literal | numeric_literal | date_time_literal`

Specify constant values. See the "Literals" section for information on specifying literals.

`aggregate_function | scalar_function`

SQL functions. See Chapter 4, "SQL-92 Functions" for more information on "Aggregate Functions" and "Scalar Functions."

`numeric_arith_expr`

Computes a value from numeric values. See "Numeric Arithmetic Expressions" for more information.

`date_arith_expr`

Computes a value from date-time values. See "Date Arithmetic Expressions" for more information.

`conditional_expr`

Evaluates a search condition or expression and returns one of multiple possible results depending on that evaluation.

`(`*`expression`*`)`

SQL evaluates expressions in parentheses first.

EXAMPLES

The following example illustrates specifying a table name for the customer_id field:

```
SELECT customer.customer_id FROM customers ;
```

You **must** qualify a column name if it occurs in more than one table specified in the FROM clause:

```
SELECT customer.customer_id FROM customers, orders ;
```

NOTES

- Qualified column names are always permitted, even when they are not required.
- You can also qualify column names with an alias. An alias is also called a *correlation name*.
- The `FROM` clause of a query expression can specify an optional alias after the table name See the "Query Expressions" section for details on query expressions. If you specify an alias, you must use it, and not the table name, to qualify column names that refer to the table. Query expressions that join a table with itself must use aliases to distinguish between references to column names.

The following example shows a query expression that joins the table customer with itself. It uses the aliases *x* and *y* and returns information on customers in the same city as customer SMITH:

```
SELECT y.cust_no, y.last_name
     FROM customer x, customer y
     WHERE x.last_name = 'SMITH'
     AND y.city = x.city ;
```

2.7.1 Numeric Arithmetic Expressions

Numeric arithmetic expressions compute a value using addition, subtraction, multiplication, and division operations on numeric literals and expressions that evaluate to any numeric data type.

This is the syntax for a *numeric_arith_expr*:

SYNTAX

```
[ + | - ] { numeric_literal | numeric_expr }
  [{ + | - | * | / } numeric_arith_expr ]
```

+ | -

Unary operators.

numeric_literal

See the "Numeric Literals" section for details on specifying numeric literals.

numeric_expr

Evaluates to a numeric data type. See the "Data Types" section for information about expressions that can evaluate to numeric data types, including:

- Column name
- Subqueries that return a single value
- Aggregate functions
- CAST or CONVERT operations to numeric data types
- Other scalar functions that return a numeric data type

+ | - | * | /

Operators for addition, subtraction, multiplication, and division. SQL evaluates numeric arithmetic expressions in the following order:

- Unary plus or minus
- Expressions in parentheses
- Multiplication and division, from left to right
- Addition and subtraction, from left to right

2.7.2 Date Arithmetic Expressions

Date arithmetic expressions compute the difference between date-time expressions in terms of days or milliseconds. SQL supports these forms of date arithmetic:

- Addition and subtraction of integers to and from date-time expressions
- Subtraction of a date-time expression from another

This is the syntax for a *date_arith_expr*:

SYNTAX

```
date_time_expr { + | - } int_expr
  | date_time_expr - date_time_expr
```

date_time_expr

Returns a value of type DATE or TIME or TIMESTAMP. A single date-time expression cannot mix data types, however. All elements of the expression must be the same data type.

Date-time expressions can contain date-time literals, but they must be converted to DATE or TIME using the CAST, CONVERT, or TO_DATE functions. See the following examples: CAST Function (SQL-92 Compatible) and CONVERT Function (Progress Extension).

int_expr

Returns an integer value. SQL interprets the integer differently depending on the data type of the date-time expression:

- For DATE expressions, integers represent days
- For TIME expressions, integers represent milliseconds
- For TIMESTAMP expressions, integers represent milliseconds

EXAMPLES

The following example manipulates DATE values using date arithmetic. SQL interprets integers as days and returns date differences in units of days:

```
SELECT C1, C2, C1-C2 FROM DTEST
c1                  c2           c1-c2
---------------------------------------
1956-05-07          1952-09-29  1316



select sysdate,
     sysdate - 3 ,
     sysdate - cast ('9/29/52' as date)
from dtest;

sysdate        sysdate-3    sysdate-convert(date,9/29/52)
-----------------------------------------------------------
1995-03-24     1995-03-21     15516
```

The following example manipulates TIME values using date arithmetic. SQL interprets integers as milliseconds and returns time differences in milliseconds:

```
select systime,
     systime - 3000,
     systime - cast ('15:28:01' as time)
from dtest;

systime      systime-3000   systime-convert(time,15:28:01)
-------------------------------------------------------------
15:28:09     15:28:06       8000
```

2.7.3 Conditional Expressions

Conditional expressions are a subset of scalar functions that generate different results depending on tests of the values of their arguments. They provide some of the flexibility of traditional programming constructs to allow expressions to return varying results depending on the value of their arguments.

The following scalar functions provide support for conditional expressions:

CASE

CASE is the most general conditional expression. It specifies a series of search conditions and associated expressions. SQL returns the value specified by the first expression whose associated search condition evaluates as true. If none of the expressions evaluate as true, the CASE expression returns a null value, or the value of some other default expression if the CASE expression includes the ELSE clause.

All the other conditional expressions can also be expressed as CASE expressions.

DECODE

DECODE is not compatible with the SQL-92 standard.

NULLIF

NULLIF substitutes a null value for an expression if it is equal to a second expression.

COALESCE

COALESCE specifies a series of expressions. SQL returns the first expression whose value is not null. If all the expressions evaluate as null, COALESCE returns a null value.

IFNULL

IFNULL substitutes a specified value if an expression evaluates as null. If the expression is not null, IFNULL returns the value of the expression.

2.8 Literals

A *literal* is a type of expression that specifies a constant value. Literals are also called constants. Generally, you can specify a literal wherever SQL syntax allows an expression. Some SQL constructs allow literals but disallow other forms of expressions.

There are three types of literals:

- Numeric
- Character string
- Date-time

The following sections discuss each type of literal.

2.8.1 Numeric Literals

A *numeric literal* is a string of digits that SQL interprets as a decimal number. SQL allows the string to be in a variety of formats, including scientific notation.

This is the syntax for a *numeric_literal*:

SYNTAX

```
[ + | - ]{[ 0-9 ] [ 0-9 ]...}
  [ . [ 0-9 ] [ 0-9 ]...]
  [{ E | e }[ + | - ][ 0-9 ]{[ 0-9 ]}]
```

EXAMPLE

The following numeric strings are all valid:

```
123
123.456
-123.456
12.34E-04
```

2.8.2 Character-String Literals

A character-string literal is a string of characters enclosed in single quotation marks (' '). To include a single quotation mark in a character-string literal, precede it with an additional single quotation mark.

EXAMPLE

The INSERT statements in the following example show embedding quotation marks in character-string literals:

```
insert into quote values('unquoted literal');
insert into quote values(''single-quoted literal'');
insert into quote values('"double-quoted literal"');
insert into quote values('O''Hare');
select * from quote;

c1
--
unquoted literal
'single-quoted literal'
"double-quoted literal"
O'Hare

4 records selected
```

NOTE

- A character string literal may contain multi-byte characters in the character set used by the SQL client. Only single-byte ASCII-encoded quote marks are valid in the syntax.

2.8.3 Date-time Literals

SQL supports special formats for literals to be used in conjunction with date-time data types. Basic predicates and the VALUES clause of INSERT statements can specify date literals directly for comparison and insertion into tables. In other cases, you need to convert date literals to the appropriate date-time data type with the CAST, CONVERT, or TO_DATE scalar functions.

Enclose date-time literals in single quotation marks (' ').

NOTES

- All text (names of days, months, ordinal number endings) in all date-format literals must be in the English Language. The default date format is American. You can explicitly request another date format by using a format string.
- Time literals are in the English Language only.

Date Literals

A *date literal* specifies a day, month, and year using any of the following formats, enclosed in single quotation marks (' ').

This is the syntax for a *date_literal*:

SYNTAX

```
{ d 'yyyy-mm-dd' }
   | mm-dd-yyyy
   | mm/dd/yyyy
   | mm-dd-yy
   | mm/dd/yy
   | yyyy-mm-dd
   | yyyy/mm/dd
   | dd-mon-yyyy
   | dd/mon/yyyy
   | dd-mon-yy
   | dd/mon/yy
```

`{ d 'yyyy-mm-dd' }`

A date literal enclosed in an escape clause is compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase d. End the literal with a close brace.

NOTE: If you use the ODBC escape clause, you must specify the date using the format *yyyy-mm-dd*.

dd

Specifies the day of month as a one or two digit number in the range 01-31.

mm

Specifies the month value as a one or two digit number in the range 01-12.

mon

Specifies the first 3 characters of the name of the month in the range 'JAN' to 'DEC'.

yy

Specifies the last two digits of the year.

yyyy

Specifies the year as a four digit number.

EXAMPLES

The following example illustrates using the date literal format with an INSERT statement:

```
INSERT INTO dtest VALUES ( { d '1998-05-07' } )
```

The INSERT and SELECT statements in the following example show some of the supported formats for date literals:

```
CREATE TABLE T2 (C1 DATE, C2 TIME);
INSERT INTO T2 (C1) VALUES('5/7/56');
INSERT INTO T2 (C1) VALUES('7/MAY/1956');
INSERT INTO T2 (C1) VALUES('1956/05/07');
INSERT INTO T2 (C1) VALUES({d '1956-05-07'});
INSERT INTO T2 (C1) VALUES('29-SEP-1952');

SELECT C1 FROM T2;

--------------
c1
1956-05-07
1956-05-07
1956-05-07
1956-05-07
1952-09-29
```

Time Literals

Time literals specify an hour, minute, second, and millisecond, using the following format, enclosed in single quotation marks (' ').

This is the syntax for a *time_literal*:

SYNTAX

```
{ t 'hh:mi:ss' } | hh:mi:ss[:mls ]
```

`{ t 'hh:mi:ss' }`

A time literal enclosed in an escape clause is compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase t. End the literal with a close brace (}).

NOTE: If you use the ODBC escape clause, you must specify the time using the format *hh*:*mi*:*ss*.

hh

Specifies the hour value as a two-digit number (in the range 00 to 23).

mi

Specifies the minute value as a two-digit number (in the range 00 to 59).

ss

Specifies the seconds value as a two-digit number (in the range 00 to 59).

mls

Specifies the milliseconds value as a two-digit number (in the range 000 to 999).

EXAMPLES

The following example illustrates using the time literal format with an INSERT statement:

```
INSERT INTO ttest VALUES ( { t '23:22:12' } ) ;
```

The INSERT statements in the following example show some of the formats SQL will and will not accept for time literals:

```
INSERT INTO T2 (C2) VALUES('3');
error(-20234): Invalid time string

INSERT INTO T2 (C2) VALUES('8:30');
error(-20234): Invalid time string

INSERT INTO T2 (C2) VALUES('8:30:1');
INSERT INTO T2 (C2) VALUES('8:30:');
error(-20234): Invalid time string


INSERT INTO T2 (C2) VALUES('8:30:00');
INSERT INTO T2 (C2) VALUES('8:30:1:1');
INSERT INTO T2 (C2) VALUES({t'8:30:1:1'});
```

This SELECT statement illustrates which INSERT statements successfully inserted a row:

```
SELECT C2 FROM T2;c2

c2
--
08:30:01
08:30:00
08:30:01
08:30:01
```

Timestamp Literals

Timestamp literals specify a date and a time separated by a space, enclosed in single quotation marks (' ').

This is the syntax for a *timestamp_literal*:

SYNTAX

```
{ ts 'yyyy-mm-dd hh:mi:ss' } | 'date_literal time_literal'
```

`{ ts 'yyyy-mm-dd hh:mi:ss' }`

A timestamp literal enclosed in an escape clause is compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase ts. End the literal with a close brace (}). Note that braces are part of the syntax.

NOTE: If you use the ODBC escape clause, you **must** specify the timestamp using the format *yyyy-mm-dd hh:mi:ss*.

date_literal

A date literal.

time_literal

A time literal.

EXAMPLES

This example illustrates how to INSERT a time-stamp literal into a column:

```
INSERT INTO DTEST
VALUES ( { ts '1956-05-07 10:41:37'} ) ;
```

This example illustrates a time-stamp literal with the ODBC escape clause:

```
SELECT * FROM DTEST WHERE C1 = {ts '1985-08-10 05:41:37'} ;
```

2.9 Date-time Format Strings

The TO_CHAR scalar function supports a variety of format strings to control the output of date and time values. The format strings consist of keywords that SQL interprets and replaces with formatted values.

The format strings are case-sensitive. For instance, SQL replaces 'DAY' with all uppercase letters, but follows the case of 'Day'.

Supply the format strings, enclosed in single quotation marks, as the second argument to the TO_CHAR function. For example:

EXAMPLE

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected


SELECT TO_CHAR(C1, 'Day, Month ddth'),
     TO_CHAR(C2, 'HH12 a.m.') FROM T2;

TO_CHAR(C1,DAY, MONTH DDTH) TO_CHAR(C2,HH12 A.M.)
--------------------------- --------------------
Monday   , September 29th  02 p.m.
1 record selected
```

For information on the TO_CHAR function, see Chapter 4, "SQL-92 Functions."

2.9.1 Date-format Strings

A *date-format string* can contain any of the following format keywords along with other characters. The format keywords in the format string are replaced by corresponding values to get the result. The other characters are displayed as literals. Table 2-3 lists the date formats, and the corresponding descriptions.

Table 2–3: Date-format Strings and Descriptions *(1 of 2)*

Date-format String	Description
CC	The century as a two-digit number.
YYYY	The year as a four-digit number.
YYY	The last three digits of the year.
YY	The last two digits of the year.
Y	The last digit of the year.
Y,YYY	The year as a four-digit number with a comma after the first digit.
Q	The quarter of the year as a one-digit number (with values 1, 2, 3, or 4).
MM	The month value as a two-digit number (in the range 01-12).
MONTH	The name of the month as a string of nine characters (’JANUARY’ to ’DECEMBER ’).
MON	The first three characters of the name of the month (in the range ’JAN’ to ’DEC’).
WW	The week of the year as a two-digit number (in the range 01-53).
W	The week of the month as a one-digit number (in the range 1-5).
DDD	The day of the year as a three-digit number (in the range 001-366).
DD	The day of month as a two-digit number (in the range 01-31).
D	The day of the week as a one-digit number (in the range 1-7, 1 for Sunday and 7 for Saturday).

Table 2–3: Date-format Strings and Descriptions *(2 of 2)*

Date-format String	Description
DAY	The day of the week as a character string of nine characters (in the range 'SUNDAY' to 'SATURDAY ').
DY	The day of the week as a character string of three characters (in the range 'SUN' to 'SAT').
J	The Julian day (number of days since DEC 31, 1899) as an eight-digit number.
TH	When added to a format keyword that results in a number, this format keyword ('TH') is replaced by the string 'ST', 'ND', 'RD', or 'TH' depending on the last digit of the number.

EXAMPLE

The following example illustrates the use of the DAY, MONTH, DD, and TH format strings:

```
SELECT C1 FROM T2;

C1
--
09/29/1952

1 record selected


SELECT TO_CHAR (C1, 'Day, Month ddth'),
   TO_CHAR (C2, 'HH12 a.m.') FROM T2;

TO_CHAR (C1,DAY, MONTH DDTH) TO_CHAR (C2,HH12 A.M.)
---------------------------- ----------------------
Monday, September 29th 02 p.m.

1 record selected
```

2.9.2 Time-format Strings

A *time format string* can contain any of the following format keywords along with other characters. The format keywords in the format string are replaced by corresponding values to get the result. The other characters are displayed as literals. Table 2-4 lists the time formats, and the corresponding descriptions.

Table 2–4: Time-format Strings and Descriptions

Time-format String	Description
AM PM	The string 'AM' or 'PM' depending on whether time corresponds to morning or afternoon.
A.M. P.M.	The string 'A.M.' or 'P.M.' depending on whether time corresponds to morning or afternoon.
HH12	The hour value as a two-digit number (in the range 00 to 11).
HH HH24	The hour value as a two-digit number (in the range 00 to 23).
MI	The minute value as a two-digit number (in the range 00 to 59).
SS	The seconds value as a two-digit number (in the range 00 to 59).
SSSSS	The seconds from midnight as a five-digit number (in the range 00000 to 86399).
MLS	The milliseconds value as a three-digit number (in the range 000 to 999).

EXAMPLE

The following example illustrates the TO_CHAR function, and the Day, Month, dd, and HH12 format strings:

```
SELECT C1 FROM T2;

C1
--
09/29/1952

1 record selected


SELECT TO_CHAR (C1, 'Day, Month ddth'),
   TO_CHAR (C2, 'HH12 a.m.') FROM T2;

TO_CHAR (C1,DAY, MONTH DDTH) TO_CHAR (C2,HH12 A.M.)
---------------------------- ----------------------
Monday , September 29th 02 p.m.
1 record selected
```

3

SQL-92 Statements

This chapter describes the function and syntax for each of these SQL-92 statements as shown in Table 3–1.

SQL-92 Statements

Table 3–1: Progress SQL-92 Statements

ALTER USER Statement	BEGIN-END DECLARE SECTION	CALL Statement
CLOSE Statement	Column Constraints	COMMIT Statement
CONNECT Statement	CREATE INDEX Statement	CREATE PROCEDURE Statement
CREATE SYNONYM Statement	CREATE TABLE Statement	CREATE TRIGGER Statement
CREATE USER Statement	CREATE VIEW Statement	DECLARE CURSOR Statement
DELETE Statement	DESCRIBE Statement	DISCONNECT Statement
DROP INDEX Statement	DROP PROCEDURE Statement	DROP SYNONYM Statement
DROP TABLE Statement	DROP TRIGGER Statement	DROP USER Statement
DROP VIEW Statement	EXEC SQL Delimiter	EXECUTE Statement
EXECUTE IMMEDIATE Statement	FETCH Statement	GET DIAGNOSTICS Statement
GET DIAGNOSTICS EXCEPTION Statement	GRANT Statement	INSERT Statement
LOCK TABLE Statement	OPEN Statement	PREPARE Statement
REVOKE Statement	ROLLBACK Statement	SELECT Statement
SET CONNECTION Statement	SET SCHEMA Statement	SET TRANSACTION ISOLATION LEVEL Statement
UPDATE Statement	UPDATE STATISTICS Statement	WHENEVER Statement

ALTER USER Statement

Changes the password for the specified user.

SYNTAX

```
ALTER USER 'username', 'old_password', 'new_password' ;
```

EXAMPLE

In this example *username* 'Jasper' changes the 'Jasper' account password from 'normandy' to 'brittany':

```
ALTER USER 'Jasper', 'normandy', 'brittany' ;
```

NOTES

- Used in conjunction with CREATE USER and DROP USER, the ALTER USER statement provides a way to change a user password through SQL-92. This SQL-only solution does not require the Progress dictionary.
- The *old_password* specification must match the current password for *username*.

AUTHORIZATION

User specified in *username*

SQL COMPLIANCE

Progress Extension

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CREATE USER Statement, DROP USER Statement

BEGIN-END DECLARE SECTION

Declares variables and types used by the precompiler. Any variables you refer to in an embedded SQL-92 statement must be declared in a DECLARE SECTION. This section starts with a BEGIN DECLARE SECTION statement and ends with an END DECLARE SECTION statement. Each variable must be declared as a host language data type.

SYNTAX

```
EXEC SQL BEGIN DECLARE SECTION
host_lang_type  variable_name ;
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

host_lang_type variable_name ;

A conventional C Language variable declaration. This form of variable declaration conforms to the ANSI standard for the C Language.

SYNTAX

```
{ char | short | long | float | double }
```

EXAMPLE

This example is a code fragment from the `StatSel` function in the sample program `2StatSel.pc`. The complete source for sample program, `2StatSel.pc`, is listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
EXEC SQL BEGIN DECLARE SECTION ;
      short InvTransNum_v ;
      short Qty_v ;
      short OrderNum_v ;
EXEC SQL END DECLARE SECTION ;
```

NOTES

- The C Language type int is not supported by ESQLC. Type int maps to 16 or 32 bits depending on the machine architecture. This can create rounding errors at runtime as values are passed across different machine architectures.
- Variables you declare in a BEGIN-END DECLARE SECTION can be used in C Language statements as if they had been declared outside the DECLARE SECTION.
- The scope of variables follows host language scoping rules. The ESQLC variables are not visible outside the file in which they are declared.
- DECLARE sections are permissible only where host language declarations are permissible in the host language syntax. This restriction is due to how DECLARE SECTION blocks are translated into the main body of host language declarations.
- Avoid DECLARE sections in header files that are included by more than one source file. This can cause duplicate variables with the same name.
- The form of the variable created by ESQLC for each type is specified so that it can be manipulated from host language statements. Declaring variables allows you to use the variables in both host language and embedded SQL-92 statements.

AUTHORIZATION

None

SQL COMPLIANCE

Declarations that use host-language types

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

Static Array Types (Chapter 14 in the *Progress Embedded SQL-92 Guide and Reference*)

CALL Statement

Invokes a stored procedure.

SYNTAX

```
CALL proc_name ( [ parameter ] [ , ... ] );
```

proc_name

The name of the procedure to invoke.

parameter

Literal or variable value to pass to the procedure.

AUTHORIZATION

Must have DBA or execute privileges

SQL COMPLIANCE

ODBC Extended SQL grammar, when enclosed in ODBC escape clause

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CREATE PROCEDURE Statement, DROP PROCEDURE Statement

CLOSE Statement

Closing a cursor changes the state of the cursor from open to closed.

SYNTAX

```
EXEC SQL CLOSE cursor_name ;
```

cursor_name

An identifier named earlier in a DECLARE CURSOR statement and an OPEN CURSOR statement.

EXAMPLE

This example is a code fragment from the seldone routine in the sample program 4DynSel.pc. The complete source for sample program, 4DynSel.pc, is listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
EXEC SQL CLOSE dyncur ;
EXEC SQL COMMIT WORK ;
```

NOTES

- Only a cursor in the open state can be set to the closed state.
- When a transaction ends, any cursors in the open state are automatically set to the closed state.
- When a cursor is in the closed state, you cannot perform FETCH, DELETE, or UPDATE operations using that cursor.
- It is good practice to close cursors explicitly.

AUTHORIZATION

None (See AUTHORIZATION for the OPEN Statement)

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

DECLARE CURSOR Statement, OPEN Statement, FETCH Statement, positioned UPDATE Statement, positioned DELETE Statement

Column Constraints

Specifies a constraint for a column that restricts the values that the column can store. INSERT, UPDATE, or DELETE statements that violate the constraint fail. The database returns a *constraint violation* error with SQLCODE of -20116.

Column constraints are similar to table constraints, but their definitions are associated with a single column rather than the entire table.

SYNTAX

```
CONSTRAINT constraint_name
   NOT NULL [ PRIMARY KEY | UNIQUE ]
  | REFERENCES [ owner_name.]table_name [ ( column_name ) ]
  | CHECK ( search_condition )
```

`CONSTRAINT` *constraint_name*

Allows you to assign a name that you choose to the column constraint. While this specification is optional, this facilitates making changes to the column definition. If you do not specify a *constraint_name*, the database assigns a name. These names can be long and unwieldy, and you must query system tables to retrieve the name.

`NOT NULL`

Restricts values in the column to values that are not null.

`NOT NULL PRIMARY KEY`

Defines the column as the primary key for the table. There can be at most one primary key for a table. A column with the `NOT NULL PRIMARY KEY` constraint should not contain null or duplicate values. See the NOTE below.

Other tables can name primary keys as foreign keys in their `REFERENCES` clauses. If they do, SQL restricts operations on the table containing the primary key in the following ways:

- `DROP TABLE` statements that delete the table fail.
- `DELETE` and `UPDATE` statements that modify values in the column that match a foreign key's value also fail.

`NOT NULL UNIQUE`

Defines the column as a unique key that cannot contain null or duplicate values. Columns with `NOT NULL UNIQUE` constraints defined for them are also called candidate keys.

Other tables can name unique keys in their `REFERENCES` clauses. If they do, SQL restricts operations on the table containing the unique key in the following ways:

`REFERENCES` *table_name* **[** (*column_name*) **]**

Defines the column as a foreign key and specifies a matching primary or unique key in another table. The `REFERENCES` clause names the matching primary or unique key.

A foreign key and its matching primary or unique key specify a *referential constraint*. A value stored in the foreign key must either be null or be equal to some value in the matching unique or primary key.

You can omit the *column_name* argument if the table specified in the `REFERENCES` clause has a primary key and you want the primary key to be the matching key for the constraint.

`CHECK (` *search_condition* `)`

Specifies a column-level check constraint. SQL restricts the form of the search condition. The search condition must not:

- Refer to any column other than the one with which it is defined
- Contain aggregate functions, subqueries, or parameter references

EXAMPLES

The following example shows the creation of a primary key column on the supplier table:

```
CREATE TABLE supplier (
     supp_no     INTEGER CONSTRAINT supp_key_con NOT NULL PRIMARY KEY,
     name        CHAR (30),
     status      SMALLINT,
     city        CHAR (20)
) ;
```

The following example creates a NOT NULL UNIQUE constraint to define the column ss_no as a unique key for the employee table.

```
CREATE TABLE employee (
     empno        INTEGER NOT NULL PRIMARY KEY,
     ss_no        INTEGER NOT NULL UNIQUE,
     ename        CHAR (19),
     sal          NUMERIC (10, 2),
     deptno       INTEGER NOT NULL
) ;
```

The following example defines order_item.orditem_order_no as a foreign key that references the primary key orders.order_no:

```
CREATE TABLE orders (
     order_no INTEGER NOT NULL PRIMARY KEY,
     order_date DATE
) ;

CREATE TABLE order_item (
     orditem_order_no INTEGER REFERENCES orders ( order_no ),
     orditem_quantity INTEGER
) ;
```

NOTE: The second CREATE TABLE statement in the previous example could have omitted the column name order_no in the REFERENCES clause, since it refers to the primary key of table orders.

The following example creates a check constraint:

```
CREATE TABLE supplier (
     supp_no     INTEGER NOT NULL,
     name        CHAR (30),
     status      SMALLINT,
     city        CHAR (20) CHECK (supplier.city <> 'BadApple')
) ;
```

NOTE: If a column is defined with a UNIQUE column constraints, no error results if more than one row has a NULL value for the column. This is in compliance with the SQL-92 standard.

COMMIT Statement

Commits a transaction explicitly after executing one or more SQL statements. Committing a transaction makes permanent any changes made by the SQL statements.

SYNTAX

```
COMMIT [ WORK ] ;
```

NOTES

- The SQL statements executed prior to executing the COMMIT statement are executed as one atomic transaction that is recoverable and durable. The transaction is serializable if you specify this isolation level.
- On a system failure and or the execution of the ROLLBACK, the transaction is rolled back to its initial state. Any changes made by the transaction are undone, restoring the database to its initial state. In the event of a system failure, the transaction will be rolled back during crash recovery when the database is restarted.
- A COMMIT operation makes any database modifications made by that transaction permanent.
- Once a COMMIT operation is executed the database modifications cannot be rolled back.
- Once a COMMIT operation is executed the transaction modifications are guaranteed durable regardless of any transient system failures.
- The atomicity applies only to the database modification and not to any direct I/O performed to devices such as the terminal, printer, and OS files by the application code.
- A COMMIT operation releases all locks implicitly or explicitly acquired by the transaction.
- A COMMIT operation closes all open cursors.

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL and interactive SQL

RELATED STATEMENTS

ROLLBACK Statement

CONNECT Statement

Establishes a connection to a database. Optionally, the CONNECT statement can also specify a name for the connection and a *username* and *password* for authentication.

SYNTAX

```
CONNECT TO connect_string
  [ AS connection_name ]
  [ USER username ]
  [ USING password ] ;
```

connect_string

SYNTAX

```
{ DEFAULT | db_name | db_type:T:host_name:port_num:db_name }
```

NOTE: Arguments to CONNECT must be either string literals enclosed in quotation marks or character-string host variables.

connect_string

Specifies to which database to connect. If the CONNECT statement specifies DEFAULT, SQL tries to connect to the environment-defined database, if any. The value of the DB_NAME environment variable specifies the default connect string.

The *connect_string* can be a simple database name or a complete *connect_string*. A complete connect string has the following components:

Connect String	Description
db_type	Type of database. The only currently supported database type is '*progress*.'
T	T directs the SQL engine to use the TCP/IP protocol.
host_name	Name of the system where the database resides.
port_num	Port number to use for the connection.
db_name	Name of the database.

connection_name

The name of the connection as either a character literal or host variable. If the CONNECT statement omits a connection name, the default is the name of the database. Connection names must be unique.

username

User name for authentication of the connection. SQL verifies the user name against a corresponding password before it connects to the database. The value of the DH_USER environment variable specifies the default user name. If DH_USER is not set, the value of the USER environment variable specifies the default user name.

password

Password for authentication of the connection. SQL verifies the password against a corresponding user name before it connects to the database.

The value of the DH_PASSWD environment variable determines the default password.

NOTES

- Arguments to CONNECT must be either string literals enclosed in quotation marks or character_string host variables.
- An application can connect to more than one database at a time, with a maximum of 10 connections. However, the application can actually gain access to only one database at a time. The database name specified in the CONNECT statement becomes the active one.
- If an application executes a SQL statement before connecting to a database, an attempt is made to connect to the environment-defined database, if any. If the connection is successful, the SQL statement is executed on that database.

EXAMPLES

The following examples illustrate the CONNECT statement:

- The first statement shown connects to the salesdb database on the local system.
- The second statement connects to the custdb database on the local system.
- The last statement connects to the environment_defined database by default.

```
CONNECT TO "salesdb" AS "sales_conn";
CONNECT TO "progress:T:localhost:custdb" AS "cust_conn";
CONNECT TO DEFAULT;
```

See also the last example for the DISCONNECT Statement which illustrates the CONNECT, SET CONNECTION, and DISCONNECT statements in combination.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL only

RELATED STATEMENTS

DISCONNECT Statement, SET CONNECTION Statement

CREATE INDEX Statement

Creates an index on the specified table using the specified columns of the table. An index improves the performance of SQL operations whose predicates are based on the indexed column. However, an index slows performance of INSERT, DELETE, and UPDATE operations.

SYNTAX

```
CREATE [ UNIQUE ] INDEX index_name
  ON table_name
  ( { column_name [ ASC | DESC ] } [, ... ] )
  [ AREA area_name ];
```

UNIQUE

Does not allow the table to contain any rows with duplicate column values for the set of columns specified for that index.

index_name

Must be unique within the local database.

table_name

The name of the table on which the index is being built.

column_name [, ...]

The columns on which searches and retrievals will be ordered. These columns are called the index key. When more than one column is specified in the CREATE INDEX statement, a concatenated index is created.

area_name

The name of the storage area where the index and its entries are stored.

ASC | DESC

Allows the index to be ordered as either ascending (ASC) or descending (DESC) on each column of the concatenated index. The default is ASC.

NOTES

- The first index you create on a table should be the most fundamental key of the table. This index (the first one created on a table) cannot be dropped except by dropping the table.
- An index slows performance of INSERT, DELETE, and UPDATE operations.

EXAMPLE

This example illustrates how to create a UNIQUE INDEX on a table:

```
CREATE UNIQUE INDEX custindex ON customer (cust_no) ;
```

AUTHORIZATION

Must have DBA privilege or INDEX privilege on the table.

SQL COMPLIANCE

ODBC Core SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CREATE TABLE Statement, DROP INDEX Statement

CREATE PROCEDURE Statement

Creates a stored procedure. Stored procedures contain a Java code snippet that is processed into a Java class definition and stored in the database in text and compiled form. SQL applications invoke stored procedures through the SQL CALL statement or the procedure-calling mechanisms of ODBC and JDBC.

SYNTAX

```
CREATE PROCEDURE [ owner_name.]procname
  ( [ parameter_decl [ , ... ] ] )
  [ RESULT  (  column_name data_type  [  , ...  ]  ) ]
  [ IMPORT
       java_import_clause  ]
   BEGIN
     java_snippet
   END
```

parameter_decl

SYNTAX

```
{ IN | OUT | INOUT } parameter_name  data_type
```

owner_name

Specifies the owner of the procedure. If the name is different from the user name of the user executing the statement, then the user must have DBA privileges.

procname

Names the stored procedure. DROP PROCEDURE statements specify the procedure name defined here. SQL also uses *procname* in the name of the Java class that it creates from the Java snippet.

IN | OUT | INOUT

Specifies whether following parameter declaration is input, output, or both.

Calling applications pass values for input parameters in the CALL statement or CALL escape sequence.

Stored procedures assign values to output parameters as part of their processing.

INOUT parameters have both a value passed in and receive a new value during procedure processing.

parameter_name data_type

Names a parameter and associates an SQL data type with it. The *data_type* must be one of the supported data types described in the "Data Types" section in Chapter 2, "SQL-92 Language Elements."

RESULT (*column_name data_type* [, ...])

Specifies columns in the result set the procedure returns. If the CREATE PROCEDURE statement includes this clause, the Java snippet must explicitly insert rows into the result set using the Java class *SQLResultSet*.

Note that the *column_name* argument is not used in the body of the stored procedure. Instead, methods of the Java classes refer to columns in the result set by ordinal number, not by name.

```
IMPORT
      java_import_clause
```

Specifies standard Java classes to import. The IMPORT keyword must be uppercase and on a separate line.

```
BEGIN
      java_snippet
END
```

The body of the stored procedure. The body is a sequence of Java statements between the BEGIN and END keywords. The Java statements become a method in a class that SQL creates and submits to the Java compiler.

NOTE: The BEGIN and END keywords must be uppercase and on separate lines.

EXAMPLE

```
CREATE PROCEDURE new_sal (
     IN  deptnum     INTEGER,
     IN  pct_incr    INTEGER,
)
RESULT (
     empname CHAR(20),
     oldsal   NUMERIC,
     newsal   NUMERIC
)
BEGIN
     String ename = new String (20) ;
     BigDecimal osal = new BigDecimal () ;
     BigDecimal nsal = new BigDecimal () ;

     SQLCursor empcursor = new SQLCursor (
          "SELECT empname, sal, (sal * ( ? /100) + NVL (comm, 0)) total,
          FROM emp WHERE deptnum = ? " ) ;

     empcursor.setParam (1, pct_incr);
     empcursor.setParam (2, deptnum);
     empcursor.open () ;
     do
     {
          empcursor.fetch ();
          if (empcursor.found ())
          {
               ename = (StringBugger) empcursor.getValue (1, CHAR);
               osal = (BigDecimal) empcursor.getValue (2, NUMERIC);
               nsal = (BigDecimal) empcursor.getValue (3, NUMERIC);

               SQLResultSet.set (1, ename);
               SQLResultSet.set (2, osal);
               SQLResultSet.set (3, nsal) ;
               SQLResultSet.insert ();
          }
     } while (empcursor.found ()) ;

     empcursor.close () ;
END
```

AUTHORIZATION

Must have DBA privilege, RESOURCE privilege, or ownership of procedure.

SQL COMPLIANCE

SQL-92, ODBC Core SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CALL Statement, DROP PROCEDURE Statement

CREATE SYNONYM Statement

Creates a synonym for the specified table, view, or synonym. A *synonym* is an alias that SQL statements can use instead of the name specified when the table, view, or synonym was created.

SYNTAX

```
CREATE [ PUBLIC ] SYNONYM synonym
  FOR [ owner_name.] {table_name | view_name  | synonym } ;
```

PUBLIC

Specifies that the synonym is public: all users can refer to the name without qualifying it. By default, the synonym is private: other users must qualify the synonym by preceding it with the user name of the user who created it.

Users must have the DBA privilege to create public synonyms.

SYNONYM *synonym*

Name for the synonym.

FOR [*owner_name.*] { *table_name* | *view_name* | *synonym* }

Table, view, or synonym for which SQL creates the new synonym.

EXAMPLE

```
CREATE SYNONYM customer FOR smith.customer ;
CREATE PUBLIC SYNONYM public_suppliers FOR smith.suppliers ;
```

AUTHORIZATION

Must have DBA privilege or RESOURCE privilege.

SQL COMPLIANCE

Progress Extension

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

DROP SYNONYM Statement

CREATE TABLE Statement

Creates a table definition. A table definition consists of a set of named column definitions for data values that will be stored in rows of the table. SQL provides two forms of the CREATE TABLE statement.

The first syntax form explicitly specifies column definitions. The second syntax form, with the AS *query_expression* clause, implicitly defines the columns using the columns in a query expression.

SYNTAX

```
CREATE TABLE [ owner_name.]table_name
  ( { column_definition | table_constraint }, ...)
    [ AREA area_name ]
    ;
CREATE TABLE [ owner_name.]table_name
    [ (column_name [ NOT NULL] , ... ) ]
    [ AREA area_name ]
    AS query_expression
    ;
```

owner_name

Specifies the owner of the table. If the name is different from the user name of the user executing the statement, then the user must have DBA privileges.

table_name

Names the table you are defining.

column_definition:

SYNTAX

```
column_name   data_type
  [ DEFAULT { literal | NULL | SYSDATE } ]
  [ column_constraint [ column_constraint , ... ] ]
```

column_name data_type

Names a column and associates a data type with it. The column names specified must be different from other column names in the table definition. The *data_type* must be one of the supported data types described in the "Data Types" section in Chapter 2, "SQL-92 Language Elements."

Note that when a table contains more than one column, a comma separator is required after each *column_definition* except for the final *column_definition*.

DEFAULT

Specifies an explicit default value for a column. The column takes on the value if an INSERT statement does not include a value for the column. If a column definition omits the DEFAULT clause, the default value is NULL.

The DEFAULT clause accepts the following arguments:

literal	An integer, numeric, or string constant.
NULL	A null value.
SYSDATE	The current date. Valid only for columns defined with DATE data types. SYSDATE is equivalent to the Progress default keyword TODAY.

column_constraint

Specifies a constraint that will be applied while inserting or updating a value in the associated column. For more information, see the "Column Constraints" section.

table_constraint

Specifies a constraint that will be applied while inserting or updating a row in the table. For more information, see the Table Constraints section.

AREA *area_name*

Specifies the name of the storage area where data stored in the table is to be stored.

If the specified area does not exist, the database returns an error. If you do not specify an area, the default area is used.

AS *query_expression*

Specifies a query expression to use for the data types and data values of the table's columns. The types and lengths of the columns of the query expression result become the types and lengths of the respective columns in the table created. The rows in the resultant set of the query expression are inserted into the table after creating the table. In this form of the CREATE TABLE statement, column names are optional. If omitted, the names of the table's columns are taken from the column names of the query expression.

EXAMPLES

In the following CREATE TABLE supplier_item example, the user issuing the CREATE TABLE statement must have REFERENCES privilege on the itemno column of the table john.item:

```
CREATE TABLE supplier_item (
     supp_no     INTEGER NOT NULL PRIMARY KEY,
     item_no     INTEGER NOT NULL REFERENCES john.item (itemno),
     qty         INTEGER
) ;
```

The following CREATE TABLE statement explicitly specifies a table owner, gus:

```
CREATE TABLE gus.account (
     account  integer,
     balance  money (12),
     info     char (84)
) ;
```

The following example shows the AS *query_expression* form of CREATE TABLE to create and load a table with a subset of the data in the customer table:

```
CREATE TABLE gus.dealer (name, street, city, state)
     AS
          SELECT name, street, city, state
          FROM customer
          WHERE customer.state IN ('CA','NY', 'TX') ;
```

The following example includes a NOT NULL column constraint and DEFAULT clauses for column definitions:

```
CREATE TABLE emp (
     empno  integer NOT NULL,
     deptno  integer DEFAULT 10,
     join_date date DEFAULT NULL
) ;
```

AUTHORIZATION

Must have DBA privilege, RESOURCE privilege or SELECT privilege.

SQL COMPLIANCE

SQL-92, ODBC Minimum SQL grammar, Progress Extensions: AREA and AS *query_expression*

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

DROP TABLE Statement, "Query Expressions" in Chapter 2, "SQL-92 Language Elements"

CREATE TRIGGER Statement

Creates a trigger for the specified table. A trigger is a special type of automatically executed stored procedure that helps ensure referential integrity for a database.

Triggers contain Java source code that can use SQL Java classes to carry out database operations. Triggers are automatically activated when an INSERT, UPDATE, or DELETE statement changes the trigger's target table. The Java source code details what actions the trigger takes when it is activated.

SYNTAX

```
CREATE TRIGGER [ owner_name.]trigname
  { BEFORE | AFTER }
  { INSERT | DELETE | UPDATE [ OF column_name [ , ... ] }
  ON table_name
  [ REFERENCING { OLDROW [ ,NEWROW ] | NEWROW [ ,OLDROW ] } ]
  [ FOR EACH { ROW | STATEMENT } ]
  [ IMPORT
      java_import_clause ]
    BEGIN
         java_snippet
    END
```

owner_name

Specifies the owner of the trigger. If the name is different from the user name of the user executing the statement, then the user must have DBA privileges.

trigname

Names the trigger. `DROP TRIGGER` statements specify the trigger name defined here. SQL also uses *trigname* in the name of the Java class that it creates from the Java snippet.

`BEFORE | AFTER`

Denotes the *trigger action time*. The trigger action time specifies whether the triggered action, implemented by *java_snippet*, executes `BEFORE` or `AFTER` the invoking `INSERT`, `UPDATE`, or `DELETE` statement.

```
INSERT | DELETE | UPDATE [ OF column_name [ , ... ] ]
```

Denotes the *trigger event*. The trigger event is the statement that activates the trigger.

If UPDATE is the triggering statement, this clause can include an optional column list. Only updates to any of the specified columns will activate the trigger. If UPDATE is the triggering statement and does not include the optional column list, then any UPDATE on the table will activate the trigger.

```
ON table_name
```

Identifies the name of the table where the trigger is defined. A triggering statement that specifies *table_name* causes the trigger to execute. *table_name* cannot be the name of a view.

```
REFERENCING   OLDROW [ , NEWROW ] | NEWROW [ , OLDROW ]
```

Provides a mechanism for SQL to pass row values as input parameters to the stored procedure implemented by *java_snippet*. The code in *java_snippet* uses the *getValue* method of the NEWROW and OLDROW objects to retrieve values of columns in rows affected by the trigger event and store them in procedure variables. This clause is allowed only if the trigger specfies the FOR EACH ROW clause.

The meaning of the OLDROW and NEWROW arguments of the REFERENCING clause depends on whether the trigger event is INSERT, UPDATE, or DELETE:

- INSERT...REFERENCING NEWROW means the triggered action can access values of columns of each row inserted. SQL passes the column values specified by the INSERT statement.
- INSERT...REFERENCING OLDROW is meaningless, since there are no existing values for a row being inserted. INSERT...REFERENCING OLDROW generates a syntax error.
- UPDATE...REFERENCING OLDROW means the triggered action can access the values of columns, before they are changed, of each row updated. SQL passes the column values of the row as it exists in the database before the update operation.
- DELETE...REFERENCING OLDROW means the triggered action can access values of columns of each row deleted. SQL passes the column values of the row as it exists in the database before the delete operation.
- DELETE...REFERENCING NEWROW is meaningless, since there are no new existing values to pass for a row being deleted. DELETE...REFERENCING OLDROW generates a syntax error.

- UPDATE is the only triggering statement that allows both NEWROW and OLDROW in the REFERENCING clause.
- UPDATE...REFERENCING NEWROW means the triggered action can access the values of columns, after they are changed, of each row updated. SQL passes the column values specified by the UPDATE statement.

NOTES

- The trigger action time (BEFORE or AFTER) does not affect the meaning of the REFERENCING clause. For instance, BEFORE UPDATE...REFERENCING NEWROW still means the values of columns after they are updated will be available to the triggered action.
- The REFERENCING clause generates an error if the trigger does not include the FOR EACH ROW clause.

```
FOR EACH { ROW | STATEMENT }
```

Controls the execution frequency of the triggered action implemented by *java_snippet*.

FOR EACH ROW means the triggered action executes once for each row being updated by the triggering statement. CREATE TRIGGER must include the FOR EACH ROW clause if it also includes a REFERENCING clause.

FOR EACH STATEMENT means the triggered action executes only once for the whole triggering statement. FOR EACH STATEMENT is the default.

```
IMPORT  java_import_clause
```

Specifies standard Java classes to import. The IMPORT keyword must be uppercase and on a separate line.

```
BEGIN
   java_snippet
END
```

Denotes the body of the trigger or the *triggered action*. The body contains the Java source code that implements the actions to be completed when a triggering statement specifies the target table. The Java statements become a method in a class that SQL creates and submits to the Java compiler.

The BEGIN and END keywords must be uppercase and on separate lines.

NOTES

- Triggers can take action on their own table so that they invoke themselves. SQL limits such recursion to five levels.
- You can you have multiple triggers on the same table. Multiple UPDATE triggers on the same table must specify different columns. SQL-92 executes all triggers applicable to a given combination of table, trigger event, and action time.
- The actions carried out by a trigger can fire another trigger. When this happens, the other trigger's actions execute before the rest of the first trigger finishes executing.
- If a constraint and trigger are both invoked by a particular SQL statement, SQL checks constraints first, so any data modification that violates a constraint does not also fire a trigger.
- To modify an existing trigger, you must delete it and issue another `CREATE TRIGGER` statement. You can query the *systrigger* system table for information about the trigger before you delete it.

EXAMPLE

This example illustrates an UPDATE trigger on a table called BUG_INFO. If the STATUS or PRIORITY fields are modified, the trigger modifies the BUG_SUMMARY and BUG_STATUS tables appropriately, based on defined conditions:

(1 of 3)

```
CREATE TRIGGER BUG_UPDATE_TRIGGER
  AFTER UPDATE OF STATUS, PRIORITY ON BUG_INFO
  REFERENCING OLDROW, NEWROW
  FOR EACH ROW

IMPORT
import java.sql.* ;
```

(2 of 3)

```
BEGIN
try
{
     // column number of STATUS is 10
     String  old_status, new_status;

     old_status = (STRING) OLDROW.GetValue(10, CHAR);
     new_status = (STRING) NEWROW.GetValue(10, CHAR);

     if ((old_status.CompareTo("OPEN") == 0) &&
     (new_status.CompareTo("FIXED") == 0))

     {
          //  If STATUS has changed from OPEN to FIXED
          //  increment the bugs_fixed_cnt by 1 in the
          //  row corresponding to current month
          //  and current year

          SQLIStatement  update_stmt (
          " update BUG_STATUS set bugs_fixed_cnt = bugs_fixed_cnt + 1 "
          " where  month = ?  and year = ?"
          );

          Integer  current_month = 10;
          Integer  current_year  = 1997;

          update_stmt.SetParam(1, current_month);
          update_stmt.SetParam(2, current_year);
          update_stmt.Execute();

          SQLIStatement  insert_stmt (
          " insert into BUG_SUMMARY values (?,?,?)"
          );

          //  Column numbers for bug_id, priority, reported_on
          //  and fixed_on are 1, 2, 5, and 6
          String  bug_id,priority;
          Date    reported_on, fixed_on;
```

(3 of 3)

```
 bug_id = (String) NEWROW.GetValue(1, CHAR);
          priority = (String) NEWROW.GetValue(2, CHAR);
          reported_on = (Date) NEWROW.GetValue(5,DATE);
          fixed_on = (Data) NEWROW.GetValue(6, DATE);

          Integer turn_around_time = fixed_on - reported_on;

           insert_stmt.SetParam(1, bug_id);
          insert_stmt.SetParam(2, priority);
          insert_stmt.SetParam(3, turn_around_time);
          insert_stmt.Execute();
          }

     // If PRIORITY has changed to URGENT,
     // increment the bugs_escalated by 1 in the month field .

     String  old_priority, new_priority;
     old_priority = (String) OLDROW.GetValue(2, CHAR);
     new_priority = (String) NEWROW.GetValue(2, CHAR);

     if ((new_priority.CompareTo("URGENT")==0)  &&
     (old_priority.CompareTo("URGENT") != 0))
     {
          //  If PRIORITY  has changed to URGENT increment the
          //  bugs_escalated  by 1 in the row corresponding to
          //  current month and current year
          SQLIStatement  update_stmt (
          " update BUG_STATUS
          set bugs_escalated_cnt = bugs_escalated_cnt + 1 "
          " where  month = ?  and year = ?"
          );

          Integer  current_month = 10;
          Integer  current_year  = 1997;

          update_stmt.SetParam(1, current_month);
          update_stmt.SetParam(2, current_year);
          update_stmt.Execute();
     }

}
catch (SQLException e)
{
     // Log the exception message from e.
     SQLException  sqle = new SQLException("UPDATE_BUG_TRIGGER failed");
     throw sqle;
}
END
```

AUTHORIZATION

Must have the DBA privilege or RESOURCE privilege

SQL COMPLIANCE

SQL-92, ODBC Core SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

DROP TRIGGER Statement

CREATE USER Statement

Creates the specified user.

SYNTAX

```
CREATE USER 'username', 'password' ;
```

EXAMPLE

In this example an account with DBA privileges creates the '*username*' 'Jasper' with '*password*' 'spaniel':

```
CREATE USER 'Jasper', 'spaniel' ;
```

NOTES

- Used in conjunction with ALTER USER Statement and DROP USER Statement, the CREATE USER statement provides a way to manage user records through SQL. This SQL_only solution does not require the Progress dictionary.
- The '*username*' and '*password*' must be enclosed in quotes.
- Before issuing the CREATE USER statement, there are no users defined in the user table and any user may log into the database.
- After issuing the CREATE USER statement, only users defined in the user table may log into the database.

AUTHORIZATION

Must have DBA privileges

SQL COMPLIANCE

Progress Extension

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

ALTER USER Statement, DROP USER Statement

CREATE VIEW Statement

Creates a view with the specified name on existing tables or views.

SYNTAX

```
CREATE VIEW [ owner_name.]view_name
  [ ( column_name, column_name,... ) ]
  AS [ ( ] query_expression [ ) ]
  [ WITH CHECK OPTION ] ;
```

owner_name

Owner of the created view

(*column_name*, *column_name*,...)

Specifies column names for the view. These names provide an alias for the columns selected by the query specification. If the column names are not specified, then the view is created with the same column names as the tables or views on which it is based.

WITH CHECK OPTION

Checks that the updated or inserted row satisfies the view definition. The row must be selectable using the view. The WITH CHECK OPTION clause is only allowed on an updatable view.

NOTES

- A view is deletable if deleting rows from that view is allowed. For a view to be deletable, the view definition has to satisfy the following conditions:
  - The first FROM clause contains only one table reference or one view reference.
  - There are no aggregate functions, DISTINCT clause, GROUP BY clause, or HAVING clause in the view definition.
  - If the first FROM clause contains a view reference, then the view referred to is deletable.

- A view is updatable if updating rows from that view is allowed. For a view to be updatable, the view has to satisfy the following conditions:
  - The view is deletable (it satisfies all the conditions specified above for deletability).
  - All the select expressions in the first SELECT clause of the view definition are simple column references.
  - If the first FROM clause contains a view reference, then the view referred to is updatable.
- A view is insertable if inserting rows into that view is allowed. For a view to be insertable, the view has to satisfy the following conditions:
  - The view is updatable (it satisfies all the conditions specified above for updatability).
  - If the first FROM clause contains a table reference, then all NOT NULL columns of the table are selected in the first SELECT clause of the view definition.
  - If the first FROM clause contains a view reference, then the view referred to is insertable.

EXAMPLE

```
CREATE VIEW ne_customers AS
          SELECT cust_no, name, street, city, state, zip
          FROM customer
          WHERE state IN ( 'NH', 'MA', 'ME', 'RI', 'CT', 'VT' )
          WITH CHECK OPTION ;
CREATE VIEW order_count (cust_number, norders) AS
          SELECT cust_no, COUNT(*)
          FROM orders
          GROUP BY cust_no;
```

AUTHORIZATION

Must have DBA privilege, RESOURCE privilege, or SELECT privilege

SQL COMPLIANCE

SQL-92, ODBC Core SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

"Query Expressions" in Chapter 2, "SQL-92 Language Elements", DROP VIEW Statement

DECLARE CURSOR Statement

Associates a cursor with a static query or a prepared dynamic query statement. The query or the prepared statement can have references to host variables.

SYNTAX

```
DECLARE cursor_name CURSOR FOR
  { query_expression [ ORDER BY clause ] [ FOR UPDATE clause ]
    | prepared_statement_name
  } ;
```

cursor_name

A name you assign to the cursor. The name must meet the requirements for an identifier.

query_expression **[** ORDER BY *clause* **]** **[**FOR UPDATE *clause* **]**

A complete query expression.

prepared_statement_name

The name assigned to a prepared SQL-92 statement in an earlier PREPARE statement.

EXAMPLES

This example illustrates static processing of a SELECT statement. This is a code fragment from the `StatSel` function in sample program `2StatSel.pc`. The complete source for this sample program is in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL DECLARE stcur CURSOR FOR
     SELECT InvTransNum, Qty, OrderNum FROM PUB.InventoryTrans ;
EXEC SQL OPEN stcur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

This example is a code fragment from the `DynSel` function in sample program `4DynSel.pc`, which illustrates dynamic processing of a SELECT statement. The complete source for this sample program is in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

NOTES

- You must declare a cursor before any OPEN Statement, FETCH Statement, or CLOSE Statement.
- The scope of the cursor declaration is the entire source file in which it is declared. The operations on the cursor such as OPEN Statement, CLOSE Statement, and FETCH Statement can occur only within the same compilation unit as the cursor declaration.
- The use of a cursor allows the execution of the positioned forms of the UPDATE and DELETE statements.
- If the DECLARE statement corresponds to a static SQL statement with parameter references:
  - The DECLARE statement must be executed before each execution of an OPEN statement for the same cursor.
  - The DECLARE statement and the OPEN statement that follows must occur within the same transaction within the same task.
  - If the statement contains parameter references to automatic variables or function arguments, the DECLARE statement and the following OPEN statement for the same cursor must occur within the same C function.
- See the "SELECT a Single Row" reference entry for descriptions of the ORDER BY Clause and FOR UPDATE Clause.

AUTHORIZATION

None (See AUTHORIZATION for the OPEN Statement)

SQL COMPLIANCE

SQL-92. Progress Extension: `prepared_statement_name`

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

PREPARE Statement, OPEN Statement, FETCH Statement, CLOSE Statement, SELECT Statement, "Query Expressions" in Chapter 2, "SQL-92 Language Elements"

DELETE Statement

Deletes zero, one, or more rows from the specified table that satisfy the search condition specified in the WHERE clause. If the optional WHERE clause is not specified, then the DELETE statement deletes all rows of the specified table.

SYNTAX

```
DELETE FROM [ owner_name.] { table_name | view_name }
  [ WHERE search_condition ] ;
```

EXAMPLE

```
DELETE FROM customer WHERE customer_name = 'RALPH' ;
```

NOTE

- If the table has primary or candidate keys and there are references from other tables to the rows to be deleted, the statement is rejected.

AUTHORIZATION

Must have DBA privilege, ownership of the table, of DELETE permission of the table.

SQL COMPLIANCE

SQL-92, ODBC Extended SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

"Search Conditions" in Chapter 2, "SQL-92 Language Elements", WHERE Clause

DESCRIBE Statement

Writes information about a prepared statement to the SQL Descriptor Area (SQLDA). You use a DESCRIBE statement in a series of steps that allows a program to accept SQL-92 statements at run time. Dynamically generated statements are not part of a program's source code; they are generated at run time.

There are two forms of the DESCRIBE statement:

- The DESCRIBE BIND VARIABLES Statement writes information about input variables in an expression to an SQLDA. These variables can be substitution variable names or parameter markers.
- The DESCRIBE SELECT LIST Statement writes information about select list items in a prepared SELECT statement to an SQLDA.

SYNTAX

```
DESCRIBE [ BIND VARIABLES | SELECT LIST ] FOR statement_name
   INTO input_sqlda_name ;
```

The SQLDA is a host language data structure used in dynamic SQL-92 processing. DESCRIBE statements write information about the number, data types, and sizes of input variables or select list items to SQLDA structures. Program logic then processes that information to allocate storage. OPEN, EXECUTE, and FETCH statements read the SQLDA structures for the addresses of the allocated storage.

For more information see the DESCRIBE BIND VARIABLES Statement and the DESCRIBE SELECT LIST Statement.

DESCRIBE BIND VARIABLES Statement

Writes information about any input variables in the prepared statement to an input SQLDA structure.

SYNTAX

```
DESCRIBE BIND VARIABLES FOR statement_name INTO input_sqlda_name ;
```

statement_name

The name of an input SQL-92 statement to be processed using dynamic SQL-92 steps. Typically, this is the same *statement_name* used in the PREPARE statement.

input_sqlda_name

The name of the SQLDA structure to which DESCRIBE will write information about input variables. Input variables represent values supplied to INSERT and UPDATE statements at run time, and to predicates in DELETE, UPDATE, and SELECT statements at run time.

To utilize the DESCRIBE BIND VARIABLES statement in your application, issue statements in the following order:

1. PREPARE
2. DESCRIBE BIND VARIABLES
3. EXECUTE or OPEN CURSOR

The DESCRIBE BIND VARIABLES statement writes the number of input variables to the sqld_nvars field of the SQLDA. If the sqld_size field of the SQLDA is not equal to or greater than this number, DESCRIBE writes the value as a negative number to sqld_nvars. Design your application to check sqld_nvars for a negative number to determine if a particular SQLDA is large enough to process the current input statement.

Input variables in dynamic SQL-92 statements are identified by parameter markers or as substitution names. See the PREPARE Statement for more information.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

PREPARE Statement, DECLARE CURSOR Statement, OPEN Statement, FETCH Statement, CLOSE Statement

DESCRIBE SELECT LIST Statement

Writes information about select list items in a prepared SELECT statement to an output SQLDA structure.

SYNTAX

```
DESCRIBE SELECT LIST FOR statement_name INTO output_sqlda_name ;
```

statement_name

The name of a SELECT statement to be processed using dynamic SQL-92 steps. Typically, this is the same *statement_name* as in the PREPARE Statement.

output_sqlda_name

The name of the SQLDA structure to which DESCRIBE will write information about select list items.

NOTE: Select list items are column names and expressions in a SELECT Statement. A FETCH Statement writes the values returned by a SELECT Statement to the addresses stored in an output SQLDA.

To utilize the DESCRIBE SELECT LIST statement in your application, issue statements in the following order:

1. DECLARE CURSOR
2. PREPARE
3. OPEN
4. DESCRIBE SELECT LIST
5. FETCH

A DESCRIBE SELECT LIST statement writes the number of select list items to the sqld_nvars field of an output SQLDA. If the sqld_size field of the SQLDA is not equal to or greater than this number, DESCRIBE writes the value as a negative number to sqld_nvars. Design your application to check sqld_nvars for a negative number to determine if a particular output SQLDA is large enough to process the current SELECT Statement.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

PREPARE Statement, DECLARE CURSOR Statement, OPEN Statement, FETCH Statement, CLOSE Statement

DISCONNECT Statement

Terminates the connection between an application and the database to which it is connected.

SYNTAX

```
DISCONNECT { 'connection_name' | CURRENT | ALL | DEFAULT } ;
```

connection_name

The name of the connection as either a character literal or host variable. See the CONNECT Statement for more information on the *connection_name*.

CURRENT

Disconnects the current connection.

ALL

Disconnects all established connections.

DEFAULT

Disconnects the connection to the default database.

EXAMPLES

The first example illustrates CONNECT TO AS '*connection_name*' and DISCONNECT '*connection_name*'.

```
EXEC SQL
     CONNECT TO 'progress:T:localhost:6745:salesdb' AS 'conn_1' ;
/*
**  C Language and embedded SQL-92 application processing against the
**  database in the connect_string
*/
   .
   .
   .
EXEC SQL
     DISCONNECT 'conn_1' ;
```

This example illustrates CONNECT TO DEFAULT and DISCONNECT DEFAULT:

```
EXEC SQL
    CONNECT TO DEFAULT ;
/*
**  C Language and embedded SQL-92 application processing against the
**  database in the connect_string
*/
   .
   .
   .
EXEC SQL
    DISCONNECT DEFAULT ;
```

This example disconnects all database connections. After you issue DISCONNECT ALL there is no current connection.

```
EXEC SQL
     DISCONNECT ALL;
```

The following example illustrates the CONNECT, SET CONNECTION, and DISCONNECT statements in combination using these steps:

1. CONNECT TO '*connect_string*' AS '*connection_name*' which establishes a *connect_string* connection to the database in the '*connect_string*'; the connection has the name 'conn_1'
2. CONNECT TO DEFAULT which establishes a connection to the DEFAULT database and sets this connection current
3. DISCONNECT DEFAULT which disconnects the connection to the DEFAULT database
4. SET CONNECTION '*connection_name*' which sets the 'conn_1' connection current
5. DISCONNECT CURRENT which disconnects the 'conn_1' connection

This example illustrates connection management statements in combination:

```
/*
**  1. CONNECT TO 'connect_string'
*/
EXEC SQL
     CONNECT TO 'progress:T:localhost:6745:salesdb' AS 'conn_1' ;
/*
**  2. CONNECT TO DEFAULT. This suspends the conn_1 connection
**     and sets the DEFAULT connection current
*/
EXEC SQL
     CONNECT TO DEFAULT ;
/*
** Application processing against the DEFAULT database
*/
   .
   .
   .
/*
**  3. DISCONNECT DEFAULT
*/
EXEC SQL
     DISCONNECT DEFAULT ;
/*
**  4. Set the first connection, conn_1, current
*/
EXEC SQL
     SET CONNECTION conn_1 ;
/*
** Application processing against the database in the connect_string
*/
   .
   .
   .
/*
**  5. DISCONNECT the conn_1 connection, which is the current connection.
*/
EXEC SQL
     DISCONNECT CURRENT ;
```

NOTES

- When you specify DISCONNECT '*connection_name*' or DISCONNECT CURRENT and there is also an established connection to the DEFAULT database, the connection to the DEFAULT database becomes the current connection. If there is no DEFAULT database there is no current connection after the SQL engine processes the DISCONNECT.
- The DISCONNECT DEFAULT statement terminates the connection to the DEFAULT database. If this connection is the current connection, there is no current connection after this DISCONNECT statement is executed.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL only

RELATED STATEMENTS

CONNECT Statement, SET CONNECTION Statement

DROP INDEX Statement

Deletes an index on the specified table.

SYNTAX

```
DROP INDEX [ index_owner_name.]index_name
    ON [ table_owner_name.]table_name  ;
```

index_owner_name

Specifies the name of the index owner. If *index_owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.

table_name

Verifies the *index_name* to correspond to the table. This argument is optional.

NOTE: You cannot drop the first index created on a table, except by dropping the table.

EXAMPLE

```
DROP INDEX custindex ON customer ;
```

AUTHORIZATION

Must have DBA privilege or ownership of the index.

SQL COMPLIANCE

ODBC Core SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL , ODBC applications, JDBC applications

RELATED STATEMENTS

CREATE INDEX Statement

DROP PROCEDURE Statement

Deletes a stored procedure.

SYNTAX

```
DROP PROCEDURE [ owner_name.]procedure_name ;
```

owner_name

Specifies the owner of the procedure.

procedure_name

Name of the stored procedure to delete.

EXAMPLE

```
DROP PROCEDURE new_sal ;
```

AUTHORIZATION

Must have DBA privilege or owner of a stored procedure.

SQL COMPLIANCE

SQL-92, ODBC Core SQL grammar

ENVIRONMENT

Embedded SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CALL Statement, CREATE PROCEDURE Statement

DROP SYNONYM Statement

Drops the specified synonym.

SYNTAX

```
DROP [ PUBLIC ] SYNONYM synonym ;
```

PUBLIC

Specifies that the synonym was created with the PUBLIC argument.

SYNONYM *synonym*

Name for the synonym.

EXAMPLE

```
DROP SYNONYM customer ;
DROP PUBLIC SYNONYM public_suppliers ;
```

NOTES

- If DROP SYNONYM specifies PUBLIC and the synonym was not a public synonym, SQL generates the **base table not found** error.
- If DROP SYNONYM does not specify public and the synonym was created with the PUBLIC argument, SQL generates the **base table not found** error.

AUTHORIZATION

Must have DBA privilege or ownership of the synonym (for DROP SYNONYM).

SQL COMPLIANCE

Progress Extension

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CREATE SYNONYM Statement

DROP TABLE Statement

Deletes the specified table.

SYNTAX

```
DROP TABLE  [ owner_name.]table_name ;
```

owner_name

Specifies the owner of the table.

table_name

Names the table to drop.

EXAMPLE

```
DROP TABLE customer ;
```

NOTES

- If *owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.
- When a table is dropped, the indexes on the table and the privileges associated with the table are dropped automatically.
- Views dependent on the dropped table are not automatically dropped, but become invalid.
- If the table is part of another table's referential constraint (if the table is named in another table's REFERENCES clause), the DROP TABLE statement fails. You must DROP the referring table first.

AUTHORIZATION

Must have DBA privilege or ownership of the table.

SQL COMPLIANCE

SQL-92, ODBC Minimum SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CREATE TABLE Statement

DROP TRIGGER Statement

Deletes a trigger.

SYNTAX

```
DROP TRIGGER [ owner_name.]trigger_name ;
```

owner_name

Specifies the owner of the trigger.

trigger_name

Names the trigger to drop.

EXAMPLE

```
DROP TRIGGER sal_check ;
```

AUTHORIZATION

Must have DBA privilege or ownership of the trigger.

SQL COMPLIANCE

Progress Extension

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CREATE TRIGGER Statement

DROP USER Statement

Deletes the specified user.

SYNTAX

```
DROP USER 'username' ;
```

'username'

Specifies the user name to delete. The `'username'` must be enclosed in quotes.

EXAMPLE

In this example, an account with DBA privileges drops the *'username'* 'Jasper':

```
DROP USER 'Jasper' ;
```

NOTES

- Used in conjunction with CREATE USER Statement and ALTER USER Statement, the DROP USER statement provides a way to manage user records through SQL. This SQL-only solution does not require the Progress dictionary.

AUTHORIZATION

Must have DBA privileges.

SQL COMPLIANCE

Progress Extension

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

ALTER USER Statement, CREATE USER Statement

DROP VIEW Statement

Deletes the view from the database.

SYNTAX

```
DROP VIEW [ owner_name.]view_name ;
```

owner_name

Specifies the owner of the view.

view_name

Names the view to drop.

EXAMPLE

```
DROP VIEW newcustomers ;
```

NOTES

- If *owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.
- When a view is dropped, other views that are dependent on this view are not dropped. The dependent views become invalid.

AUTHORIZATION

Must have DBA privilege or ownership of the view.

SQL COMPLIANCE

SQL-92, ODBC Core SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

CREATE VIEW Statement

EXEC SQL Delimiter

In C Language programs, you must precede embedded SQL-92 statements with the EXEC SQL delimiter so that the precompiler can distinguish statements from the host language statements.

NOTE: Constructs within a BEGIN-END DECLARE SECTION do not require the EXEC SQL delimiter.

SYNTAX

```
EXEC SQL sql_statement ;
```

sql_statement

An SQL-92 statement to be processed by the ESQLC precompiler. You must terminate each SQL-92 statement with a semicolon to mark the end of the statement.

EXAMPLE

This example is a code fragment from the `DynSel` function in sample program `4DynSel.pc`, which illustrates dynamic processing of a SELECT statement. The complete source for sample program, `2StatSel.pc`, is listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
EXEC SQL WHENEVER SQLERROR GOTO selerr ;
EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

NOTE

- In general, the ESQLC precompiler does not parse host language statements and therefore does not detect any syntax or semantic errors in host language statements. The exceptions to this rule are:
  - Recognition of host language blocks. The precompiler recognizes host language blocks in order to determine the scope of variables and types.
  - Constants defined with the #define preprocessor command. To evaluate these constants, the ESQLC precompiler invokes the C Language preprocessor before beginning embedded SQL-92 processing.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

None

EXECUTE Statement

Executes the statement specified in *statement_name*. This is the syntax for an EXECUTE statement:

SYNTAX

```
EXECUTE statement_name
  [ USING
    { [ SQL ] DESCRIPTOR structure_name
      | :host_variable [ [ INDICATOR ] :ind_variable ] , ... }
  ] ;
```

statement_name

Name of the prepared SQL-92 statement.

structure_name

Name of an SQL-92 descriptor area (SQLDA).

EXAMPLE

This example is a code fragment from the `DynUpd` function in sample program `3DynUpd.pc`, which illustrates dynamic processing of an UPDATE statement. The complete source for sample program, `2StatSel.pc`, is listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
/*
** Process the non-SELECT input statement
**    PREPARE the statement
**    EXECUTE the prepared statement
**    COMMIT WORK
*/

      EXEC SQL PREPARE dynstmt FROM :sql_stmt_v ;
      EXEC SQL EXECUTE dynstmt ;
      EXEC SQL COMMIT WORK ;
```

NOTES

- A statement must be processed with a PREPARE statement before it can be processed with an EXECUTE statement.
- A prepared statement can be executed multiple times in the same transaction. Typically each call to the EXECUTE statement supplies a different set of host variables.
- If there is no DESCRIPTOR in the USING clause, the EXECUTE statement is restricted to the number of variables specified in the host variable list. The number and type of the variables must be known at compile time. The host variables must be declared in the DECLARE SECTION before they can be used in the USING clause of the EXECUTE statement.
- If there is a DESCRIPTOR in the USING clause, the program can allocate space for the input host variables at runtime.

AUTHORIZATION

(See AUTHORIZATION for the relevant statement)

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

EXECUTE IMMEDIATE Statement, PREPARE Statement, SQLCA structure

EXECUTE IMMEDIATE Statement

Executes the statement specified in a *statement_string* or *host_variable*.

SYNTAX

```
EXECUTE IMMEDIATE { statement_string | host_variable } ;
```

statement_name

Name of the prepared SQL-92 statement.

structure_name

Name of an SQL-92 descriptor area (SQLDA).

NOTES

- The character string form of the statement is referred to as a *statement string*. An EXECUTE IMMEDIATE statement accepts either a statement string or a host variable as input.
- A statement string must not contain host variable references or parameter markers.
- A statement string must not begin with EXEC SQL Delimiter and must not end with a semicolon.
- When an EXECUTE IMMEDIATE statement is executed, the SQL engine parses the statement and checks it for errors. Any error in the execution of the statement is reported in the SQLCA.
- If the same SQL-92 statement is to be executed multiple times, it is more efficient to use PREPARE and EXECUTE statements, rather than an EXECUTE IMMEDIATE statement.

AUTHORIZATION

(See AUTHORIZATION for the relevant statement)

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

EXECUTE Statement, SQLCA structure

FETCH Statement

Moves the position of the cursor to the next row of the active set and fetches the column values of the current row into the specified host variables.

SYNTAX

```
FETCH cursor_name
  { USING SQL DESCRIPTOR structure_name
    | INTO :host_var_ref [ [ INDICATOR ] :ind_var_ref ] , ...
  } ;
```

cursor_name

A name identified in an earlier DECLARE CURSOR statement and an OPEN CURSOR statement.

USING SQL DESCRIPTOR *structure_name*

Directs the SQL engine to FETCH data into storage addressed by an SQLDA structure.

INTO :*host_var_ref* [[INDICATOR] :*ind_var_ref*]

Directs the SQL engine to FETCH data into the identified host variables, and to set values in the identified indicator variables.

EXAMPLE

The complete source for sample program `4DynSel.pc` is listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*. This example is a code fragment from the `dynsel` function in that sample program:

```
/*
**  One way to limit the number of rows returned is to
**  set a new value for "j" here. As supplied in the SPORTS200 database,
**  the PUB.InventoryTrans table contains 75 rows.
*/
      j = 100;
      for (i = 0; i < j; i++)
           {
           EXEC SQL FETCH dyncur INTO
             :int_p1_v, :int_p2_v, :char_p_v ;
           if (i == 0)
               {
               printf (" 1st col  2nd col  3rd col");
               printf (" -------  -------  --------");
               }
           printf (" %d  %d  %s ",
                   int_p1_v, int_p2_v, char_p_v) ;
           }
```

NOTES

- A FETCH operation requires that the cursor be open.
- The positioning of the cursor for each FETCH operation is as follows:
  - The first time you execute a FETCH statement after opening the cursor, the cursor is positioned to the first row of the active set.
  - Subsequent FETCH operations advance the cursor position in the active set. The next row becomes the current row.
  - When the current row is deleted using a positioned DELETE statement, the cursor is positioned before the row after the deleted row in the active set.
- The cursor can only be moved forward in the active set by executing FETCH statements. To move the cursor to the beginning of the active set, you must CLOSE the cursor and OPEN it again.

- If the cursor is positioned on the last row of the active set or if the active set does not contain any rows, executing a FETCH will return the status code SQL_NOT_FOUND in the SQLCA.
- After a successful FETCH, the total row count fetched so far for this cursor is returned in `sqlca.sqlerrd[2]`. The count is set to zero after an OPEN cursor operation.
- You can FETCH multiple rows in one FETCH operation by using array variables in the INTO clause. The SQL_NOT_FOUND status code is returned in the SQLCA when the end of the active set is reached, even if the current FETCH statement returns one or more rows.
- If you use array variables in a FETCH Statement, the array sizes are set to the number of rows fetched after the FETCH statement is executed.

AUTHORIZATION

None (See AUTHORIZATION for the OPEN Statement)

SQL COMPLIANCE

SQL-92; Progress Extension: USING DESCRIPTOR clause

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

DECLARE CURSOR Statement, OPEN Statement, CLOSE Statement

GET DIAGNOSTICS Statement

Retrieves information about the execution of the previous SQL statement from the SQL diagnostics area. The diagnostics area is a data structure that contains information about the execution status of the most recent SQL statement. Specifically, GET DIAGNOSTICS extracts information about the SQL statement as a whole from the SQL diagnostics area's header component.

NOTE: The GET DIAGNOSTICS EXCEPTION *number* extracts detail information.

SYNTAX

```
GET DIAGNOSTICS
  :param = header_info_item
    [ , :param = header_info_item ] , ... ;
```

:param

A host-language variable to receive the information returned by the GET DIAGNOSTICS statement. The host-language program must declare a *param* compatible with the SQL data type of the information item.

header_info_item

One of the following keywords, which returns associated information about the diagnostics area or the SQL statement:

SYNTAX

```
NUMBER | MORE | COMMAND_FUNCTION | DYNAMIC_FUNCTION | ROW_COUNT
```

NUMBER

The number of detail areas in the diagnostics area. Currently, NUMBER is always 1. NUMBER is type NUMERIC with a scale of 0.

MORE

A one-character string with a value of **Y** (all conditions are detailed in the diagnostics area) or **N** (all conditions are not detailed) that tells whether the diagnostics area contains information on all the conditions resulting from the statement.

COMMAND_FUNCTION

Contains the character-string code for the statement (as specified in the SQL-92 standard), if the statements was a static SQL statement. If the statement was a dynamic statement, contains either the character string 'EXECUTE' or 'EXECUTE IMMEDIATE'.

DYNAMIC_FUNCTION

Contains the character-string code for the statement (as specified in the SQL-92 standard). For dynamic SQL statements only (as indicated by 'EXECUTE' or 'EXECUTE IMMEDIATE' in the COMMAND_FUNCTION item).

ROW_COUNT

The number of rows affected by the SQL statement.

EXAMPLE

The GET DIAGNOSTICS example extracts header information about the last SQL statement executed. The information is assigned to host variables that are defined in the DECLARE SECTION of an embedded SQL program:

```
GET DIAGNOSTICS :num = NUMBER, :cmdfunc = COMMAND_FUNCTION ;
```

For information on defining and using host variables, see Chapter 7, "Query Statements," in the *Progress Embedded SQL-92 Guide and Reference*.

NOTE: The GET DIAGNOSTICS statement itself does not affect the contents of the diagnostics area. This means applications can issue multiple GET DIAGNOSTICS statements to retrieve different items of information about the same SQL statement.

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL

RELATED STATEMENTS

GET DIAGNOSTICS EXCEPTION Statement, WHENEVER Statement

GET DIAGNOSTICS EXCEPTION Statement

Retrieves information about the execution of the previous SQL statement from the SQL diagnostics area. The diagnostics area is a data structure that contains information about the execution status of the most recent SQL statement. Specifically, GET DIAGNOSTICS EXCEPTION extracts information about the SQL statement as a whole from the SQL diagnostics area's detail component.

The detail area contains information for a particular condition (an error, warning, or success condition) associated with execution of the last SQL statement. The diagnostics area can potentially contain multiple detail areas corresponding to multiple conditions generated by the SQL statement described by the header. The SQL diagnostics area currently supports only one detail area.

NOTE: The GET DIAGNOSTICS statement extracts header information.

SYNTAX

```
GET DIAGNOSTICS EXCEPTION number
  :param = detail_info_item
    [, :param = detail_info_item ] , ... ;
```

`EXCEPTION` *number*

Specifies that `GET DIAGNOSTICS EXCEPTION` extracts detail information. *number* specifies which of multiple detail areas `GET DIAGNOSTICS` extracts. Currently, *number* must be the integer 1.

:param

Receives the information returned by the `GET DIAGNOSTICS EXCEPTION` statement. The host-language program must declare a *param* compatible with the SQL data type of the information item.

detail_info_item

One of the following keywords, which returns associated information about the particular error condition:

SYNTAX

```
CONDITION_NUMBER
   | RETURNED_SQLSTATE
   | CLASS_ORIGIN
   | SUBCLASS_ORIGIN
   | ENVIRONMENT_NAME
   | CONNECTION_NAME
   | CONSTRAINT_CATALOG
   | CONSTRAINT_SCHEMA
   | CONSTRAINT_NAME
   | CATALOG_NAME
   | SCHEMA_NAME
   | TABLE_NAME
   | COLUMN_NAME
   | CURSOR_NAME
   | MESSAGE_TEXT
   | MESSAGE_LENGTH
   | MESSAGE_OCTET_LENGTH
```

`CONDITION_NUMBER`

The sequence of this detail area in the diagnostics area. Currently, `CONDITION_NUMBER` is always 1.

`RETURNED_SQLSTATE`

The `SQLSTATE` value that corresponds to the condition.

`CLASS_ORIGIN`

The general type of error. For example, 'connection exception,' or 'data exception.'

`SUBCLASS_ORIGIN`

The specific error. Usually the same as the message text.

ENVIRONMENT_NAME

Not currently supported.

CONNECTION_NAME

Not currently supported.

CONSTRAINT_CATALOG

Not currently supported.

CONSTRAINT_SCHEMA

Not currently supported.

CONSTRAINT_NAME

Not currently supported.

CATALOG_NAME

Not currently supported.

SCHEMA_NAME

Not currently supported.

TABLE_NAME

The name of the table, if the error condition involves a table.

COLUMN_NAME

The name of the affected columns, if the error condition involves a column.

CURSOR_NAME

Not currently supported.

MESSAGE_TEXT

The associated message text for the error condition.

MESSAGE_LENGTH

The length in characters of the message in the MESSAGE_LENGTH item.

MESSAGE_OCTET_LENGTH

Not currently supported.

EXAMPLE

THE GET DIAGNOSTICS EXCEPTION example extracts detail information into host variables that are defined in the DECLARE SECTION of an embedded SQL program:

```
GET DIAGNOSTICS EXCEPTION :num :sstate = RETURNED_SQLSTATE,
     :msgtxt = MESSAGE_TEXT ;
```

For information on defining and using host variables, see Chapter 7, "Query Statements," in the *Progress Embedded SQL-92 Guide and Reference*.

NOTE: The GET DIAGNOSTICS statement itself does not affect the contents of the diagnostics area. This means applications can issue multiple GET DIAGNOSTICS statements to retrieve different items of information about the same SQL statement.

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL

RELATED STATEMENTS

GET DIAGNOSTICS Statement, WHENEVER Statement

GRANT Statement

Grants various privileges to the specified users of the database. There are two forms of the GRANT statement:

- Grant database_wide privileges, either system administration (DBA) or general creation (RESOURCE), or both.
- Grant various privileges on specific tables and views. Privilege definitions are stored in the system tables SYSDBAUTH, SYSTABAUTH, and SYSCOLAUTH for the database, tables, and columns, respectively.

This is the syntax to GRANT database-wide privileges:

SYNTAX

```
GRANT { RESOURCE, DBA } TO username [ , username ] , ... ;
```

This is syntax to GRANT privileges on specific tables and views:

SYNTAX

```
GRANT { privilege [, privilege ] , ... | ALL [ PRIVILEGES] }
  ON table_name
  TO  { username [, username ] , ... |  PUBLIC }
  [ WITH GRANT OPTION ] ;
```

privilege:

SYNTAX

```
{ SELECT | INSERT | DELETE | INDEX
   | UPDATE [ ( column , column , ... ) ]
   | REFERENCES [ ( column , column , ... ) ] }
```

RESOURCE

Allows the specified users to issue CREATE statements.

DBA

Allows the specified users to create, access, modify, or delete any database object, and to grant other users any privileges.

TO username [, *username*] , ...

Grants the specified privileges on the table or view to the specified list of users.

SELECT

Allows the specified users to read data from the table or view.

INSERT

Allows the specified users to add new rows to the table or view.

DELETE

Allows the specified users to delete rows from the table or view.

INDEX

Allows the specified users to create an index on the table or view.

UPDATE [(*column* , *column* , ...)]

Allows the specified users to modify existing rows in the table or view. If followed by a column list, the users can modify values only in the columns named.

REFERENCES [(*column* , *column* , ...)]

Allows the specified users to refer to the table from other tables' constraint definitions. If followed by a column list, constraint definitions can refer only to the columns named. For more detail on constraint definitions, see the Column Constraints and Table Constraints section.

ALL

Grants all privileges for the table or view.

`TO PUBLIC`

Grants the specified privileges on the table or view to any user with access to the system.

`WITH GRANT OPTION`

Allows the specified users to grant their privileges or a subset of their privileges to other users.

EXAMPLE

```
GRANT DELETE ON cust_view TO dbuser1 ;
GRANT SELECT ON newcustomers TO dbuser2 ;
```

NOTE: If the *username* specified in a RESOURCE or DBA GRANT operation does not already exist, the GRANT statement creates a row in the SYSDBAUTH system table for the new *username*. This row is not deleted by a subsequent REVOKE operation.

AUTHORIZATION

Must have the DBA privilege, ownership of the table, or all the specified privileges on the table (granted with the WITH GRANT OPTION clause).

SQL COMPLIANCE

SQL-92, ODBC Core SQL grammar. Extensions: INDEX, RESOURCE, DBA privileges

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

REVOKE Statement

INSERT Statement

Inserts new rows into the specified table or view that will contain either the explicitly specified values or the values returned by the query expression.

SYNTAX

```
INSERT INTO [ owner_name.]{table_name|view_name }
  [ ( column_name [, column_name ] , ... ) ]
  { VALUES ( value [, value ] , ... ) | query_expression } ;
```

EXAMPLES

```
INSERT INTO customer (cust_no, name, street, city, state)
     VALUES
         (1001, 'RALPH', '#10 Columbia Street', 'New York', 'NY') ;

INSERT INTO neworders (order_no, product, qty)
     SELECT order_no, product, qty
          FROM orders
          WHERE order_date = SYSDATE ;
```

NOTES

- If the optional list of column names is specified, then only the values for those columns are required. The rest of the columns of the inserted row will contain NULL values, provided that the table definition allows NULL values and there is no DEFAULT clause for the columns. If a DEFAULT clause is specified for a column and the column name is not present in the optional column list, then the column is given the default value.
- If the optional list is not specified, then all the column values must be either explicitly specified or returned by the query expression. The order of the values should be the same as the order in which the columns are declared in the declaration of the table or view.
- The VALUES (**...**) form for specifying the column values inserts one row into the table. The query expression form inserts all the rows from the query results.
- If the table contains a foreign key and there is no corresponding primary key that matches the values of the foreign key in the record being inserted, the insert operation is rejected.

AUTHORIZATION

Must have DBA privilege, ownership of the table, INSERT privilege on the table, or SELECT privilege on all the tables or views referred to in the *query_expression*, if a query_expression is specified.

SQL COMPLIANCE

SQL-92, ODBC Core SQL grammar

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

"Query Expressions" in Chapter 2, "SQL-92 Language Elements"

LOCK TABLE Statement

Explicitly locks one or more specified tables for shared or exclusive access.

SYNTAX

```
LOCK TABLE table_name [ , table_name ] , ...
  IN { SHARE | EXCLUSIVE } MODE ;
```

table_name

The table in the database that you want to lock explicitly. You can specify one table or a comma-separated list of tables.

SHARE MODE

Allows all transactions to read the table(s). Prohibits all **other** transactions from modifying the table(s). After you acquire an explicit lock on a table in SHARE MODE, any SELECT statements in your transaction can read rows and **do not** implicitly acquire individual record locks. Any INSERT, UPDATE, and DELETE statements **do** acquire record locks.

EXCLUSIVE MODE

Allows the current transaction to read and modify the table(s), and prohibits any other transactions from reading or modifying the table(s). After you acquire an explicit lock on a table in EXCLUSIVE MODE, you can SELECT, INSERT, UPDATE, and DELETE rows and your transaction **does not** implicitly acquire individual record locks for these operations.

EXAMPLES

Unless another transaction holds an EXCLUSIVE lock on the teratab and megatab tables, the SHARE MODE example explicitly locks the tables. The shared lock allows all transactions to read the tables. Only the current transaction can modify the tables:

```
LOCK TABLE teratab, megatab IN SHARE MODE ;
```

Unless another transaction holds a lock on the teratab table, the EXCLUSIVE MODE example locks the teratab table for exclusive use by the current transaction. No other transactions can read or modify the teratab table:

```
LOCK TABLE teratab IN EXCLUSIVE MODE ;
```

Without a table lock, the first SELECT statement in this example could exceed the limits of the record lock table. The LOCK TABLE statement prevents the subsequent SELECT statement from consuming the record lock table:

```
-- Without a table lock, this SELECT statement creates an
-- entry in the record lock table for every row in teratab.

SELECT COUNT (*) FROM teratab ;

-- The LOCK TABLE IN SHARE MODE operation preserves the
-- record lock table resource.

LOCK TABLE teratab IN SHARE MODE ;
SELECT COUNT (*) FROM teratab ;
```

NOTES

- The LOCK TABLE statement may encounter a locking conflict with another transaction.
- The SHARE MODE option detects a locking conflict if another transaction:
  - Locked the table in EXCLUSIVE MODE and has not issued a COMMIT or ROLLBACK
  - Inserted, updated, or deleted rows in the table and has not issued a COMMIT or ROLLBACK
- The EXCLUSIVE MODE option detects a locking conflict if another transaction:
  - Locked the table in SHARE MODE or EXCLUSIVE MODE and has not issued a COMMIT or ROLLBACK
  - Read from, inserted, updated, or deleted rows and has not issued a COMMIT or ROLLBACK
- When there is a locking conflict, the transaction is suspended for up to five seconds until the requested lock can be acquired. If after five seconds the lock is still not available, the database returns an error.

- You can use explicit table locking to improve the performance of a single transaction, at the cost of decreasing the concurrency of the system and potentially blocking other transactions. It is more efficient to lock a table explicitly if you know that the transaction will be updating a substantial part of a table. You gain efficiency by decreasing the overhead of the implicit locking mechanism, and by decreasing any potential wait time for acquiring individual record locks on the table.

- You can use explicit table locking to minimize potential deadlocks in situations where a transaction is modifying a substantial part of a table. Before making a choice between explicit or implicit locking, compare the benefits of table locking with the disadvantages of losing concurrency.

- The database releases explicit and implicit locks only when the transaction ends with a COMMIT or ROLLBACK operation.

- See the SET TRANSACTION ISOLATION LEVEL Statement for information on isolation levels and the inconsistencies allowed by each:

- The isolation level in effect determines the record locking scheme.

  – READ UNCOMMITTED ensures that when a record is read, no record locks are acquired.

  – READ COMMITTED ensures that when a record is read a share lock is acquired on that record; the duration of the lock varies.

  – REPEATABLE READ ensures that when a record is read, a share lock is acquired on that record and held until the end of the current transaction.

  – SERIALIZABLE ensures that when a table is accessed the entire table is locked with a lock of appropriate strength; the lock is held until the end of the transaction.

- With READ COMMITTED, the intent is to hold the share lock on a record until the application reads the next record. This behavior is not always achieved.

  Under specific conditions, the server might release a share record lock before the record is returned to the client. This can lead to unreliable results when the application intends to update the rows being fetched. To prevent early release of the share lock when READ COMMITTED is in effect the following must be true:

  – A SELECT Statement references one table only.

  – The statement does not contain an ORDER BY Clause.

  - The statement does not contain a GROUP BY CLAUSE.
  - A WHERE Clause does not contain any subqueries.

  If an application requires a query that does not satisfy the conditions listed and the intent is to update the fetched rows, the application should set the isolation level to REPEATABLE READ.

- To achieve better performance, some client configurations prefetch records from the server. With isolation level READ COMMITTED, it is possible for the server to release a share lock on a record that an application is processing. When prefetching is in effect, a user application should not assume that the current row is locked. If a Progress C Language embedded SQL-92 application uses the array fetch feature, prefetching is enabled.
- If an application employs prefetching and has a requirement for updating the fetched rows, the application should set the transaction isolation level to REPEATABLE READ.
- As illustrated in the third example, some SELECT statements place an entry in the record lock table for every row in a table. For these queries, consider issuing a LOCK TABLE SHARE MODE before the SELECT statement.

AUTHORIZATION

Must have DBA privilege or SELECT privilege on the table.

SQL COMPLIANCE

Progress Extension

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

COMMIT Statement, ROLLBACK Statement, SET TRANSACTION ISOLATION LEVEL Statement

OPEN Statement

Executes a prepared SQL-92 query associated with a cursor and creates a result set composed of the rows that satisfy the query. This set of rows is called the *active set*.

SYNTAX

```
OPEN cursor_name
  [ USING { [ SQL ] DESCRIPTOR structure_name
    | :host_variable [ [ INDICATOR ] :ind_variable ] , ... } ] ;
```

cursor_name

An identifier named in an earlier DECLARE CURSOR statement.

USING [SQL] DESCRIPTOR *structure_name*

Directs the SQL engine to create the result set in storage addressed by the identified SQLDA structure.

USING :*host_variable* [[INDICATOR] :*ind_variable*]

Directs the SQL engine to create the result set in storage addressed by host variables.

EXAMPLE

This example is a code fragment from the `StatSel` function in sample program `2StatSel.pc`, which illustrates static processing of a SELECT statement. The complete source for sample program, `2StatSel.pc`, is listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
/*
**    5.  Name WHENEVER routine to handle SQLERROR.
**
**    6.  DECLARE cursor for the SELECT statement.
**        NOTE: You must set input parameter values before OPEN CURSOR.
**        The static query in this program does not have input parameters.
**
**    7.  OPEN the cursor.
**        NOTE: For static statements, if a DECLARE CURSOR
**              statement contains references to automatic variables,
**              the OPEN CURSOR statement must be in the same C function.
**
**    8.  Name WHENEVER routine to handle NOT FOUND condition.
*/

      EXEC SQL WHENEVER SQLERROR GOTO selerr ;
      EXEC SQL DECLARE stcur CURSOR FOR
            SELECT InvTransNum, Qty,
            OrderNum FROM PUB.InventoryTrans ;
      EXEC SQL OPEN stcur ;
      EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

NOTES

- Executing an OPEN cursor statement sets the cursor to the open state.
- After the OPEN cursor statement is executed, the cursor is positioned just before the first row of the active set.
- For a single execution of an OPEN cursor statement, the active set does not change and the host variables are not re_examined.
- If you elect to retrieve a new active set and a host variable value has changed, you must CLOSE the cursor and OPEN it again.
- Execution of a COMMIT Statement or ROLLBACK Statement implicitly closes the cursors which have been opened in the current transaction.
- It is good practice to CLOSE cursors explicitly.

- When a cursor is in the open state, executing an OPEN statement on that cursor results in an error.
- If a DECLARE cursor statement is associated with a static SQL-92 statement containing parameter markers, the following requirements apply:
  - You must execute the DECLARE statement before executing the OPEN statement for that cursor.
  - The DECLARE cursor statement and the OPEN statement for the same cursor must occur in the same transaction.
  - If the statement contains parameter markers for stack variables, the DECLARE cursor statement and the following OPEN statement for the same cursor must occur in the same C Language function.

AUTHORIZATION

Must have DBA privilege of SELECT privilege on all the tables and views referenced in the SELECT statement associated with the cursor.

SQL COMPLIANCE

SQL-92. Progress Extension: USING DESCRIPTOR clause

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

DECLARE CURSOR Statement, CLOSE Statement, FETCH Statement, positioned UPDATE Statement, positioned DELETE Statement

PREPARE Statement

Parses and assigns a name to an ad hoc or dynamically generated SQL-92 statement for execution. You use a PREPARE statement in a series of steps that allows a program to accept or generate SQL-92 statements at run time.

SYNTAX

```
PREPARE statement_name FROM statement_string ;
```

statement_name

A name for the dynamically generated statement. DESCRIBE, EXECUTE, and DECLARE CURSOR statements refer to this *statement_name*. A *statement_name* must be unique in a program.

statement_string

Specifies the SQL-92 statement to be prepared for dynamic execution. You can use either the name of a C Language string variable containing the SQL-92 statement, or you can specify the SQL-92 statement as a quoted literal. If there is an SQL-92 syntax error, the PREPARE statement returns an error in the SQLCA.

SYNTAX

```
{ :host_variable | quoted_literal }
```

EXAMPLES

The first example is a code fragment from the DynUpd function in sample program 3DynUpd.pc, which illustrates dynamic processing of an UPDATE statement:

```
/*
** Process a dynamic non-SELECT input statement
**    PREPARE the statement
**    EXECUTE the prepared statement
**    COMMIT WORK
*/

EXEC SQL PREPARE dynstmt FROM :sql_stmt_v ;
EXEC SQL EXECUTE dynstmt ;
EXEC SQL COMMIT WORK ;
```

This example is a code fragment from the DynSel function in sample program 4DynSel.pc, which illustrates dynamic processing of a SELECT statement:

```
/*
**    PREPARE a the dynamic SELECT statement.
**    DECLARE cursor for the prepared SELECT statement.
**    NOTE: You must set input parameter values before OPEN CURSOR.
**    If your query has input parameters, you must define them in
**    the DECLARE SECTION.
**    OPEN the declared cursor.
**    NOTE: For static statements, if a DECLARE CURSOR
**    statement contains references to automatic variables,
**    the OPEN CURSOR statement must be in the same C function.
**
**    Name WHENEVER routine for NOT FOUND condition.
**    FETCH a row and print results until no more rows.
*/

EXEC SQL PREPARE stmtid from :sel_stmt_v ;
EXEC SQL DECLARE dyncur CURSOR FOR stmtid ;
EXEC SQL OPEN dyncur ;
EXEC SQL WHENEVER NOT FOUND GOTO seldone ;
```

The complete source for four sample programs are listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

NOTES

- A statement string can have one or more references to input variables. These variables represent values supplied at run time to:
  - INSERT and UPDATE statements
  - Predicates in DELETE, UPDATE, and SELECT statements
- A program supplies an input variable to a PREPARE statement either as a *substitution name* or as a parameter marker.
  - A substitution name is a name preceded by a colon (:) in a statement string. This name does not refer to a C Language variable, but acts only as a placeholder for input variables.
  - A parameter marker is a question mark (?) in the statement string, serving as a placeholder for input variables.

- The USING clauses of EXECUTE and OPEN statements identify host language storage. The values in this storage expand a statement string, replacing a substitution name or a parameter marker. You can design your program to execute the same prepared statement many times in a transaction, supplying different values for input variables for each execution. If you COMMIT or ROLLBACK the transaction, you must PREPARE the statement string again.

AUTHORIZATION

Must have DBA privilege or Authorization for SQL-92 statement being prepared (See the AUTHORIZATION section for the relevant statement).

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

EXECUTE Statement, OPEN Statement, CLOSE Statement, FETCH Statement, and SQLCA structure

REVOKE Statement

Revokes various privileges from the specified users of the database. There are two forms of the REVOKE statement.

- Revoke database_wide privileges, either system administration (DBA), or general creation (RESOURCE), or both.
- Revoke various privileges on specific tables and views.

This is syntax to REVOKE system administration privileges (DBA) or general creation privileges (RESOURCE), or both:

SYNTAX

```
REVOKE { RESOURCE , DBA } FROM { username [ , username ] , ... } ;
```

RESOURCE

Revokes from the specified users the privilege to issue CREATE statements.

DBA

Revokes from the specified users the privilege to create, access, modify, or delete any database object, and revokes the privilege to grant other users any privileges.

FROM *username* [, *username*] , ...

Revokes the specified privileges on the table or view from the specified list of users.

This is the syntax to REVOKE privileges on specific tables and views:

SYNTAX

```
REVOKE [ GRANT OPTION FOR ]
   { privilege [, privilege ] , ...  | ALL [ PRIVILEGES ] }
   ON table_name
   FROM { username [ , username ] , ... | PUBLIC }
    [ RESTRICT | CASCADE ] ;
```

GRANT OPTION FOR

Revokes the GRANT option for the privilege from the specified users. The actual privilege itself is not revoked. If specified with RESTRICT, and the privilege is passed on to other users, the REVOKE statement fails and generates an error. Otherwise, GRANT OPTION FOR implicitly revokes any privilege the user might have given to other users.

privilege

SYNTAX

```
{ SELECT | INSERT | DELETE | INDEX
  | UPDATE [ ( column , column , ... ) ]
  | REFERENCES [ ( column , column , ... ) ] } ;
```

privilege [, *privilege*] , ... | ALL [PRIVILEGES]

List of privileges to be revoked. See the description in the GRANT statement. Revoking RESOURCE and DBA privileges can only be done by the administrator or a user with DBA privileges.

If more than one user grants access to the same table to a user, then all the grantors must perform a revoke for the user to lose access to the table.

Using the keyword ALL revokes all the privileges granted on the table or view.

FROM PUBLIC

Revokes the specified privileges on the table or view from any user with access to the system.

RESTRICT | CASCADE

Prompts SQL to check to see if the privilege being revoked was passed on to other users. This is possible only if the original privilege included the WITH GRANT OPTION clause. If so, the REVOKE statement fails and generates an error. If the privilege was not passed on, the REVOKE statement succeeds.

If the REVOKE statement specifies CASCADE, revoking the access privileges from a user also revokes the privileges from all users who received the privilege from that user.

If the REVOKE statement specifies neither RESTRICT nor CASCADE, the behavior is the same as for CASCADE.

EXAMPLE

```
REVOKE INSERT ON customer FROM dbuser1 ;
REVOKE DELETE ON cust_view FROM dbuser2 ;
```

NOTE: If the *username* specified in a GRANT DBA or GRANT RESOURCE operation does not already exist, the GRANT statement creates a row in the SYSDBAUTH system table for the new *username*. This row is not deleted by a subsequent REVOKE operation.

AUTHORIZATION

Must have the DBA privilege (to revoke DBA or RESOURCE privileges), ownership of the table (to revoke privileges on a table),

SQL COMPLIANCE

SQL-92, ODBC Core SQL grammar. Progress Extensions: INDEX, RESOURCE, DBA privileges

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

GRANT Statement

ROLLBACK Statement

Ends the current transaction and undoes any database changes performed during the transaction.

SYNTAX

```
ROLLBACK [ WORK ] ;
```

NOTES

- Under certain circumstances, SQL marks a transaction for abort but does not actually roll it back immediately. Without an explicit ROLLBACK, any subsequent updates do not take effect. A COMMIT statement causes SQL to recognize the transaction as marked for abort and instead implicitly rolls back the transaction.
- SQL marks a transaction for abort in the event of a hardware or software system failure. This transaction is rolled back during recovery.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL, interactive SQL

RELATED STATEMENTS

COMMIT Statement

SELECT Statement

Selects the specified column values from one or more rows contained in the tables or views specified in the query expression. The selection of rows is restricted by the WHERE clause. The temporary table derived through the clauses of a select statement is called a *result table*.

SYNTAX

```
SELECT column_list
  FROM table_list
    [ WHERE search_condition ]
    [ GROUP BY grouping_condition ]
    [ HAVING search_condition ]
    [ORDER BY ordering_condition ]
    [FOR UPDATE update_condition ]
;
```

column_list

See the COLUMN_LIST Clause.

FROM *table_list*

See the FROM Clause.

WHERE *search_condition*

See the WHERE Clause.

GROUP BY *grouping_condition*

See the GROUP BY CLAUSE.

HAVING *search_condition*

See the HAVING CLAUSE.

ORDER BY *ordering_condition*

See the ORDER BY Clause .

FOR UPDATE *update_condition*

See the FOR UPDATE Clause.

AUTHORIZATION

Must have DBA privilege or SELECT permission on all the tables or views referred to in the *query_expression*.

SQL COMPLIANCE

SQL-92. Progress Extensions: FOR UPDATE clause, ODBC Extended SQL grammar

ENVIRONMENT

Embedded SQL (within DECLARE), interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

"Query Expressions" in Chapter 2, "SQL-92 Language Elements", DECLARE CURSOR Statement, OPEN Statement, FETCH Statement, CLOSE Statement

COLUMN_LIST Clause

Specifies which columns to retrieve by the SELECT statement.

SYNTAX

```
[ ALL | DISTINCT ]
  { * | { table_name | alias.}* [ , { table_name.| alias.} * ] ...
     | expr [ [ AS ] [ ' ] column_title [ ' ] ]
       [, expr [ [ AS ] [ ' ] column_title [ ' ] ] ] ...
     |  [ table| alias.]column_name , ... ]
}
```

[ALL | DISTINCT]

Indicates whether a result table omits duplicate rows. ALL is the default and specifies that the result table includes all rows. DISTINCT specifies taht a table omits duplicate rows.

*** | { *table_name.* | *alias.* } ***

Specifies that the result table includes all columns from all tables named in the FROM clause.

*** *expr* [[AS] ['] *column_title* [']]**

Specifies a list of expressions, called a *select list*, whose results will form columns of the result table. Typically, the expression is a column name from a table named in the FROM clause. The expression can also be any supported mathematical expression, scalar function, or aggregate function that returns a value.

The optional '*column_title*' argument specifies a new heading for the associated column in the result table. You can also use the *column_title* in an ORDER BY clause. Enclose the new title in single or double quotation marks if it contains spaces or other special characters.

[*table* | *alias.*]*column_name* , ...]

Specifies a list columns from a particular table or alias.

EXAMPLES

Both these statement return all the columns in the customer table to the select list.

```
SELECT * FROM customer;

SELECT customer.* FROM customer;
```

The *table_name*.* syntax is useful when the select list refers to columns in multiple tables and you want to specify all the columns in one of those tables:

```
SELECT Customer.CustNum, Customer.Name, Invoice.*
     FROM Customer, Invoice ;
```

The following example illustrates using the *column_title* option to change the name of the column.

```
-- Illustrate optional 'column_title' syntax
select
        FirstName as 'First Name',
        LastName as 'Last Name',
        state as 'New England State'
from employee
        where state = 'NH' or state = 'ME' or state = 'MA'
        or state = 'VT' or state = 'CT' or state = 'RI';

First Name       Last Name        New England State
---------------- ---------------- ------------------
Justine          Smith            MA
Andy             Davis            MA
Marcy            Adams            MA
Larry            Dawsen           MA
John             Burton           NH
Mark             Hall             MA
Stacey           Smith            MA
Scott            Abbott           MA
Meredith         White            NH
Heather          White            NH
```

You **must** qualify a column name if it occurs in more than one table specified in the FROM clause:

```
select customer.customer_id from customer ;
```

```
-- Table name qualifier required
-- Customer table has city and state columns
-- Billto table has city and state columns

select
        Customer.custnum,
        Customer.city as "Customer City",
        Customer.State as 'Customer State',
        Billto.City as "Bill City",
        Billto.State as 'Bill State'
from Customer, Billto
        where Customer.City = 'Clinton';

 CustNum Customer City    Customer State   Bill City        Bill State
-------- ---------------- ---------------- ------------------------------
    1272 Clinton           MS               Montgomery       AL
    1272 Clinton           MS               Atlanta          GA
    1421 Clinton           SC               Montgomery       AL
    1421 Clinton           SC               Atlanta          GA
    1489 Clinton           OK               Montgomery       AL
    1489 Clinton           OK               Atlanta          GA
```

FROM Clause

Specifies one or more table references. Each table reference resolves to one table (either a table stored in the database or a virtual table resulting from processing the table reference) whose rows the query expression uses to create the result table.

SYNTAX

```
FROM table_ref [, table_ref ] ... [ { NO REORDER } ]
```

table_ref

There are three forms of table references:

- A direct reference to a table, view, or synonym.
- A derived table specified by a query expression in the FROM clause.
- A joined table that combines rows and columns from multiple tables.

If there are multiple table references, SQL joins the tables to form an intermediate result table that is used as the basis for evaluating all other clauses in the query expression. That intermediate result table is the *Cartesian product* of rows in the tables in the FROM clause, formed by concatenating every row of every table with all other rows in all tables.

SYNTAX

```
table_name [ AS ] [ alias [ ( column_alias [ ... ] ) ] ]
  | ( query_expression ) [ AS ] alias [ (column_alias [ ... ] ) ]
  | [ ( ] joined_table [ ) ]
```

```
FROM table_name [ AS ] [ alias  [ ( column_alias  [ ... ] ) ] ]
```

Explicitly names a table. The name can be a table name, a view name, or a synonym.

alias

A name used to qualify column names in other parts of the query expression. Aliases are also called *correlation names*.

If you specify an alias, you must use it, and not the table name, to qualify column names that refer to the table. Query expressions that join a table with itself must use aliases to distinguish between references to column names.

Similar to table aliases, the *column_alias* provides an alternative name to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in *table_name*. Also, if you specify column aliases in the FROM clause, you must use **them**, and not the column names, in references to the columns.

FROM (*query_expression*) **[** AS **]** **[** *alias* **[** (*column_alias* **[** **...** **]**) **]** **]**

Specifies a derived table through a query expression. With derived tables, you **must** specify an alias to identify the derived table.

Derived tables can also specify column aliases. Column aliases provide alternative names to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in the result table of the query expression. Also, if you specify column aliases in the FROM clause, you must use them, and not the column names, in references to the columns.

FROM **[** (**]** *joined_table* **[**) **]**

Combines data from two table references by specifying a join condition.

SYNTAX

```
{ table_ref CROSS JOIN table_ref
  | table_ref [ INNER | LEFT [ OUTER ] ] JOIN
    table_ref ON search_condition
}
```

The syntax currently allowed in the FROM clause supports only a subset of possible join conditions:

- CROSS JOIN specifies a Cartesian product of rows in the two tables. Every row in one table is joined to every row in the other table.
- INNER JOIN specifies an inner join using the supplied search condition.
- LEFT OUTER JOIN specifies a left outer join using the supplied search condition.

You can also specify these and other join conditions in the WHERE clause of a query expression. See the "Outer Joins" section in Chapter 2, "SQL-92 Language Elements" for more information on both ways of specifying outer joins.

{ NO REORDER }

Disables join order optimization for the FROM clause. Use NO REORDER when you choose to override the join order chosen by the optimizer. The braces are part of the syntax for this optional clause.

EXAMPLE

For customers with orders, retrieve their names and order info:

```
SELECT customer.cust_no, customer.name, orders.order_no,orders.order_date
     FROM customers, orders
     WHERE customer.cust_no = orders.cust_no ;
```

WHERE Clause

Specifies a *search_condition* that applies conditions to restrict the number of rows in the result table. If the query expression does not specify a WHERE clause, the result table includes all the rows of the specified table reference in the FROM clause.

SYNTAX

```
WHERE search_condition
```

search_condition

Applied to each row of the result table set of the FROM clause. Only rows that satisfy the conditions become part of the result table. If the result of the *search_condition* is NULL for a row, the row is not selected.

Search conditions can specify different conditions for joining two or more tables. See the "Outer Joins" and "Search Conditions" sections in Chapter 2, "SQL-92 Language Elements" for information on the different kinds of search conditions.

GROUP BY CLAUSE

Specifies grouping of rows in the result table.

SYNTAX

```
GROUP BY [ table_name.]column_name  ...
```

NOTES

- For the first column specified in the GROUP BY clause, SQL arranges rows of the result table into groups whose rows all have the same values for the specified column.
- If you specify a second GROUP BY column, SQL groups rows in each main group by values of the second column.
- SQL groups rows for values in additional GROUP BY columns in a similar fashion.
- All columns named in the GROUP BY clause must also be in the select list of the query expression. Conversely, columns in the select list must also be in the GROUP BY clause or be part of an aggregate function.

HAVING CLAUSE

Allows you to set conditions on the groups returned by the GROUP BY clause. If the HAVING clause is used without the GROUP BY clause, the implicit group against which the search condition is evaluated is all the rows returned by the WHERE clause.

SYNTAX

```
HAVING search_condition
```

NOTE: A condition of the HAVING clause can compare one aggregate function value with another aggregate function value or a constant.

EXAMPLE

The HAVING clause in the following example compares the value of an aggregate function (COUNT (*)) to a constant (10). The query returns the customer number and number of orders for all customers who had more than 10 orders before March 31st, 1999.

```
SELECT cust_no, count(*)
     FROM orders
     WHERE order_date < TO_DATE ('3/31/1999')
     GROUP BY cust_no
     HAVING COUNT (*) > 10 ;
```

ORDER BY Clause

Allows ordering of the rows selected by the SELECT statement. Unless an ORDER BY clause is specified, the rows of the result set might be returned in an unpredictable order as determined by the access paths chosen and other decisions made by the query optimizer. The decisions made will be affected by the statistics generated from table and index data examined by the UPDATE STATISTICS command.

SYNTAX

```
ORDER BY { expr | posn } [ ASC | DESC ]
  [ , { expr | posn } [ ASC | DESC ] , ... ]
```

expr

Expression of one or more columns of the tables specified in the FROM clause of the SELECT statement.

posn

Integer column position of the columns selected by the SELECT statement.

ASC | DESC

Indicates whether to order by ascending order (ASC) or descending order. The default is ASC.

EXAMPLES

```
-- Produce a list of customers sorted by last_name.
SELECT last_name, street, city, state, zip
     FROM customer
     ORDER BY last_name ;

-- Produce a merged list of customers and suppliers sorted by last_name.
SELECT last_name, street, state, zip
     FROM customer
     UNION
     SELECT last_name, street, state, zip
     FROM supplier
     ORDER BY 1 ;
```

NOTES

- The ORDER BY clause, if specified, should follow all other clauses of the SELECT statement.
- The selected rows are ordered on the basis of the first *expr* or *posn*. If the values are the same, then the second *expr* or *posn* is used in the ordering.
- A query expression can be followed by an optional ORDER BY clause. If the query expression contains set operators, then the ORDER BY clause can specify only the positions.

FOR UPDATE Clause

Specifies update intention on the rows selected by the SELECT statement.

SYNTAX

```
FOR UPDATE [ OF [ table.]column_name , ... ] [ NOWAIT ]
```

OF [*table*.]*column_name* , ...

Specifies the table's column name to be updated.

NOWAIT

Causes the SELECT statement to return immediately with an error if a lock cannot be acquired on a row in the selection set because of the lock held by some other transaction. The default behavior is for the transaction to wait until it gets the required lock or until it times out waiting for the lock.

NOTE: If you specify FOR UPDATE, the database acquires exclusive locks on all the rows satisfying the SELECT statement. The database does **not** acquire row level locks if there is an exclusive lock on the table. See the LOCK TABLE Statement for information on table locking.

SET CONNECTION Statement

Switches the application from one established connection to another. This resumes the connection associated with the specified *connection_name*, restoring the context of that database connection to the same state it was in when suspended.

SYNTAX

```
SET CONNECTION { 'connection_name' | DEFAULT } ;
```

connection_name

The name of the connection as either a character literal or host variable. If the SET CONNECTION statement omits a connection name, the default is the name of the database. Connection names must be unique.

DEFAULT

Sets the DEFAULT connection as the current connection.

EXAMPLES

The first example shows how to establish a database as the current database. The SET CONNECTION command sets the database associated with the connection named conn_1 to the status of current database. The connection named conn_1 must be associated with an established connection:

```
EXEC SQL
     SET CONNECTION 'conn_1' ;
```

Use SET CONNECTION DEFAULT to **set current** the database associated with the DEFAULT connection. This statement suspends the conn_1 connection, which had been current.

```
EXEC SQL
     SET CONNECTION DEFAULT ;
```

See also the last example for the DISCONNECT Statement, which illustrates the CONNECT, SET CONNECTION, and DISCONNECT statements in combination.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL only

RELATED STATEMENTS

CONNECT Statement, DISCONNECT Statement

SET SCHEMA Statement

Sets the default owner, also known as schema, for unqualified table references.

SYNTAX

```
SET SCHEMA { 'string_literal' | ? | :host_var | USER }
```

'string_literal'

Specifies the name for the default owner as a string literal, enclosed in single or double quotes.

?

Indicates a parameter marker to contain the default owner. The actual replacement value for the owner name is supplied in a subsequent SQL-92 operation.

:host_var

Host variable reference declared in a DECLARE SECTION. The SET SCHEMA *:host_var* option is valid only in an embedded SQL-92 program.

USER

Directs the database to set the default owner back to the *username* that established the session.

EXAMPLE

This example sets the default schema name to 'White'. Subsequent SQL-92 statements with unqualified table references will use the owner name 'White'. The SELECT statement in this example returns all rows in the 'White.customer' table. The *username* establishing the original session is still the current user.

```
SET SCHEMA 'White' ;
COMMIT ;

SELECT * from customer ;
```

NOTES

- For authorization purposes, invoking SET SCHEMA does **not** change the *username* associated with the current session.
- You can set the default schema name to the *username* associated with the session by using a SET SCHEMA USER statement.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

None

SET TRANSACTION ISOLATION LEVEL Statement

Explicitly sets the isolation level for a transaction. Isolation levels specify the degree to which one transaction can modify data or database objects in use by another concurrent transaction.

SYNTAX

```
SET TRANSACTION ISOLATION LEVEL isolation_level_name ;
```

isolation_level_name:

SYNTAX

```
READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE
```

`READ UNCOMMITTED`

Allows dirty reads, nonrepeatable reads, and phantoms. When a record is read, no record locks are aquired. This forces read-only use. Allows a user application to read records that were modified by other applications but have not yet been committed.

`READ COMMITTED`

Prohibits dirty reads; allows nonrepeatable reads and phantoms. Whenever a record is read, a share lock is acquired on that record. The duration of the lock varies. Disallows the reading of uncommitted modified records. However, if a record is read multiple times in the same transaction, the state of the record can be different each time.

`REPEATABLE READ`

Prohibits dirty reads and nonrepeatable reads; allows phantoms. Whenever a record is read, a share lock is acquired on that record and held until the end of the current transaction. Disallows the reading of uncommitted modified records. If a record is read multiple times in the same transaction, the state of the record remains the same.

`REPEATABLE READ` is the default isolation level.

`SERIALIZABLE`

Prohibits dirty reads, nonrepeatable reads, and phantoms. If an application executes the same `SELECT` statement more than once within the same transaction, the same set of rows is retrieved every time. Guarantees that concurrent transactions will not affect each other, and that they will behave as if they were executing serially, not concurrently.

Whenever a table is accessed, the entire table is locked with an appropriate lock. The table lock is held until the end of the current transaction.

NOTES

- Progress Software recommends that you specify the transaction isolation level number by *isolation_level_name*.
- See the LOCK TABLE Statement for information on record locking schemes used by each isolation level.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92. The semantics to which it corresponds are standard.

The isolation level SERIALIZABLE guarantees the highest consistency. The isolation level READ UNCOMMITTED guarantees the least consistency. The default isolation level is REPEATABLE READ, which prohibits non-repeatable read operations. The ANSI/ISO SQL standard defines isolation levels in terms of the inconsistencies they allow:

Dirty read

Allows the transaction to read a row that has been inserted or modified by another transaction, but not committed. If the other transaction rolls back its changes, the transaction will read a row that never existed because it never committed.

Nonrepeatable read

Allows the transaction to read a row that another transaction modifies or deletes before the next read operation. If the other transaction commits the change, the transaction receives modified values or discovers the row is deleted on subsequent read operations.

Phantom

Allows the transaction to read a range of rows that satisfies a given search condition, but to which another transaction adds rows before another read operation using the same search condition. The transaction receives a different collection of rows with the same search condition.

ENVIRONMENT

Embedded SQL only, interactive SQL

RELATED STATEMENTS

COMMIT Statement, LOCK TABLE Statement, ROLLBACK Statement

Table Constraints

Specifies a constraint for a table that restricts the values that the table can store. INSERT, UPDATE, or DELETE statements that violate the constraint fail. SQL returns a *constraint violation* error.

Table constraints have syntax and behavior similar to Column Constraints. Note the following differences:

- The definitions of the table constraints are separated from the column definitions by commas.
- Table constraint definitions can include more than one column, and SQL evaluates the constraint based on the combination of values stored in all the columns.

SYNTAX

```
CONSTRAINT constraint_name
    PRIMARY KEY ( column [ , ... ] )
  | UNIQUE ( column [ , ... ] )
  | FOREIGN KEY ( column [, ... ] )
      REFERENCES [ owner_name.]table_name [ ( column [ , ... ] ) ]
  | CHECK ( search_condition )
```

`CONSTRAINT` *constraint_name*

Allows you to assign a name that you choose to the table constraint. While this specification is optional, this facilitates making changes to the table definition, since the name you specify is in your source `CREATE TABLE` statement. If you do not specify a *constraint_name*, the database assigns a name. These names can be long and unwieldy, and you must query system tables to determine the name.

`PRIMARY KEY (` *column* `[ , ... ] )`

Defines the column list as the primary key for the table. There can be at most one primary key for a table.

All the columns that make up a table level primary key must be defined as `NOT NULL`, or the `CREATE TABLE` statement fails. The combination of values in the columns that make up the primary key must be unique for each row in the table.

Other tables can name primary keys in their REFERENCES clauses. If they do, SQL restricts operations on the table containing the primary key in the following ways:

- DROP TABLE statements that delete the table fail.
- DELETE and UPDATE statements that modify values in the combination of columns that match a foreign key's value also fail.

UNIQUE (*column* [, ...])

Defines the column list as a unique, or candidate, key for the table. Unique key table-level constraints have the same rules as primary key table-level constraints, except that you can specify more than one UNIQUE table-level constraint in a table definition.

FOREIGN KEY (*column* [, ...]) REFERENCES [*owner_name.*] *table_name* [(*column* [, ...])]

Defines the first column list as as a foreign key and, in the REFERENCES clause, specifies a matching primary or unique key in another table.

A foreign key and its matching primary or unique key specify a *referential constraint*. The combination of values stored in the columns that make up a foreign key must either:

- Have at least one of the column values be null
- Be equal to some corresponding combination of values in the matching unique or primary key

You can omit the column list in the REFERENCES clause if the table specified in the REFERENCES clause has a primary key and you want the primary key to be the matching key for the constraint.

CHECK (*search_condition*)

Specifies a table level check constraint. The syntax for table level and column level check constraints is identical. Table level check constraints must be separated by commas from surrounding column definitions.

SQL restricts the form of the search condition. The search condition must not:

- Refer to any column other than columns that precede it in the table definition.
- Contain aggregate functions, subqueries, or parameter references.

EXAMPLES

The following example shows creation of a table level primary key. Note that its definition is separated from the column definitions by a comma:

```
CREATE TABLE supplier_item (
     supp_no   INTEGER NOT NULL,
     item_no   INTEGER NOT NULL,
     qty       INTEGER NOT NULL DEFAULT 0,
          PRIMARY KEY (supp_no, item_no)
) ;
```

The following example shows how to create a table with two `UNIQUE` table level constraints:

```
CREATE TABLE order_item (
     order_no    INTEGER NOT NULL,
     item_no     INTEGER NOT NULL,
     qty         INTEGER NOT NULL,
     price       MONEY NOT NULL,
          UNIQUE (order_no, item_no),
          UNIQUE (qty, price)
) ;
```

The following example defines the combination of columns student_courses.teacher and student_courses.course_title as a foreign key that references the primary key of the courses table. Note that this REFERENCES clause does not specify column names because the foreign key refers to the primary key of the courses table:

```
CREATE TABLE courses (
     teacher        CHAR (20) NOT NULL,
     course_title   CHAR (30) NOT NULL,
     PRIMARY KEY (teacher, course_title)
) ;
CREATE TABLE student_courses (
     student_id     INTEGER,
     teacher        CHAR (20),
     course_title   CHAR (30),
     FOREIGN KEY (teacher, course_title) REFERENCES courses
) ;
```

SQL evaluates the referential constraint to see if it satisfies the following search condition:

```
(student_courses.teacher IS NULL
     OR student_courses.course_title IS NULL)
     OR EXISTS (SELECT * FROM student_courses WHERE
          (student_courses.teacher = courses.teacher AND
          student_courses.course_title = courses.course_title)
          )
```

NOTE: INSERT, UPDATE, or DELETE statements that cause the search condition to be false violate the constraint, fail, and generate an error.

The following example creates a table with two column level check constraints and one table level check constraint. Each constraint is defined with a name.

```
CREATE TABLE supplier (
     supp_no   INTEGER NOT NULL,
     name      CHAR (30),
     status    SMALLINT CONSTRAINT status_check_con
                   CHECK ( supplier.status BETWEEN 1 AND 100 ),
     city      CHAR (20) CONSTRAINT city_check_con CHECK
                    ( supplier.city IN ('NEW YORK', 'BOSTON', 'CHICAGO')),
     CONSTRAINT supp_tab_check_con CHECK (supplier.city <> 'CHICAGO'
          OR supplier.status = 20)
) ;
```

UPDATE Statement

Updates the rows and columns of the specified table with the given values for rows that satisfy the *search_condition*. This is the syntax for the UPDATE statement:

SYNTAX

```
UPDATE table_name
  SET assignment [, assignment ] , ...
  [ WHERE search_condition ] ;
```

assignment:

SYNTAX

```
column = { expr | NULL }
  | ( column [, column ] , ... ) = ( expr [, expr ] )
  | ( column [, column ] , ... ) = ( query_expression )
```

NOTES

- If you specify the optional WHERE clause, only rows that satisfy the *search_condition* are updated. If you do not specify a WHERE clause all rows of the table are updated.
- If the expressions in the SET clause are dependent on the columns of the target table, the expressions are evaluated for each row of the table.
- If a query expression is specified on the right_hand side of an assignment, the number of expressions in the first SELECT clause of the query expression must be the same as the number of columns listed on the left_hand side of the assignment.
- If a query expression is specified on the right_hand side of an assignment, the query expression must return one row.
- If a table has check constraints and if the columns to be updated are part of a check expression, then the check expression is evaluated. If the result of the evaluation is FALSE, the UPDATE statement fails.
- If a table has primary or candidate keys and if the columns to be updated are part of the primary or candidate key, SQL checks to determine if there is a corresponding row in the referencing table. If there is a corresponding row the UPDATE operation fails.

EXAMPLES

```
UPDATE orders
          SET qty = 12000
          WHERE order_no = 1001 ;


UPDATE orders
          SET (product) =
                    (SELECT item_name
                     FROM items
                     WHERE item_no = 2401 )
          WHERE order_no = 1002 ;


UPDATE orders
          SET (amount) = (2000 * 30)
          WHERE order_no = 1004 ;


UPDATE orders
          SET (product, amount) =
                              (SELECT item_name, price * 30
                               FROM items
                               WHERE item_no = 2401 )
          WHERE order_no = 1002 ;
```

AUTHORIZATION

Must have DBA privilege or UPDATE privileges on all the specified columns of the target table, and SELECT privilege on all the other tables referred to in the statement.

SQL COMPLIANCE

SQL-92. ODBC Extended SQL grammar. Progress Extensions: assignments of the form (*column* , *column* , **...**) = (*expr* , *expr* , **...**)

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

SELECT Statement, OPEN Statement, FETCH Statement, "Search Conditions" and "Query Expressions" in Chapter 2, "SQL-92 Language Elements"

UPDATE STATISTICS Statement

Queries data tables and updates the following table and column statistics:

- The number of rows in the table (the cardinality)
- The value ranges corresponding to 10% of the rows of a table, for each indexed column. The UPDATE STATISTICS operation produces a 10-step histogram of the column values.

SYNTAX

```
UPDATE STATISTICS [ FOR table_name ] ;
```

table_name

Specifies a single table on which to update statistics. If you do not specify a *table_name*, the UPDATE STATISTICS statement updates statistics on all tables in the database.

NOTES

- The time required to gather the statistics is highly dependent on the number of tables specified and on the number of rows in those tables.
- The optimizer uses the information from UPDATE STATISTICS to make decisions about the best query strategy to use when executing a particular SQL-92 statement.
- Until a user, application, or SQL-92 script issues an UPDATE STATISTICS statement, the optimizer bases query strategies on values it derives from various defaults. These values might not lead to the best performance, so database administrators should issue an UPDATE STATISTICS operation periodically.

AUTHORIZATION

Must have DBA privilege, SELECT privilege, or ownership of table (to issue UPDATE STATISTICS statement for a specific table).

SQL COMPLIANCE

Progress Extension

ENVIRONMENT

Embedded SQL, interactive SQL, ODBC applications, JDBC applications

RELATED STATEMENTS

None

WHENEVER Statement

Specifies actions for three SQL-92 run-time exceptions. This is the syntax for a WHENEVER statement:

SYNTAX

```
WHENEVER
  { NOT FOUND | SQLERROR | SQLWARNING }
  { STOP | CONTINUE | { GOTO | GO TO } host_lang_label } ;
```

{ NOT FOUND | SQLERROR | SQLWARNING }

- The NOT FOUND exception is set when sqlca.sqlcode is set to SQL_NOT_FOUND.
- The SQLERROR exception is set when sqlca.sqlcode is set to a negative value.
- The SQLWARNING exception is set when sqlca.sqlwarn[0] is set to 'W' after a statement is executed.

{ STOP | CONTINUE | GOTO | GO TO } *host_lang_label* }

- The STOP exception results in the ESQLC program stopping execution.
- The CONTINUE exception results in the ESQLC program continuing execution. The default exception is to CONTINUE.
- GOTO | GO *host_lang_label* results in the ESQLC program execution to branch to the statement corresponding to the *host_lang_label*.

EXAMPLES

The first example is a code fragment from the main function in sample program 1StatUpd.pc. The complete source for sample program, 1StatUpd.pc, is listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
/*
**  Name WHENEVER routine to handle SQLERROR condition.
*/
EXEC SQL WHENEVER SQLERROR GOTO mainerr ;
```

This example is a code fragment from the dynupd function in sample program 3DynUpd.pc, which illustrates dynamic processing of an UPDATE statement. The complete source for sample program, 3DynUpd.pc, is listed in Appendix A of the *Progress Embedded SQL-92 Guide and Reference*.

```
/*
**    Name WHENEVER routines to handle NOT FOUND and SQLERROR
*/

EXEC SQL WHENEVER SQLERROR GOTO nodyn ;
EXEC SQL WHENEVER NOT FOUND GOTO nodyn ;
```

NOTES

- You can place multiple WHENEVER statements for the same exception in a source file. Each WHENEVER statement overrides the previous WHENEVER statement specified for the same exception.
- Correct operation of a WHENEVER statement with a GOTO *host_language_label* or a GO TO *host_language_label* is subject to the scoping rules of the C Language. The *host_language_label* must be within the scope of all SQL-92 statements for which the action is active. The GO TO or GOTO action is active starting from the corresponding WHENEVER statement until another WHENEVER statement for the same exception, or until end of file.

AUTHORIZATION

None

SQL COMPLIANCE

SQL-92. Progress Extensions: SQLWARNING exception condition and STOP action.

ENVIRONMENT

Embedded SQL-92 only

RELATED STATEMENTS

FETCH Statement

4

SQL-92 Functions

A *function* is a type of SQL expression that returns a value based on the arguments supplied to the function. Progress SQL-92 supports two types of functions:

- **Aggregate Functions** — Calculate a single value for a collection of rows in a result table. If the function is in a statement with a GROUP BY clause, it returns a value for each group in the result table. Aggregate functions are also called set or statistical functions. Aggregate functions cannot be nested. This chapter describes the Aggregate Functions listed in Table 4–1.

- **Scalar Functions** — Calculate a value based on another single value. Scalar functions are also called value functions. Scalar functions can be nested. This chapter describes the Scalar Functions listed in Table 4–2.

SQL-92 Functions

Table 4–1: Progress SQL-92 Aggregate Functions

AVG Function	COUNT Function	MAX Function
MIN Function	SUM Function	–

Table 4–2: Progress SQL-92 Scalar Functions *(1 of 2)*

ABS Function	ACOS Function	ADD_MONTHS Function
ASCII Function	ASIN Function	ATAN Function
ATAN2 Function	CASE Function	CAST Function
CEILING Function	CHAR Function	CHR Function
COALESCE Function	CONCAT Function	CONVERT Function (ODBC Compatible)
CONVERT Function (Progress Extension)	COS Function	CURDATE Function
CURTIME Function	DATABASE Function	DAYNAME Function
DAYOFMONTH Function	DAYOFWEEK Function	DAYOFYEAR Function
DB_NAME Function	DECODE Function	DEGREES Function
EXP Function	FLOOR Function	GREATEST Function
HOUR FUNCTION	IFNULL Function	INITCAP Function
INSERT Function	INSTR Function	LAST_DAY Function
LCASE Function	LEAST Function	LEFT Function
LENGTH Function	LOCATE Function	LOG10 Function
LOWER Function	LPAD Function	LTRIM Function
MINUTE Function	MOD Function	MONTH Function

Table 4–2: Progress SQL-92 Scalar Functions *(2 of 2)*

MONTHNAME Function	MONTHS_BETWEEN Function	NEXT_DAY Function
NOW Function	NULLIF Function	NVL Function
PI Function	POWER Function	PREFIX Function
PRO_* Functions	QUARTER Function	RADIANS Function
RAND Function	REPEAT Function	REPLACE Function
RIGHT Function	ROUND Function	ROWID Function
RPAD Function	RTRIM Function	SECOND Function
SIGN Function	SIN Function	SQRT Function
SUBSTR Function	SUBSTRING Function (ODBC Compatible)	SUFFIX Function
SYSDATE Function	SYSTIME Function	SYSTIMESTAMP Function
TAN Function	TO_CHAR Function	TO_DATE
TO_NUMBER Function	TO_TIME Function	TO_TIMESTAMP Function
TRANSLATE Function	UCASE Function	UPPER Function
USER Function	WEEK Function	YEAR Function

AVG Function

Computes the average of a collection of values. The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the average.

SYNTAX

```
AVG ( { [ ALL ] expression } | { DISTINCT column_ref } )
```

EXAMPLE

This example illustrates the AVG function:

```
SELECT AVG (salary)
     FROM employee
     WHERE deptno = 20 ;
```

NOTES

- Null values are eliminated before the average value is computed. If all the values are null, the result is null.
- The argument to the function must be of type SMALLINT, INTEGER, NUMERIC, REAL, or FLOAT.
- The result is of type NUMERIC.

COUNT Function

Computes either the number of rows in a group of rows or the number of non-null values in a group of values.

SYNTAX

```
COUNT ( { [ ALL ] expression } | { DISTINCT column_ref } | * )
```

EXAMPLE

This example illustrates the COUNT function:

```
SELECT COUNT (*)
     FROM orders
     WHERE order_date = SYSDATE ;
```

NOTES

- The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the count.
- If the argument to COUNT function is '*', then the function computes the count of the number of rows in a group.
- If the argument to COUNT function is not '*', then null values are eliminated before the number of rows is computed.
- The argument *column_ref* or *expression* can be of any type.
- The result of the function is of integer type. The result is never null.

MAX Function

Returns the maximum value in a group of values.

SYNTAX

```
COUNT ( { [ ALL ] expression } | { DISTINCT column_ref } | * )
```

EXAMPLE

This example illustrates the MAX function:

```
SELECT order_date, product, MAX (qty)
     FROM orders
     GROUP BY order_date, product ;
```

NOTES

- Specifying DISTINCT has no effect on the result.
- The argument *column_ref* or *expression* can be of any type.
- The result of the function is of the same data type as that of the argument.
- The result is null if the result set is empty or contains only null values.

MIN Function

Returns the minimum value in a group of values.

SYNTAX

```
MIN ( { [ ALL ] expression } | { DISTINCT column_ref } )
```

EXAMPLE

This example illustrates the MIN function:

```
SELECT MIN (salary)
     FROM employee
     WHERE deptno = 20 ;
```

NOTES

- Specifying DISTINCT has no effect on the result.
- The argument *column_ref* or *expression* can be of any type.
- The result of the function is of the same data type as that of the argument.
- The result is null if the result set is empty or contains only null values.

SUM Function

Returns the sum of the values in a group. The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the sum.

SYNTAX

```
SUM ( { [ALL] expression } | { DISTINCT column_ref } )
```

EXAMPLE

This example illustrates the SUM function:

```
SELECT SUM (amount)
     FROM orders
     WHERE order_date = SYSDATE ;
```

NOTES

- The argument *column_ref* or *expression* can be of any type.
- The result of the function is of the same data type as that of the argument except that the result is of type INTEGER when the argument is of type `SMALLINT` or `TINYINT`.
- The result can have a null value.

ABS Function

Computes the absolute value of *expression*.

SYNTAX

```
ABS ( expression )
```

EXAMPLE

This example illustrates the ABS function:

```
SELECT ABS (MONTHS_BETWEEN (SYSDATE, order_date))
    FROM orders
    WHERE  ABS (MONTHS_BETWEEN (SYSDATE, order_date)) > 3 ;
```

NOTES

- The argument to the function must be of type TINYINT, SMALLINT, INTEGER, NUMERIC, REAL, or FLOAT.
- The result is of type NUMERIC.
- If the argument *expression* evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

ACOS Function

Returns the arccosine of *expression*.

SYNTAX

```
ACOS ( expression )
```

EXAMPLE

This example illustrates two ways to use the acos function. The first SELECT statement returns the arcosine in radians, and the second returns the arcosine in degrees.

```
select acos (.5) 'Arccosine in radians' from syscalctable;

ARCCOSINE IN RADIANS
--------------------
1.047197551196598

1 record selected


select acos (.5) * (180/ pi()) 'Arccosine in degrees' from syscalctable;

ARCCOSINE IN DEGREES
--------------------
59.999999999999993
1 record selected
```

NOTES

- ACOS takes the ratio (*expression*) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.
- The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.
- The expression must be in the range -1 to 1.
- The expression must evaluate to an approximate numeric data type.

COMPATIBILITY

ODBC Compatible

ADD_MONTHS Function

Adds to the date value specified by the *date_expression*, the given number of months specified by *integer_expression*, and returns the resultant date value.

SYNTAX

```
ADD_MONTHS ( date_expression , integer_expression )
```

EXAMPLE

This example illustrates the ADD_MONTHS function:

```
SELECT *
    FROM  customer
    WHERE ADD_MONTHS (start_date, 6) > SYSDATE ;
```

NOTES

- The first argument must be of DATE type.
- The second argument to the function must be of NUMERIC type.
- The result is of type DATE.
- If any of the arguments evaluates to null, the result is null.

COMPATIBILITY

Progress Extension

ASCII Function

Returns the ASCII value of the first character of the given character expression.

SYNTAX

```
ASCII ( char_expression )
```

EXAMPLE

The following example shows how to use the ASCII function:

```
SELECT ASCII ( zip )
     FROM customer ;
```

NOTES

- The argument to the function must be of type CHARACTER.
- The result is of type INTEGER.
- If the argument *char_expression* evaluates to null, the result is null.
- The ASCII function is character-set dependent, and supports multi-byte characters. The function returns the character encoding integer value of the first character of *char_expression* in the current character set. If *char_expression* is a literal string, the result is determined by the character set of the SQL client. If *char_expression* is a column in the database, the character set of the database determines the result.

COMPATIBILITY

ODBC Compatible

ASIN Function

Returns the arcsine of *expression*.

SYNTAX

```
ASIN ( expression )
```

EXAMPLE

The following example shows how to use the ASIN function. The first SELECT statement returns the arcsine in degrees, and the second returns the arcsine in radians.

```
select asin (1) * (180/ pi()) 'Arcsine in degrees' from syscalctable;

ARCSINE IN DEGREES
------------------
90.000000000000000

1 record selected


select asin (1) 'Arcsine in radians' from syscalctable;

ARCSINE IN RADIANS
------------------
1.570796326794897

1 record selected
```

ASIN takes the ratio (*expression*) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.

The result is expressed in radians and is in the range -pi/2 to pi/2 radians. To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

NOTES

- The *expression* must be in the range -1 to 1.
- The *expression* must evaluate to an approximate numeric data type.

COMPATIBILITY

ODBC Compatible

ATAN Function

Returns the arctangent of *expression*.

SYNTAX

```
ATAN ( expression )
```

EXAMPLE

The following example illustrates two ways to use the ATAN function:

```
select atan (1) * (180/ pi()) 'Arctangent in degrees' from syscalctable;

ARCTANGENT IN DEGREES
---------------------
45.000000000000000

1 record selected


select atan (1) 'Arctangent in radians' from syscalctable;

ARCTANGENT IN RADIANS
---------------------
0.785398163397448

1 record selected
```

ATAN takes the ratio (*expression*) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

NOTES

- The *expression* must be in the range -1 to 1.
- The *expression* must evaluate to an approximate numeric data type.

COMPATIBILITY

ODBC Compatible

ATAN2 Function

Returns the arctangent of the x and y coordinates specified by *expression1* and *expression2*.

SYNTAX

```
ATAN2 ( expression1 , expression2 )
```

EXAMPLE

The following example illustrates two ways to use the ATAN2 function:

```
select atan2 (1,1) * (180/ pi()) 'Arctangent in degrees' from syscalctable;

ARCTANGENT IN DEGREES
---------------------
45.000000000000000

1 record selected


select atan2 (1,1) 'Arctangent in radians' from syscalctable;

ARCTANGENT IN RADIANS
---------------------
0.785398163397448

1 record selected
```

NOTES

- ATAN2 takes the ratio of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.
- *expression1* and *expression2* specify the x and y coordinates of the end of the hypotenuse opposite the angle.
- The result is expressed in radians and is in the range -pi/2 to pi/2 radians. To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.
- Both *expression1* and *expression2* must evaluate to approximate numeric data types.

COMPATIBILITY

ODBC Compatible

CASE Function

Specifies a series of search conditions and associated result expressions. The general form is called a searched case expression. SQL returns the value specified by the first result expression whose associated search condition evaluates as true. If none of the search conditions evaluates as true, the CASE expression returns a null value, or the value of some other default expression if the CASE expression includes the ELSE clause.

CASE also supports syntax for a shorthand notation, called a simple case expression, for evaluating whether one expression is equal to a series of other expressions.

SYNTAX

```
searched_case_expr | simple_case_expr
```

searched_case_expr

SYNTAX

```
CASE
  WHEN search_condition THEN { result_expr | NULL }
  [ ... ]
  [ ELSE expr | NULL ]
END
```

simple_case_expr

SYNTAX

```
CASE primary_expr
  WHEN expr THEN { result_expr | NULL }
  [ ... ]
  [ ELSE expr | NULL ]
```

CASE

Specifies a searched case expression. It must be followed by one or more WHEN-THEN clauses, each specifying a search condition and corresponding expression.

WHEN *search_condition* THEN { *result_expr* | NULL }

Specifies a search condition and corresponding expression. SQL evaluates *search_condition*. If *search_condition* evaluates as true, CASE returns the value specified by *result_expr*, or null, if the clause specifies THEN NULL.

If *search_condition* evaluates as false, SQL evaluates the next WHEN-THEN clause, if any, or the ELSE clause, if it is specified.

CASE *primary_expr*

Specifies a simple case expression. In a simple case expression, one or more WHEN-THEN clauses specify two expressions.

WHEN *expr* THEN { *result_expr* | NULL }

Prompts SQL to evaluate *expr* and compare it with *primary_expr* specified in the CASE clause. If they are equal, CASE returns the value specified by *result_expr* (or null, if the clause specifies THEN NULL).

If *expr* is not equal to *primary_expr*, SQL evaluates the next WHEN-THEN clause, if any, or the ELSE clause, if it is specified.

ELSE { *expr* | NULL }

Specifies an optional expression whose value SQL returns if none of the conditions specified in WHEN-THEN clauses are satisfied. If the CASE expression omits the ELSE clause, it is the same as specifying ELSE NULL.

EXAMPLES

A simple case expression can always be expressed as a searched case expression. This example illustrates a simple case expression:

```
CASE primary_expr
     WHEN expr1 THEN result_expr1
     WHEN expr2 THEN result_expr2
     ELSE expr3
END
```

The simple case expression in the preceding CASE example is equivalent to the following searched case expression:

```
CASE
     WHEN primary_expr = expr1 THEN result_expr1
     WHEN primary_expr = expr2 THEN result_expr2
     ELSE expr3
END
```

The following example shows a searched case expression that assigns a label denoting suppliers as 'In Mass' if the state column value is 'MA':

```
select lastname, city,
     case
          when state = 'MA' then 'In Mass' else 'Not in Mass'
     end
from supplier;

Lastname         City             searched_case(State,MA,In Mass,)
---------------- ---------------- --------------------------------
GolfWorld Suppl  Boston             In Mass
Pool Swimming S  Valkeala           Not in Mass
Nordic Ski Whol  Hingham            In Mass
Champion Soccer  Harrow             Not in Mass
ABC Sports Supp  Boston             In Mass
Seasonal Sports  Bedford            In Mass
Tennis Supplies  Boston             In Mass
Boating Supplie  Jacksonville       Not in Mass
Aerobic Supplie  Newport Beach      Not in Mass
Sports Unlimite    Irving               Not in Mass
```

The following example shows the equivalent simple case expression:

```
SELECT lastname, city
     case state
          when 'MA' then 'In Mass' else 'Not in Mass'
     end
FROM supplier;

Lastname         City             simple_case(State,MA,In Mass,)
---------------- ---------------- ------------------------------
GolfWorld Suppl  Boston             In Mass
Pool Swimming S  Valkeala           Not in Mass
Nordic Ski Whol  Hingham            In Mass
Champion Soccer  Harrow             Not in Mass
ABC Sports Supp  Boston             In Mass
Seasonal Sports  Bedford            In Mass
Tennis Supplies  Boston             In Mass
Boating Supplie  Jacksonville       Not in Mass
Aerobic Supplie  Newport Beach      Not in Mass
Sports Unlimite    Irving               Not in Mass
```

NOTES

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.

COMPATIBILITY

SQL-92 Compatible

CAST Function

Converts an expression to another data type. The first argument is the expression to be converted. The second argument is the target data type.

The length option for the data_type argument specifies the length for conversions to CHAR and VARCHAR data types. If omitted, the default is 1 byte.

If the expression evaluates to null, the result of the function is null. Specifying NULL with the CAST function is useful for set operations, such as UNION, that require two tables to have the same structure. CAST NULL allows you to specify a column of the correct data type, so a table with a similar structure to another, but with fewer columns, can be in a union operation with the other table.

The CAST function provides a data-type-conversion mechanism compatible with the SQL-92 standard.

Use the CONVERT function, enclosed in the ODBC escape clause { fn }, to specify ODBC-compliant syntax for data type conversion. See the ODBC compatible CONVERT function for more information.

SYNTAX

```
CAST ( { expression | NULL } AS data_type [ ( length )] )
```

EXAMPLE

The following SQL example uses CAST to convert an integer field from a catalog table to a CHARACTER data type:

```
SELECT CAST(fld AS CHAR(25)), fld FROM sysprogress.syscalctable;

CONVERT(CHARACTER(25),FLD)     FLD
--------------------------     ---
100                            100

1 record selected
```

COMPATIBILITY

SQL-92 Compatible

CEILING Function

Returns the smallest integer greater than or equal to *expression*.

SYNTAX

```
CEILING ( expression )
```

EXAMPLE

This example illustrates the CEILING function:

```
SELECT CEILING (32.5) 'Ceiling'
     FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTE

- The expression must evaluate to a numeric data type.

COMPATIBILITY

ODBC Compatible

CHAR Function

Returns a character string with the first character having an ASCII value equal to the argument expression. CHAR is identical to CHR but provides ODBC-compatible syntax.

SYNTAX

```
CHAR ( integer_expression )
```

EXAMPLE

This example illustrates the CHAR function:

```
SELECT *
    FROM  customer
    WHERE SUBSTR (zip, 1, 1) = CHAR (53) ;
```

NOTES

- The argument to the function must be of type INTEGER, TINYINT, or SMALLINT.
- The result is of type CHARACTER.
- If the argument *integer_expression* evaluates to null, the result is null.
- The CHAR and CHR functions are character-set dependent, and support single-byte and multi-byte characters. If *integer_expression* is a valid character encoding integer value in the current SQL server character set, the function returns the correct character. If it is not a valid character the function returns a NULL value.

COMPATIBILITY

ODBC Compatible

CHR Function

Returns a character string with the first character having an ASCII value equal to the argument expression.

SYNTAX

```
CHR ( integer_expression )
```

EXAMPLE

This example illustrates the CHR function and the SUBSTR (substring) function:

```
SELECT *
    FROM  customer
    WHERE SUBSTR (zip, 1, 1) = CHR (53) ;
```

NOTES

- The argument to the function must be of type INTEGER, TINYINT, or SMALLINT.
- The result is of type CHARACTER.
- If the argument *integer_expression* evaluates to null, the result is null.
- The CHR and CHAR functions are character-set dependent, and support multi-byte characters. If *integer_expression* is a valid character encoding integer value in the current SQL server character set, the function returns the correct character. If it is not a valid character the function returns a NULL value.

COMPATIBILITY

Progress Extension

COALESCE Function

Specifies a series of expressions, and returns the first expression whose value is not null. If all the expressions evaluate as null, COALESCE returns a null value.

SYNTAX

```
COALESCE ( expression1, expression2 [ . . . ] )
```

NOTE: The COALESCE syntax is shorthand notation for a common case that can also be represented in a CASE expression. The following two formulations are equivalent:

```
COALESCE ( expression1 , expression2 , expression3 )
```

```
CASE
    WHEN expression1 IS NOT NULL THEN expression1
    WHEN expression2 IS NOT NULL THEN expression2
    ELSE expression3
END
```

EXAMPLE

This example illustrates the COALESCE function:

```
SELECT COALESCE (end_date, start_date) from job_hist;
```

NOTES

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.

COMPATIBILITY

SQL-92 Compatible

CONCAT Function

Returns a concatenated character string formed by concatenating argument one with argument two.

SYNTAX

```
CONCAT ( char_expression , char_expression )
```

EXAMPLE

This example illustrates the CONCAT function:

```
SELECT last_name, empno, salary
    FROM customer
    WHERE project = CONCAT('US',proj_nam);
```

NOTES

- Both of the arguments must be of type CHARACTER or VARCHAR.
- The result is of type VARCHAR.
- If any of the argument expressions evaluate to null, the result is null.
- The two *char_expression* expressions and the result of the CONCAT function may contain multi-byte characters.

COMPATIBILITY

ODBC Compatible

CONVERT Function (ODBC Compatible)

Converts an expression to another data type. The first argument is the expression to be converted. The second argument is the target data type.

If the expression evaluates to null, the result of the function is null.

The ODBC CONVERT function provides ODBC-compliant syntax for data type conversions. You must enclose the function with the ODBC escape clause { fn } to use ODBC-compliant syntax.

SYNTAX

```
{ fn CONVERT ( expression , data_type ) }
```

NOTE: Braces are part of the actual syntax.

data_type

SYNTAX

```
SQL_BINARY | SQL_BIT | SQL_CHAR | SQL_DATE | SQL_DECIMAL
  | SQL_DOUBLE | SQL_FLOAT | SQL_INTEGER | SQL_REAL
  | SQL_SMALLINT | SQL_TIME | SQL_TIMESTAMP | SQL_TINYINT
  | SQL_VARBINARY | SQL_VARCHAR
```

COMPATIBILITY

ODBC Compatible

CONVERT Function (Progress Extension)

Converts an expression to another data type. The first argument is the target data type. The second argument is the expression to be converted to that type.

The length option for the *data_type* argument specifies the length for conversions to CHAR and VARCHAR data types. If omitted, the default is 30 bytes.

If the expression evaluates to null, the result of the function is null.

The CONVERT function syntax is similar to but not compatible with the ODBC CONVERT function. Enclose the function in the ODBC escape clause { fn } to specify ODBC-compliant syntax. See the ODBC compatible CONVERT function for more information.

SYNTAX

```
CONVERT ( 'data_type [ ( length ) ]', expression )
```

EXAMPLE

The following SQL example uses the CONVERT function to convert an INTEGER field from a system table to a character string:

```
SELECT CONVERT('CHAR', fld), fld FROM sysprogress.syscalctable;

CONVERT(CHAR,FLD)               FLD
-----------------               ---
100                             100

1 record selected


SELECT CONVERT('CHAR(35)', fld), fld FROM sysprogress.syscalctable;

CONVERT(CHAR(35),FLD)               FLD
---------------------               ---
100                                 100

1 record selected
```

NOTE

- When *data_type* is CHARACTER(*length*) or VARCHAR(*length*), the *length* specification represents the number of characters. The converted result may contain multi-byte characters.

COMPATIBILITY

Progress Extension

COS Function

Returns the cosine of *expression*.

SYNTAX

```
COS ( expression )
```

EXAMPLE

This example illustrates the COS function:

```
select cos(45 * pi()/180) 'Cosine of 45 degrees'
     from sysprogress.syscalctable;

COSINE OF 45 DEG
----------------
0.707106781186548

1 record selected
```

NOTES

- COS takes an angle *expression* and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.
- The expression specifies an angle in radians.
- The expression must evaluate to an approximate numeric data type.
- To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

COMPATIBILITY

ODBC Compatible

CURDATE Function

Returns the current date as a DATE value. This function takes no arguments.

SYNTAX

```
CURDATE ( )
```

NOTE: SQL statements can refer to CURDATE anywhere they can refer to a DATE expression.

EXAMPLE

The following example shows how to use the CURDATE function:

```
INSERT INTO objects (object_owner, object_id, create_date)
     VALUES (USER, 1001, CURDATE()) ;
```

COMPATIBILITY

ODBC Compatible

CURTIME Function

Returns the current time as a TIME value. This function takes no arguments.

SYNTAX

```
CURTIME ( )
```

NOTE: SQL statements can refer to CURTIME anywhere they can refer to a TIME expression.

EXAMPLE

This example illustrates how to use the CURTIME function to INSERT the current time into the create_time column of the objects table:

```
INSERT INTO objects (object_owner, object_id, create_time)
     VALUES (USER, 1001, CURTIME()) ;
```

COMPATIBILITY

ODBC Compatible

DATABASE Function

Returns the name of the database corresponding to the current connection name. This function takes no arguments, and the trailing parentheses are optional.

SYNTAX

```
DATABASE [ ( ) ]
```

EXAMPLE

The following example shows how to use the DATABASE function:

```
select database() from t2;

DATABASE
--------
steel

1 record selected
```

COMPATIBILITY

ODBC Compatible

DAYNAME Function

Returns a character string containing the name of the day (for example, Sunday through Saturday) for the day portion of *date_expression*. The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

SYNTAX

```
DAYNAME ( date_expression )
```

EXAMPLE

This example illustrates the DAYNAME function:

```
SELECT *
     FROM orders
     WHERE order_no = 342 and DAYNAME(order_date)='SATURDAY';


ORDER_NO  ORDER_DATE  REFERENCE         CUST_NO
--------  ----------  ----------        --------
342       08/10       tdfg/101          10001

1 record selected
```

COMPATIBILITY

ODBC Compatible

DAYOFMONTH Function

Returns the day of the month in the argument as a short integer value in the range of 1-31. The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

SYNTAX

```
DAYOFMONTH ( date_expression )
```

EXAMPLE

This example illustrates the DAYOFMONTH function:

```
SELECT *
    FROM orders
    WHERE DAYOFMONTH (order_date) = 14 ;
```

NOTES

- The *date_expression* argument must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM). See syntax for *date_literal* in the "Date-time Literals" section in Chapter 2, "SQL-92 Language Elements."
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

DAYOFWEEK Function

Returns the day of the week in the argument as a short integer value in the range of 1-7.

The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

SYNTAX

```
DAYOFWEEK ( date_expression )
```

EXAMPLE

The following example shows how to use the DAYOFWEEK function:

```
SELECT *
    FROM orders
    WHERE DAYOFWEEK (order_date) = 2 ;
```

NOTES

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM). See the syntax for *date_literal* in the "Date-time Literals" section in Chapter 2, "SQL-92 Language Elements."
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

DAYOFYEAR Function

Returns the day of the year in the argument as a short integer value in the range of 1-366. The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

SYNTAX

```
DAYOFYEAR ( date_expression )
```

EXAMPLE

This example illustrates the DAYOFYEAR function:

```
SELECT *
    FROM orders
    WHERE DAYOFYEAR (order_date) = 300 ;
```

NOTES

- The argument to the function must be of type DATE.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

DB_NAME Function

Returns the name of the database corresponding to the current connection name. It provides compatibility with the Sybase SQL Server function db_name.

SYNTAX

```
DB_NAME ( )
```

EXAMPLE

This example illustrates the DB_NAME function:

```
SELECT DB_NAME() FROM T2;

DB_NAME
-------
demo

1 record selected
```

COMPATIBILITY

Progress Extension

DECODE Function

Compares the value of the first argument *expression* with each *search_expression* and, if a match is found, returns the corresponding *match_expression*. If no match is found, then the function returns the *default_expression*. If a *default_expression* is not specified and no match is found, the function returns a null value.

SYNTAX

```
DECODE ( expression, search_expression, match_expression
   [ , search_expression, match_expression . . . ]
   [ , default_expression ] )
```

EXAMPLE

This example illustrates one way to use the DECODE function:

```
SELECT ename, DECODE (deptno,
             10, 'ACCOUNTS  ',
             20, 'RESEARCH  ',
             30, 'SALES     ',
             40, 'SUPPORT   ',
             'NOT ASSIGNED'
          )
   FROM employee ;
```

NOTES

- Use a simple case expression when SQL-92-compatible syntax is a requirement.
- The first argument *expression* can be of any type. The types of all *search_expressions* must be compatible with the type of the first argument.
- The *match_expressions* can be of any type. The types of all *match_expressions* must be compatible with the type of the first *match_expression*.
- The type of the *default_expression* must be compatible with the type of the first *match_expression*.

- The type of the result is the same as that of the first *match_expression*.
- If the first argument *expression* is null, then the value of the *default_expression* is returned, if it is specified. Otherwise null is returned.

COMPATIBILITY

Progress Extension

DEGREES Function

Returns the number of degrees in an angle specified in radians by *expression*.

SYNTAX

```
DEGREES ( expression )
```

EXAMPLE

This example illustrates the DEGREES function:

```
SELECT DEGREES(3.14159265359) 'Degrees in pi Radians'
     FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTES

- The *expression* specifies an angle in radians.
- The *expression* must evaluate to a numeric data type.

COMPATIBILITY

ODBC Compatible

EXP Function

Returns the exponential value of *expression* (e raised to the power of *expression*).

SYNTAX

```
EXP ( expression )
```

EXAMPLE

This example illustrates the EXP function:

```
SELECT EXP( 4 ) 'e to the 4th power' from sysprogress.syscalctable;
```

NOTE

- *expression* must evaluate to an approximate numeric data type.

COMPATIBILITY

ODBC Compatible

FLOOR Function

Returns the largest integer less than or equal to *expression*.

SYNTAX

```
FLOOR ( expression )
```

EXAMPLE

This example illustrates the FLOOR function:

```
SELECT FLOOR (32.5) 'Floor' from sysprogress.syscalctable ;
```

NOTE

- *expression* must evaluate to a numeric data type.

COMPATIBILITY

ODBC Compatible

GREATEST Function

Returns the greatest value among the values of the given expressions.

SYNTAX

```
GREATEST ( expression , expression ... )
```

EXAMPLE

This example illustrates the GREATEST function:

```
SELECT cust_no, last_name,
    GREATEST (ADD_MONTHS (start_date, 10), SYSDATE)
    FROM customer ;
```

NOTES

- The first argument to the function can be of any type. However, the types of the subsequent arguments must be compatible with that of the first argument.
- The type of the result is the same as that of the first argument.
- If any of the argument expressions evaluate to null, the result is null.
- When the data type of an *expression* is either CHARACTER(*length*) or VARCHAR(*length*), the expression may contain multi-byte characters. The sort weight for each character is determined by the collation table in the database.

COMPATIBILITY

Progress Extension

HOUR FUNCTION

Returns the hour in the argument as a short integer value in the range of 0-23.

SYNTAX

```
HOUR ( time_expression )
```

EXAMPLE

This example illustrates the HOUR function:

```
SELECT *
    FROM arrivals
    WHERE HOUR (in_time) < 12 ;
```

NOTES

- The argument to the function must be of type TIME.
- The argument must be specified in the format *hh:mi:ss*.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

IFNULL Function

Returns *value* if *expr* is null. If *expr* is not null, IFNULL returns *expr*.

SYNTAX

```
IFNULL( expr, value)
```

EXAMPLE

This example illustrates the IFNULL function. The SELECT statement returns three rows with a NULL value in column C1, and two non-NULL values:

```
SELECT C1, IFNULL(C1, 9999) FROM TEMP ORDER BY C1;

C1    IFNULL(C1,9999)
--    ---------------
      9999
      9999
      9999
1     1
3     3
```

NOTE

- The data type of *value* must be compatible with the data type of *expr*.

COMPATIBILITY

ODBC Compatible

INITCAP Function

Returns the result of the argument character expression after converting the first character to uppercase and the subsequent characters to lowercase.

SYNTAX

```
INITCAP ( char_expression )
```

EXAMPLE

The following example shows how to use the INITCAP function:

```
SELECT INITCAP (last_name)
     FROM customer ;
```

NOTES

- The *char_expression* must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.
- A *char_expression* and the result may contain multi-byte characters. The uppercase conversion for the first character and the lowercase conversion for the rest of the characters is based on the case table in the *convmap* file. The default case table is BASIC.

COMPATIBILITY

Progress Extension

INSERT Function

Returns a character string where *length* number of characters have been deleted from *string_exp1* beginning at *start_pos*, and *string_exp2* has been inserted into *string_exp1*, beginning at *start_pos*.

SYNTAX

```
INSERT( string_exp1 , start_pos , length , string_exp2 )
```

EXAMPLE

This example illustrates the INSERT function. The two letters 'o' and 'l' are deleted from the name 'Goldman' in the last_name column, and the letters 'xx' are inserted into the last_name column, beginning at the fourth character, overlaying the letters 'd' and 'm':

```
SELECT INSERT(last_name,2,4,'xx')
     FROM customer
     WHERE last_name = 'Goldman';

INSERT LAST_NAME,2,4,XX)
------------------------
Gxxan

1 record selected
```

NOTES

- The *string_exp* can be type fixed-length or-variable length CHARACTER.
- The *start_pos* and *length* can be of data type INTEGER, SMALLINT, or TINYINT.
- The result string is of the type *string_exp1*.
- If any of the argument expressions evaluate to null, the result is null.
- If *start_pos* is negative or zero, the result string evaluates to null.

- If *length* is negative, the result evaluates to null.
- *string_exp1* and *string_exp2* and the result may contain multi-byte characters. This is determined by the character set of the SQL server. The *length* argument specifies a number of characters.

COMPATIBILITY

ODBC Compatible

INSTR Function

Searches character string *char_expression1* for the character string *char_expression2*. The search begins at *start_pos* of *char_expression1*. If *occurrence* is specified, then INSTR searches for the *n*th occurrence, where *n* is the value of the fourth argument.

The position (with respect to the start of *char_expression1*) is returned if a search is successful. Zero is returned if no match can be found.

SYNTAX

```
INSTR ( char_expression1 , char_expression2
   [ , start_pos [ , occurrence ] ] )
```

EXAMPLE

This example illustrates the INSTR function:

```
SELECT cust_no, last_name
    FROM customer
    WHERE INSTR (LOWER (addr), 'heritage') > 0 ;
```

NOTES

- The first and second arguments must be CHARACTER data type.
- The third and fourth arguments, if specified, must be SMALLINT or TINYINT data type.
- The value for start position in a character string is the ordinal number of the character in the string. The very first character in a string is at position 1, the second character is at position 2, the *n*th character is at position *n*.
- If you do not specify *start_pos*, a default value of 1 is assumed.
- If you do not specify *occurrence*, a default value of 1 is assumed.
- The result is INTEGER data type.

- If any of the argument expressions evaluate to null, the result is null.
- *A char_expression* and the result may contain multi-byte characters.

COMPATIBILITY

Progress Extension

LAST_DAY Function

Returns the date corresponding to the last day of the month containing the argument date.

SYNTAX

```
LAST_DAY ( date_expression )
```

EXAMPLE

This example illustrates the LAST_DAY function:

```
SELECT *
    FROM orders
    WHERE LAST_DAY (order_date) + 1 = '08/01/1999' ;
```

NOTES

- The argument to the function must be of type DATE.
- The result is of type DATE.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

Progress Extension

LCASE Function

Returns the result of the argument character expression after converting all the characters to lowercase. LCASE is the same as LOWER but provides ODBC-compatible syntax.

SYNTAX

```
LCASE ( char_expression )
```

EXAMPLE

This example illustrates the LCASE function:

```
SELECT *
    FROM customer
    WHERE LCASE (last_name) = 'smith' ;
```

NOTES

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.
- A *char_expression* and the result may contain multi-byte characters. The lowercase conversion is determined by the case table in the `convmap` file. The default case table is BASIC.

COMPATIBILITY

ODBC Compatible

LEAST Function

Returns the lowest value among the values of the given expressions.

SYNTAX

```
LEAST ( expression , expression, ... )
```

EXAMPLE

This example illustrates the LEAST function:

```
SELECT cust_no, last_name,
     LEAST (ADD_MONTHS (start_date, 10), SYSDATE)
     FROM customer ;
```

NOTES

- The first argument to the function can be of any type. However, the types of the subsequent arguments must be compatible with that of the first argument.
- The type of the result is the same as that of the first argument.
- If any of the argument expressions evaluate to null, the result is null.
- When the data type of an *expression* is either CHARACTER(*length*) or VARCHAR(*length*), the *expression* may contain multi-byte characters. The sort weight for each character is determined by the collation table in the database.

COMPATIBILITY

Progress Extension

LEFT Function

Returns the leftmost count of characters of *string_exp*.

SYNTAX

```
LEFT ( string_exp , count )
```

EXAMPLE

The following example shows how to use the LEFT function:

```
SELECT LEFT(last_name,4) FROM  customer WHERE last_name = 'Goldman';

LEFT(LAST_NAME),4)
------------------
Gold

1 record selected
```

NOTES

- *string_exp* can be fixed-length or variable-length CHARACTER data types.
- *count* can be INTEGER, SMALLINT, or TINYINT data types.
- If any of the arguments of the expression evaluate to null, the result is null.
- If the *count* is negative, the result evaluates to null.
- The *string_exp* and the result may contain multi-byte characters. The function returns the number of characters.

COMPATIBILITY

ODBC Compatible

LENGTH Function

Returns the string length of the value of the given character expression.

SYNTAX

```
LENGTH ( char_expression )
```

EXAMPLE

This example illustrates the LENGTH function:

```
SELECT last_name 'LONG LAST_NAME'
     FROM customer
     WHERE LENGTH (last_name) > 5 ;
```

NOTES

- The argument to the function must be of type CHARACTER or VARCHAR.
- The result is of type INTEGER.
- If the argument expression evaluates to null, the result is null.
- *char_expression* may contain multi-byte characters. The function returns a number of characters.

COMPATIBILITY

ODBC Compatible

LOCATE Function

Returns the location of the first occurrence of *char_expr1* in *char_expr2*. If the function includes the optional integer argument *start_pos*, LOCATE begins searching *char_expr2* at that position. If the function omits the *start_pos* argument, LOCATE begins its search at the beginning of *char_expr2*.

LOCATE denotes the first character position of a character expression as 1. If the search fails, LOCATE returns 0. If either character expression is null, LOCATE returns a null value.

SYNTAX

```
LOCATE( char_expr1 , char_expr2 , [ start_pos ] )
```

EXAMPLE

The following example uses two string literals as character expressions. LOCATE returns a value of 6:

```
SELECT LOCATE('this', 'test this test', 1) FROM TEST;

LOCATE(THIS,
------------
6
1 record selected
```

NOTE

- *char_expr1* and *char_expr2* may contain multi-byte characters. The *start_pos* argument specifies the position of a starting character, not a byte position. The search is case-sensitive. Character comparisons use the collation table in the database.

COMPATIBILITY

ODBC Compatible

LOG10 Function

Returns the base 10 logarithm of *expression*.

SYNTAX

```
LOG10 ( expression )
```

EXAMPLE

This example illustrates the LOG10 function:

```
SELECT LOG10 (100) 'Log base 10 of 100' FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTE

- The *expression* must evaluate to an approximate numeric data type.

COMPATIBILITY

ODBC Compatible

LOWER Function

Returns the result of the argument *char_expression* after converting all the characters to lowercase.

SYNTAX

```
LOWER ( char_expression )
```

EXAMPLE

This example illustrates the LOWER function:

```
SELECT *
    FROM customer
    WHERE LOWER (last_name) = 'smith' ;
```

NOTES

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

SQL-92 Compatible

LPAD Function

Pads the character string corresponding to the first argument on the left with the character string corresponding to the third argument. After the padding, the length of the result is *length*.

SYNTAX

```
LPAD ( char_expression , length [ , pad_expression ] )
```

EXAMPLE

This example illustrates two ways to use the LPAD function:

```
SELECT LPAD (last_name, 30) FROM customer ;

SELECT LPAD (last_name, 30, '.') FROM customer ;
```

NOTES

- The first argument to the function must be of type CHARACTER. The second argument to the function must be of type INTEGER. The third argument, if specified, must be of type CHARACTER. If the third argument is not specified, the default value is a string of length 1 containing one blank.
- If *L1* is the length of the first argument and *L2* is the value of the second argument:
  - If *L1* is less than *L2*, the number of characters padded is equal to *L2* minus *L1*.
  - If *L1* is equal to *L2*, no characters are padded and the result string is the same as the first argument.
  - If *L1* is greater than *L2*, the result string is equal to the first argument truncated to the first *L2* characters.

- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.
- The *char_expression* and *pad_expression* may contain multi-byte characters. The *length* specifies a number of characters.

COMPATIBILITY

Progress Extension

LTRIM Function

Removes all the leading characters in *char_expression* that are present in *char_set* and returns the resulting string. The first character in the result is guaranteed not to be in *char_set*. If you do not specify the *char_set* argument, leading blanks are removed.

SYNTAX

```
LTRIM ( char_expression [ , char_set ] )
```

EXAMPLE

This example illustrates the LTRIM function:

```
SELECT last_name, LTRIM (addr, ' ')
     FROM customer ;
```

NOTES

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.
- The *char_expression*, the character set specified by *char_set*, and the result may contain multi-byte characters. Character comparisons are case-sensitive and are determined by the collation table in the database.

COMPATIBILITY

ODBC Compatible

MINUTE Function

Returns the minute value in the argument as a short integer in the range of 0-59.

SYNTAX

```
MINUTE ( time_expression )
```

EXAMPLE

This example illustrates the MINUTE function:

```
SELECT *
     FROM arrivals
     WHERE MINUTE (in_time) > 10 ;
```

NOTES

- The argument to the function must be of type TIME.
- The argument must be specified in the format *HH:MI:SS*.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

MOD Function

Returns the remainder of *expression1* divided by *expression2*.

SYNTAX

```
MOD ( expression1 , expression2 )
```

The following example shows how to use the MOD function:

EXAMPLE

This example illustrates the MOD function:

```
SELECT MOD (11, 4) 'Modulus' FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTES

- Both *expression1* and *expression2* must evaluate to exact numeric data types.
- If *expression2* evaluates to zero, MOD returns zero.

COMPATIBILITY

ODBC Compatible

MONTH Function

Returns the month in the year specified by the argument as a short integer value in the range of 1-12.

SYNTAX

```
MONTH ( date_expression )
```

EXAMPLE

This example illustrates the MONTH function:

```
SELECT * FROM orders WHERE MONTH (order_date) = 6 ;
```

NOTES

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a time literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM). See the syntax for *date_literal* in the "Date-time Literals" section in Chapter 2, "SQL-92 Language Elements."
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

MONTHNAME Function

Returns a character string containing the name of the month (for example, January through December) for the month portion of *date_expression*. Argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

SYNTAX

```
MONTHNAME ( date_expression )
```

EXAMPLE

This example illustrates the MONTHNAME function. The query returns all rows where the name of the month in the order_date column is equal to 'June':

```
SELECT *
FROM orders
WHERE order_no =346 and MONTHNAME(order_date)='JUNE';

ORDER_NO ORDER_DATE  REFERENCE       CUST_NO
-------- ---------   --------        -------
346      06/01/1991    87/rd          10002

1 record selected
```

COMPATIBILITY

ODBC Compatible

MONTHS_BETWEEN Function

Computes the number of months between two date values corresponding to the first and second arguments.

SYNTAX

```
MONTHS_BETWEEN ( date_expression, date_expression )
```

EXAMPLE

This example illustrates the MONTHS_BETWEEN function:

```
SELECT MONTHS_BETWEEN (SYSDATE, order_date)
    FROM orders
    WHERE order_no = 1002 ;
```

NOTES

- The first and second arguments to the function must be of type DATE.
- The result is of type INTEGER.
- The result is negative if the date corresponding to the second argument is greater than that corresponding to the first argument.
- If any of the argument expressions evaluates to null, the result is null.

COMPATIBILITY

Progress Extension

NEXT_DAY Function

Returns the minimum date that is greater than the date corresponding to the first argument where the day of the week is the same as that specified by the second argument.

SYNTAX

```
NEXT_DAY ( date_expression, day_of_week )
```

EXAMPLE

This example illustrates the NEXT_DAY function:

```
SELECT NEXT_DAY (order_date, 'MONDAY') FROM orders ;
```

NOTES

- The first argument to the function must be of type DATE.
- The second argument to the function must be of type CHARACTER. The result of the second argument must be a valid day of the week ('SUNDAY', 'MONDAY' etc.).
- The result is of type DATE.
- If any of the argument expressions evaluate to null, the result is null.

COMPATIBILITY

Progress Extension

NOW Function

Returns the current date and time as a TIMESTAMP value. This function takes no arguments.

SYNTAX

```
NOW ( )
```

COMPATIBILITY

ODBC Compatible

NULLIF Function

Returns a null value for *expression1* if it is equal to *expression2*. It is useful for converting values to null from applications that use some other representation for missing or unknown data. The NULLIF scalar function is a type of conditional expression.

SYNTAX

```
NULLIF ( expression1, expression2 )
```

EXAMPLE

This example uses the NULLIF scalar function to insert a null value into an address column if the host-language variable contains a single space character:

```
INSERT INTO employee (add1) VALUES (NULLIF (:address1, ' '));
```

NOTES

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.
- The NULLIF expression is shorthand notation for a common case that can also be represented in a CASE expression, as follows:

```
CASE
    WHEN expression1 = expression2 THEN NULL
    ELSE expression1
END
```

COMPATIBILITY

SQL-92 Compatible

NVL Function

Returns the value of the first expression if the first expression value is not null. If the first expression value is null, the value of the second expression is returned.

SYNTAX

```
NVL ( expression , expression )
```

NOTE: The NVL function is not ODBC compatible. Use the IFNULL function when ODBC-compatible syntax is required.

EXAMPLE

This example illustrates the NVL function:

```
SELECT salary + NVL (comm, 0) 'TOTAL SALARY' FROM employee ;
```

NOTES

- The first argument to the function can be of any type.
- The type of the second argument must be compatible with that of the first argument.
- The type of the result is the same as the first argument.

COMPATIBILITY

Progress Extension

PI Function

Returns the constant value of pi as a floating-point value.

SYNTAX

```
PI ( )
```

EXAMPLE

This example illustrates the PI function:

```
SELECT PI () FROM SYSPROGRESS.SYSCALCTABLE;
```

COMPATIBILITY

ODBC Compatible

POWER Function

Returns *expression1* raised to the power of *expression2*.

SYNTAX

```
POWER ( expression1 , expression2 )
```

EXAMPLE

This example illustrates the POWER function, raising ’3’ to the second power:

```
SELECT POWER ( 3 , 2) ’3 raised to the 2nd power’
     FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTES

- *expression1* must evaluate to a numeric data type.
- *expression2* must evaluate to an exact numeric data type.

PREFIX Function

Returns the substring of a character string, starting from the position specified by *start_pos* and ending before the specified character.

SYNTAX

```
PREFIX ( char_expression , start_pos , char_expression )
```

char_expression

Evaluates to a character string, typically a character-string literal or column name. If the expression evaluates to null, PREFIX returns null.

start_pos

Evaluates to an integer value. PREFIX searches the string specified in the first argument starting at that position. A value of 1 indicates the first character of the string.

char_expression

Evaluates to a single character. PREFIX returns the substring that ends before that character. If PREFIX does not find the character, it returns the substring beginning at *start_pos*, to the end of the string. If the expression evaluates to more than one character, PREFIX ignores all but the first character.

EXAMPLE

The following example shows one way to use the PREFIX function:

```
create table prefix_table
     (
     colstring varchar(20),
     colchar char(1)
     );

insert into prefix_table values ('string.with.dots', '.');

insert into prefix_table values ('string-with-dashes', '-');

select colstring, colchar, prefix(colstring, 1, '.') from prefix_table;

COLSTRING            COLCHAR              prefix(COLSTRING,1,.)
-------------------- -------------------- ---------------------
string.with.dots     .                    string
string-with-dashes   -                    string-with-dashes


select colstring, colchar, prefix(colstring, 1, colchar) from prefix_table;

COLSTRING            COLCHAR              prefix(COLSTRING,1,COLCHAR)
-------------------- -------------------- ---------------------------
string.with.dots     .                    string
string-with-dashes   -                    string


select colstring, colchar, prefix(colstring, 1, 'X') from prefix_table;

COLSTRING            COLCHAR              prefix(COLSTRING,1,X)
-------------------- -------------------- ---------------------
string.with.dots     .                    string.with.dots
string-with-dashes   -                    string-with-dashes
```

NOTE

- Each *char_expression* and the result may contain multi-byte characters. The *start_pos* argument specifies the character position, not a byte position. Character comparisons are case-sensitive and are determined by sort weights in the collation table in the database.

COMPATIBILITY

Progress Extension

PRO_* Functions

Provide limited support for the ARRAY data type. The PRO_ELEMENT, PRO_ARR_ESCAPE, and PRO_ARR_DESCAPE functions are documented in Appendix C, "Data Type Compatibility Issues with Previous Versions of Progress."

NOTE: For a specific example on how to have separate elements in an array field, see the Technical Support Kbase solution #19061.

QUARTER Function

Returns the quarter in the year specified by the argument as a short integer value in the range of 1-4.

SYNTAX

```
QUARTER ( date_expression )
```

EXAMPLE

This example illustrates the QUARTER function. The query requests all rows in the orders table where the order_date is in the third quarter of the year:

```
SELECT *
    FROM orders
    WHERE QUARTER (order_date) = 3 ;
```

NOTES

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM). See the syntax for *date_literal* in the "Date-time Literals" section in Chapter 2, "SQL-92 Language Elements."
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

RADIANS Function

Returns the number of radians in an angle specified in degrees by *expression*.

SYNTAX

```
RADIANS ( expression )
```

EXAMPLE

This example illustrates the RADIANS function:

```
SELECT RADIANS(180) 'Radians in 180 degrees' FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTES

- *expression* specifies an angle in degrees.
- *expression* must evaluate to a numeric data type.

COMPATIBILITY

ODBC Compatible

RAND Function

Returns a randomly generated number, using *expression* as an optional seed value.

SYNTAX

```
RAND ( [ expression ] )
```

EXAMPLE

This example illustrates the RAND function, supplying an optional seed value of '3':

```
SELECT RAND(3) 'Random number using 3 as seed value'
     FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTE

- *expression* must evaluate to an exact numeric data type.

COMPATIBILITY

ODBC Compatible

REPEAT Function

Returns a character string composed of *string_exp* repeated *count* times.

SYNTAX

```
REPEAT ( string_exp , count )
```

EXAMPLE

The following example shows how to use the REPEAT function:

```
SELECT REPEAT(fld1,3) FROM test100WHERE fld1 = 'Afghanistan' ;

REPEAT(FLD1,3)
------------------
AfghanistanAfghanistanAfghanistan

1 record selected
```

NOTES

- The *string_exp* can be of the type fixed length or variable length CHARACTER .
- The count can be of type INTEGER, SMALLINT, or TINYINT.
- If any of the arguments of the expression evaluates to a null, the result is null.
- If the count is negative or zero, the result evaluates to null.
- *string_exp* and the result may contain multi-byte characters.

COMPATIBILITY

ODBC Compatible

REPLACE Function

Replaces all occurrences of *string_exp2* in *string_exp1* with *string_exp3*.

SYNTAX

```
REPLACE ( string_exp1 , string_exp2 , string_exp3 )
```

EXAMPLE

This example illustrates the REPLACE function, replacing the letters 'mi' in the last_name 'Smith' with the letters 'moo':

```
SELECT REPLACE ( last_name,'mi','moo' )
     FROM customer WHERE last_name = 'Smith';

REPLACE(LAST_NAME,MI,MOO)
-------------------------
Smooth

1 record selected
```

NOTES

- *string_exp* can be fixed-length or variable-length CHARACTER data types.
- If any of the arguments of the expression evaluates to null, the result is null.
- If the replacement string is not found in the search string, it returns the original string.
- Each occurrence of *string_exp* and the result may contain multi-byte characters. Character comparisons are case-sensitive and are determined by sort weights in the collation table in the database.

COMPATIBILITY

ODBC Compatible

RIGHT Function

Returns the rightmost count of characters of *string_exp*.

SYNTAX

```
RIGHT ( string_exp , count )
```

EXAMPLE

This example illustrates the RIGHT function, selecting the rightmost six letters from the string 'Afghanistan':

```
SELECT RIGHT(fld1,6) FROM  test100 WHERE fld1 = 'Afghanistan';

RIGHT(FLD1,6)
---------------
nistan

1 record selected
```

NOTES

- The *string_exp* can be fixed-length or variable-length CHARACTER data types.
- The *count* can be INTEGER, SMALLINT, or TINYINT data types.
- If any of the arguments of the expression evaluate to null, the result is null.
- If *count* is negative, the result evaluates to null.
- *string_exp* and the result may contain multi-byte characters. *count* represents the number of characters.

COMPATIBILITY

ODBC Compatible

ROUND Function

Returns the rounded value of a numeric expression.

SYNTAX

```
ROUND ( num_expression [, rounding_factor ] ) ;
```

EXAMPLE

This example illustrates four calls to the ROUND function. In each case the *num_expression* is 2953861.8320. In the first call the *rounding_factor* is 2, in the second call the *rounding_factor* is -2, in the third call the *rounding_factor* is 0, and in the fourth call no *rounding_factor* is specified.

```
-- rounding_factor 2 returns 2953861.83
ROUND ( 2953861.8320, 2 )


-- rounding_factor -2 returns 2953900.00
ROUND ( 2953861.8320, -2 )


-- rounding_factor 0 returns 2953862.00
ROUND ( 2953861.8320, 0 )


-- No rounding_factor argument also returns 2953862.00
ROUND ( 2953861.8320 )
```

NOTES

- *num_expression* must be numeric or must be convertible to numeric.
- *num_expression* must be one of these supported data types:
  - INTEGER
  - TINYINT
  - SMALLINT
  - NUMBER

  - FLOAT
  - DOUBLE PRECISION

- If the data type of *num_expression* is not a supported type, ROUND returns an error message.
- The *num_expression* is rounded to the next higher digit when:
  - The digit before a negative *rounding_factor* is 5 or greater
  - The digit after a positive *rounding_factor* is 5 or greater
- The *num_expression* is rounded to the next lower digit when:
  - The digit before a negative *rounding_factor* is 4 or less
  - The digit after a positive *rounding_factor* is 4 or less
- *rounding_factor* is an integer between -32 and +32 inclusive, and indicates the digit position to which you want to round *num_expression*. Figure 4–1 illustrates how the digit positions are numbered. In the figure, the *num_expression* is 2953861.8320.

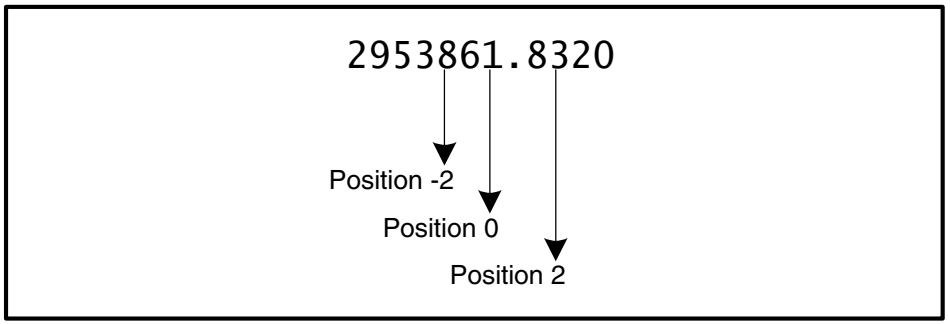


Figure 4–1: ROUND Function Digit Positions

- If you do not specify a *rounding_factor*, the function rounds *num_expression* to digit 0 (the ones place).
- To round to the right of the decimal point specify a positive *rounding_factor*.
- To round to the left of the decimal specify a negative *rounding_factor*.

COMPATIBILITY

Progress Extension

ROWID Function

Returns the row identifier of the current row in a table. This function takes no arguments. The ROWID of a row is determined when the row is inserted into the table. Once assigned, the ROWID remains the same for the row until the row is deleted. At any given time, each row in a table is uniquely identified by its ROWID.

NOTE: Using its ROWID is the most efficient way of selecting the row.

SYNTAX

```
ROWID
```

EXAMPLE

This example illustrates the ROWID function, returning all columns from the row in the customers table where the rowid = ’10’:

```
SELECT *
    FROM customers
    WHERE ROWID = ’10’;
```

COMPATIBILITY

Progress Extension

RPAD Function

Pads the character string corresponding to the first argument on the right with the character string corresponding to the third argument. After the padding, the length of the result is equal to the value of the second argument *length*.

SYNTAX

```
RPAD ( char_expression, length [ , pad_expression ] )
```

EXAMPLE

This example illustrates two ways to use the RPAD function:

```
SELECT RPAD (last_name, 30)
     FROM customer ;

SELECT RPAD (last_name, 30, '.')
     FROM customer ;
```

NOTES

- The first argument to the function must be of type CHARACTER. The second argument to the function must be of type INTEGER. The third argument, if specified, must be of type CHARACTER. If the third argument is not specified, the default value is a string of length 1 containing one blank.
- If *L1* is the length of the first argument and *L2* is the value of the second argument:
  - If *L1* is less than *L2*, the number of characters padded is equal to *L2* minus *L1*.
  - If *L1* is equal to *L2*, no characters are padded and the result string is the same as the first argument.
  - If *L1* is greater than *L2*, the result string is equal to the first argument truncated to the first *L2* characters.
- The result is of type CHARACTER.

- If the argument expression evaluates to null, the result is null.
- *char_expression* and *pad_expression* may contain multi-byte characters. *length* represents the number of characters in the result.

COMPATIBILITY

Progress Extension

RTRIM Function

Removes all the trailing characters in *char_expression* that are present in *char_set* and returns the resultant string. The last character in the result is guaranteed not to be in *char_set*. If you do not specify a *char_set*, trailing blanks are removed.

SYNTAX

```
RTRIM ( char_expression [ , char_set ] )
```

EXAMPLE

This example illustrates the RTRIM function:

```
SELECT RPAD ( RTRIM (addr, ' '), 30, '.')
     FROM customer ;
```

NOTES

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.
- The *char_expression*, the character set specified by *char_set*, and the result may contain multi-byte characters. Character comparisons are case-sensitive and are determined by the collation table in the database.

COMPATIBILITY

ODBC Compatible

SECOND Function

Returns the seconds in the argument as a short integer value in the range of 0-59.

SYNTAX

```
SECOND ( time_expression )
```

EXAMPLE

This example illustrates the SECOND function, requesting all columns from rows in the arrivals table where the in_time column is less than or equal to ’40’:

```
SELECT * FROM arrivals WHERE SECOND (in_time) <= 40 ;
```

NOTES

- The argument to the function must be of type TIME.
- The argument must be specified in the format *HH:MI:SS*.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

SIGN Function

Returns 1 if *expression* is positive, -1 if *expression* is negative, or zero if *expression* is zero.

SYNTAX

```
SIGN ( expression )
```

EXAMPLE

This example illustrates the SIGN function:

```
SELECT SIGN(-14) 'Sign' FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTE

- *expression* must evaluate to a numeric data type.

COMPATIBILITY

ODBC Compatible

SIN Function

Returns the sine of *expression*.

SYNTAX

```
SIN ( expression )
```

EXAMPLE

This example illustrates the SIN trigonometric function:

```
select sin(45 * pi()/180) 'Sine of 45 degrees' from syscalctable;

SINE OF 45 DEGREES
------------------
0.707106781186547

1 record selected
```

NOTES

- SIN takes an angle (*expression*) and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.
- *expression* specifies an angle in radians
- *expression* must evaluate to an approximate numeric data type.
- To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

COMPATIBILITY

ODBC Compatible

SQRT Function

Returns the square root of *expression*.

SYNTAX

```
SQRT ( expression )
```

EXAMPLE

This example illustrates the SQRT function, requesting the square root of the value ’28’:

```
SELECT SQRT(28) ’square root of 28’ FROM SYSPROGRESS.SYSCALCTABLE;
```

NOTES

- The value of *expression* must be positive.
- *expression* must evaluate to an approximate numeric data type.

COMPATIBILITY

ODBC Compatible

SUBSTR Function

Returns the substring of the character string corresponding to the first argument starting at *start_pos* and *length* characters long. If the third argument *length* is not specified, the substring starting at *start_pos* up to the end of *char_expression* is returned.

SYNTAX

```
SUBSTR ( char_expression, start_pos [ , length ] )
```

EXAMPLE

This example illustrates the SUBSTR function:

```
SELECT last_name, '(', SUBSTR (phone, 1, 3) , ')',
          SUBSTR (phone, 4, 3), '-',
          SUBSTR (phone, 7, 4)
  FROM customer ;
```

NOTES

- The first argument must be of type CHARACTER.
- The second argument must be of type INTEGER.
- The third argument, if specified, must be of type INTEGER.
- The values for specifying position in the character string start from 1. The first character in a string is at position 1, the second character is at position 2, and so on.
- The result is of type CHARACTER.
- If any of the argument expressions evaluate to null, the result is null.
- *char_expression* and the result may contain multi-byte characters. *length* specifies a number of characters. Character comparisons are case-sensitive and are determined by sort weights in the collation table in the database.

COMPATIBILITY

Progress Extension

SUBSTRING Function (ODBC Compatible)

Returns the substring of the character string corresponding to the first argument starting at *start_pos* and *length* characters long. If the third argument *length* is not specified, the substring starting at *start_pos* up to the end of *char_expression* is returned.

SYNTAX

```
SUBSTRING ( char_expression, start_pos [ , length ] )
```

EXAMPLE

This example illustrates the SUBSTRING function:

```
SELECT last_name, '(', SUBSTRING (phone, 1, 3) , ')',
          SUBSTRING (phone, 4, 3), '-',
          SUBSTRING (phone, 7, 4)
  FROM customer ;
```

NOTES

- The first argument must be of type CHARACTER.
- The second argument must be of type INTEGER.
- The third argument, if specified, must be of type INTEGER.
- The values for specifying position in the character string start from 1. The very first character in a string is at position 1, the second character is at position 2, and so on.
- The result is of type CHARACTER.
- If any of the argument expressions evaluate to null, the result is null.
- *char_expression* and the result may contain multi-byte characters. *start_pos* is the character position, and *length* specifies a number of characters.

COMPATIBILITY

ODBC Compatible

SUFFIX Function

Returns the substring of a character string starting after the position specified by *start_pos* and the second *char_expression*, to the end of the string.

SYNTAX

```
SUFFIX (char_expression , start_pos , char_expression )
```

char_expression

Evaluates to a character string, typically a character-string literal or column name. If the expression evaluates to null, SUFFIX returns null.

start_pos

Evaluates to an integer value. SUFFIX searches the string specified in the first argument starting at that position. A value of 1 indicates the first character of the string.

char_expression

Evaluates to a single character. SUFFIX returns the substring that begins with that character. If SUFFIX does not find the character after *start_pos*, it returns null. If the expression evaluates to more than one character, SUFFIX ignores all but the first character.

EXAMPLE

This example illustrates two ways to use the SUFFIX function:

```
SELECT C1, C2, SUFFIX(C1, 6, '.') FROM T1;
C1      C2 SUFFIX(C1,6,.
--      -- -------------
test.pref  .
pref.test  s

2 records selected


SELECT C1, C2, SUFFIX(C1, 1, C2) FROM T1;

C1      C2 SUFFIX(C1,1,C
--      -- -------------
test.pref  . pref
pref.test  s  t

2 records selected
```

NOTE

- Each *char_expression* and the result may contain multi-byte characters. The *start_pos* argument specifies the character position, not a byte position. Character comparisons are case-sensitive and are determined by sort weights in the collation table in the database.

COMPATIBILITY

Progress Extension

SYSDATE Function

Returns the current date as a DATE value. This function takes no arguments, and the trailing parentheses are optional.

SYNTAX

```
SYSDATE [ ( ) ]
```

EXAMPLE

This example illustrates the SYSDATE function, inserting a new row into the objects table, setting the create_date column to the value of the current date:

```
INSERT INTO objects (object_owner, object_id, create_date)
     VALUES (USER, 1001, SYSDATE) ;
```

COMPATIBILITY

Progress Extension

SYSTIME Function

Returns the current time as a TIME value to the nearest second. This function takes no arguments, and the trailing parentheses are optional.

NOTE: SQL statements can refer to SYSTIME anywhere they can refer to a TIME expression.

SYNTAX

```
SYSTIME [ ( ) ]
```

EXAMPLE

This example illustrates the SYSTIME function, inserting a new row into the objects table, setting the create_time column to the value of the current time:

```
INSERT INTO objects (object_owner, object_id, create_time)
     VALUES (USER, 1001, SYSTIME) ;
```

COMPATIBILITY

Progress Extension

SYSTIMESTAMP Function

Returns the current date and time as a TIMESTAMP value. This function takes no arguments, and the trailing parentheses are optional.

SYNTAX

```
SYSTIMESTAMP [ ( ) ]
```

EXAMPLE

This example illustrates different formats for SYSDATE, SYSTIME, and SYSTIMESTAMP:

```
SELECT SYSDATE FROM test;

SYSDATE
-------
09/13/1994

1 record selected


SELECT SYSTIME FROM test;

SYSTIME
-------
14:44:07:000

1 record selected


SELECT SYSTIMESTAMP FROM test;

SYSTIMESTAMP
------------
1994-09-13 14:44:15:000

1 record selected
```

COMPATIBILITY

Progress Extension

TAN Function

Returns the tangent of *expression*.

SYNTAX

```
TAN ( expression )
```

EXAMPLE

The following example shows how to use the TAN function:

```
select tan(45 * pi()/180) 'Tangent of 45 degrees' from
     sysprogress.syscalctable;

TANGENT OF 45 DEGREES
---------------------
1.000000000000000

1 record selected
```

NOTES

- TAN takes an angle (*expression*) and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.
- *expression* specifies an angle in radians.
- *expression* must evaluate to an approximate numeric data type.
- To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

COMPATIBILITY

ODBC Compatible

TO_CHAR Function

Converts the given expression to character form and returns the result. The primary use for TO_CHAR is to format the output of date-time expressions through the *format_string* argument.

SYNTAX

```
TO_CHAR ( expresion [ , format_string ] )
```

expression

Converts to character form. It must evaluate to a value of the date or time data type to use the format_string.

format_string

Specifies the format of the output. See the "Date-format Strings" section and the "Time-format Strings" section in Chapter 2, "SQL-92 Language Elements" for more information on format strings.

SQL ignores the format string if the *expression* argument does not evaluate to a date or time.

NOTES

- The first argument to the function can be of any type.
- The second argument, if specified, must be of type CHARACTER.
- The result is of type CHARACTER.
- The *format* argument can be used only when the type of the first argument is DATE.
- If any of the argument expressions evaluates to null, the result is null.

COMPATIBILITY

Progress Extension

TO_DATE

Converts the given date literal to a date value.

SYNTAX

```
TO_DATE ( date_literal )
```

EXAMPLE

This example illustrates the TO_DATE function, returning all columns from rows in the orders table where the order_date column is earlier or equal to the date '12/31/1999':

```
SELECT *
    FROM orders
    WHERE order_date <= TO_DATE ('12/31/1999') ;
```

NOTES

- The result is of type DATE.
- Supply the date literal in any valid format.

COMPATIBILITY

Progress Extension

TO_NUMBER Function

Converts the given character expression to a number value.

SYNTAX

```
TO_NUMBER ( char_expression )
```

EXAMPLE

This example illustrates the TO_NUMBER function and the SUBSTR function:

```
SELECT *
    FROM customer
    WHERE TO_NUMBER (SUBSTR (phone, 1, 3)) = 603 ;
```

NOTES

- The argument to the function must be of type CHARACTER.
- The result is of type NUMERIC.
- If any of the argument expressions evaluates to null, the result is null.

COMPATIBILITY

Progress Extension

TO_TIME Function

Converts the given time literal to a time value.

SYNTAX

```
TO_TIME ( time_literal )
```

EXAMPLE

The following example shows how to use the TO_DATE and the TO_TIME functions:

```
SELECT * FROM orders
    WHERE order_date < TO_DATE ('05/15/1991')
    AND order_time < TO_TIME ('12:00:00') ;
```

NOTES

- The result is of type TIME.
- Supply the time literal in any valid format.

COMPATIBILITY

Progress Extension

TO_TIMESTAMP Function

Converts the given timestamp literal to a timestamp value.

SYNTAX

```
TO_TIMESTAMP ( timestamp_lit )
```

EXAMPLE

The following example shows how to use the TO_TIMESTAMP function:

```
SELECT * FROM DTEST WHERE C3 = TO_TIMESTAMP('4/18/99 10:41:19')
```

NOTES

- The result is of type TIME.
- Supply the timestamp literal in any valid format.

COMPATIBILITY

Progress Extension

TRANSLATE Function

Translates each character in *char_expression* that is in *from_set* to the corresponding character in *to_set*. The translated character string is returned as the result.

SYNTAX

```
TRANSLATE ( char_expression , from_set , to_set )
```

EXAMPLE

This example substitutes underscores for spaces in customer names.

```
SELECT TRANSLATE (customer_name, ' ', '_')
   "TRANSLATE Example" from customers;

TRANSLATE EXAMPLE
-----------------
Sports_Cars_Inc._____________________________
Mighty_Bulldozer_Inc._____________________________
Ship_Shapers_Inc._____________________________
Tower_Construction_Inc._____________________________
Chemical_Construction_Inc._____________________________
Aerospace_Enterprises_Inc._____________________________
Medical_Enterprises_Inc._____________________________
Rail_Builders_Inc._____________________________
Luxury_Cars_Inc._____________________________
Office_Furniture_Inc._____________________________

10 records selected
```

NOTES

- *char_expression*, *from_set*, and *to_set* can be any character expression.
- For each character in *char_expression*, TRANSLATE checks for the same character in *from_set*:
- If it is in *from_set*, TRANSLATE translates it to the corresponding character in *to_set* (if the character is the *n*th character in *from_set*, the *n*th character in *to_set*).
- If the character is not in *from_set* TRANSLATE does not change it.

- If *from_set* is longer than *to_set*, TRANSLATE does not change trailing characters in *from_set* that do not have a corresponding character in *to_set*.
- If either *from_set* or *to_set* is null, TRANSLATE does nothing.

COMPATIBILITY

Progress Extension

UCASE Function

Returns the result of the argument character expression after converting all the characters to uppercase. UCASE is identical to UPPER, but provides ODBC-compatible syntax.

SYNTAX

```
UCASE ( char_expression )
```

EXAMPLE

The example illustrates the UCASE function, returning columns from rows in the customer table where the last_name column, after being converted to uppercase, is equal to 'SMITH':

```
SELECT *
    FROM customer
    WHERE UCASE (last_name) = 'SMITH' ;
```

NOTES

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.
- A *char_expression* and the result may contain multi-byte characters. The uppercase conversion is determined by the case table in the *convmap* file. The default case table is BASIC.

COMPATIBILITY

ODBC Compatible

UPPER Function

Returns the result of the argument character expression after converting all the characters to uppercase.

SYNTAX

```
UPPER ( char_expression )
```

EXAMPLE

The example illustrates the UPPER function, returning columns from rows in the customer table where the last_name column, after being converted to uppercase, is equal to 'SMITH':

```
SELECT *
    FROM customer
    WHERE UPPER (last_name) = 'SMITH' ;
```

NOTES

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.
- A *char_expression* and the result may contain multi-byte characters. The uppercase conversion is determined by the case table in the `convmap` file. The default case table is BASIC.

COMPATIBILITY

SQL-92 Compatible

USER Function

Returns a character-string identifier for the user of the current transaction, as determined by the host operating system. This function takes no arguments, and the trailing parentheses are optional.

SYNTAX

```
USER [ ( ) ]
```

NOTE: SQL statements can refer to USER anywhere they can refer to a character string expression.

COMPATIBILITY

ODBC Compatible

WEEK Function

Returns the week of the year as a short integer value in the range of 1-53.

SYNTAX

```
WEEK ( time_expression )
```

EXAMPLE

The example illustrates the WEEK function. The query returns all columns from rows in the orders table where the order_date is in the fifth week of the year:

```
SELECT *
    FROM orders
    WHERE WEEK (order_date) = 5 ;
```

NOTES

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (DD) precedes the month specification (MM). See the syntax for a *date_literal* in the “Date Literals” section.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

COMPATIBILITY

ODBC Compatible

YEAR Function

Returns the year as a short integer value in the range of 0-9999.

SYNTAX

```
YEAR ( date_expression )
```

EXAMPLE

The example illustrates the YEAR function. The query returns all columns in rows in the orders table where the year in the order_date column is equal to '1992':

```
SELECT *
    FROM orders
    WHERE YEAR (order_date) = 1992;
```

NOTES

- The argument to the function must be of type DATE.
- If *date_expression* is supplied as a date literal, it can be any of the valid *date_literal* formats where the day specification (*DD*) precedes the month specification (*MM*). See the syntax for a *date_literal* in the "Date Literals" section.
- The result is of type SHORT.
- If the argument expression evaluates to NULL, the result is NULL.

COMPATIBILITY

ODBC Compatible

5

Java Stored Procedures and Triggers

This chapter describes when and how to use Java stored procedures and triggers. The elements are described in the following sections:

- Definitions of Java stored procedures and triggers
- Advantages of stored procedures
- How Progress SQL-92 interacts with Java
- Using stored procedures
- Stored procedure basics
- Using the Progress SQL-92 Java Classes
- Using triggers

See Chapter 6, "Java Class Reference," for detailed information on supported Java methods and classes.

5.1 Definitions of Java Stored Procedures and Triggers

Stored procedures and *triggers* are Java routines that are executed by SQL-92 server processes. A trigger is a procedure that is invoked automatically when certain database events occur. A stored procedure is a procedure that is explicitly invoked by a client application, another stored procedure, or a trigger procedure.

A stored procedure is a snippet of Java code embedded in an CREATE PROCEDURE statement. The Java snippet can use all standard Java features as well as use Progress SQL-supplied Java classes for processing any number of SQL statements.

A trigger is a special type of stored procedure that helps to ensure referential integrity for a database. Like stored procedures, triggers also contain Java code embedded in a CREATE TRIGGER statement and use Progress SQL Java classes. However, triggers are automatically invoked (fired) by certain SQL operations: an INSERT, UPDATE, or DELETE operation on the trigger's target table.

5.2 Advantages of Stored Procedures

Stored procedures provide a very flexible, general mechanism to store a collection of SQL-92 statements and Java program constructs that enforce business rules and perform administrative tasks in a database. The ability to write stored procedures and triggers expands the flexibility and performance of applications that access the Progress SQL-92 environment.

The following is a list of additional advantages of utilizing stored procedures:

- In a client server environment, client applications make a single request for the entire procedure, instead of one or more requests for each SQL statement in the stored procedure or trigger.
- Stored procedures and triggers are stored in compiled form as well as in source text form, so execution is faster than a corresponding SQL script would be.
- Stored procedures can implement elaborate algorithms to enforce complex business rules. The details of the procedure implementation can change without requiring changes in an application that calls the procedure.

5.3 How Progress SQL-92 Interacts with Java

Progress SQL-92 stored procedures allow the use of standard Java programming constructs along with standard SQL statements. To do this, the SQL server interacts with Java in the following ways:

- When you create a stored procedure, the SQL server processes the Java code, submits it to the Java compiler, receives the compiled result, and stores the result in the database.
- When an application calls a stored procedure, the SQL server interacts with the Java Virtual Machine (JVM) to execute the stored procedure and receive any result.

5.3.1 Creating Stored Procedures

The Java source text that makes up the body of a stored procedure is not a complete Java program, but a program fragment or snippet that the Progress SQL-92 server converts into a complete Java class when it processes a CREATE PROCEDURE statement. Creating a stored procedure involves the following general steps:

1. A client application or tool (SQL Explorer, an ODBC application, or a JDBC application) issues a CREATE PROCEDURE statement that contains the Java source text.
2. The Progress SQL-92 server adds code to the Java snippet to create a complete Java class and submits the combined code to the Java compiler.
3. Presuming there are no Java compilation errors, the Java compiler returns compiled bytecode back to the Progress SQL-92 server. If there are compilation errors, the Progress SQL-92 server passes the first error message generated by the compiler back to the application or tool that issued the CREATE PROCEDURE statement.
4. The Progress SQL-92 server stores both the Java source text and the bytecode form of the procedure in the database.

Figure 5–1 illustrates the general steps for creating a Java stored procedure.

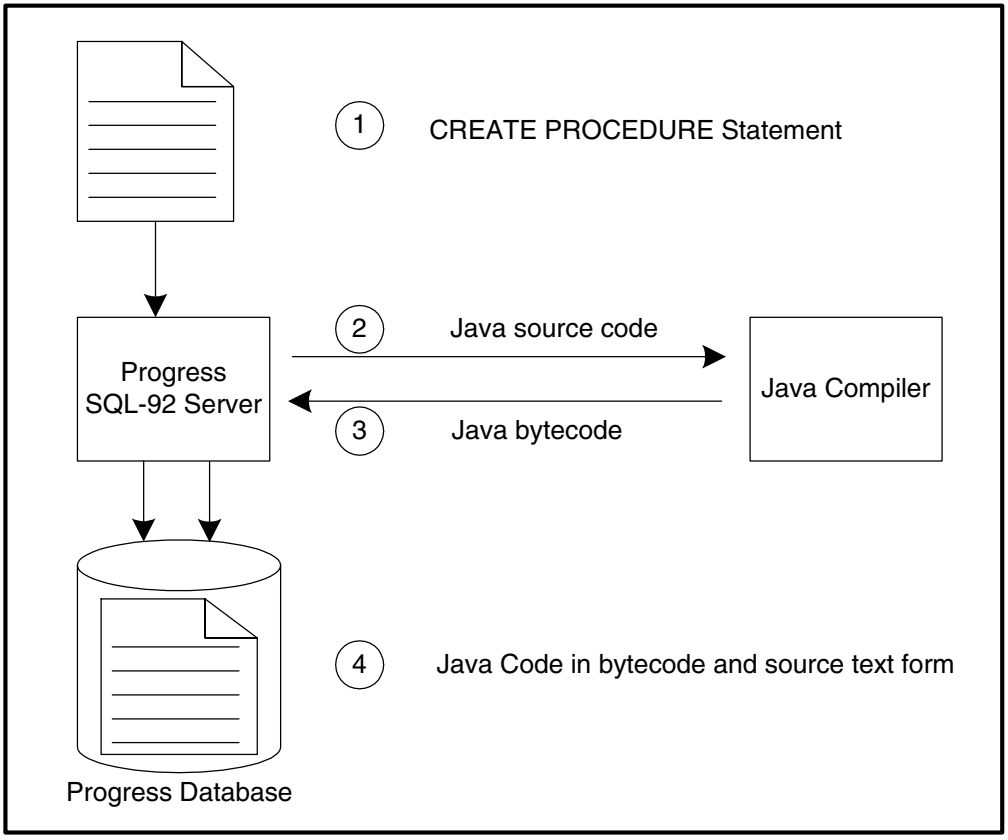


Figure 5–1: Creating Stored Procedures

5.3.2 Calling Stored Procedures

Once a stored procedure is created and stored in the database, any application or other stored procedure can execute it by calling it. You can call stored procedures from ODBC applications, JDBC applications, ESQL-92 applications, or directly from the SQL-92 Explorer.

EXAMPLE

This example shows an excerpt from an ODBC application that calls a stored procedure (order_parts) using the ODBC syntax { call *procedure_name* (*param*) }:

```
SQLUINTEGER Part_num;
SQLINTEGER  Part_numInd = 0;
// Bind the parameter.
     SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT,
          SQL_C_SLONG, SQL_INTEGER, 0, 0, &Part_num, 0, Part_numInd);
// Place the department number in Part_num.
Part_num = 318;
// Execute the statement.
SQLExecDirect(hstmt, "{call order_parts(?)}", SQL_NTS);
```

Executing a stored procedure involves the following general steps:

1. The application calls the stored procedure through its native calling mechanism. The previous example uses the ODBC call escape sequence.

2. Progress SQL-92 retrieves the compiled bytecode form of the procedure and submits it to the Java Virtual Machine for execution.

3. For every SQL statement in the procedure, the Java Virtual Machine calls Progress SQL-92.

4. Progress SQL-92 manages the interaction of the stored procedure with the database and execution of the SQL statements, and returns any result to the Java Virtual Machine.

5. The Java Virtual Machine returns result (output parameters and result sets) of the procedure to Progress SQL-92, which in turn passes them to the calling application.

Figure 5–2 illustrates the steps in executing a stored procedure.

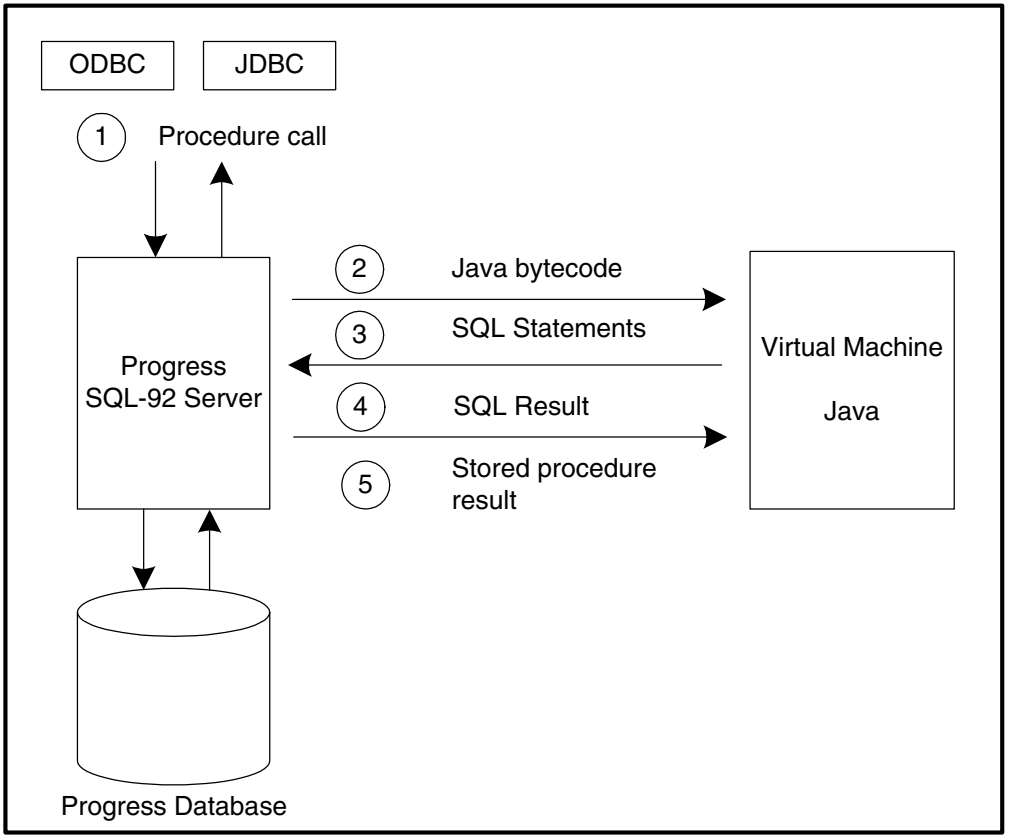


Figure 5–2: Executing Stored Procedures

5.4 Using Stored Procedures

This section describes basic stored procedure operations such as creating and deleting, and provides a tutorial on using Progress SQL-92-supplied classes. See Chapter 6, "Java Class Reference" for detailed reference information on the classes and their methods.

Stored procedures extend the SQL capabilities of a database by adding control flow through Java program constructs that enforce business rules and perform administrative tasks.

Stored procedures can take advantage of the power of Java programming features. Stored procedures can:

- Receive and return input and output parameters
- Handle exceptions
- Include any number and kind of SQL statements to access the database
- Return a procedure result set to the calling application
- Make calls to other procedures
- Use predefined and external Java classes

Progress SQL-92 provides support for SQL statements in Java through several classes. See Chapter 6, "Java Class Reference" for detailed reference information.

Table 5–1 summarizes the functionality of these Progress SQL-92-supplied classes.

Table 5–1: Summary of Progress SQL-92 Java Classes

Functionality	Progress SQL-92 Java Class
Immediate (one-time) execution of SQL statements	SQLIStatement
Prepared (repeated) execution of SQL statements	SQLPStatement
Retrieval of SQLPrepared (repeated) execution of SQL statements result sets	SQLCursor
Returning a procedure result set to the application	DhSQLResultSet
Exception handling for SQL statements	DhSQLException

5.4.1 Stored Procedure Basics

This section discusses how to get started writing stored procedures.

The SQL CREATE PROCEDURE statement provides the basic framework for stored procedures. Use the CREATE PROCEDURE statement to submit a Java code snippet that will be compiled and stored in the database.

This is syntax for the CREATE PROCEDURE statement.

SYNTAX

```
CREATE PROCEDURE [ owner_name.]procname
   ( [ parameter_decl [ , ... ] ] )
   [ RESULT ( column_name data_type [ , ... ] ) ]
   [ IMPORT
      java_import_clause ]
   BEGIN
      java_snippet
   END
```

parameter_decl

SYNTAX

```
{ IN | OUT | INOUT } parameter_name data_type
```

5.4.2 What Is a Java Snippet?

The core of the stored procedure is the *java_snippet*. The snippet contains a sequence of Java statements. When it processes a CREATE PROCEDURE statement, Progress SQL-92 adds header and footer "wrapper" code to the Java snippet. This wrapper code:

- Declares a class with the name *username_procname_SP*, where *username* is the user name of the database connection that issued the CREATE PROCEDURE statement and ***procname*** is the name supplied in the CREATE PROCEDURE statement.
- Declares a method within that class that includes the Java snippet. When an application calls the stored procedure, the SQL server calls the Java virtual machine to invoke the method of the *username_procname_SP* class.

5.4.3 Structure of Stored Procedures

There are two parts to any stored procedure:

- The *procedure specification* provides the name of the procedure and can include other optional clauses:
  - Parameter declarations
  - Procedure result set declaration
  - Import clause
- The *procedure body* contains the Java code that executes when an application invokes the procedure.

EXAMPLE

A simple stored procedure requires the procedure name in the specification, and a statement requiring no parameters in the body. The procedure in this example assumes the existence of a table named HelloWorld, and inserts a quoted string into that table:

```
CREATE PROCEDURE HelloWorld ()
BEGIN
    SQLIStatement Insert_HelloWorld = new SQLIStatement
    ("INSERT INTO HelloWorld(fld1) values ('Hello World!')");
END
```

EXAMPLE

Subsequently, from SQL Explorer you can execute the procedure like this:

```
SQLExplorer> CREATE TABLE helloworld (fld1 CHAR(100));
SQLExplorer> CALL HelloWorld();
0 records returned

SQLExplorer> SELECT * FROM helloworld;
FLD1

----


Hello World!

1 record selected
```

The procedure specification can also contain other clauses:

- *Parameter declarations* specify the name and type of parameters that the calling application will pass and receive from the procedure. Parameters can be input, output, or both.
- The *procedure result set declaration* details the names and types of fields in a result set the procedure generates. The result set is a set of rows that contain data generated by the procedure. If a procedure retrieves rows from a database table, for instance, it can store the rows in a result set for access by applications and other procedures. Note that the names specified in the result-set declaration are not used within the stored procedure body. Instead, methods of the Progress SQL-92 Java classes refer to fields in the result set by ordinal number, not by name.
- The *import clause* specifies which packages the procedure needs from the Java core API. By default, the Java compiler imports the `java.lang` package. The IMPORT clause must list any other packages the procedure uses. Progress SQL-92 automatically imports the packages it requires.

EXAMPLE

The following example shows a more complex procedure specification that contains these elements:

```
CREATE PROCEDURE new_sal (
    IN  deptnum    INTEGER,
    IN  pct_incr   INTEGER
)
RESULT  (
    empname  CHAR (20),
    oldsal  NUMERIC,
    newsal  NUMERIC
)
IMPORT
    import java.dbutils.SequenceType;

BEGIN
   .
   .
   .
END
```

5.4.4 Writing Stored Procedures

Use any text editor to write the CREATE PROCEDURE statement and save the source text as a text file. That way, you can easily modify the source text and try again if it generates syntax or Java compilation errors.

EXAMPLE

From the command prompt, you can invoke SQL Explorer and submit the file containing the CREATE PROCEDURE statement as an input script, as shown in the following example:

```
$ sqlexp -infile hello_world_script.sql example_db
```

From the command prompt, you can invoke SQL Explorer and submit the file containing the CREATE PROCEDURE statement as an input script, as shown in the following example:

```
-- File name: hello_world_script.sql
-- Purpose: Illustrate a CREATE PROCEDURE statement.
@echo true;
@autocommit true;
CREATE PROCEDURE HelloWorld ()

BEGIN
     SQLIStatement Insert_HelloWorld = new SQLIStatement (
     "INSERT INTO HelloWorld(fld1) values ('Hello World!')");
     Insert_HelloWorld.execute();
END
;
COMMIT WORK;
```

The Java snippet within the CREATE PROCEDURE statement does not execute as a standalone program. Instead, it executes in the context of an application call to the method of the class created by the SQL server. This characteristic has the following implications:

- It is meaningless for a snippet to declare a *main* method, since it will never be executed.
- If the snippet declares any classes, it must instantiate them within the snippet to invoke their methods.
- The SQL server redirects the standard output stream to a file. This means method invocations such as System.out.println will not display messages on the screen, but instead writes them to that file.

5.4.5 Invoking Stored Procedures

How applications call stored procedures depends on their environment.

From ODBC

From ODBC, applications use the ODBC call escape sequence:

SYNTAX

```
{ CALL proc_name [ ( parameter [ , ... ] ) ] } ;
```

Use parameter markers (question marks used as placeholders) for input or output parameters to the procedure. You can also use literal values for input parameters only. Progress SQL-92 stored procedures do not support return values in the ODBC escape sequence. See the *Microsoft ODBC Programmer's Reference*, Version 3.0, for more information on calling procedures from ODBC applications.

Embed the escape sequence in an ODBC SQLExecDirect call to execute the procedure.

EXAMPLE

This example shows a call to a stored procedure named order_parts that passes a single input parameter using a parameter marker:

```
SQLUINTEGER Part_num;
SQLINTEGER  Part_numInd = 0;

// Bind the parameter.
     SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT,
     SQL_C_SLONG, SQL_INTEGER,
     0, 0, &Part_num, 0, Part_numInd);

// Place the department number in Part_num.
Part_num = 318;
// Execute the statement.
SQLExecDirect(hstmt, "{ call order_parts(?) } ", SQL_NTS);
```

From JDBC

The JDBC call escape sequence is the same as in ODBC:

SYNTAX

```
{ CALL proc_name [ ( parameter [ , ... ] ) ] } ;
```

Embed the escape sequence in a JDBC CallableStatement.prepareCall method invocation.

EXAMPLE

This example shows the JDBC code parallel to the ODBC code excerpt shown in the previous example:

```
try
{
     CallableStatement statement;
     int Part_num = 318;

     // Associate the statement with the procedure call
     // (conn is a previously-instantiated connection object)
     statement = conn.prepareCall("{call order_parts(?)}");

     // Bind the parameter.
     statement.setInt(1, Part_num);

     // Execute the statement.
     statement.execute();
}
```

From SQL Explorer

From SQL Explorer, issue the SQL CALL statement.

EXAMPLE

This example shows the CALL statement that invokes the order_parts stored procedure, using a literal value instead of a parameter marker:

```
CALL order_parts (318);
```

5.4.6 Modifying and Deleting Stored Procedures

There is no ALTER PROCEDURE statement. This means that to modify a procedure, you must drop and recreate it. To recreate the procedure, you need the original source of the CREATE PROCEDURE statement. Before you drop the procedure, decide which approach you will use to recreate it:

- If you kept the original procedure definition in an SQL script file, edit the file and resubmit it through SQL Explorer.
- Query system tables to extract the source of the CREATE PROCEDURE statement to a file.

The SQL DROP PROCEDURE statement deletes stored procedures from the database. Exercise care in dropping procedures, since any procedure that calls the dropped procedure will raise an error condition when the now nonexistent stored procedure is invoked.

5.4.7 Stored Procedure Security

- To create a stored procedure, a user must have RESOURCE or DBA privileges.
- The DBA privilege entitles a user to execute any stored procedure.
- The DBA privilege entitles a user to drop any stored procedure.
- The owner of a stored procedure is given EXECUTE privilege on that procedure at creation time, by default.
- The privileges on a procedure can be granted to another user or to public either by the owner of that procedure or by the DBA.
- Stored procedures are executed with the definer's rights, not the invoker's. In other words, when a procedure is being executed on behalf of a user with EXECUTE privilege on that procedure, for the objects that are accessed by the procedure, the procedure owner's privileges are checked and not the user's. This enables a user to execute a procedure successfully even when the user does not have the privileges to directly access the objects that are accessed by the procedure, so long as the user has EXECUTE privilege on the procedure.

5.5 Using the Progress SQL-92 Java Classes

This section describes how you use the Progress SQL-92 Java classes to issue and process SQL statements in Java stored procedures.

To process SQL statements in a stored procedure, you must know whether the SQL statement generates output (in other words, if the statement is a query) or not. SELECT statements, for example, generate results: they retrieve data from one or more database tables and return the results as rows in a table.

Whether a statement generates such an SQL result set determines which Progress SQL-92 Java classes you should use to issue it:

- To issue SQL statements that do not generate results (such as INSERT, GRANT, or CREATE), use the SQLIStatement class for one-time execution, or the SQLPStatement class for repeated execution.
- To issue SQL statements that generate results (SELECT and, in some cases, CALL), use the SQLCursor class to retrieve rows from a database or another procedure's result set.

In either case, if you want to return a result set to the application, use the DhSQLResultSet class to store rows of data in a procedure result set. You must use DhSQLResultSet methods to transfer data from an SQL result set to the procedure result set for the calling application to process it. You can also use DhSQLResultSet methods to store rows of data generated internally by the procedure.

In addition, Progress SQL-92 provides the DhSQLException class so procedures can process and generate Java exceptions through the try, catch, and throw constructs.

5.5.1 Passing Values to SQL-92 Statements

Stored procedures must be able to pass and receive values from SQL statements they execute. They do this through the setParam and getValue methods.

The setParam Method: Pass Input Values to SQL Statements

The setParam method sets the value of an SQL statement's parameter marker to the specified value (a literal value, a procedure variable, or a procedure input parameter).

The setParam method takes two arguments. This is the syntax for setParam:

SYNTAX

```
setParam ( marker_num , value ) ;
```

marker_num

Specifies the ordinal number of the parameter marker in the SQL statement that is to receive the value as an integer. 1 denotes the first parameter marker, 2 denotes the second, *n* denotes the *nth*.

value

Specifies a literal, variable name, or input parameter that contains the value to be assigned to the parameter marker.

EXAMPLE

This example shows an excerpt from a stored procedure that uses setParam to assign values from two procedure variables to the parameter markers in an SQL INSERT statement. When the procedure executes, it substitutes the value of the cust_number procedure variable for the first parameter marker and the value of the cust_name variable for the second parameter marker:

```
SQLIStatement insert_cust = new SQLIStatement (
     "INSERT INTO customer VALUES (?,?) ");
     insert_cust.setParam (1, cust_number);
     insert_cust.setParam (2, cust_name);
```

The getValue Method: Pass Values from SQL Result Sets to Variables

The getValue method of the SQLCursor class assigns a single value from an SQL result set (returned by an SQL query or another stored procedure) to a procedure variable or output parameter.

SYNTAX

```
getValue ( col_num , sql_data_type ) ;
```

col_num

Specifies the desired column of the result set as integer. getValue retrieves the value in the currently fetched record of the column denoted by *col_num*. '1' denotes the first column of the result set, '2' denotes the second, *n* denotes the *nth*.

sql_data_type

Specifies the corresponding SQL data type. For a complete list of appropriate data types, refer to Table 5–2.

EXAMPLE

This example shows how the getValue() method works. This method returns a java object that must be cast to the corresponding SQL data type :

```
cnum = (Integer) NEWROW.getValue(1, INTEGER);
cnname = (String) NEWROW.getValue(1, CHARACTER);
```

5.5.2 Passing Values to and from Stored Procedures: Input and Output Parameters

Applications need to pass and receive values from the stored procedures they call. They do this through input and output parameters declared in the procedure specification.

Applications can pass and receive values from stored procedures using input and output parameters declared in the stored procedure specification. When it processes the CREATE PROCEDURE statement, the SQL server declares Java variables of the same name. This means the body of the stored procedure can refer to input and output parameters as if they were Java variables declared in the body of the stored procedure.

Procedure result sets are another way for applications to receive output values from a stored procedure. Procedure result sets provide output in a row-oriented tabular format. See the "Returning a Procedure Result Set to Applications: The RESULT Clause and DhSQLResultSet" section for more information.

Parameter declarations include the parameter type (IN, OUT, or INOUT), the parameter name, and SQL data type See the "Implicit Data Type Conversion Between SQL-92 and Java Types" section for details of how SQL data types map to Java data types.

EXAMPLE

Declare input and output parameters in the specification section of a stored procedure, as shown in the following example:

```
CREATE PROCEDURE order_entry (
     IN  cust_name    CHAR(20),
     IN  item_num     INTEGER,
     IN  quantity     INTEGER,
     OUT status_code  INTEGER,
     INOUT order_num  INTEGER
)
```

When the order_entry stored procedures executes, the calling application passes values for the cust_name, item_num, quantity, and order_num input parameters. The body of the procedure refers to them as Java variables. Similarly, Java code in the body of order_entry processes and returns values in the status_code and order_num output parameters.

5.5.3 Implicit Data Type Conversion Between SQL-92 and Java Types

When the SQL server creates a stored procedure, it converts the type of any input and output parameters. See the "Passing Values to SQL-92 Statements" section.

The *java.lang* package, part of the Java core classes, defines classes for all the primitive Java types that "wrap" values of the corresponding primitive type in an object. The SQL server converts the SQL data types declared for input and output parameters to one of these wrapper types, as shown in Table 5–2.

Be sure to use wrapper types when declaring procedure variables to use as arguments to the getValue, setParam, and set methods. These methods take objects as arguments and will generate compilation errors if you pass a primitive type to them. See also the example at the end of the SQLCursor section i n Chapter 6, "Java Class Reference."

EXAMPLE

The following example illustrates the use of the Java wrapper type Long for a SQL type BIGINT:

```
CREATE PROCEDURE proc1(INOUT f1 char(50), INOUT f2 BIGINT)

BEGIN
   f1 = new String("new rising sun");
   f2 = new Java.math.BigInteger("999");
END

CREATE PROCEDURE proc2()

BEGIN
   String in1 = new String("String type");
   String out1 = new String();
   Long out2 = new Long("0");

  SQLCursor call_proc = new SQLCursor("call proc1(?, ?)");
   call_proc.setParam(1,in1);
   // In setParam you can use either String or String type
   // for SQL types CHAR, and VARCHAR

   call_proc.setParam(2,new Long("111"));
   call_proc.open();

   out1 = (String)call_proc.getParam(1,CHAR);

   // getParam requires String type for CHAR
     out2 = (Long)call_proc.getParam(2,BIGINT);

   call_proc.close();
END
```

When the SQL server submits the Java class it creates from the stored procedure to the Java compiler, the compiler checks for data-type consistency between the converted parameters and variables you declare in the body of the stored procedure.

To avoid type mismatch errors, use the data-type mappings shown in Table 5–2 for declaring parameters and result-set fields in the procedure specification and the Java variables in the procedure body:

Table 5–2: Mapping Between SQL-92 and Java Data Types

SQL Type	Java Methods	Java Wrapper Type
CHAR, VARCHAR	all	String
CHAR, VARCHAR	set, setParam	String
NUMERIC	all	java.math.BigDecimal
DECIMAL	all	java.math.BigDecimal
BIT	all	Boolean
TINYINT	all	Byte[1]
SMALLINT	all	Integer
INTEGER	all	Integer
REAL	all	Float
FLOAT	all	Double
DOUBLE PRECISION	all	Double
BINARY	all	Byte[ ]
VARBINARY	all	Byte[ ]
DATE	all	java.sql.Date
TIME	all	java.sql.Time
TIMESTAMP	all	java.sql.Timestamp

5.5.4 Executing an SQL-92 Statement

If an SQL-92 statement does not generate a result set, stored procedures can execute the statement in one of two ways:

- Immediate execution. Using methods of the SQLIStatement class, the procedure executes a statement once.
- Prepared execution. Using methods of the SQLPStatement class, the procedure prepares a statement for multiple executions in a procedure loop.

Table 5–3 shows the SQL statements that do not generate result sets. You can execute these statements in a stored procedure using either the SQLIStatement or the SQLPStatement class.

Table 5–3: Executable SQL-92 Statements

ALTER USER	CREATE INDEX
CREATE PROCEDURE	CREATE SYNONYM
CREATE TABLE	CREATE TRIGGER
CREATE USER	CREATE VIEW
DELETE	DROP INDEX
DROP PROCEDURE	DROP TABLE
DROP TRIGGER	DROP USER
DROP VIEW	GRANT
INSERT	REVOKE
UPDATE	UPDATE STATISTICS

Immediate Execution

Use immediate execution when a procedure needs to execute an SQL statement only once.

EXAMPLE

This stored procedure in this sample script inserts a row in a table. The constructor for SQLIStatement takes the SQL INSERT statement as its only argument. In this example, the statement includes five parameter markers:

TeamProc.sql

```
------------------------------------------------------------------------
-- File Name: TeamProc.sql
--
-- Purpose:  Creates the insert_team stored procedure
--           in the current database.
--
-- Requirements:
--           (1) Database must be created
--           (2) Database must be running
--           (3) Submit this script to SQL Explorer as an input file
--               (character mode) or as a run file (GUI mode).
--
-- Revision History:
--
-- Date               Author                    Change
-- ----               ------                    ------
--
-- 11/99              DB-DOC                    Created, V9.1A
------------------------------------------------------------------------

CREATE PROCEDURE insert_team(
     IN  empnum      INTEGER not null,
     IN  FirstName   VARCHAR(30) not null,
     IN  LastName    VARCHAR(50) not null,
     IN  State       VARCHAR(50) not null,
     IN  Sport       CHAR(20)
) ;

BEGIN
     SQLIStatement insert_team = new SQLIStatement (
        "INSERT INTO team (empnum, FirstName, LastName, State, Sport)
          VALUES ( ?,?,?,?,? ) ");
     insert_team.setParam (1, empnum);
     insert_team.setParam (2, FirstName);
     insert_team.setParam (3, LastName);
     insert_team.setParam (4, State);
     insert_team.setParam (5, Sport) ;
     insert_team.execute ();
END

COMMIT WORK ;
```

Prepared Execution

Use prepared execution when you need to execute the same SQL statement repeatedly. Prepared execution avoids the overhead of creating multiple SQLIStatement objects for a single statement.

There is an advantage to prepared execution when you execute the same SQL statement from within a loop. Instead of creating an object with each iteration of the loop, prepared execution creates an object once and supplies input parameters for each execution of the statement.

Once a stored procedure creates an SQLPStatement object, you can execute the object multiple times, supplying different values for each execution.

EXAMPLE

This code fragment extends the previous example to use prepared execution:

```
CREATE PROCEDURE prepared_insert_customer (
   IN  cust_number INTEGER,
   IN  cust_name   CHAR(20)
)

BEGIN
   SQLPStatement p_insert_cust = new SQLPStatement (
   "INSERT INTO customer VALUES (?,?) ");
   .
   .
   .
int i;
for (i = 0; i < new_custs.length; i++)
{
   p_insert_cust.setParam (1, new_custs[i].cust_number);
   p_insert_cust.setParam (2, new_custs[i].cust_name);
   p_insert_cust.execute ();
}
END
```

5.5.5 Retrieving Data: the SQLCursor Class

Methods of the SQLCursor class let stored procedures retrieve rows of data. When stored procedures create an object from the SQLCursor class, they pass as an argument an SQL statement that generates a result set. The SQL statement is either a SELECT or CALL statement:

- A SELECT statement queries the database and returns data that meets the criteria specified by the query expression in the SELECT statement.

- A CALL statement invokes another stored procedure that returns a result set specified by the RESULT clause of the CREATE PROCEDURE statement.

Either way, once the procedure creates an object from the SQLCursor class, the processing of result sets follows the same steps:

1. Open the cursor with the SQLCursor.Open method.
2. Check whether there are any records in the result set with the SQLCursor.Found method.
3. If there are records in the result set, loop through the result set:
   - Try to fetch a record with the SQLCursor.Fetch method.
   - Check whether the fetch returned a record with the SQLCursor.Found method.
   - If the fetch operation returned a record, assign values from the result-set record's fields to procedure variables or procedure output parameters with the SQLCursor.getValue method.
   - Process the data.
   - If the fetch operation did not return a record, exit the loop.
4. Close the cursor with the SQLCursor.close method.

EXAMPLE

This example uses SQLCursor to process the result set returned by an SQL SELECT statement:

```
CREATE PROCEDURE get_sal ()

BEGIN
    String ename = new String (20) ;
    BigDecimal esal = new BigDecimal () ;
    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.Open () ;
    empcursor.fetch ();
        while (empcursor.found ())
        {
            empcursor.getValue (1, ename);
            empcursor.getValue (2, esal);
            //  do something with the values here
        }
    empcursor.close () ;
END
```

Stored procedures also use SQLCursor objects to process a result set returned by another stored procedure. Instead of a SELECT statement, the SQLCursor constructor includes a CALL statement that invokes the desired procedure.

EXAMPLE

The following example shows an excerpt from a stored procedure that processes the result set returned by another procedure, get_customers:

```
SQLCursor cust_cursor = new SQLCursor (
   "CALL get_customers (?) ") ;
    cust_cursor.setParam (1, "NE");
    cust_cursor.Open () ;
    for (;;)
    {
        cust_cursor.Fetch ();
        if (cust_cursor.Found ())
        {
            cust_number = (Integer) cust_cursor.getValue (1, INTEGER);
            cust_name = (String) cust_cursor.getValue (2, CHAR) ;
        }
        else
                break;
    }
    cust_cursor.close () ;
```

5.5.6 Returning a Procedure Result Set to Applications: The RESULT Clause and DhSQLResultSet

The get_sal procedure in the previous example with a CREATE PROCEDURE uses the SQLCursor.getValue method to store the values of a database record in individual variables. But the procedure did not do anything with those values, and they would be overwritten in the next iteration of the loop that fetches records.

The DhSQLResultSet class provides a way for a procedure to store rows of data in a procedure result set so the rows can be returned to the application that calls it. There can only be one procedure result set in a stored procedure.

A stored procedure must explicitly process a result set to return it to the calling application:

- Declare the procedure result set through the RESULT clause of the procedure specification.
- Populate the procedure result set in the body of the procedure using the methods of the DhSQLResultSet class.

When the SQL server creates a Java class from a CREATE PROCEDURE statement that contains the RESULT clause, it implicitly instantiates an object of type DhSQLResultSet, and calls it SQLResultSet. Invoke methods of the SQLResultSet instance to populate fields and rows of the procedure result set.

EXAMPLE

The next example extends the get_sal procedure to return a procedure result set. For each row of the SQL result set assigned to procedure variables, the procedure:

- Assigns the current values in the procedure variables to corresponding fields in the procedure result set with the DhSQLResultSet.Set method
- Inserts a row into the procedure result set with the DhSQLResultSet.Insert method

\

```
CREATE PROCEDURE get_sal2 ()
RESULT (
    empname CHAR(20),
    empsal   NUMERIC
)
BEGIN
    String ename = new String (20) ;
    BigDecimal esal = new BigDecimal () ;
    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.Open () ;
    do
    {
        empcursor.Fetch ();
        if (empcursor.found ())
        {
               ename = (String) empcursor.getValue (1, CHAR);
               esal = (BigDecimal) empcursor.getValue (2, NUMERIC);
              // NUMERIC and DECIMAL are synonyms
               SQLResultSet.Set (1, ename);
               SQLResultSet.Set (2, esal);
               SQLResultSet.Insert ();
        }
    } while (empcursor.found ()) ;
    empcursor.close () ;
END
```

5.5.7 Handling Null Values

Stored procedures routinely need to set and detect null values.

- Stored procedures might need to set the values of SQL statement input parameters or procedure result fields to null.
- Stored procedures must check if the value of a field in an SQL result set is null before assigning it through the SQLCursor.getValue method. The SQL server generates a run-time error if the result-set field specified in getValue is null.

Setting SQL Statement Input Parameters and Procedure Result Set Fields to Null

Both the setParam method and set method take objects as their value arguments. For SQL statement input parameters; see the "Using the Progress SQL-92 Java Classes" section. For procedure result set fields, see the "Returning a Procedure Result Set to Applications: The RESULT Clause and DhSQLResultSet" section. You can pass a null reference directly to the method or pass a variable which has been assigned the null value.

EXAMPLE

This example shows using both techniques to set an SQL input parameter to null:

```
CREATE TABLE t1 (
     c1 INTEGER,
     c2 INTEGER,
     c3 INTEGER)

CREATE PROCEDURE test_nulls( )
BEGIN
     Integer pvar_int ;
     pvar_int = null ;

     SQLIStatement insert_t1 = new SQLIStatement
     ( "INSERT INTO t1 (c1, c2, c3) values (?,?,?) ");

     // Set to non-null value
     insert_t1.setParam(1, new Integer(1));

     // Set directly to null
     insert_t1.setParam(2, null);

     // Set indirectly to null
     insert_t1.setParam(3, pvar_int);

     insert_t1.execute();
END
```

Assigning Null Values from SQL Result Sets: The SQLCursor.wasNULL Method

If the value of the field argument to the SQLCursor.getValue method is null, the SQL server returns a run-time error.

EXAMPLE

This example illustrates the error returned when the argument to SQLCursor.getValue is null:

```
(error(-20144): Null value fetched.)
```

This means you must always check whether a value is null before attempting to assign a value in an SQL result set to a procedure variable or output parameter. The SQLCursor class provides the wasNULL method for this purpose.

The SQLCursor.wasNULL method returns TRUE if a field in the result set is null. It takes a single integer argument that specifies which field of the current row of the result set to check.

EXAMPLE

The next example illustrates using the wasNULL method:

```
CREATE PROCEDURE test_nulls2( )
RESULT ( res_int1 INTEGER ,
         res_int2 INTEGER ,
         res_int3 INTEGER )
BEGIN
     Integer pvar_int1      = new Integer(0);
     Integer pvar_int2      = new Integer(0);
     Integer pvar_int3      = new Integer(0);
     SQLCursor select_t1 = new SQLCursor
     ( "SELECT c1, c2, c3 from t1" );
     select_t1.open();
     select_t1.fetch();
     while ( select_t1.found() )
     {
     // Assign values from the current row of the SQL result set
     // to the pvar_intx procedure variables. Must first check
     // whether the values fetched are null: if they are, must set
     // pvars explicitly to null.
     if ((select_t1.wasNULL(1)) == true)
         pvar_int1 = null;
     else
         pvar_int1 = (Integer) select_t1.getValue(1, INTEGER);
     if ((select_t1.wasNULL(2)) == true)
         pvar_int2 = null;
     else
         pvar_int2 = (Integer) select_t1.getValue(2, INTEGER);
     if ((select_t1.wasNULL(3)) == true)
         pvar_int3 = null;
     else
         pvar_int3 = (Integer) select_t1.getValue(3, INTEGER);
       else
            select_t1.getValue(3,pvar_int3);
         // Transfer the value from the procedure variables to the
         // columns of the current row of the procedure result set.
         SQLResultSet.set(1,pvar_int1);
         SQLResultSet.set(2,pvar_int2);
         SQLResultSet.set(3,pvar_int3);
         // Insert the row into the procedure result set.
         SQLResultSet.insert();
          select_t1.fetch();
     }
     // Close the SQL result set.
     select_t1.close();
END
```

5.5.8 Handling Errors

Progress SQL-92 stored procedures use standard Java try catch constructs to process exceptions.

Any errors in SQL statement execution result in the creation of a DhSQLException class object. When Progress SQL detects an error in an SQL statement, it throws an exception. The stored procedure should use try/catch constructs to process such exceptions. The getDiagnostics method of the DhSQLException class object provides a mechanism to retrieve different details of the error.

The getDiagnostics method takes a single argument whose value specifies which error message detail it returns. See Table 5–4 for explanations of the getDiagnostics error handling options.

Table 5–4: getDiagnostics Error Handling Options

Argument Value	Returns
RETURNED_SQLSTATE	The SQLSTATE returned by execution of the previous SQL statement
MESSAGE_TEXT	The condition indicated by RETURNED_SQLSTATE
CLASS_ORIGIN	Not currently used. Always returned null
SUBCLASS_ORIGIN	Not currently used. Always returned null

The error messages and the associated SQLSTATE and Progress SQL-92 error code values are documented in Appendix A, “Progress SQL-92 Reference Information.”

EXAMPLE

This example shows an excerpt from a stored procedure that uses DhSQLException.getDiagnostics:

```
try
{
     SQLIStatement insert_cust = new SQLIStatement (
     "INSERT INTO customer VALUES (1,2) ");
}
catch (DhSQLException e)
{
     errstate = e.getDiagnostics (RETURNED_SQLSTATE) ;
     errmesg  = e.getDiagnostics (MESSAGE_TEXT) ;
          .
          .
          .
}
```

Stored procedures can also throw their own exceptions by instantiating a DhSQLException object and throwing the object when the procedure detects an error in execution. The conditions under which the procedure throws the exception object are completely dependent on the procedure.

EXAMPLE

This example illustrates using the DhSQLException constructor to create an exception object called excep. It then throws the excep object under all conditions:

```
CREATE PROCEDURE sp1_02()
BEGIN
// raising exception
     DhSQLException excep =
        new DhSQLException(666,new String("Entered the tst02 procedure"));
     if (true)
     throw excep;
END
```

5.5.9 Calling Stored Procedures from Stored Procedures

Stored procedures and triggers can call other stored procedures. Nesting procedures lets you take advantage of existing procedures. Instead of rewriting the code, procedures can simply issue CALL statements to the existing procedures.

Another use for nesting procedures is to assemble result sets generated by queries on different databases into a single result set. With this technique, the stored procedure processes multiple SELECT statements through multiple instances of the SQLCursor class. For each of the instances, the procedure uses the DhSQLResultSet class to add rows to the result set returned by the procedure.

Stored Procedure Parameter Requirements and Usage

When one stored procedure is calling another stored procedure, the following requirements **must** be met for using the three parameter types in order to properly allocate the SQLDA structure to the correct size:

- An IN parameter calls only the SetParam function.
- An OUT parameter calls only the RegisterOutParam funciton.
- An INOUT parameter calls both the SetParam and RegisterOutParam functions in any order.

5.5.10 INOUT and OUT Parameters When One Java Stored Procedure Calls Another

If an OUT or INOUT parameter is of data type CHARACTER, then getParam() returns a Java String Object. You must declare a procedure variable of type String, and explicitly cast the value returned by getParam to type String. Before calling getParam() you must call the SQLCursor.wasNULL method to test whether the returned value is null. If getParam() is called for a null value, it raises a DhSQLException.

The getParam() method returns the value of an INOUT or OUT parameter identified by the number you specify in the *fieldIndex* parameter. getParam() returns the value as an object of the datatype you specify in the *fieldType* parameter. Since getParam() returns the result as an instance of class Object, you must explicitly cast your *inout_var* variable to the correct datatype.

See the SQLCursor.getParam section in Chapter 6, "Java Class Reference" for a complete description of syntax and parameters for the getParam method.

These are the general steps to follow when calling one Java stored procedure from another:

1. Register OUT parameters in the calling stored procedure
2. Declare Java variables in the snippet of the calling procedure
3. Invoke the other stored procedure

EXAMPLE

This example illustrates the steps required for calling one Java stored procedure from another:

```
create procedure lotusp(
IN f1 char(50),
INOUT f2 char(50),
OUT f3 char(50)
)
RESULT(f4 char(50))
BEGIN
     f2 = new String("new rising sun");
     f3 = new String("new rising lotus");
     SQLResultSet.set(1, new String("the fog - the snow - the ice"));
     SQLResultSet.insert();
END

commit work;

create procedure proc1()
BEGIN
     String inout_param = new String("sun");
     String out_param = new String();

     SQLCursor call_proc = new SQLCursor("call lotusp(?,?,?)");
     call_proc.setParam(1, new String("moon"));
     call_proc.setParam(2, inout_param);
     call_proc.registerOutParam(3, CHAR);
     //  OR you can specify the optional scale parameter
     //  call_proc.registerOutParam(3, CHAR, 15);
     call_proc.open();
     inout_param = (String)call_proc.getParam(2, CHAR);
     out_param = (String)call_proc.getParam(3, CHAR);
     call_proc.close();
END
```

5.6 Using Triggers

Triggers are a special type of stored procedure used to maintain database integrity.

Like stored procedures, triggers also contain Java code (embedded in a CREATE TRIGGER statement) and use Progress SQL-92 Java classes. However, triggers are automatically invoked (fired) by certain SQL operations (an insert, update, or delete operation) on the trigger's target table.

This section provides a general description of triggers and discusses in detail where trigger procedures differ from stored procedures. Unless otherwise noted, the material in earlier sections of this chapter also applies to triggers.

5.6.1 Trigger Basics

Use the SQL CREATE TRIGGER statement to create a trigger. This is the syntax for the CREATE TRIGGER statement.

SYNTAX

```
CREATE TRIGGER [ owner_name. ] trigname
 { BEFORE | AFTER }
 { INSERT | DELETE | UPDATE [ OF column_name [ , ... ] ] }
ON table_name
 [ { OLDROW [ , NEWROW ] | NEWROW [ OLDROW ] } ]
 [ FOR EACH { ROW | STATEMENT } ]
[ IMPORT
    java_import_clause ]
BEGIN
 java_snippet
END
```

5.6.2 Structure of Triggers

Like a stored procedure, a trigger has a specification and a body.

The body of a trigger is the same as that of a stored procedure: BEGIN and END delimiters enclosing a Java snippet. The Java code in the snippet defines the triggered action that executes when the trigger is fired.

As with stored procedures, when it processes a CREATE TRIGGER statement, Progress SQL-92 adds wrapper code to create a Java class and method that is invoked when the trigger is fired.

The trigger specification, however, is different from a stored procedure specification. It contains the following elements:

- The CREATE clause specifies the name of the trigger. Progress SQL-92 stores the CREATE TRIGGER statement in the database under trigname. It also uses trigname in the name of the Java class that Progress SQL-92 declares to wrap around the Java snippet. The class name uses the format username_trigname_TP, where username is the user name of the database connection that issued the CREATE TRIGGER statement.
- The BEFORE or AFTER keywords specify the trigger action time: whether the triggered action implemented by java_snippet executes before or after the triggering INSERT, UPDATE, or DELETE statement.
- The INSERT, DELETE, or UPDATE keyword specifies which data modification command activates the trigger. If UPDATE is the trigger event, this clause can include an optional column list. Updates to any of the specified columns will activate the trigger. (Updates to other columns in the table will not activate the trigger.) If UPDATE is the triggering statement and does not include the optional column list, then the UPDATE statement must specify all the table columns in order to activate the trigger.
- The ON *table_name* clause specifies the table for which the specified trigger event activates the trigger. The ON clause cannot specify a view or a remote table.
- The optional REFERENCING clause is allowed only if the trigger also specifies the FOR EACH ROW clause. It provides a mechanism for SQL to pass row values as input parameters to the stored procedure implemented by java_snippet. The code in *java_snippet* uses the getValue method of the NEWROW and OLDROW objects to retrieve values of columns in rows affected by the trigger event and store them in procedure variables. See the "OLDROW and NEWROW Objects: Passing Values to Triggers" section for information.
- The FOR EACH clause specifies the frequency with which the triggered action implemented by java_snippet executes.
- FOR EACH ROW means the triggered action executes once for each row being updated by the triggering statement. CREATE TRIGGER must include the FOR EACH ROW clause if it also includes a REFERENCING clause.
- FOR EACH STATEMENT means the triggered action executes only once for the whole triggering statement. FOR EACH STATEMENT is the default.
- The IMPORT clause is the same as in stored procedures. It specifies standard Java classes to import.

EXAMPLE

This example shows the elements of a trigger:

```
CREATE TRIGGER BUG_UPDATE_TRIGGER
     AFTER
     UPDATE OF STATUS REPORT, PRIORITY
     ON BUG_INFO
      REFERENCING OLDROW, NEWROW
     FOR EACH ROW
     IMPORT
    import java.sql.*. ;

BEGIN
   .
   .
   .
END
```

5.6.3 Triggers Versus Stored Procedures Versus Constraints

Triggers are identical to stored procedures in many respects. There are three main differences:

- Triggers are automatic. When the trigger event (an INSERT, UPDATE, or DELETE statement) affects the specified table (and, optionally in UPDATE operations, the specified columns), the Java code contained in the body of the trigger executes. Stored procedures, on the other hand, must be explicitly invoked by an application or another procedure.
- Triggers cannot have output parameters or a result set. Since triggers are automatic, there is no calling application to process any output they might generate. The practical consequence of this is that the Java code in the trigger body cannot invoke methods of the DhSQLResultSet class.
- Triggers have limited input parameters. The only possible input parameters for triggers are values of columns in the rows affected by the trigger event. If the trigger includes the REFERENCING clause, Progress SQL-92 passes the values (either as they existed in the database or are specified in the INSERT or UPDATE statement) of each row affected. The Java code in the trigger body can use those values in its processing by invoking the getValue method of the OLDROW and NEWROW objects. See the "Using Stored Procedures" section.

The automatic nature of triggers makes them well suited for enforcing referential integrity. In this regard they are like constraints, since both triggers and constraints can help ensure that a value stored in the foreign key of a table must either be null or be equal to some value in the matching unique or primary key of another table. However, triggers differ from constraints in the following ways:

- Triggers are active, while constraints are passive. Constraints prevent updates that violate referential integrity, and triggers perform explicit actions in addition to the update operation.
- Triggers can do much more than enforce referential integrity. Because they are passive, constraints are limited to preventing updates in a narrow set of conditions. Triggers are more flexible. The next section outlines some common uses for triggers.

5.6.4 Typical Uses for Triggers

Typical uses for triggers include combinations of the following:

Cascading Deletes

A delete operation on one table causes additional rows to be deleted from other tables that are related to the first table by key values. This is an active way of enforcing referential integrity that a table constraint enforces passively.

Cascading Updates

An update operation on one table causes additional rows to be updated in other tables that are related to the first table by key values. These updates are commonly limited to the key fields themselves. This is an active way of enforcing referential integrity that a table constraint enforces passively.

Summation Updates

An update operation in one table causes an update operation in a row of another table. The second value is increased or decreased.

Automatic Archiving

A delete operation on one table creates an identical row in an archive table that is not otherwise used by the database.

5.6.5 OLDROW and NEWROW Objects: Passing Values to Triggers

The OLDROW and NEWROW objects allow SQL to pass row values as input parameters to the stored procedure in a trigger that executes once for each affected row. If the CREATE TRIGGER statement contains the REFERENCING clause, the SQL server implicitly instantiates an OLDROW or NEWROW object (or both, depending on the arguments to the REFERENCING clause) when it creates the Java class.

This allows the Java code in the snippet to use the getValue method of those objects to retrieve values of columns in rows affected by the trigger event and store them in procedure variables:

- The OLDROW object contains values of a row as it exists in the database before an update or delete operation. It is instantiated when triggers specify an UPDATE...REFERENCING OLDROW or DELETE...REFERENCING OLDROW clause. It is meaningless and not available for insert operations.

- The NEWROW object contains values of a row as specified in an INSERT or UPDATE statement. It is instantiated when triggers specify an UPDATE...REFERENCING NEWROW or INSERT...REFERENCING NEWROW clause. It is meaningless and not available for delete operations.

UPDATE is the only triggering statement that allows both NEWROW and OLDROW in the REFERENCING clause.

Triggers use the OLDROW.getValue and NEWROW.getValue methods to assign a value from a row being modified to a procedure variable. The format and arguments for getValue are the same as in other Progress SQL-92 Java classes:

SYNTAX

```
getValue ( col_num , sql_data_type ) ;
```

col_num

Specifes the integer column number of the affected row. getValue retrieves the value in the column denoted by *col_num*. '1' denotes the first column of the result set, '2' denotes the second, *n* denotes the *nth*.

sql_data_type

Specifies the corresponding SQL data type. For a complete list of appropriate data types, refer to Table 5–2.

EXAMPLE

This example shows an excerpt from a trigger that uses getValue to assign values from both OLDROW and NEWROW objects:

```
CREATE TRIGGER BUG_UPDATE_TRIGGER
AFTER UPDATE OF STATUS, PRIORITY ON BUG_INFO
REFERENCING OLDROW, NEWROW
FOR EACH ROW

IMPORT
import java.sql.* ;
BEGIN
try
{
     // column number of STATUS is 10
     String  old_status, new_status;
     old_status = (String) OLDROW.getValue(10, CHAR);
     new_status = (String) NEWROW.getValue(10, CHAR);
     if ((old_status.CompareTo("OPEN") == 0) &&
     (new_status.CompareTo("FIXED") == 0))
     {
  //  If STATUS has changed from OPEN to FIXED
  //  increment the bugs_fixed_cnt by 1 in the
  //  row corresponding to current month
  //  and current year
     SQLIStatement  update_stmt (
     " update BUG_STATUS set bugs_fixed_cnt = bugs_fixed_cnt + 1 "
     " where  month = ?  and year = ?"
     );
          .
          .
          .
```

6

Java Class Reference

This reference chapter provides information on the Progress SQL Java classes and methods. The Java Class Reference section lists all the methods in the Progress SQL Java classes and shows which classes declare them. This chapter covers the following methods:

setParam	open	wasNULL	getDiagnostics
makeNULL	close	getValue	log
execute	fetch	set	err
rowCount	found	insert	–

Subsequent sections describe each Java class and its methods in more detail. Some Java methods are common to more than one class. The Java classes are presented in this order:

- DhSQLException
- DhSQLResultSet
- SQLCursor
- SQLIStatement
- SQLPStatement

Java Class Reference

This section provides reference material on the Progress SQL Java classes and methods. This section lists all the methods in the Progress SQL Java classes and shows which classes declare them. Subsequent sections are arranged alphabetically and describe each class and its methods in more detail. Some Java methods are common to more than one class.

setParam

Sets the value of an SQL statement's input parameter to the specified value; a literal, procedure variable, or procedure input parameter. The following Java classes declare setParam:

- SQLIStatement
- SQLPStatement
- SQLCursor

makeNULL

Sets the value of an SQL statement's input parameter to null. The following Java classes declare makeNull:

- SQLIStatement
- SQLPStatement
- SQLCursor

Set a field of the currently-active row in a procedure's result set to null:

- DhSQLResultSet

execute

Executes the SQL statement. The following Java classes declare execute:

- SQLIStatement
- SQLPStatement

rowCount

Returns the number of rows deleted, inserted, or updated by the SQL statement. The following Java classes declare rowCount:

- SQLIStatement
- SQLPStatement
- SQLCursor

open

Opens the result set specified by the SELECT or CALL statement. The following Java class declares open:

- SQLCursor

close

Closes the result set specified by the SELECT or CALL statement. The following Java class declares close:

- SQLCursor

fetch

Fetches the next record in a result set. The following Java class declares fetch:

- SQLCursor

found

Checks whether a fetch operation returned to a record. The following Java class declares found:

- SQLCursor

wasNULL

Checks if the value in a fetched field is null. The following Java class declares wasNull:

- SQLCursor

getValue

Stores the value of a fetched field in the specified procedure variable or procedure output parameter. The following Java class declares getValue:

- SQLCursor

set

Sets the field in the currently active row of a procedure's result set a literal, procedure variable, or procedure input parameter. The following Java class declares set:

- DhSQLResultSet

insert

Inserts the currently active row into the result set of a procedure. The following Java class declares insert:

- DhSQLResultSet

getDiagnostics

Returns the specified detail of an error message. The following Java class declares getDiagnostics:

- DhSQLException

log

Writes a message to the log. The following Java classes inherit the log:

- SQLIStatement
- SQLPStatement
- SQLCursor
- DhSQLResult Set
- DhSQLException

err

Writes a message to the log. The following Java classes write to the log: :

- SQLIStatement
- SQLPStatement
- SQLCursor
- DhSQLResult Set
- DhSQLException

DhSQLException

Extends the general `java.lang`. exception class to provide detail about errors in SQL statement execution. Any such errors raise an exception with an argument that is an SQLException class object. The getDiagnostics() method retrieves details of the error.

Constructors

```
public DhSQLException(int ecode, String errMsg)
```

Parameters

ecode

The error number associated with the exception condition

errMsg

The error message associated with the exception condition

EXAMPLE

This example illustrates using the DhSQLException constructor to create an exception object called excep. It then throws the excep object under all conditions:

```
CREATE PROCEDURE sp1_02()
BEGIN
     // raising exception
     DhSQLException excep = new DhSQLException(666,new String
          ("Entered the tst02 procedure"));
     if (true)
     throw excep;
END
```

DhSQLException.getDiagnostics

Returns the requested detail about an exception.

Format

```
public String getDiagnostics(int diagType)
```

Returns

A string containing the information specified by the *diagType* parameter as shown in Table 6–1.

Parameters

diagType

One of the argument values listed in Table 6–1.

Table 6–1: Argument Values for DhSQLException.getDiagnostics

Argument Value	Returns
RETURNED_SQLSTATE	The SQLSTATE returned by execution of the previous SQL statement
MESSAGE_TEXT	The condition indicated by RETURNED_SQLSTATE
CLASS_ORIGIN	Not currently used. Always returns null
SUBCLASS_ORIGIN	Not currently used. Always returns null

Throws

DhSQLException

EXAMPLE

This code fragment illustrates DhSQLException.getDiagnostics:

```
try
{
     SQLIStatement insert_cust = new SQLIStatement (
     "INSERT INTO customer VALUES (1,2) ");
}
catch (DhSQLException e)
{
     errstate = e.getDiagnostics (RETURNED_SQLSTATE) ;
     errmesg  = e.getDiagnostics (MESSAGE_TEXT) ;
          .
          .
          .
}
```

DhSQLResultSet

Provides the stored procedure with a result set to return to the application that called the procedure.

The Java code in a stored procedure does not explicitly create DhSQLResultSet objects. Instead, when the SQL server creates a Java class from a CREATE PROCEDURE statement that contains a Result clause, it implicitly instantiates an object of type DhSQLResultSet, and calls it SQLResultSet.

Procedures invoke methods of the SQLResultSet instance to populate fields and rows of the result set.

Constructors

No explicit constructor

Parameters

None

Throws

DhSQLException

DhSQLResultSet.insert

Inserts the currently active row into a procedure's result set.

Format

```
public void insert()
```

Returns

None

Parameters

None

Throws

DhSQLException

EXAMPLE

This code fragment illustrates SQLResultSet.set and SQLResultSet.insert:

```
CREATE PROCEDURE get_sal2 ()
RESULT (
    empname CHAR(20),
    empsal   NUMERIC,
)
BEGIN
    String ename = new String (20) ;
    BigDecimal esal = new BigDecimal () ;
    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.Open () ;
    do
    {
        empcursor.Fetch ();
        if (empcursor.found ())
        {
                empcursor.getValue (1, ename);
                empcursor.getValue (2, esal);
            SQLResultSet.Set (1, ename);
            SQLResultSet.Set (2, esal);
            SQLResultSet.Insert ();
        }
    } while (empcursor.found ()) ;
    empcursor.close () ;
END
```

DhSQLResultSet.makeNULL

Sets a field of the currently-active row in a procedure's result set to null. This method is redundant with using the DhSQLResultSet.set method to set a procedure result-set field to null.

Format

```
public void makeNULL(int field)
```

Returns

None

Parameters

field

An integer that specifies which field of the result-set row to set to null. 1 denotes the first field in the row, 2 denotes the second, *n* denotes the *nth*.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates SQLResultSet.set and SQLResultSet.makeNULL:

```
CREATE PROCEDURE test_makeNULL2(
    IN char_in CHAR(20)
RESULT ( res_char CHAR(20) , res_vchar VARCHAR(30))

BEGIN
    SQLResultSet.set(1,char_in);
    SQLResultSet.makeNULL(2);
END
```

DhSQLResultSet.set

Sets the field in the currently-active row of a procedure's result set to the specified value (a literal, procedure variable, or procedure input parameter).

Format

public void set(int *field*, Object *val*)

Returns

None

Parameters

field

An integer that specifies which field of the result-set row to set to the value specified by val. (1 denotes the first field in the row, 2 denotes the second, and so on.)

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the field.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates SQLResultSet.Set:

```
CREATE PROCEDURE get_sal2 ()
RESULT (
     empname CHAR(20),
     empsal   NUMERIC,
)
BEGIN
     String ename = new String (20) ;
     BigDecimal esal = new BigDecimal () ;
     SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

     empcursor.Open () ;
     do
     {
        empcursor.Fetch ();
        if (empcursor.found ())
          {
               empcursor.getValue (1, ename);
               empcursor.getValue (2, esal);
               SQLResultSet.Set (1, ename);
               SQLResultSet.Set (2, esal);
               SQLResultSet.Insert ();
          }
     } while (empcursor.found ()) ;
     empcursor.close () ;
END
```

SQLCursor

Allows rows of data to be retrieved from a database or another stored procedure's result set.

Constructors

SQLCursor (String *statement*)

Parameters

statement

Generates a result set. Enclose the SQL statement in double quotes. The SQL statement is either a SELECT or CALL statement.

NOTES

- A SELECT statement queries the database and returns data that meets the criteria specified by the query expression in the SELECT statement.
- A CALL statement invokes another stored procedure that returns a result set specified by the RESULT clause of the CREATE PROCEDURE statement.

Throws

DhSQLException

EXAMPLES

The following excerpt from a stored procedure instantiates an SQLCursor object called cust_cursor that retrieves data from a database table:

```
SQLCursor empcursor = new SQLCursor ( "SELECT name, sal FROM emp " ) ;
```

The following excerpt from a stored procedure instantiates an SQLCursor object called cust_cursor that calls another stored procedure:

```
t_cursor = new SQLCursor ( "CALL get_customers (?) ") ;
```

SQLCursor.close

Closes the result set specified by a SELECT or CALL statement.

Format

```
public void close()
```

Returns

None

Parameters

None

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the getValue and close methods:

```
{
     if (cust_cursor.Found ())
     {
          cust_cursor.getValue (1, cust_number);
          cust_cursor.getValue (2, cust_name) ;
     }
     else
          break;
}
     cust_cursor.close () ;
```

SQLCursor.fetch

Fetches the next record in a result set, if there is one.

Format

```
public void fetch()
```

Returns

None

Parameters

None

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the fetch method and the getValue method:

```
for (;;)
{
     cust_cursor.Fetch ();
     if (cust_cursor.Found ())
     {
          cust_cursor.getValue (1, cust_number);
          cust_cursor.getValue (2, cust_name) ;
     }
     else
          break;
}
```

SQLCursor.found

Checks whether a fetch operation returned a record.

Format

```
public boolean found ()
```

Returns

True if the previous call to fetch() returned a record, false otherwise.

Parameters

None

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the fetch, found, and getValue methods:

```
for (;;)
{
     cust_cursor.Fetch ();
     if (cust_cursor.Found ())
     {
          cust_cursor.getValue (1, cust_number);
          cust_cursor.getValue (2, cust_name) ;
     }
     else
          break;
}
```

SQLCursor.getParam

Retrieves the values of Java OUT and INOUT parameters.

Format

```
inout_var = getParam(  int  fieldIndex,  short  fieldType   )  ;
```

Returns

OUT or INOUT variable

Parameters

inout_var

The target variable into which the value of an OUT or INOUT parameter is stored.

fieldIndex

An integer that specifies the position of the parameter in the parameter list.

fieldType

A short integer that specifies the datatype of the parameter. The allowable defined values for *fieldType* are listed in Table 6–2, grouped by category of data type.

Table 6–2: Allowable Values for *fieldType* in getParam

Character	Exact Numeric	Approximate Numeric	Date-time	Bit String
CHAR	INTEGER	REAL	DATE	BIT
CHARACTER	SMALLINT	FLOAT	TIME	BINARY
VARCHAR	TINYINT	DOUBLE	TIMESTAMP	VARBINARY
–	NUMERIC	–	–	LVARBINARY
–	DECIMAL	–	–	–

Throws

DhSQLException

NOTES

- The getParam() method returns the value of an INOUT or OUT parameter identified by the number you specify in the *fieldIndex* parameter. getParam() returns the value as an object of the datatype you specify in the *fieldType* parameter. Since getParam() returns the result as an instance of class Object, you must explicitly cast your *inout_var* variable to the correct data type.
- If the OUT or INOUT parameter is of datatype CHARACTER, then getParam returns a Java String Object. You must declare a procedure variable of type String, and explicitly cast the value returned by getParam to type String. Before calling getParam() you must call the SQLCursor.wasNULL method to test whether the returned value is null. If getParam() is called for a null value, it raises a DhSQLException.
- See the "INOUT and OUT Parameters When One Java Stored Procedure Calls Another," section in Chapter 6, "Java Class Reference" for an example of how and when to use SQLCursor.getParam().

SQLCursor.getValue

Assigns a single value from a SQL result set to a procedure variable. The single field value is the result of an SQL query or the result from another stored procedure.

Format

```
public Object getValue(  int  fieldNum,  short  fieldType  )
```

Returns

Object

Parameters

fieldNum

An integer that specifies the position of the field to retrieve from the fetched record.

fieldType

A short integer that specifies the data type of the parameter. The allowable defined values for *fieldType* are listed in Table 6–3, grouped by category of data type.

Table 6–3: Allowable Values for *fieldType* in getValue

Character	Exact Numeric	Approximate Numeric	Date-time	Bit String
CHAR	INTEGER	REAL	DATE	BIT
CHARACTER	SMALLINT	FLOAT	TIME	BINARY
VARCHAR	TINYINT	DOUBLE	TIMESTAMP	VARBINARY
–	NUMERIC	–	–	LVARBINARY
–	DECIMAL	–	–	–

Throws

DhSQLException

NOTES

- Before invoking getValue, you must test for the null condition by calling the SQLCursor.wasNULL method. If the value returned is null, you must explicitly set the target variable in the stored procedure to null.
- The getValue method returns a value from the result set identified by the number you specify in the *fieldNum* parameter. getValue returns the value as an object of the datatype you specify in the *fieldType* parameter. Since getValue returns the result as an instance of class Object, you must explicitly cast your return value to the correct datatype.
- If the returned value is of datatype CHARACTER, then getValue returns a Java String Object. You must declare a procedure variable of type String and explicitly cast the value returned by getValue to type String.

EXAMPLE

This example illustrates testing for null and invoking the Java getValue method:

```
Integer  pvar_int = new Integer(0);
String   pvar_str = new String();
SQLCursor select_t1 = new SQLCursor
     ("select int_col, char_col from T1");

Select_t1.open();
Select_t1.fetch();

while(select_t1.found())
{
     // Assign values from the current row of the SQL result set
     // to the procedure variables. First check whether
     // the values fetched are null. If null then explicitly
     // set the procedure variables to null.

     if ((select_t1.wasNULL(1)) == true)
          pvar_int = null;
     else
          pvar_int = (Integer)select_t1.getValue(1, INTEGER);
     if ((select_t1.wasNULL(2)) == true)
          pvar_str = null;
     else
          pvar_str = (String)select_t1.getValue(1, CHAR);
}
```

SQLCursor.makeNULL

Sets the value of an SQL statement's input parameter to null. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes. This method is redundant with using the setParam method to set an SQL statement's input parameter to null.

Format

```
public void makeNULL(int f)
```

Returns

None

Parameters

f

An integer that specifies which input parameter of the SQL statement string to set to null. '1' denotes the first input parameter in the statement, '2' denotes the second, *n* denotes the *nth*.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the makeNULL method:

```
CREATE PROCEDURE sc_makeNULL()
BEGIN
     SQLCursor select_btypes = new SQLCursor (
         "SELECT small_fld from sfns where small_fld = ? ");
     select_btypes.makeNULL(1);
     select_btypes.open();
     select_btypes.fetch();
   .
   .
   .
     select_btypes.close();
END
```

SQLCursor.open

Opens the result set specified by the SELECT or CALL statement.

Format

```
public void open()
```

Returns

None

Parameters

None

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the open method:

```
SQLCursor empcursor = new SQLCursor ( "SELECT name, sal FROM emp " ) ;
empcursor.Open () ;
```

SQLCursor.registerOutParam

Register OUT parameters.

Format

```
registerOutParam( int fieldIndex, short fieldType [ , short scale ] )
```

Returns

None

Parameters

fieldIndex

An integer that specifies the position of the parameter in the parameter list.

fieldType

A short integer that specifies the datatype of the parameter. The allowable defined values for *fieldType* are listed in Table 6–4, grouped by category of data type:

Table 6–4: Allowable Values for *fieldType* in registerOutParam

Character	Exact Numeric	Approximate Numeric	Date-time	Bit String
CHAR	INTEGER	REAL	DATE	BIT
CHARACTER	SMALLINT	FLOAT	TIME	BINARY
VARCHAR	TINYINT	DOUBLE	TIMESTAMP	VARBINARY
–	NUMERIC	–	–	LVARBINARY
–	DECIMAL	–	–	–

Throws

DhSQLException

See the "INOUT and OUT Parameters When One Java Stored Procedure Calls Another" section in Chapter 5, "Java Stored Procedures and Triggers" for an example of how and when to use SQLCursor.registerOutParam().

SQLCursor.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public int rowCount()
```

Returns

An integer indicating the number of rows.

Parameters

None

Throws

DhSQLException

EXAMPLE

This example uses the rowCount method of the SQLIStatement class. It nests the method invocation within SQLResultSet.set to store the number of rows affected (1, in this case) in the procedure's result set:

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs INTEGER )

BEGIN
     SQLCursor insert_test103 = new SQLIStatement (
     "INSERT INTO test103 (fld1) values (17)");
     insert_test103.execute();
     SQLResultSet.set(1,new Long(insert_test103.rowCount()));
     SQLResultSet.insert();
END
```

SQLCursor.setParam

Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public void setParam(int f, Object val)
```

Returns

None

Parameters

f

An integer that specifies which parameter marker in the SQL statement is to receive the value. '1' denotes the first parameter marker, '2' denotes the second, *n* denotes the *nth*.

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the setParam method:

```
CREATE PROCEDURE sps_setParam()

BEGIN
// Assign local variables to be used as SQL input parameter references
     Integer ins_fld_ref   = new Integer(1);
     Integer ins_small_fld = new Integer(3200);
     Integer ins_int_fld   = new Integer(21474);
     Double  ins_doub_fld  = new Double(1.797E+30);
     String  ins_char_fld  = new String("Athula");
     String  ins_vchar_fld = new String("Scientist");
     Float   ins_real_fld  = new Float(17);
     SQLIStatement insert_sfns1 = new SQLIStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );
     insert_sfns1.setParam(1,ins_fld_ref);
     insert_sfns1.setParam(2,ins_small_fld);
     insert_sfns1.setParam(3,ins_int_fld);
     insert_sfns1.setParam(4,ins_doub_fld);
     insert_sfns1.setParam(5,ins_char_fld);
     insert_sfns1.setParam(6,ins_vchar_fld);
     insert_sfns1.execute();
END
```

SQLCursor.wasNULL

Checks if the value in a fetched field is null.

Format

```
public boolean wasNULL(int field)
```

Returns

True if the field is null, false otherwise.

Parameters

field

An integer that specifies which field of the fetched record is of interest. (1 denotes the first column of the result set, 2 denotes the second, and so on.) wasNULL checks whether the value in the currently-fetched record of the column denoted by field is null.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the wasNULL method:

```
CREATE PROCEDURE test_wasNULL()

BEGIN
     int small_sp      = 0;
     SQLCursor select_btypes =
          new SQLCursor ("SELECT small_fld from sfns");
     select_btypes.open();
     select_btypes.fetch();
     if ((select_btypes.wasNULL(1)) == true)
            small_sp = null;
     else
            select_btypes.getValue(1,small_sp);
     select_btypes.close();
END
```

SQLIStatement

Allows immediate (one-time) execution of SQL statements that do not generate a result set.

Constructors

SQLIStatement (String *statement*)

Parameters

`statement`

An SQL statement that does not generate a result set. Enclose the SQL statement in double quotes.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the SQLIStatement class:

```
CREATE PROCEDURE insert_customer (
IN  cust_number INTEGER,
IN  cust_name   CHAR(20)
)

BEGIN
     SQLIStatement insert_cust = new SQLIStatement (
        "INSERT INTO customer VALUES (?,?) ");
END
```

SQLIStatement.execute

Executes the SQL statement. This method is common to the SQLIStatement and SQLPStatement classes.

Format

```
public void execute()
```

Returns

None

Parameters

None

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the setParam and execute methods:

```
CREATE PROCEDURE insert_customer (
IN  cust_number INTEGER,
IN  cust_name   CHAR(20)
)

BEGIN
     SQLIStatement insert_cust = new SQLIStatement (
            "INSERT INTO customer VALUES (?,?) ");
     insert_cust.setParam (1, cust_number);
     insert_cust.setParam (2, cust_name);
     insert_cust.execute ();
END
```

SQLIStatement.makeNULL

Sets the value of an SQL statement's input parameter to null. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes. This method is redundant with using the setParam method to set an SQL statement's input parameter to null.

Format

```
public void makeNULL(int f)
```

Returns

None

Parameters

f

An integer that specifies which input parameter of the SQL statement string to set to null. '1' denotes the first input parameter in the statement, '2' denotes the second, *n* denotes the *nth*.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the makeNULL method:

```
CREATE PROCEDURE sis_makeNULL()

BEGIN
     SQLIStatement insert_sfns1 = new SQLIStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );
     insert_sfns1.setParam(1,new Integer(66));
     insert_sfns1.makeNULL(2);
     insert_sfns1.makeNULL(3);
     insert_sfns1.makeNULL(4);
     insert_sfns1.makeNULL(5);
     insert_sfns1.makeNULL(6);
     insert_sfns1.execute();
END
```

SQLIStatement.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public int rowCount()
```

Returns

An integer indicating the number of rows.

Parameters

None

Throws

DhSQLException

EXAMPLE

This example uses the rowCount method of the SQLIStatement class. It nests the method invocation within SQLResultSet.set to store the number of rows affected (1, in this case) in the procedure's result set:

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs INTEGER )
BEGIN
     SQLIStatement insert_test103 = new SQLIStatement (
     "INSERT INTO test103 (fld1) values (17)");
     insert_test103.execute();
     SQLResultSet.set(1,new Long(insert_test103.rowCount()));
     SQLResultSet.insert();
END
```

SQLIStatement.setParam

Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public void setParam(int f, Object val)
```

Returns

None

Parameters

f

An integer that specifies which parameter marker in the SQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the setParam method:

```
CREATE PROCEDURE sps_setParam()

BEGIN
// Assign local variables to be used as
// SQL input parameter references
     Integer ins_fld_ref   = new Integer(1);
     Integer ins_small_fld = new Integer(3200);
     Integer ins_int_fld   = new Integer(21474);
     Double  ins_doub_fld  = new Double(1.797E+30);
     String  ins_char_fld  = new String("Athula");
     String  ins_vchar_fld = new String("Scientist");
     Float   ins_real_fld  = new Float(17);

     SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );

     insert_sfns1.setParam(1,ins_fld_ref);
     insert_sfns1.setParam(2,ins_small_fld);
     insert_sfns1.setParam(3,ins_int_fld);
     insert_sfns1.setParam(4,ins_doub_fld);
     insert_sfns1.setParam(5,ins_char_fld);
     insert_sfns1.setParam(6,ins_vchar_fld);
     insert_sfns1.execute();
END
```

SQLPStatement

Allows prepared (repeated) execution of SQL statements that do not generate a result set.

Constructors

SQLPStatement (String *statement*)

Parameters

statement

An SQL statement that does not generate a result set. Enclose the SQL statement in double quotes.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the SQLPStatement class:

```
SQLPStatement pstmt = new SQLPStatement ( "INSERT INTO T1 VALUES (?, ?) ") ;
```

SQLPStatement.execute

Executes the SQL statement. This method is common to the SQLIStatement and SQLPStatement classes.

Format

```
public void execute()
```

Returns

None

Parameters

None

Throws

DhSQLException

EXAMPLE

This code fragment illustrates the execute and setParam methods in the SQLPStatement class:

```
SQLPStatement pstmt = new SQLPStatement (
     "INSERT INTO T1 VALUES (?, ?) ") ;
pstmt.setParam (1, 10);
pstmt.setParam (2, 10);
pstmt.execute ();
pstmt.setParam (1, 20);
pstmt.setParam (2, 20);
pstmt.execute ();
```

SQLPStatement.makeNULL

Sets the value of an SQL statement's input parameter to null. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes. This method is redundant with using the setParam method to set an SQL statement's input parameter to null.

Format

```
public void makeNULL(int f)
```

Returns

None

Parameters

f

An integer that specifies which input parameter of the SQL statement string to set to null. (1 denotes the first input parameter in the statement, 2 denotes the second, and so on.)

Throws

DhSQLException

EXAMPLE

This code fragment illustrates SQLPStatement.makeNULL:

```
CREATE PROCEDURE sps_makeNULL()

BEGIN
     SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );
     insert_sfns1.setParam(1,new Integer(666));
     insert_sfns1.makeNULL(2);
     insert_sfns1.makeNULL(3);
     insert_sfns1.makeNULL(4);
     insert_sfns1.makeNULL(5);
     insert_sfns1.makeNULL(6);
     insert_sfns1.execute();
END
```

SQLPStatement.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public int rowCount()
```

Returns

An integer indicating the number of rows.

Parameters

None

Throws

DhSQLException

EXAMPLE

This example uses the rowCount method of the SQLPStatement class. It nests the method invocation within SQLResultSet.set to store the number of rows affected (1, in this case) in the procedure's result set:

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs INTEGER )

BEGIN
     SQLPStatement insert_test103 = new SQLPStatement (
     "INSERT INTO test103 (fld1) values (17)");
     insert_test103.execute();
     SQLResultSet.set(1,new Long(insert_test103.rowCount()));
     SQLResultSet.insert();
END
```

SQLPStatement.setParam

Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public void setParam(int f, Object val)
```

Returns

None

Parameters

f

An integer that specifies which parameter marker in the SQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

Throws

DhSQLException

EXAMPLE

This code fragment illustrates SQLPStatement.setParam:

```
CREATE PROCEDURE sps_setParam()

BEGIN
// Assign local variables to be used as
// SQL input parameter references
     Integer ins_fld_ref   = new Integer(1);
     Integer ins_small_fld = new Integer(3200);
     Integer ins_int_fld   = new Integer(21474);
     Double  ins_doub_fld  = new Double(1.797E+30);
     String  ins_char_fld  = new String("Athula");
     String  ins_vchar_fld = new String("Scientist");
     Float   ins_real_fld  = new Float(17);
     SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)
      values (?,?,?,?,?,?)" );

     insert_sfns1.setParam(1,ins_fld_ref);
     insert_sfns1.setParam(2,ins_small_fld);
     insert_sfns1.setParam(3,ins_int_fld);
     insert_sfns1.setParam(4,ins_doub_fld);
     insert_sfns1.setParam(5,ins_char_fld);
     insert_sfns1.setParam(6,ins_vchar_fld);
     insert_sfns1.execute();
END
```

A

Progress SQL-92 Reference Information

This appendix provides reference information for Progress SQL-92. Specifically, it lists:

- Progress SQL-92 Reserved Words
- Progress SQL-92 System Limits
- Progress SQL-92 Error Messages

Progress SQL-92 Reserved Words

Reserved words are keywords. You can use keywords as identifiers in SQL statements only if you delimit them with double quotation marks. If you use keywords without delimiting them, the statement generates one of the following errors:

```
error(-20003): Syntax error
error(-20049): Keyword used for a name
```

Table A–1 is a list of Progress SQL-92 reserved words.

Table A–1: Progress SQL-92 Reserved Words *(1 of 4)*

A	ABS	ACOS	ADD
ADD_MONTHS	AFTER	ALL	ALTER
AN	AND	ANY	ARRAY
AS	ASC	ASCII	ASIN
ATAN	ATAN2	AVG	BEFORE
BEGIN	BETWEEN	BIGINT	BINARY
BIND	BINDING	BIT	BY
CALL	CASCADE	CASE	CAST
CEILING	CHAR	CHAR_LENGTH	CHARACTER
CHARACTER_LENGTH	CHARTOROWID	CHECK	CHR
CLEANUP	CLOSE	CLUSTERED	COALESCE
COLGROUP	COLLATE	COMMIT	COMPLEX
COMPRESS	CONCAT	CONNECT	CONSTRAINT
CONTAINS	CONTINUE	CONVERT	COS
COUNT	CREATE	CROSS	CURDATE
CURRENT	CURSOR	CURTIME	CVAR
DATABASE	DATAPAGES	DATE	DAYNAME
DAYOFMONTH	DAYOFWEEK	DAYOFYEAR	DB_NAME

Table A–1: Progress SQL-92 Reserved Words *(2 of 4)*

DBA	DEC	DECIMAL	DECLARATION
DECLARE	DECODE	DEFAULT	DEFINITION
DEGREES	DELETE	DESC	DESCRIBE
DESCRIPTOR	DHTYPE	DIFFERENCE	DISTINCT
DOUBLE	DROP	EACH	ELSE
END	ESCAPE	EXCLUSIVE	EXEC
EXECUTE	EXISTS	EXIT	EXP
EXPLICIT	EXTENT	FETCH	FIELD FILE
FLOAT	FLOOR	FOR	FOREIGN
FOUND	FROM	FULL	GO
GOTO	GRANT	GREATEST	GROUP
HASH	HAVING	HOUR	IDENTIFIED
IFNULL	IMMEDIATE	IN	INDEX
INDEXPAGES	INDICATOR	INITCAP	INNER
INOUT	INPUT	INSERT	INSTR
INT	INTEGER	INTERFACE	INTERSECT
INTO	IS	JOIN	KEY
LAST_DAY	LCASE	LEAST	LEFT
LENGTH	LIKE	LINK	LIST
LOCATE	LOCK	LOG	LOG10
LONG	LOWER	LPAD	LTRIM
LVARBINARY	LVARCHAR	MAIN	MAX
METADATA_ONLY	MIN	MINUS	MINUTE
MOD	MODE	MODIFY	MONEY
MONTH	MONTHNAME	MONTHS_BETWEEN	NAME

Table A–1: Progress SQL-92 Reserved Words *(3 of 4)*

NATIONAL	NATURAL	NCHAR	NEWROW
NEXT_DAY	NOCOMPRESS	NOT	NOW
NOWAIT	NULL	NULLIF	NULLVALUE
NUMBER	NUMERIC	NVL	OBJECT_ID
ODBC_CONVERT	ODBCINFO	OF	OLDROW
ON	OPEN	OPTION	OR
ORDER	OUT	OUTER	OUTPUT
PCTFREE	PI	POWER	PRECISION
PREFIX	PREPARE	PRIMARY	PRIVILEGES
PROCEDURE	PUBLIC	QUARTER	RADIANS
RAND	RANGE	RAW	REAL
RECORD	REFERENCES	REFERENCING	RENAME
REPEAT	REPLACE	RESOURCE	RESTRICT
RESULT	RETURN	REVOKE	RIGHT
ROLLBACK	ROW	ROWID	ROWIDTOCHAR
ROWNUM	RPAD	RTRIM	SEARCHED_CASE
SECOND	SECTION	SELECT	SERVICE
SET	SHARE	SHORT	SIGN
SIMPLE_CASE	SIN	SIZE	SMALLINT
SOME	SOUNDEX	SPACE	SQL
SQL_BIGINT	SQL_BINARY	SQL_BIT	SQL_CHAR
SQL_DATE	SQL_DECIMAL	SQL_DOUBLE	SQL_FLOAT
SQL_INTEGER	SQL_LONGVARBINARY	SQL_LONGVARCHAR	SQL_NUMERIC
SQL_REAL	SQL_SMALLINT	SQL_TIME	SQL_TIMESTAMP
SQL_TINYINT	SQL_TSI_DAY	SQL_TSI_FRAC_SECOND	SQL_TSI_HOUR

Table A–1: Progress SQL-92 Reserved Words *(4 of 4)*

SQL_TSI_MINUTE	SQL_TSI_MONTH	SQL_TSI_QUARTER	SQL_TSI_SECOND
SQL_TSI_WEEK	SQL_TSI_YEAR	SQL_VARBINARY	SQL_VARCHAR
SQLERROR	SQLWARNING	SQRT	START
STATEMENT	STATISTICS	STOP	STORAGE_ATTRIBUTES
STORAGE_MANAGER	STORE_IN_PROGRESS	SUBSTR	SUBSTRING
SUFFIX	SUM	SUSER_NAME	SYNONYM
SYSDATE	SYSTIME	SYSTIMESTAMP	TABLE
TAN	THEN	TIME	TIMEOUT
TIMESTAMP	TIMESTAMPADD	TIMESTAMPDIFF	TINYINT
TO	TO_CHAR	TO_DATE	TO_NUMBER
TO_TIME	TO_TIMESTAMP	TPE	TRANSACTION
TRANSLATE	TRIGGER	TYPE	UCASE
UID	UNION	UNIQUE	UNSIGNED
UPDATE	UPPER	USER	USER_ID
USER_NAME	USING	UUID	VALUES
VARBINARY	VARCHAR	VARIABLES	VARYING
VERSION	VIEW	WEEK	WHEN
WHENEVER	WHERE	WITH	WORK
YEAR	–	–	–

Progress SQL-92 System Limits

Table A–2 is a list of the maximum sizes for various attributes of the Progress SQL-92 database environment, and for elements of SQL-92 queries addressed to this environment. Attribute names may be seen in ESQL/C.

Table A–2: Progress SQL-92 System Limits *(1 of 2)*

Attribute	Name	Value
Maximum number of procedure arguments in an SQL CALL statement	TPE_MAX_PROC_ARGS	50
Maximum length of an SQL statement	TPE_MAX_SQLSTMTLEN	131000
Maximum length of a column in a table	TPE_MAX_FLDLEN	2000
Maximum length of default value specification	TPE_MAX_DFLT_LEN	250
Maximum length of a connect string	TPE_MAX_CONNLEN	100
Maximum length for a table name	TPE_MAX_IDLEN	32
Maximum length for an area name	TPE_MAX_AREA_NAME	32
Maximum length for a username in a connect string	TPE_UNAME_LEN	32
Maximum length of an error message	TPE_MAX_ERRLEN	256
Maximum number of columns in a table	TPE_MAX_FIELDS	500
Maximum length of a CHECK constraint clause	SQL_MAXCHKCL_SZ	240
Maximum number of nesting levels in an SQL statement	SQL_MAXLEVELS	25
Maximum number of table references in an SQL statement: Microsoft Windows	SQL_MAXTBLREF	50

Table A–2: Progress SQL-92 System Limits *(2 of 2)*

Attribute	Name	Value
Maximum number of table references in an SQL statement: other platforms	SQL_MAXTBLREF	250
Maximum size of input parameters for an SQL statement	SQL_MAXIPARAMS_SZ	512
Maximum number of outer references in an SQL statement	SQL_MAX_OUTER_REF	25
Maximum nesting level for view references	MAX_VIEW_LEVEL	25
Maximum number of check constraints in a table	SQL_MAXCHKCNSTRS	1000 total constraints per table[1]
Maximum number of foreign constraints in a table	SQL_MAXFRNCNSTRS	1000 total constraints per table[1]

[1] The limit per table for the sum of (the number of check constraints plus the number of foreign constraints) is 1000.

Progress SQL-92 Error Messages

This section lists the error messages generated by the various components of Progress SQL-92. Error messages generated by the Progress SQL-92 ODBC driver are documented in the *Progress ODBC Driver Guide*.

Overview

In addition to the Progress SQL-92-specific error codes, error conditions have an associated *SQLSTATE* value. SQLSTATE is a five-character status parameter whose value indicates the condition status returned by the most recent SQL statement. The first two characters of the SQLSTATE value specify the class code and the last three characters specify the subclass code:

- Class codes of a–h and 0–4 are reserved by the SQL-92 standard. For those class codes only, subclass codes of a–h and 0–4 are also reserved by the standard.
- Subclasses S and T and class IM are reserved by the ODBC standard.
- Class codes of i–z and 5–9 are specific to database implementations such as Progress SQL-92. All subclass codes in those classes are implementation defined except as noted for ODBC.

Error Codes, SQLSTATE Values, and Messages

Table A–3 is a list of Progress SQL-92 error messages, ordered by error code number. The table shows the corresponding SQLSTATE value for each message.

Table A–3: Progress SQL-92 Error Codes and Messages *(1 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
00000	00000	Successful completion	***status okay
100L	02000	no data	**sql not found.
10002	22503	Data exception	Tuple Not Found for the Specified TID
10012	N0N12	Flag	ETPL_SCAN_EOP

Table A–3: Progress SQL-92 Error Codes and Messages *(2 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
10013	22914	Data Exception	No more records to be fetched
10100	2150b	Cardinality violation	Too many fields exist
10101	70701	Progress/SQL MM error	No more records exist
10102	2350i	Integrity constraint	Duplicate primary/index key value
10104	M0M06	Progress/SQL rss error	Specified index method is not supported
10107	N0N07	Flag	EIX_SCAN_EOP flag is set
10108	50903	Progress/SQL rds error	Duplicate record specified
10301	M0901	Progress/SQL rss error	Table is locked and LCK_NOWAIT
10400	22501	Data exception	Invalid file size for alter log statement
10920	22521	Data exception	Already existing value specified
11100	50901	Progress/SQL rds error	Invalid transaction id
11102	50903	Progress/SQL rds error	TDS area specified is not found
11103	50504	Progress/SQL rds error	TDS not found for binding
11104	50505	Progress/SQL rds error	Transaction aborted
11105	50506	Progress/SQL rds error	Transaction error
11109	50510	Progress/SQL rds error	Invalid transaction handle
11111	50912	Progress/SQL rds error	Invalid isolation level
11300	M0M00	Progress/SQL rss error	Specified INFO type is not supported

Table A–3: Progress SQL-92 Error Codes and Messages *(3 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
11301	M0M01	Progress/SQL rss error	Specified index type is not supported
16001	22701	Data exception	MM– No data block
16002	70702	Progress SQL-92 MM error	MM– Bad swap block
16003	70703	Progress SQL-92 MM error	MM– No cache block
16004	22704	Data exception	MM– Invalid row number
16005	70705	Progress SQL-92 MM error	MM– Invalid cache block
16006	70706	Progress SQL-92 MM error	MM– Bad swap file
16007	70707	Progress SQL-92 MM error	MM– Row too big
16008	70708	Progress SQL-92 MM error	MM– Array initialized
16009	70709	Progress SQL-92 MM error	MM– Invalid chunk number
16010	70710	Progress SQL-92 MM error	MM– Cannot create table
16011	70711	Progress SQL-92 MM error	MM– Cannot alter table
16012	70712	Progress SQL-92 MM error	MM– Cannot drop table
16020	70713	Progress SQL-92 MM error	MM– TPL ctor error
16021	70714	Progress SQL-92 MM error	MM– Insertion error
16022	70715	Progress SQL-92 MM error	MM– Deletion error
16023	70716	Progress SQL-92 MM error	MM– Updation error
16024	70717	Progress SQL-92 MM error	MM– Fetching error
16025	70718	Progress SQL-92 MM error	MM– Sorting error
16026	70719	Progress SQL-92 MM error	MM– Printing error

Table A–3: Progress SQL-92 Error Codes and Messages *(4 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
16027	70720	Progress SQL-92 MM error	MM– TPLSCAN ctor error
16028	70721	Progress SQL-92 MM error	MM– Scan fetching error
16030	70722	Progress SQL-92 MM error	MM– Can't create index
16031	70723	Progress SQL-92 MM error	MM– Can't drop index
16032	70724	Progress SQL-92 MM error	MM– IXSCAN ctor error
16033	70725	Progress SQL-92 MM error	MM– IX ctor error
16034	70726	Progress SQL-92 MM error	MM– IX deletion error
16035	70727	Progress SQL-92 MM error	MM– IX appending error
16036	70728	Progress SQL-92 MM error	MM– IX insertion error
16037	70729	Progress SQL-92 MM error	MM– IX scan fetching error
16040	70730	Progress SQL-92 MM error	MM– Begin transaction
16041	70731	Progress SQL-92 MM error	MM– Commit transaction
16042	40000	Transaction rollback	***MM– Rollback transaction
16043	70732	Progress SQL-92 MM error	MM– Mark point
16044	70733	Progress SQL-92 MM error	MM– Rollback savepoint
16045	70734	Progress SQL-92 MM error	MM– Set & Get isolation
16050	70735	Progress SQL-92 MM error	MM– TID to char
16051	70736	Progress SQL-92 MM error	MM– char to TID
20000	50501	Progress SQL-92 rds error	SQL internal error
20001	50502	Progress SQL-92 rds error	Memory allocation failure
20002	50503	Progress SQL-92 rds error	Open database failed

Table A–3: Progress SQL-92 Error Codes and Messages *(5 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20003	2a504	Syntax error	Syntax error
20004	28505	Invalid auth specs	User not found
20005	22506	Data exception	Table/View/Synonym not found
20006	22507	Data exception	Column not found/specified
20007	22508	Data exception	No columns in table
20008	22509	Data exception	Inconsistent types
20009	22510	Data exception	Column ambiguously specified
20010	22511	Data exception	Duplicate column specification
20011	22512	Data exception	Invalid length
20012	22513	Data exception	Invalid precision
20013	22514	Data exception	Invalid scale
20014	22515	Data exception	Missing input parameters
20015	22516	Data exception	Subquery returns multiple rows
20016	22517	Data exception	Null value supplied for a mandatory (not null) column
20017	22518	Data exception	Too many values specified
20018	22519	Data exception	Too few values specified
20019	50520	Progress SQL-92 rds error	Cannot modify table referred to in subquery
20020	42521	Access rule violation	Bad column specification for group by clause
20021	42522	Access rule violation	Non-group-by expression in having clause

Table A–3: Progress SQL-92 Error Codes and Messages *(6 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20022	42523	Access rule violation	Non-group-by expression in select clause
20023	42524	Access rule violation	Aggregate function not allowed here
20024	0a000	Feature not supported	Sorry, operation not yet implemented
20025	42526	Access rule violation	Aggregate functions nested
20026	50527	Progress SQL-92 rds error	Too many table references
20027	42528	Access rule violation	Bad field specification in order by clause
20028	50529	Progress SQL-92 rds error	An index with the same name already exists
20029	50530	Progress SQL-92 rds error	Index referenced not found
20030	22531	Data exception	Table space with same name already exists
20031	50532	Progress SQL-92 rds error	Cluster with same name already exists
20032	50533	Progress SQL-92 rds error	No cluster with this name
20033	22534	Data exception	Table space not found
20034	50535	Progress SQL-92 rds error	Bad free <*specification_name*> specification
20035	50536	Progress SQL-92 rds error	At least column spec or null clause should be specified
20036	07537	Dynamic sql error	Statement not prepared
20037	24538	Invalid cursor state	Executing select statement

Table A–3: Progress SQL-92 Error Codes and Messages *(7 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20038	24539	Invalid cursor state	Cursor not closed
20039	24540	Invalid cursor state	Open for nonselect statement
20040	24541	Invalid cursor state	Cursor not opened
20041	22542	Data exception	Table/View/Synonym already exists
20042	2a543	Syntax error	Distinct specified more than once in query
20043	50544	Progress SQL-92 rds error	Tuple size too high
20044	50545	Progress SQL-92 rds error	Array size too high
20045	08546	Connection exception	File does not exist or not accessible
20046	50547	Progress SQL-92 rds error	Field value not null for some tuples
20047	42548	Access rule violation	Granting to self not allowed
20048	42549	Access rule violation	Revoking for self not allowed
20049	22550	Data exception	Keyword used for a name
20050	21551	Cardinality violation	Too many fields specified
20051	21552	Cardinality violation	Too many indexes on this table
20052	22553	Data exception	Overflow error
20053	08554	Connection exception	Database not opened
20054	08555	Connection exception	Database not specified or improperly specified
20055	08556	Connection exception	Database not specified or Database not started
20056	28557	Invalid auth specs	No DBA access rights

Table A–3: Progress SQL-92 Error Codes and Messages *(8 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20057	28558	Invalid auth specs	No RESOURCE privileges
20058	40559	Transaction rollback	Executing SQL statement for an aborted transaction
20059	22560	Data exception	No files in the table space
20060	22561	Data exception	Table not empty
20061	22562	Data exception	Input parameter size too high
20062	42563	Syntax error	Full pathname not specified
20063	50564	Progress SQL-92 rds error	Duplicate file specification
20064	08565	Connection exception	Invalid attach type
20065	26000	Invalid SQL statement name	Invalid statement type
20066	33567	Invalid SQL descriptor name	Invalid sqlda
20067	08568	Connection exception	More than one database cannot be attached locally
20068	42569	Syntax error	Bad arguments
20069	33570	Invalid SQL descriptor name	SQLDA size not enough
20070	33571	Invalid SQL descriptor name	SQLDA buffer length too high
20071	42572	Access rule violation	Specified operation not allowed on the view
20072	50573	Progress SQL-92 rds error	Server is not allocated
20073	2a574	Access rule violation	View query specification for view too long

Table A–3: Progress SQL-92 Error Codes and Messages *(9 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20074	2a575	Access rule violation	View column list must be specified as expressions are given
20075	21576	Cardinality violation	Number of columns in column list is less than in select list
20076	21577	Cardinality violation	Number of columns in column list is more than in select list
20077	42578	Access rule violation	Check option specified for noninsertable view
20078	42579	Access rule violation	Given SQL statement is not allowed on the view
20079	50580	Progress SQL-92 rds error	More tables cannot be created
20080	44581	Check option violation	View check option violation
20081	22582	Data exception	Number of expressions projected on either side of set-op do not match
20082	42583	Access rule violation	Column names not allowed in order by clause for this statement
20083	42584	Access rule violation	Outerjoin specified on a complex predicate
20084	42585	Access rule violation	Outerjoin specified on a sub_query
20085	42586	Access rule violation	Invalid Outerjoin specification
20086	42587	Access rule violation	Duplicate table constraint specification
20087	21588	Cardinality violation	Column count mismatch
20088	28589	Invalid auth specs	Invalid user name
20089	22590	Data exception	System date retrieval failed

Table A–3: Progress SQL-92 Error Codes and Messages *(10 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20090	42591	Access rule violation	Table column list must be specified as expressions are given
20091	2a592	Access rule violation	Query statement too long
20092	2d593	Invalid transaction termination	No tuples selected by the subquery for update
20093	22594	Data exception	Synonym already exists
20094	hz595	Remote database access	Database link with same name already exists
20095	hz596	Remote database access	Database link not found
20096	08597	Connection exception	Connect String not specified/incorrect
20097	hz598	Remote database access	Specified operation not allowed on a remote table
20098	22599	Data exception	More than one row selected by the query
20099	24000	Invalid cursor state	Cursor not positioned on a valid row
20100	4250a	Access rule violation	Subquery not allowed here
20101	2350b	Integrity constraint	No references for the table
20102	2350c	Integrity constraint	Primary/Candidate key column defined null
20103	2350d	Integrity constraint	No matching key defined for the referenced table
20104	2350e	Integrity constraint	Keys in reference constraint incompatible

Table A–3: Progress SQL-92 Error Codes and Messages *(11 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20105	5050f	Progress SQL-92 rds error	Statement not allowed in read only isolation level
20106	2150g	Cardinality violation	Invalid ROWID
20107	hz50h	Remote database access	Remote database not started
20108	0850i	Connection exception	Remote Network Server not started
20109	hz50j	Remote database access	Remote database Name not valid
20110	0850k	Connection exception	TCP/IP Remote HostName is unknown
20114	33002	Invalid SQL descriptor name	Fetched Value NULL & indicator var not defined
20115	5050l	Progress SQL-92 rds error	References to the table/record present
20116	2350m	Integrity constraint	Constraint violation
20117	2350n	Integrity constraint	Table definition not complete
20118	4250o	Access rule violation	Duplicate constraint name
20119	2350p	Integrity constraint	Constraint name not found
20120	22000	Data exception	**Use of reserved word
20121	5050q	Progress SQL-92 rds error	Permission denied
20122	5050r	Progress SQL-92 rds error	Procedure not found
20123	5050s	Progress SQL-92 rds error	Invalid arguments to procedure
20124	5050t	Progress SQL-92 rds error	Query conditionally terminated
20125	0750u	Dynamic sql-error	Number of open cursors exceeds limit

Table A–3: Progress SQL-92 Error Codes and Messages *(12 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20126	34000	Invalid cursor name	***Invalid cursor name
20127	07001	Dynamic sql-error	Bad parameter specification for the statement
20128	2250x	Data Exception	Numeric value out of range
20129	2250y	Data Exception	Data truncated
20132	5050u	Progress SQL-92 rds error	Revoke failed because of restrict
20134	5050v	Progress SQL-92 rds error	Invalid long datatype column references
20135	5050x	Progress SQL-92 rds error	Contains operator is not supported in this context
20135	m0m01	Progress SQL-92 diagnostics error	Diagnostics statement failed
20136	5050z	Progress SQL-92 rds error	Contains operator is not supported for this datatype
20137	50514	Progress SQL-92 rds error	Index is not defined or does not support CONTAINS
20138	50513	Progress SQL-92 rds error	Index on long fields requires that it can push down only CONTAINS
20140	50512	Progress SQL-92 rds error	Procedure already exists
20141	85001	Progress SQL-92 Stored procedure Compilation	Error in Stored Procedure Compilation
20142	86001	Progress SQL-92 Stored procedure Execution	Error in Stored Procedure Execution
20143	86002	Progress SQL-92 Stored procedure Execution	Too many recursions in call procedure

Table A–3: Progress SQL-92 Error Codes and Messages *(13 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20144	86003	Progress SQL-92 Stored procedure Execution	Null value fetched.
20145	86004	Progress SQL-92 Stored procedure Execution	Invalid field reference
20146	86005	Progress SQL-92 Triggers	Trigger with this name already exists
20147	86006	Progress SQL-92 Triggers	Trigger with this name does not exist
20148	86007	Progress SQL-92 Triggers	Trigger Execution Failed
20211	22800	Data exception	Remote procedure call error
20212	08801	Connection exception	SQL client bind to daemon failed
20213	08802	Connection exception	SQL client bind to SQL server failed
20214	08803	Connection exception	SQL NETWORK service entry is not available
20215	08804	Connection exception	Invalid TCP/IP hostname
20216	hz805	Remote database access	Invalid remote database name
20217	08806	Connection exception	Network error on server
20218	08807	Connection exception	Invalid protocol
20219	2e000	Invalid connection name	***Invalid connection name
20220	08809	Connection exception	Duplicate connection name
20221	08810	Connection exception	No active connection
20222	08811	Connection exception	No environment defined database
20223	08812	Connection exception	Multiple local connections

Table A–3: Progress SQL-92 Error Codes and Messages *(14 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20224	08813	Connection exception	Invalid protocol in connect_string
20225	08814	Connection exception	Exceeding permissible number of connections
20226	80815	Progress SQL-92 snw error	Bad database handle
20227	08816	Connection exception	Invalid host name in connect string
20228	28817	Invalid auth specs	Access denied (Authorization failed)
20229	22818	Data exception	Invalid date value
20230	22819	Data exception	Invalid date string
20231	22820	Data exception	Invalid number strings
20232	22821	Data exception	Invalid number string
20233	22822	Data exception	Invalid time value
20234	22523	Data exception	Invalid time string
20235	22007	Data exception	Invalid time stamp string
20236	22012	Data exception	Division by zero attempted
20238	22615	Data exception	Error in format type
20239	2c000	Invalid character set name	Invalid character set name specified.
20240	5050y	Progress SQL-92 rds error	Invalid collation name specified
20241	08815	Connection Exception	Service in use
20300	90901	DBS error	Column group column does not exist

Table A–3: Progress SQL-92 Error Codes and Messages *(15 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
20301	90902	DBS error	Column group column already specified
20302	90903	DBS error	Column group name already specified
20303	90904	DBS error	Column groups have not covered all columns
20304	90905	DBS error	Column groups are not implemented in Progress storage
23000	22563	Progress SQL-92 Data exception	Table create returned invalid table id
23001	22564	Progress SQL-92 Data exception	Index create returned invalid index id
25128	j0j28	Progress SQL-92 odbc trans layer	Query terminated as max row limit exceeded for a remote table
25131	j0j29	Progress SQL-92 odbc trans layer	Unable to read column info from remote table
30001	5050w	Progress SQL-92 rds error	Query aborted on user request
30002	k0k02	Progress SQL-92 network interface	Invalid network handle
30003	k0k03	Progress SQL-92 network interface	Invalid sqlnetwork INTERFACE
30004	k0k04	Progress SQL-92 network interface	Invalid sqlnetwork INTERFACE procedure
30005	k0k05	Progress SQL-92 network interface	INTERFACE is already attached
30006	k0k06	Progress SQL-92 network interface	INTERFACE entry not found

Table A–3: Progress SQL-92 Error Codes and Messages *(16 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
30007	k0k07	Progress SQL-92 network interface	INTERFACE is already registered
30008	k0k08	Progress SQL-92 network interface	Mismatch in pkt header size and total argument size
30009	k0k09	Progress SQL-92 network interface	Invalid server id
30010	k0k10	Progress SQL-92 network interface	Reply does not match the request
30011	k0k02	Progress SQL-92 network interface	Memory allocation failure
30031	k0k11	Progress SQL-92 network interface	Error in transmission of packet
30032	k0k12	Progress SQL-92 network interface	Error in reception of packet
30033	k0k13	Progress SQL-92 network interface	No packet received
30034	k0k14	Progress SQL-92 network interface	Connection reset
30051	k0k15	Progress SQL-92 network interface	Network handle is inprocess handle
30061	k0k16	Progress SQL-92 network interface	Could not connect to sql network daemon
30062	k0k17	Progress SQL-92 network interface	Error in number of arguments
30063	k0k18	Progress SQL-92 network interface	Requested INTERFACE not registered
30064	k0k19	Progress SQL-92 network interface	Invalid INTERFACE procedure id

Table A–3: Progress SQL-92 Error Codes and Messages *(17 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
30065	k0k20	Progress SQL-92 network interface	Requested server executable not found
30066	k0k21	Progress SQL-92 network interface	Invalid configuration information
30067	k0k22	Progress SQL-92 network interface	INTERFACE not supported
30091	k0k23	Progress SQL-92 network interface	Invalid service name
30092	k0k24	Progress SQL-92 network interface	Invalid host
30093	k0k25	Progress SQL-92 network interface	Error in tcp/ip accept call
30094	k0k26	Progress SQL-92 network interface	Error in tcp/ip connect call
30095	k0k27	Progress SQL-92 network interface	Error in tcp/ip bind call
30096	k0k28	Progress SQL-92 network interface	Error in creating socket
30097	k0k29	Progress SQL-92 network interface	Error in setting socket option
30101	k0k30	Progress SQL-92 network interface	Interrupt occurred
40001	L0L01	Progress SQL-92 env error	Error in reading configuration
210001	08P00	Connection exception	Failure to acquire share schema lock during connect
210002	08004	Connection exception	Failure in finding DLC environment variable

Table A–3: Progress SQL-92 Error Codes and Messages *(18 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
210003	08004	Connection exception	DLC environment variable exceeds maximum size <*max_size*> -> <*DLC path*>
210004	08004	Connection exception	Error opening convmap.cp file <*filename*> <*path*>
210005	P1000	Unavailable resource	Failure getting lock table on table <*table_name*>
210011	08004	Internal error	Fatal error identifying database log in SQL
210012	22P00	Data exception	Column <*column_name*> in table <*table_name*> has value exceeding its max length or precision
210013	08004	Connection exception	Unable to complete server connection. Function<*function_name*>; reason <*summary_of_reason*>
210014	22P01	Data exception	Column values too big to make key. Table <*table_name*>; index <*index_name*>
210015	P1000	Unavailable resource	Failure getting record lock on a record table <*table_name*>
210016	P1001	Unavailable resource	Lock table is full
210017	P1002	Unavailable resource	Failure to acquire exclusive schema lock for DDL operation
210018	0AP01	Unsupported feature	Update of word indexes not yet supported. Table <*table_name*>, index <*index_name*>
210019	0A000	Unsupported feature	Scan of word indexes not yet supported. Table <*table_name*>, index <*index_name*>

Table A–3: Progress SQL-92 Error Codes and Messages *(19 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
210020	0AP03	Unsupported feature	The first index created for a table may not be dropped
210021	85001	Progress/SQL Stored Procedure compilation	Location of the Java compiler was not specified
211013	3F001	Bad schema reference	SQL-92 cannot alter or drop a table created by 4GL or SQL-89
211014	3F002	Bad schema reference	Incorrect view owner name on CREATE VIEW – cannot be PUB or _FOREIGN
211015	3F003	Bad schema reference	Database table or view owned by "sysprogress" cannot be created, dropped or altered
211016	3F004	Bad schema reference	Database schema table cannot be created, dropped or altered
211017	3F004	Bad schema reference	Attempt to insert, update, or delete a row in a schema table
211018	0A000	Array reference error	Array reference/update incorrect
218001	P8P18	Progress I18N NLS error	Failure to create a NLS character set conversion handler
219901	P0000	Internal error	Internal error *<error_num1>* *<error_meaning>* in SQL from subsystem <*subsystem_name*> function <*function_name*> called from <*calling_function*> on <*object_2*> for <*object_1*>. Save log for Progress technical support
219902	P0001	Internal error	Failure reading schema during DDL operation

Table A–3: Progress SQL-92 Error Codes and Messages *(20 of 20)*

Error Code	SQL STATE Value	Class Condition	Subclass Message
219903	P0002	Internal error	Inconsistent metadata - contact Progress technical support
219951	40P00	Transaction rollback	Fatal error *<error_num> <error_meaning>* in SQL from subsystem <*subsystem_name*> function <*function_name*> called from <*calling_function*> on <*object_2*> for *<object_1>*. Save log for Progress technical support

B

Progress SQL-92 System Catalog Tables

Progress SQL-92 maintains a set of system tables for storing information about tables, columns, indexes, constraints, and privileges. This appendix describes those System Catalog Tables.

Overview of System Catalog Tables

Progress SQL-92 maintains a set of system tables for storing information about tables, columns, indexes, constraints, and privileges.

All users have read access to the system catalog tables. SQL Data Definition Language (DDL) statements and GRANT and REVOKE statements modify system catalog tables. The system tables are modified in response to these statements, as the database evolves and changes.

The **owner** of the system tables is *sysprogress*. If you connect to a Progress SQL-92 environment with a *username* other than *sysprogress*, you must use the *owner* qualifier when you reference a system table in a SQL query. Alternatively, you can issue a SET SCHEMA 'sysprogress' statement to set the default *username* for unqualified table names to 'sysprogress'.

Core tables store information on the tables, columns, and indexes that make up the database. The remaining tables contain detailed information on database objects, and statistical information.

Table B–1 lists the system catalog tables in the same order that they are presented in following sections.

Table B–1: System Tables and Descriptions *(1 of 4)*

System Table	Summary Description
SYSTABLES Core SystemTable	Core system table. One row for each TABLE in the database.
SYSCOLUMNS Core SystemTable	Core system table. One row for each COLUMN of each table in the database.
SYSINDEXES Core SystemTable	Core system table. One row for each component of each INDEX in the database.
SYSCALCTABLE System Table	A single row with a single column set to the value 100.
SYSCHARSTAT System Table	One row for each CHARACTER column in the database.
SYSCOLAUTH System Table	One row for each column for each user holding privileges on the column.
SYSCOLSTAT System Table	Provides mappings between SYSCOLUMNS table and SYS*STAT tables.

Table B–1: System Tables and Descriptions *(2 of 4)*

System Table	Summary Description
SYSCOLUMNS_FULL System Table	Superset of information in core system table SYSCOLUMNS.
SYSDATATYPES System Table	Information on supported data types.
SYSDATESTAT System Table	One row for each DATE column in the database.
SYSDBAUTH System Table	One row for each user with database-wide privileges.
SYSFLOATSTAT System Table	One row for each FLOAT column in the database.
SYSIDXSTAT System Table	Information on each index in the database.
SYSINTSTAT System Table	One row for each INTEGER column in the database.
SYSNUMSTAT System Table	One row for each NUMERIC column in the database.
SYSPROCBIN System Table	One row for each compiled Java stored procedure or trigger in the database.
SYSPROCCOLUMNS System Table	One row for each column in the result set of a stored procedure.
SYSPROCEDURES System Table	One row for each stored procedure in the database.
SYSPROCTEXT System Table	One row for each Java source code for a stored procedure or trigger in the database.
SYSREALSTAT System Table	One row for each REAL column in the database.
SYSSMINTSTAT System Table	One row for each SMALLINT column in the database.

Table B–1: System Tables and Descriptions *(3 of 4)*

System Table	Summary Description
SYSSYNONYMS System Table	One row for each SYNONYM in the database.
SYSTABAUTH System Table	One row for each unique user/table combination holding table privileges on a table in the database.
SYSTABLES_FULL System Table	Superset of information in core system table SYSTABLES.
SYSTBLSTAT System Table	Contains statistics for each user table in the database.
SYSTIMESTAT System Table	One row for each TIME column in the database.
SYSTINYINTSTAT System Table	One row for each TINYINT column in the database.
SYSTRIGCOLS System Table	One row for each column specified in each trigger in the database.
SYSTRIGGER System Table	One row for each trigger in the database.
SYSTSSTAT System Table	One row for each TIMESTAMP column in the database.
SYSVARCHARSTAT System Table	One row for each VARCHAR column in the database.
SYSVIEWS System Table	One row for each VIEW in the database.
SYS_CHKCOL_USAGE System Table	One row for each CHECK CONSTRAINT defined on a column in the database.
SYS_CHK_CONSTRS System Table	One row for each CHECK CONSTRAINT defined on a user table in the database.
SYS_KEYCOL_USAGE System Table	One row for each column in the database defined with a PRIMARY KEY or FOREIGN KEY.

Table B–1: System Tables and Descriptions *(4 of 4)*

System Table	Summary Description
SYS_REF_CONSTRS System Table	One row for each table in the database defined with a REFERENTIAL INTEGRITY CONSTRAINT.
SYS_TBL_CONSTRS System Table	One row for each CONSTRAINT defined on a table in the database.

SYSTABLES Core SystemTable

Contains one row for each table in the database.

Table B–2: SYSTABLES Core System Table

Column Name	Column Data Type	Column Size
creator	VARCHAR	32
has_ccnstrs	VARCHAR	1
has_fcnstrs	VARCHAR	1
has_pcnstrs	VARCHAR	1
has_ucnstrs	VARCHAR	1
id	INTEGER	4
owner	VARCHAR	32
rssid	INTEGER	4
segid	INTEGER	4
tbl	VARCHAR	32
tbl_status	VARCHAR	1
tbltype	VARCHAR	1

SYSCOLUMNS Core SystemTable

Contains one row for each column of every table in the database.

Table B–3: SYSCOLUMNS Core System Table

Column Name	Column Data Type	Column Size
charset	VARCHAR	32
col	VARCHAR	32
collation	VARCHAR	32
coltype	VARCHAR	10
dflt_value	VARCHAR	250
id	INTEGER	4
nullflag	VARCHAR	1
owner	VARCHAR	32
scale	INTEGER	4
tbl	VARCHAR	32
width	INTEGER	4

SYSINDEXES Core SystemTable

Contains one row for each component of an index in the database. For an index with *n* components, there will be *n* rows in this table.

Table B–4: SYSINDEXES Core System Table

Column Name	Column Data Type	Column Size
abbreviate	BIT	1
active	BIT	1
creator	VARCHAR	32
colname	VARCHAR	32
desc	VARCHAR	144
id	INTEGER	4
idxcompress	VARCHAR	1
idxmethod	VARCHAR	2
idxname	VARCHAR	32
idxorder	CHARACTER	1
idxowner	VARCHAR	32
idxsegid	INTEGER	4
idxseq	INTEGER	4
ixcol_user_misc	VARCHAR	20
rssid	INTEGER	4
tbl	VARCHAR	32
tblowner	VARCHAR	32

SYSCALCTABLE System Table

Contains exactly one row with a single column with a value of 100.

Table B–5: SYSCALCTABLE System Table

Column Name	Column Data Type	Column Size
fld	INTEGER	4

SYSCHARSTAT System Table

Contains one row for each column in the database with datatype CHAR. Used by the optimizer, each row contains a sampling of values in the column.

Table B–6: SYSCHARSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	VARCHAR	2000
val2	VARCHAR	2000
val3	VARCHAR	2000
val4	VARCHAR	2000
val5	VARCHAR	2000
val6	VARCHAR	2000
val7	VARCHAR	2000
val8	VARCHAR	2000
val9	VARCHAR	2000
val10	VARCHAR	2000

SYSCOLAUTH System Table

Contains one row for the update privileges held by users on individual columns of tables in the database.

Table B–7: SYSCOLAUTH System Table

Column Name	Column Data Type	Column Size
col	VARCHAR	32
grantee	VARCHAR	32
grantor	VARCHAR	32
ref	VARCHAR	1
sel	VARCHAR	1
tbl	VARCHAR	32
tblowner	VARCHAR	32
upd	VARCHAR	1

SYSCOLSTAT System Table

Provides mapping information between *syscolumns* and *sys*stat* tables.

Table B–8: SYSCOLSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
coltype	VARCHAR	10
rssid	INTEGER	4
tblid	INTEGER	4

SYSCOLUMNS_FULL System Table

A superset of information in the SYSCOLUMNS core system table.

Table B–9: SYSCOLUMNS_FULL System Table *(1 of 2)*

Column Name	Column Data Type	Column Size
array_extent	INTEGER	4
charset	VARCHAR	32
col	VARCHAR	32
collation	VARCHAR	32
coltype	VARCHAR	10
col_label	VARCHAR	60
col_label_sa	VARCHAR	12
col_subtype	INTEGER	4
description	VARCHAR	144
dflt_value	VARCHAR	323
dflt_value_sa	VARCHAR	12
display_order	INTEGER	4
field_rpos	INTEGER	4
format	VARCHAR	60
format_sa	VARCHAR	12
help	VARCHAR	126
help_sa	VARCHAR	12
id	VARCHAR	4
label	VARCHAR	60

Table B–9: SYSCOLUMNS_FULL System Table *(2 of 2)*

Column Name	Column Data Type	Column Size
label_sa	VARCHAR	12
nullflag	CHARACTER	2
owner	VARCHAR	32
scale	INTEGER	4
tbl	VARCHAR	32
user_misc	VARCHAR	20
valexp	VARCHAR	144
valmsg	VARCHAR	144
valmsg_sa	VARCHAR	12
view_as	VARCHAR	100
width	INTEGER	4

SYSDATATYPES System Table

Contains information on each data type supported by the database.

Table B–10: SYSDATATYPES System Table

Column Name	Column Data Type	Column Size
autoincr	SMALLINT	2
casesensitive	SMALLINT	2
createparams	VARCHAR	32
datatype	SMALLINT	2
dhtypename	VARCHAR	32
literalprefix	VARCHAR	1
literalsuffix	VARCHAR	1
localtypename	VARCHAR	1
nullable	SMALLINT	2
odbcmoney	SMALLINT	2
searchable	SMALLINT	2
typeprecision	INTEGER	4
unsignedattr	SMALLINT	2

SYSDATESTAT System Table

Contains one row for for each column of datatype DATE. Used by the optimizer, each row contains a sampling of values in the column.

Table B–11: SYSDATESTAT System Table

Column Name	Column Data Type	Column Size
rssid	INTEGER	4
colid	INTEGER	4
tblid	INTEGER	4
val1	DATE	4
val2	DATE	4
val3	DATE	4
val4	DATE	4
val5	DATE	4
val6	DATE	4
val7	DATE	4
val8	DATE	4
val9	DATE	4
val10	DATE	4

SYSDBAUTH System Table

Contains the database-wide privileges held by users.

Table B–12: SYSDBAUTH System Table

Column Name	Column Data Type	Column Size
dba_acc	VARCHAR	1
grantee	VARCHAR	32
res_acc	VARCHAR	1

SYSFLOATSTAT System Table

Contains one row for each column of datatype FLOAT. Used by the optimizer, each row contains a sampling of values in the column.

Table B–13: SYSFLOATSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	FLOAT	4
val2	FLOAT	4
val3	FLOAT	4
val4	FLOAT	4
val5	FLOAT	4
val6	FLOAT	4
val7	FLOAT	4
val8	FLOAT	4
val9	FLOAT	4
val10	FLOAT	4

SYSIDXSTAT System Table

Contains statistics for each index in the database.

Table B–14: SYSIDXSTAT System Table

Column Name	Column Data Type	Column Size
idxid	INTEGER	4
nleaf	INTEGER	4
nlevels	SMALLINT	2
recsz	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4

SYSINTSTAT System Table

Contains one row for each column of datatype INTEGER. Used by the optimizer, each row contains a sampling of values in the column.

Table B–15: SYSINTSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	INTEGER	4
val2	INTEGER	4
val3	INTEGER	4
val4	INTEGER	4
val5	INTEGER	4
val6	INTEGER	4
val7	INTEGER	4
val8	INTEGER	4
val9	INTEGER	4
val10	INTEGER	4

SYSNUMSTAT System Table

Contains one row for each column of datatype NUMERIC. Used by the optimizer, each row contains a sampling of values in the column.

Table B–16: SYSNUMSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	NUMERIC	32
val2	NUMERIC	32
val3	NUMERIC	32
val4	NUMERIC	32
val5	NUMERIC	32
val6	NUMERIC	32
val7	NUMERIC	32
val8	NUMERIC	32
val9	NUMERIC	32
val10	NUMERIC	32

SYSPROCBIN System Table

Contains one or more rows for each stored procedure and trigger in the database. Each row contains compiled Java bytecode for its procedure or trigger.

Table B–17: SYSPROCBIN System Table

Column Name	Column Data Type	Column Size
id	INTEGER	4
proc_bin	VARBINARY	2000
proc_type	CHARACTER	2
rssid	INTEGER	4
seq	INTEGER	4

SYSPROCCOLUMNS System Table

Contains one row for each column in the result set of a stored procedure.

Table B–18: SYSPROCCOLUMNS System Table

Column Name	Column Data Type	Column Size
argtype	VARCHAR	32
col	VARCHAR	32
datatype	VARCHAR	32
dflt_value	VARCHAR	250
id	INTEGER	4
nullflag	CHAR	1
proc_id	INTEGER	4
rssid	INTEGER	4
scale	INTEGER	4
width	INTEGER	4

SYSPROCEDURES System Table

Contains one row for each stored procedure in the database.

Table B–19: SYSPROCEDURES System Table

Column Name	Column Data Type	Column Size
creator	VARCHAR	32
has_resultset	CHARACTER	1
has_return_val	CHARACTER	1
owner	VARCHAR	32
proc_id	INTEGER	4
proc_name	VARCHAR	32
proc_type	VARCHAR	32
rssid	INTEGER	4

SYSPROCTEXT System Table

Contains one or more rows for each stored procedure and trigger in the database. The row contains the Java source code for a procedure or trigger.

Table B–20: SYSPROCTEXT System Table

Column Name	Column Data Type	Column Size
id	INTEGER	4
proc_text	VARCHAR	2000
proc_type	CHARACTER	2
rssid	INTEGER	4
seq	INTEGER	4

SYSREALSTAT System Table

Contains one row for each column of datatype REAL. Used by the optimizer, each row contains a sampling of values in the column.

Table B–21: SYSREALSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	REAL	4
val2	REAL	4
val3	REAL	4
val4	REAL	4
val5	REAL	4
val6	REAL	4
val7	REAL	4
val8	REAL	4
val9	REAL	4
val10	REAL	4

SYSSMINTSTAT System Table

Contains one row for each column of datatype SMALLINT. Used by the optimizer, each row contains a sampling of values in the column.

Table B–22: SYSSMINTSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	SMALLINT	2
val2	SMALLINT	2
val3	SMALLINT	2
val4	SMALLINT	2
val5	SMALLINT	2
val6	SMALLINT	2
val7	SMALLINT	2
val8	SMALLINT	2
val9	SMALLINT	2
val10	SMALLINT	2

SYSSYNONYMS System Table

Contains one row for each synonym in the database.

Table B–23: SYSSYNONYMS System Table

Column Name	Column Data Type	Column Size
ispublic	SMALLINT	2
screator	VARCHAR	32
sname	VARCHAR	32
sowner	VARCHAR	32
sremdb	VARCHAR	32
stbl	VARCHAR	32
stblowner	VARCHAR	32

SYSTABAUTH System Table

Contains information about table privileges for each user in the database.

Table B–24: SYSTABAUTH System Table

Column Name	Column Data Type	Column Size
alt	VARCHAR	1
del	VARCHAR	1
exe	CHAR	1
grantee	VARCHAR	32
grantor	VARCHAR	32
ins	VARCHAR	1
ndx	VARCHAR	1
ref	VARCHAR	1
sel	VARCHAR	1
tbl	VARCHAR	32
tblowner	VARCHAR	32
upd	VARCHAR	1

SYSTABLES_FULL System Table

A superset of information in the SYSTABLES core system table.

Table B–25: SYSTABLES_FULL System Table *(1 of 2)*

Column Name	Column Data Type	Column Size
can_dump	VARCHAR	126
can_load	VARCHAR	126
creator	VARCHAR	32
description	VARCHAR	144
dump_name	VARCHAR	16
file_label	VARCHAR	60
file_label_sa	VARCHAR	12
frozen	BIT	1
has_ccnstrs	VARCHAR	1
has_fcnstrs	VARCHAR	1
has_pcnstrs	VARCHAR	1
has_ucnstrs	VARCHAR	1
hidden	BIT	1
id	INTEGER	4
last_change	INTEGER	4
owner	VARCHAR	32
prime_index	INTEGER	4
rssid	INTEGER	4
segid	INTEGER	4

Table B–25: SYSTABLES_FULL System Table *(2 of 2)*

Column Name	Column Data Type	Column Size
tbl	VARCHAR	32
tbltype	VARCHAR	1
tbl_status	VARCHAR	1
user_misc	VARCHAR	20
valexp	VARCHAR	144
valmsg	VARCHAR	144
valmsg_sa	VARCHAR	12

SYSTBLSTAT System Table

Contains statistics for each user table.

Table B–26: SYSTBLSTAT System Table

Column Name	Column Data Type	Column Size
card	INTEGER	4
npages	INTEGER	4
pagesz	INTEGER	4
recsz	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4

SYSTIMESTAT System Table

Contains one row for each column of datatype TIME. Used by the optimizer, each row contains a sampling of values in the column.

Table B–27: SYSTIMESTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	TIME	4
val2	TIME	4
val3	TIME	4
val4	TIME	4
val5	TIME	4
val6	TIME	4
val7	TIME	4
val8	TIME	4
val9	TIME	4
val10	TIME	4

SYSTINYINTSTAT System Table

Contains one row for each column of datatype TINYINT. Used by the optimizer, each row contains a sampling of values in the column.

Table B–28: SYSTINYINTSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
tblid	INTEGER	4
val1	TINYINT	1
val2	TINYINT	1
val3	TINYINT	1
val4	TINYINT	1
val5	TINYINT	1
val6	TINYINT	1
val7	TINYINT	1
val8	TINYINT	1
val9	TINYINT	1
val10	TINYINT	1

SYSTRIGCOLS System Table

Contains one row for each column specified in each UPDATE trigger in the database.

Table B–29: SYSTRIGCOLS System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
owner	VARCHAR	32
triggername	VARCHAR	32

SYSTRIGGER System Table

Contains one row for each trigger in the database.

Table B–30: SYSTRIGGER System Table

Column Name	Column Data Type	Column Size
fire_4gl	BIT	1
owner	VARCHAR	32
refers_to_new	CHAR	1
refers_to_old	CHAR	1
rssid	INTEGER	4
statement_or_row	CHAR	1
tbl	VARCHAR	32
tblowner	VARCHAR	32
triggerid	INTEGER	4
triggername	VARCHAR	32
trigger_event	VARCHAR	1
trigger_time	VARCHAR	1

SYSTSSTAT System Table

Contains one row for each column of datatype TIMESTAMP. Used by the optimizer, each row contains a sampling of values in the column.

Table B–31: SYSTSSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	TIMESTAMP	8
val2	TIMESTAMP	8
val3	TIMESTAMP	8
val4	TIMESTAMP	8
val5	TIMESTAMP	8
val6	TIMESTAMP	8
val7	TIMESTAMP	8
val8	TIMESTAMP	8
val9	TIMESTAMP	8
val10	TIMESTAMP	8

SYSVARCHARSTAT System Table

Contains one row for each column of datatype VARCHAR. Used by the optimizer, each row contains a sampling of values in the column.

Table B–32: SYSVARCHARSTAT System Table

Column Name	Column Data Type	Column Size
colid	INTEGER	4
rssid	INTEGER	4
tblid	INTEGER	4
val1	VARCHAR	2000
val2	VARCHAR	2000
val3	VARCHAR	2000
val4	VARCHAR	2000
val5	VARCHAR	2000
val6	VARCHAR	2000
val7	VARCHAR	2000
val8	VARCHAR	2000
val9	VARCHAR	2000
val10	VARCHAR	2000

SYSVIEWS System Table

Contains one row for each VIEW in the database.

Table B–33: SYSVIEWS System Table

Column Name	Column Data Type	Column Size
creator	VARCHAR	32
owner	VARCHAR	32
seq	INTEGER	4
viewname	VARCHAR	32
viewtext	VARCHAR	2000

SYS_CHKCOL_USAGE System Table

Contains one row for each column on which a check constraint is specified.

Table B–34: SYS_CHKCOL_USAGE System Table

Column Name	Column Data Type	Column Size
cnstrname	VARCHAR	32
colname	VARCHAR	32
owner	VARCHAR	32
tblname	VARCHAR	32

SYS_CHK_CONSTRS System Table

Contains one row for each CHECK CONSTRAINT specified on a user table. The chkclause column contains the content of the CHECK clause.

Table B–35: SYS_CHK_CONSTRS System Table

Column Name	Column Data Type	Column Size
chkclause	VARCHAR	2000
chkseq	INTEGER	4
cnstrname	VARCHAR	32
owner	VARCHAR	32
tblname	VARCHAR	32

SYS_KEYCOL_USAGE System Table

Contains one row for each column on which a PRIMARY KEY or FOREIGN KEY is specified.

Table B–36: SYS_KEYCOL_USAGE System Table

Column Name	Column Data Type	Column Size
cnstrname	VARCHAR	32
colname	VARCHAR	32
colposition	INTEGER	4
owner	VARCHAR	32
tblname	VARCHAR	32

SYS_REF_CONSTRS System Table

Contains one row for each REFERENTIAL INTEGRITY CONSTRAINT specified on a user table.

Table B–37: SYS_REF_CONSTRS System Table

Column Name	Column Data Type	Column Size
cnstrname	VARCHAR	32
deleterule	VARCHAR	1
owner	VARCHAR	32
refcnstrname	VARCHAR	32
refowner	VARCHAR	32
reftblname	VARCHAR	32
tblname	VARCHAR	32

SYS_TBL_CONSTRS System Table

Contains one row for each table constraint in the database.

Table B–38: SYS_TBL_CONSTRS System Table

Column Name	Column Data Type	Column Size
cnstrname	VARCHAR	32
cnstrtype	VARCHAR	1
idxname	VARCHAR	32
owner	VARCHAR	32
tblname	VARCHAR	32

C

Data Type Compatibility Issues with Previous Versions of Progress

This appendix addresses compatibility issues when using the Progress SQL-92 environment and earlier versions of the Progress database. Specifically, it discusses:

- Mapping between Progress 4GL supported data types and the corresponding Progress SQL-92 data types.
- Progress SQL-92 support for the ARRAY data type created using the Progress 4GL Dictionary.

C.1 Supported Data Types and Corresponding SQL-92 Data Types

Progress SQL-92 supports many data types that do not correspond to Progress 4GL data types. Table C–1 lists the Progress data types that **do** correspond to Progress SQL-92 data types.

Table C–1: Progress 4GL Supported Data Types and Corresponding Progress SQL-92 Data Types

Progress 4GL Data Type	Progress SQL-92 Data Type
ARRAY	See the "Support for the ARRAY Data Type" section
CHARACTER	VARCHAR
DATE	DATE
DECIMAL	DECIMAL or NUMERIC
INTEGER	INTEGER
LOGICAL	BIT
RAW	VARBINARY
RECID	INTEGER

NOTES

- All other SQL-92 types are not compatible with 4GL. In particular, Progress SQL-92 CHARACTER data is not compatible with the 4GL. Use SQL-92 type VARCHAR to map 4GL CHARACTER data.
- Data columns created using the Progress SQL-92 environment that have a data type not supported in the Progress 4GL environment are not accessible through Progress 4GL applications and utilities.

C.2 Support for the ARRAY Data Type

For compatibility with earlier Progress databases, Progress SQL-92 provides limited support for the ARRAY data type.

C.2.1 Overview

Array fields are created using the Progress 4GL Dictionary. Progress SQL-92 provides a mechanism for referencing and updating these arrays. Subscripted references are not supported. Progress SQL-92 manages the arrays as NVARCHAR strings, and the string representation is the concatenation of each array element, with a semicolon (;) separating each element.

To escape an instance of a semicolon in the data of a Progress CHARACTER array, use the escape character tilde before the semicolon (~;). An unquoted question mark represents a NULL element. To distinguish a NULL element from a question mark, use a tilde as an escape character for a question mark in the data (~?). A tilde is also the escape character for a tilde (~~).

Progress SQL-92 supplies three built-in functions for extracting an element, and for adding escape characters to and removing escape characters from a single element of a character array. The PRO_ELEMENT, PRO_ARR_ESCAPE, and PRO_ARR_DESCAPE functions provide full internationalization support. A description of each function follows.

C.2.2 PRO_ELEMENT Function

Extracts one or more elements from an array column and returns the NVARCHAR or VARCHAR string between the specified positions, including any internal separator characters and any internal escape characters.

SYNTAX

```
PRO_ELEMENT ( 'array_style_expression', start_position, end_position ) ;
```

array_style_expression

A string of datatype VARCHAR or CHAR, with a semi-colon (;) separating each element of the array.

start_position

The position in the string marking the beginning of the element PRO_ELEMENT is to extract.

end_position

The position in the string marking the end of the element to be extracted.

EXAMPLES

The following example returns the string ’bb’:

```
PRO_ELEMENT('aa;bb;cc', 2, 2) ;
```

The next example returns the string ’aa;bb’:

```
PRO_ELEMENT('aa;bb;cc', 1, 2) ;
```

The third example returns the string ’aa~;aa’:

```
PRO_ELEMENT('aa~;aa;bb;cc', 1, 1) ;
```

NOTES

- The *array_style_expression* must be data type NVARCHAR, VARCHAR, or CHAR.
- The returned string does not include the leading separator of the first element, or the trailing separator (;) of the last element.
- Even if you are extracting only one element, the escape characters are included in the result.
- You must invoke PRO_ARR_DESCAPE to remove any escape characters.

C.2.3 PRO_ARR_ESCAPE Function

Adds required escape characters to a single element of a character array.

PRO_ARR_ESCAPE scans the *char_element* looking for the separator character (;) or an escape character (~). The function **inserts** an additional escape character, when it finds any of these constructs:

- Escape character followed by a separator character (~;)
- Escape character followed by another escape character (~~)
- Escape character followed by a NULL terminator (~\0)

SYNTAX

```
PRO_ARR_ESCAPE( 'char_element' )  ;
```

char_element

The character representation of an array element, without any leading or trailing separators. Must be data type NVARCHAR or VARCHAR or CHAR.

EXAMPLES

The following example returns the string 'aa~;aa':

```
PRO_ARR_ESCAPE('aa;aa') ;
```

The following example returns the string 'aa~aa'. There is no change, since another special character does not follow the escape character:

```
PRO_ARR_ESCAPE('aa~aa') ;
```

The third example returns the string 'aa~~;aa':

```
PRO_ARR_ESCAPE('aa~;aa') ;
```

NOTES

- *char_element* must be data type NVARCHAR, VARCHAR, or CHAR.
- *char_element* must not be the name of an array column, since the column contains true separators which would be destroyed by this function.

C.2.4 PRO_ARR_DESCAPE Function

Removes escape characters from a single element of a character array. PRO_ARR_DESCAPE scans the *char_element* looking for the separator character (;) or an escape character (~). The function **removes** an escape character, when it finds any of these constructs:

- Escape character followed by a separator character (~;)
- Escape character followed by another escape character (~~)
- Escape character followed by a NULL terminator (~\0)

SYNTAX

```
PRO_ARR_DESCAPE( 'char_element' ) ;
```

char_element

The character representation of an array element, without any leading or trailing separators. Must be data type NVARCHAR or VARCHAR or CHAR.

EXAMPLES

The following example returns the string 'aa;aa':

```
PRO_ARR_DESCAPE('aa~;aa') ;
```

The following example returns the string 'aa~aa'. There is no change, since another special character does not follow the escape character:

```
PRO_ARR_DESCAPE('aa~aa') ;
```

The third example returns the string 'aa~;aa':

```
PRO_ARR_DESCAPE('aa~~;aa') ;
```

NOTE: *char_element* should not be the name of an array column, since the column contains true separators which would be destroyed by this function.

C.2.5 Unsubscripted Array References

When there is a reference to an unsubscripted Progress array column, Progress SQL-92 performs these operations:

- Extracts each element from the Progress 4GL array
- Converts the element to a DT_VARCHAR data type
- Passes the element to the PRO_ARR_ESCAPE function for the addition of any necessary escape characters
- Concatenates the result to a composite with the separator character (;) between this element and the next element

This results in dual data types for array fields. The fetch type is TPE_DT_NVARCHAR for the unsubscripted references. The underlying Progress 4GL data type is the native type.

EXAMPLES

The first example assumes a character array named ARRAYCHAR containing three elements in a row in the customer table where the cust_num column is equal to 88, and with values:

ARRAYCHAR[1] = ’aa’

ARRAYCHAR[2] = ’bb’

ARRAYCHAR[3] = ’cc’

The ARRAYCHAR example returns the value ’aa;bb;cc’.

```
SELECT arraychar FROM customer WHERE cust_num = 88 ;

ARRAYCHAR

---------
aa;bb;cc

1 record selected
```

To retrieve an individual element from an array with any escape characters removed, use the PRO_ELEMENT and PRO_ARR_DESCAPE functions. For example, ARRAYTYPE[2] contains the value 'aa;bb'. The PRO_ELEMENT function in the ARRAYTYPE example returns 'aa~;bb' and PRO_ARR_DESCAPE('aa~;bb') removes the escape character (~), returning the element value 'aa;bb'.

```
select pro_arr_descape(pro_element(arraytype,2,2)) from customer  ;
ARRAYTYPE

---------
aa;bb

1 record selected
```

C.2.6 Unsubscripted Array Updates and Inserts

Progress SQL-92 applies the reverse of the fetch algorithm for updates to unsubscripted Progress 4GL array columns. An SQL-92 operation accepts an NVARCHAR string, complete with separators and any required escape characters. Progress SQL-92 converts the string to the underlying data type and stores it in the current element of the array in the Progress database. SQL-92 repeats this operation for each element of the array until all elements have been inserted.

This is the SQL-92 syntax for an unsubscripted array update:

SYNTAX

```
UPDATE table_name SET array_name = ('char_element' ) WHERE where_criteria ;
```

EXAMPLE

The ARRAYINT example assumes an integer array named ARRAYINT. The result of the UPDATE operation is:

ARRAYINT[1] = 13

ARRAYINT[2] = 15

ARRAYINT[3] = 19

```
UPDATE customer SET arrayint = '13;15;19' WHERE cust_num = 77 ;
```

If the number of elements in the NVARCHAR string does not match the number of elements in the target column for the update, Progress SQL-92 returns an error, unless there is exactly one element in the NVARCHAR string for an INSERT operation.

If there is a single element in the NVARCHAR string, you can use the Progress SQL-92 INSERT statement to propagate the value to all elements of the array.

This is the SQL-92 syntax for this short form of an INSERT assignment:

SYNTAX

```
INSERT INTO table_name ( arr_col_name ) VALUES ( 'one_value' ) ;
```

EXAMPLE

The following INSERT example illustrates how you can assign values to an entire date array from a single value in the VALUES clause of an SQL-92 INSERT statement:

```
INSERT INTO customer ( begin_quota_date ) VALUES ( '01/01/00' ) ;
```

This INSERT example assigns the value '01/01/00' to every element of the date array 'begin_quota_date' column in the customer table.

NOTE: Progress SQL-92 does **not** support the assignment of a single value to an entire array using an SQL-92 UPDATE statement.

Updating a Single Element of an Array

To update a single element of an array, you must construct a string for the entire array, and assign the string to the array using an SQL-92 UPDATE statement.

This is the SQL-92 syntax for updating a single element of an array:

SYNTAX

```
UPDATE table_name SET array_col = literal_string ;
```

EXAMPLES

To assign a value to the first element of an array of size three, construct a literal string that concatenates these components:

- Update value for the first element in the array
- Semicolon separator
- Values for elements two and three in the array

Use PRO_ARR_ESCAPE to insert any necessary escape characters into the new value. Use PRO_ELEMENT to extract the values for elements two and three from the array. This example assigns the value 'aaa' to the first element of the arraychar array for customer 99, and retains the existing values for elements two and three.

```
UPDATE customer SET arraychar =
     PROARR_ESCAPE('aaa')
     || ';'
     || PRO_ELEMENT(arraychar,2,3)
WHERE cust_num = 99 ;
```

To assign a value to the second element of an array of size three, construct a string that concatenates these components:

- Value of the first element in the array
- Semicolon separator
- Update value for the second element
- Semicolon separator
- Value of the third element in the array

Use PRO_ARR_ESCAPE to insert any necessary escape characters into the new value. Use PRO_ELEMENT to extract the first and third elements from the array. This example assigns the value 'bbb' to the second element of the arraychar array for customer 99, and retains the existing values for elements one and three in arraychar.

```
UPDATE customer SET arraychar =
     PRO_ELEMENT(arraychar,1,1)
     || ';'
     || PROARR_ESCAPE('bbb')
     || ';'
     || PRO_ELEMENT(arraychar,3,3)
WHERE cust_num = 99 ;
```

D

Progress SQL-92 Elements and Statements in Backus Naur Form (BNF)

This reference appendix presents Progress SQL-92 language elements and SQL-92 statements in Backus Naur Form (BNF). The elements and statements are presented in this order:

- Data types
- Expressions
- Literals
- Query expressions
- Search conditions
- Statements, DDL and DML

Data Types Syntax in BNF

DATA TYPE

SYNTAX

```
data_type ::=
char_data_type
   | exact_numeric_data_type | approx_numeric_data_type
   | date_time_data_type | bit_string_data_type
```

CHARACTER DATA TYPE

SYNTAX

```
char_data_type ::=
{ CHARACTER | CHAR } [ ( length ) ]
   | { CHARACTER VARYING | CHAR VARYING | VARCHAR }
   [ (length) ]
```

EXACT NUMERIC DATA TYPE

SYNTAX

```
exact_numeric_data_type ::=
TINYINT
   | SMALLINT
   | INTEGER
   | NUMERIC | NUMBER [ ( precision [ , scale ] ) ]
   | DECIMAL [ ( precision , scale ) ]
```

APPROXIMATE NUMERIC DATA TYPE

SYNTAX

```
approx_numeric_data_type ::=
   { REAL | DOUBLE PRECISION | FLOAT [ ( precision ) ] }
```

DATE-TIME DATA TYPE

SYNTAX

```
date_time_data_type ::
DATE | TIME | TIMESTAMP
```

BIT STRING DATA TYPE

SYNTAX

```
bit_string_data_type ::=
BIT | BINARY [ ( length ) ] | VARBINARY [( length ) ]
    | LONG VARBINARY [( length ) ]
```

Expressions Syntax in BNF

EXPRESSION (expr)

SYNTAX

```
expr ::=
[ { table_name.| alias.}] column_name
  | character_literal
  | numeric_literal
  | date-time_literal
  | aggregate_function
  | scalar_function
  | numeric_arith_expr
  | date_arith_expr
  | conditional_expr
  | (expr)
```

NUMERIC ARITHMETIC EXPRESSION

SYNTAX

```
numeric_arith_expr ::=
[ + | - ] { numeric_literal | numeric_expr }
  [ { + | - | * | / } numeric_arith_expr ]
```

DATE ARITHMETIC EXPRESSION

SYNTAX

```
date_arith_expr ::=
date_time_expr { + | - } int_expr
  | date_time_expr - date_time_expr
```

CONDITIONAL EXPRESSION

CASE EXPRESSION

A type of conditional expression.

SYNTAX

```
case_expr ::=
searched_case_expr | simple_case_expr
```

SEARCHED CASE EXPRESSION

SYNTAX

```
searched_case_expr ::=
CASE
  WHEN search_condition THEN { result_expr | NULL } [ , ... ]
[ ELSE expr | NULL ]
END
```

SIMPLE CASE EXPRESSION

SYNTAX

```
simple_case_expr ::=
CASE primary_expr
  WHEN expr THEN { result_expr | NULL } [ , ... ]
[ ELSE expr | NULL ]
END
```

Literals Syntax in BNF

DATE-TIME LITERAL Syntax

DATE LITERAL

SYNTAX

```
date-literal ::=
{ d 'yyyy-mm-dd' }
   | mm-dd-yyyy
   | mm/dd/yyyy
   | mm-dd-yy
   | mm/dd/yy
   | yyyy-mm-dd
   | yyyy/mm/dd
   | dd-mon-yyyy
   | dd/mon/yyyy
   | dd-mon-yy
   | dd/mon/yy
```

TIME LITERAL

SYNTAX

```
time_literal ::=
{ t 'hh:mi:ss' } | hh:mi:ss[:mls ]
```

Query Expressions Syntax in BNF

QUERY EXPRESSION

SYNTAX

```
query_expression ::=
query_specification
  |  query_expression set_operator query_expression
  |  ( query_expression )
```

SET OPERATOR

SYNTAX

```
set_operator ::=
{ UNION [ ALL ] | INTERSECT | MINUS }
```

QUERY SPECIFICATION

SYNTAX

```
query_specification ::=
SELECT [ ALL | DISTINCT ]
  { *
    | { table_name.| alias.}* [, { table_name.| alias.}* ]  ,...
    | expr [ [ AS ] [ ']column_title[' ] ]
      [, expr [ [ AS ] [ ']column_title[' ] ] ] ,...
  }
FROM table_ref [, table_ref ] ...
[ WHERE search_condition ]
[ GROUP BY [ table.]column_name
  [, [ table.]column_name ] ,...
[ HAVING search_condition ]
;
```

TABLE REFERENCE

SYNTAX

```
table_ref ::=
table_name [ AS ] [ alias [ (column_alias [, ... ] ) ] ]
  | (query_expression) [ AS ] alias [ (column_alias [ ,... ] ) ]
  | [ ( ] joined_table [ ) ]
```

JOINED TABLE

SYNTAX

```
joined_table ::=
{ table_ref CROSS JOIN table_ref
  |table_ref [ INNER | LEFT [ OUTER ] ] JOIN
    table_ref ON search_condition
}
```

FROM CLAUSE INNER JOIN

SYNTAX

```
from_clause_inner_join ::=
{ FROM table_ref CROSS JOIN table_ref
  | FROM table_ref [ INNER ] JOIN table_ref
    ON search_condition
}
```

WHERE CLAUSE INNER JOIN

SYNTAX

```
where_clause_inner_join ::=
FROM table_ref, table_ref WHERE search_condition
```

FROM CLAUSE OUTER JOIN

SYNTAX

```
from_clause_outer_join ::=
FROM table_ref LEFT OUTER JOIN table_ref
  ON search_condition
```

WHERE CLAUSE OUTER JOIN

SYNTAX

```
where_clause_outer_join ::=
WHERE [ table_name.]column (+) = [ table_name.]column
  | WHERE [ table_name.]column = [ table_name.]column (+)
```

Search Conditions Syntax in BNF

SEARCH CONDITION

SYNTAX

```
search_condition ::=
[ NOT ] predicate
  [ { AND | OR } { predicate | ( search_condition ) } ]
```

PREDICATE

SYNTAX

```
predicate ::=
basic_predicate
   | quantified_predicate
   | between_predicate
   | null_predicate
   | like_predicate
   | exists_predicate
   | in_predicate
   | outer_join_predicate
```

RELATIONAL OPERATOR

SYNTAX

```
relop ::=
=  | <> | != | ^= | < | <= | > | >=
```

BASIC PREDICATE

SYNTAX

```
basic_predicate ::=
expr relop { expr | ( query_expression ) }
```

QUANTIFIED PREDICATE

SYNTAX

```
quantified_predicate ::=
expr relop { ALL | ANY | SOME } ( query_expression )
```

BETWEEN PREDICATE

SYNTAX

```
between_predicate ::=
expr [ NOT ] BETWEEN expr AND expr
```

NULL PREDICATE

SYNTAX

```
null_predicate ::=
column_name IS [ NOT ] NULL
```

LIKE PREDICATE

SYNTAX

```
like_predicate ::=
column_name [ NOT ] LIKE string_constant
  [ ESCAPE escape_character ]
```

EXISTS PREDICATE

SYNTAX

```
exists_predicate ::=
EXISTS (query_expression)
```

IN PREDICATE

SYNTAX

```
in_predicate ::=
expr [ NOT ] IN
  { (query_expression) | (constant , constant [ , ... ] ) }
```

OUTER JOIN PREDICATE

SYNTAX

```
outer_join_predicate ::=
[ table_name.]column = [ table_name.]column (+)
  | [ table_name.]column (+) = [ table_name.]column
```

Statements, DDL and DML Syntax in BNF

This sections lists Progress SQL-92 Data Definition Language (DDL) and Data Manipulation Language (DML) statements in Backus-Naur Form (BNF).

ALTER USER Statement

SYNTAX

```
alter user statement ::=
ALTER USER 'username', 'old_password', 'new_password' ;
```

BEGIN-END DECLARE SECTION

SYNTAX

```
begin declare section ::=
EXEC SQL BEGIN DECLARE SECTION
host_lang_type variable_name ;
  .
  .
  .
END DECLARE SECTION ::=
EXEC SQL END DECLARE SECTION
```

Host Language Type

SYNTAX

```
host language type ::=
{   char
  | short
  | long
  | float
  | double
}
```

CALL Statement

SYNTAX

```
call statement ::=
CALL proc_name ( [ parameter ] [ , ... ] ) ;
```

CLOSE Statement

SYNTAX

```
close ::=
EXEC SQL CLOSE cursor_name ;
```

COMMIT Statement

SYNTAX

```
commit statement ::=
COMMIT [ WORK ] ;
```

CONNECT Statement

SYNTAX

```
connect statement ::=
CONNECT TO connect_string
  [ AS connection_name ]
  [ USER user_name ]
  [ USING password ] ;
```

CONNECT STRING Statement

SYNTAX

```
connect_string ::=
{ DEFAULT | db_name
  | db_type:T:host_name:port_num:db_name }
```

CREATE INDEX Statement

SYNTAX

```
create index statement ::=
CREATE [ UNIQUE ] INDEX index_name
  ON table_name
  ( { column_name [ ASC | DESC ] } [, ... ] )
  [ AREA area_name ] ;
```

CREATE PROCEDURE Statement

SYNTAX

```
create procedure statement ::=
CREATE PROCEDURE [ owner_name.]procname
  ( [ parameter_decl [ , ..... ] ]
  )
    [ RESULT ( column_name data_type [ , ... ] ) ]
    [ IMPORT
      java_import_clause ]
  BEGIN
      java_snippet
  END
```

Parameter Declaration

SYNTAX

```
parameter_decl ::=
{ IN | OUT | INOUT } parameter_name data_type
```

CREATE SYNONYM Statement

SYNTAX

```
create synonym statement ::=
CREATE [ PUBLIC ] SYNONYM synonym
  FOR [ owner_name.] {table_name |view_name | synonym } ;
```

CREATE TABLE Statement

SYNTAX

```
create table statement ::=
CREATE TABLE [ owner_name.]table_name
  ( { column_definition | table_constraint }, ... )
    [ AREA area_name ]
;

create table statement ::=
CREATE TABLE [ owner_name.]table_name
  [ (column_name [ NULL | NOT NULL] , ...) ]
  [ AREA area_name ]
  AS query_expression
;
```

Column Definition

SYNTAX

```
column_definition ::=
column_name data_type
  [ DEFAULT { literal | NULL | SYSDATE } ]
  [ column_constraint [ column_constraint ... ] ]
```

Column Constraint

SYNTAX

```
column_constraint ::=
[ CONSTRAINT constraint_name ]
   NOT NULL [ PRIMARY KEY | UNIQUE ]
  | REFERENCES [ owner_name.]table_name [ ( column_name ) ]
  | CHECK ( search_condition )
```

Tabel Constraint

SYNTAX

```
table_constraint ::=
[ CONSTRAINT constraint_name ]
    PRIMARY KEY ( column [, ... ] )
  | UNIQUE ( column [, ..... ] )
  | FOREIGN KEY ( column [, ... ] )
    REFERENCES [ owner_name.]table_name [ ( column [, ... ] ) ]
  | CHECK ( search_condition )
```

CREATE TRIGGER Statement

SYNTAX

```
create trigger statement ::=
CREATE TRIGGER [ owner_name.]trigname
  { BEFORE | AFTER }
  { INSERT | DELETE | UPDATE [ OF column_name [ , ... ] }
  ON table_name
  [ REFERENCING { OLDROW [, NEWROW ] | NEWROW [, OLDROW ] } ]
  [ FOR EACH { ROW | STATEMENT } ]
  [ IMPORT
    java_import_clause ]
    BEGIN
          java_snippet
    END
```

CREATE USER Statement

SYNTAX

```
create user statement ::=
CREATE USER 'username', 'password' ;
```

CREATE VIEW Statement

SYNTAX

```
create view statement ::=
CREATE VIEW [ owner_name.]view_name
  [ ( column_name, column_name,... ) ]
  AS [ ( ] query_expression [ ) ]
  [ WITH CHECK OPTION ] ;
```

DECLARE CURSOR Statement

SYNTAX

```
declare cursor ::=
EXEC SQL DECLARE cursor_name CURSOR FOR
  { query_expr [ ORDER BY clause ] [ FOR UPDATE clause ]
  | prepared_statement_name
  } ;
```

DELETE Statement

SYNTAX

```
delete statement ::=
DELETE FROM [ owner_name.] {table_name |view_name }
  [ WHERE search_condition ] ;
```

DESCRIBE BIND VARIABLES Statement

SYNTAX

```
describe bind variables ::=
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
  INTO input_sqlda_name ;
```

DESCRIBE SELECT LIST Statement

SYNTAX

```
describe select list ::=
EXEC SQL DESCRIBE SELECT LIST FOR statement_name
  INTO output_sqlda_name ;
```

DISCONNECT Statement

SYNTAX

```
disconnect statement ::=
  DISCONNECT { 'connection_name' | CURRENT | ALL | DEFAULT } ;
```

DROP INDEX Statement

SYNTAX

```
drop index statement ::=
DROP INDEX [ index_owner_name.] index_name
  [ ON [ table_owner_name.] table_name ]
```

DROP PROCEDURE Statement (ODBC Core SQL Grammar)

SYNTAX

```
drop procedure statement ::=
DROP PROCEDURE [ owner_name.] procedure_name ;
```

DROP SYNONYM Statement

SYNTAX

```
drop synonym statement ::=
DROP [ PUBLIC ] SYNONYM synonym ;
```

DROP TABLE Statement

SYNTAX

```
drop table statement ::=
DROP TABLE  [ owner_name.] table_name ;
```

DROP TRIGGER Statement

SYNTAX

```
drop trigger statement ::=
DROP TRIGGER [ owner_name.] trigger_name ;
```

DROP USER Statement

SYNTAX

```
drop user statement ::=
DROP USER 'username' ;
```

DROP VIEW Statement

SYNTAX

```
drop view statement ::=
DROP VIEW [ owner_name.] view_name ;
```

EXEC SQL Statement

SYNTAX

```
EXEC SQL ::=
EXEC SQL  sql_statement ;
```

EXECUTE Statement

SYNTAX

```
EXECUTE ::=
EXEC SQL EXECUTE statement_name
  [ USING { [ SQL ] DESCRIPTOR structure_name
    | :host_variable [ [ INDICATOR ] :ind_variable ] , ... }
  ] ;
```

EXECUTE IMMEDIATE Statement

SYNTAX

```
EXECUTE IMMEDIATE ::=
EXEC SQL EXECUTE IMMEDIATE
  { statement_string | host_variable } ;
```

FETCH Statement

SYNTAX

```
fetch ::=
EXEC SQL FETCH cursor_name
  { USING SQL DESCRIPTOR structure_name
    | INTO :host_var_ref [ [ INDICATOR ] :ind_var_ref ] , ...
  } ;
```

GET DIAGNOSTICS Statement

SYNTAX

```
get diagnostics statement ::=
GET DIAGNOSTICS
  :param = header_info_item
    [ , :param = header_info_item ] ,...
;
```

Header Info Item

SYNTAX

```
header_info_item ::=
{ NUMBER
  | MORE
  | COMMAND_FUNCTION
  | DYNAMIC_FUNCTION
  | ROW_COUNT
}
```

GET DIAGNOSTICS EXCEPTION Syntax

SYNTAX

```
get diagnostics exception statement ::=
GET DIAGNOSTICS EXCEPTION number
  :param = detail_info_item
    [, :param = detail_info_item ] , ...
;
```

Detail Info Item Syntax

SYNTAX

```
detail_info_item ::=
{ CONDITION_NUMBER
  | RETURNED_SQLSTATE
  | CLASS_ORIGIN
  | SUBCLASS_ORIGIN
  | ENVIRONMENT_NAME
  | CONNECTION_NAME
  | CONSTRAINT_CATALOG
  | CONSTRAINT_SCHEMA
  | CONSTRAINT_NAME
  | CATALOG_NAME
  | SCHEMA_NAME
  | TABLE_NAME
  | COLUMN_NAME
  | CURSOR_NAME
  | MESSAGE_TEXT
  | MESSAGE_LENGTH
  | MESSAGE_OCTET_LENGTH
}
```

GRANT RESOURCE, DBA Statement

SYNTAX

```
grant resource, dba statement ::=
GRANT { RESOURCE, DBA } TO user_name [ , user_name  ] , ...
;
```

GRANT PRIVILEGE Statement

SYNTAX

```
grant privilege statement ::=
GRANT { privilege [, privilege ], ... | ALL [ PRIVILEGES] }
  ON table_name
  TO  { user_name [, user_name ] , ... | PUBLIC }
  [WITH GRANT OPTION] ;
```

PRIVILEGE

SYNTAX

```
privilege ::=
{ SELECT | INSERT | DELETE | INDEX
  |UPDATE [ ( column , column , ... ) ]
  | REFERENCES [ ( column , column , ... ) ] }
```

INSERT Statement

SYNTAX

```
insert statement ::=
INSERT INTO [ owner_name.] {table_name |view_name }
  [ ( column_name [, column_name ] , ... ) ]
  { VALUES ( value [, value ] ,... ) | query_expression } ;
```

See also the Query Expressions Syntax in BNF.

LOCK TABLE Statement

SYNTAX

```
lock table statement ::=
LOCK TABLE table_name [ , table_name ] , ... IN { SHARE | EXCLUSIVE } MODE ;
```

OPEN Statement

SYNTAX

```
open ::=
EXEC SQL OPEN cursor_name
  [ USING { [ SQL ] DESCRIPTOR structure_name
    | :host_variable [ [ INDICATOR ] :ind_variable ] , ... }
  ] ;
```

PREPARE Statement

SYNTAX

```
prepare ::=
EXEC SQL PREPARE statement_name FROM statement_string ;
```

REVOKE RESOURCE, DBA Statement

SYNTAX

```
revoke resource, dba statement ::=
REVOKE  { RESOURCE | DBA }
    FROM { user_name [ , user_name ] ,... } ;
```

REVOKE PRIVILEGE Statement

SYNTAX

```
revoke privilege statement ::=
REVOKE [ GRANT OPTION FOR ]
  { privilege [, privilege , ] , ... | ALL [ PRIVILEGES ] }
  ON table_name
  FROM { user_name [ , user_name ] , ... | PUBLIC }
    [ RESTRICT | CASCADE ] ;
```

PRIVILEGE Syntax

SYNTAX

```
privilege ::=
{ SELECT | INSERT | DELETE | INDEX
  |UPDATE [ ( column , column , ... ) ]
  | REFERENCES [ ( column , column , ... ) ] }
```

ROLLBACK Statement

SYNTAX

```
rollback statement ::=
ROLLBACK [ WORK ] ;
```

SELECT Statement

SYNTAX

```
select statement ::=
query_expression
  ORDER BY { expr | posn } [ ASC | DESC ]
    [ , { expr | posn } [ ASC | DESC ] , ... ]
  FOR UPDATE [ OF [ table.]column_name , ... ] [ NOWAIT ]
;
```

See also section Query Expressions Syntax in BNF.

SET CONNECTION Statement

SYNTAX

```
set connection statement ::=
  SET CONNECTION { 'connection_name' | DEFAULT } ;
```

SET SCHEMA Statement

SYNTAX

```
set schema statement ::=
SET SCHEMA { 'string_literal' | ? | :host_var | USER }
```

SET TRANSACTION ISOLATION LEVEL Statement

SYNTAX

```
set transaction isolation level statement ::=
SET TRANSACTION ISOLATION LEVEL isolation_level_name ;
```

ISOLATION LEVEL NAME

SYNTAX

```
isolation_level_name ::=
  READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE
```

UPDATE Statement

SYNTAX

```
update statement ::=
UPDATE table_name
  SET assignment [,assignment ], ... [ WHERE search_condition ] ;
```

ASSIGNMENT CLAUSE

SYNTAX

```
assignment ::=
column = { expr | NULL }
  | ( column [, column ], ... ) = ( expr [, expr ], ... )
  | ( column [, column ], ... ) = ( query_expression )
```

UPDATE STATISTICS Statement

SYNTAX

```
update statistics statement ::=
UPDATE STATISTICS [ FOR table_name ]
```

WHENEVER Statement

SYNTAX

```
whenever ::=
EXEC SQL WHENEVER
  { NOT FOUND | SQLERROR | SQLWARNING }
  { STOP | CONTINUE | { GOTO | GO TO } host_lang_label } ;
```

E

Compliance with Industry Standards

This appendix identifies the level of ANSI SQL-92 compliance and ODBC SQL Grammar compliance for Progress SQL-92 statements, and the SQL-92 and ODBC compatibility for Progress SQL-92 scalar functions.

Scalar Functions

This section listsProgress SQL-92 scalar functions. Table E–1 identifies the compatibility of the function as SQL-92 compatible, ODBC compatible, or a Progress extension.

Table E–1: Compatibility of SQL-92 Scalar Functions *(1 of 5)*

Scalar Function	SQL-92	ODBC	Progress Extension	Notes
ABS	–	√	–	–
ACOS	–	√	–	–
ADD_MONTHS	–	–	√	–
ASCII	–	√	–	–
ASIN	–	√	–	–
ATAN	–	√	–	–
ATAN2	–	√	–	–
CAST	√	–	–	–
CEILING	–	√	–	–
CHAR	–	√	–	–
CHR	–	–	√	–
COALESCE	√	–	–	–
CONCAT	–	√	–	–
CONVERT	–	√	–	Requires ODBC escape clause – { fn }
CONVERT	–	–	√	Not compatible with ODBC CONVERT
COS	–	√	–	–
CURDATE	–	√	–	–

Table E–1: Compatibility of SQL-92 Scalar Functions *(2 of 5)*

Scalar Function	SQL-92	ODBC	Progress Extension	Notes
CURTIME	–	√	–	–
DATABASE	–	√	–	–
DAYNAME	–	√	–	–
DAYOFMONTH	–	√	–	–
DAYOFWEEK	–	√	–	–
DAYOFYEAR	–	√	–	–
DB_NAME	–	–	√	–
DECODE	–	–	√	–
DEGREES	–	√	–	–
EXP	–	√	–	–
FLOOR	–	√	–	–
GREATEST	–	–	√	–
HOUR	–	√	–	–
IFNULL	–	√	–	–
INITCAP	–	–	√	–
INSERT	–	√	–	–
INSTR	–	–	√	–
LAST_DAY	–	–	√	–
LCASE	–	√	–	–
LEAST	–	–	√	–
LEFT	–	√	–	–

Table E–1: Compatibility of SQL-92 Scalar Functions *(3 of 5)*

Scalar Function	SQL-92	ODBC	Progress Extension	Notes
LENGTH	–	√	–	–
LOCATE	–	√	–	–
LOG10	–	√	–	–
LOWER	√	–	–	–
LPAD	–	–	√	–
LTRIM	–	√	–	–
MINUTE	–	√	–	–
MOD	–	√	–	–
MONTH	–	√	–	–
MONTHNAME	–	√	–	–
MONTHS_BETWEEN	–	–	√	–
NEXT_DAY	–	–	√	–
NOW	–	√	–	–
NULLIF	√	–	–	–
NVL	–		√	–
PI	–	√	–	–
POWER	–	–	√	–
PREFIX	–	–	√	–
PRO_ARR_DESCAPE	–	–	√	–
PRO_ARR_ESCAPE	–	–	√	–
PRO_ELEMENT	–	–	√	–

Table E–1: Compatibility of SQL-92 Scalar Functions *(4 of 5)*

Scalar Function	SQL-92	ODBC	Progress Extension	Notes
QUARTER	–	√	–	–
RADIAN	–	√	–	–
RAND	–	√	–	–
REPEAT	–	√	–	–
REPLACE	–	√	–	–
RIGHT	–	√	–	–
ROUND	–	–	√	–
ROWID	–	–	√	–
RPAD	–	–	√	–
RTRIM	–	√	–	–
SECOND	–	√	–	–
SIGN	–	√	–	–
SIN	–	√	–	–
SQRT	–	√	–	–
SUBSTR	–	–	√	–
SUBSTRING	–	√	–	–
SUFFIX	–	–	√	–
SYSDATE	–	–	√	–
SYSTIME	–	–	√	–
SYSTIMESTAMP	–	–	√	–
TAN	–	√	–	–

Table E–1: Compatibility of SQL-92 Scalar Functions *(5 of 5)*

Scalar Function	SQL-92	ODBC	Progress Extension	Notes
TO_CHAR	–	–	√	–
TO_DATE	–	–	√	–
TO_NUMBER	–	–	√	–
TO_TIME	–	–	√	–
TO_TIMESTAMP	–	–	√	–
TRANSLATE	–	–	√	–
UCASE	–	√	–	–
UPPER	√	–	–	–
USER	√	√	√	–
WEEK	–	√	–	–
YEAR	–	√	–	–

SQL-92 DDL and DML Statements

This section lists Progress SQL-92 DDL and DML Statements. Table E–2 identifies the compliance of each statement as SQL-92, a level of ODBC SQL Grammar, or as a Progress extension.

Table E–2: Compliance of SQL-92 DDL and DML Statements *(1 of 4)*

SQL-92 Statement	SQL-92	ODBC SQL Grammar	Progress Extension	Notes
ALTER USER	–	–	√	–
CALL	–	Extended	–	Must enclose in an ODBC escape clause { fn }
BEGIN-END DECLARE SECTION	√	–	–	Compliant if C Language types used Embedded SQL-92 only
CLOSE	√	–	–	Embedded SQL-92 only
COMMIT	√	–	–	–
CONNECT	√	–	`USING password`	–
CREATE INDEX	–	Core	`AREA area_name`	–
CREATE PROCEDURE	–	Core	√	–
CREATE SYNONYM	–	–	√	–
CREATE TABLE	√	Minimum	`AREA AS query_expression`	–
CREATE TRIGGER	–	Core	√	–

Table E–2: Compliance of SQL-92 DDL and DML Statements *(2 of 4)*

SQL-92 Statement	SQL-92	ODBC SQL Grammar	Progress Extension	Notes
CREATE USER	–	–	√	–
CREATE VIEW	√	Core	–	–
DECLARE CURSOR	√	–	*prepared_stmt_name*	Embedded SQL-92 only
DELETE	√	Extended	–	–
DESCRIBE	√	–	–	Embedded SQL-92 only
DISCONNECT	√	–	–	–
DROP INDEX	–	Core	–	–
DROP PROCEDURE	–	Core	√	–
DROP SYNONYM	–	–	√	–
DROP TABLE	–	Minimum	√	–
DROP TRIGGER	–	–	√	–
DROP USER	–	–	√	–
DROP VIEW	–	Core	√	–
EXEC SQL	√	–	–	Embedded SQL-92 only
EXECUTE	√	–	–	Embedded SQL-92 only
EXECUTE IMMEDIATE	√	–	–	Embedded SQL-92 only
FETCH	√	–	USING DESCRIPTOR	Embedded SQL-92 only

Table E–2: Compliance of SQL-92 DDL and DML Statements *(3 of 4)*

SQL-92 Statement	SQL-92	ODBC SQL Grammar	Progress Extension	Notes
GET DIAGNOSTICS	–	–	√	Embedded SQL-92 only

GRANT	√	Core	`INDEX`
			`RESOURCE`
			`DBA`
				–

INSERT	√	Core	–	–
LOCK TABLE	–	–	√	–
OPEN	√	–	`USING DESCRIPTOR`	Embedded SQL-92 only
PREPARE	√	–	–	Embedded SQL-92 only

REVOKE	√	Core	`INDEX`
			`RESOURCE`
			`DBA`
				–

ROLLBACK	√	–	–	–
SELECT	√	Extended	`FOR UPDATE`	–
SET CONNECTION	√	–	–	–
SET SCHEMA	√	–	–	–
SET TRANSACTION ISOLATION LEVEL	–	–	√	–
UPDATE	√	Extended	assignments of form: *(column, column)* *= (expr, expr)*	–

Table E–2: Compliance of SQL-92 DDL and DML Statements *(4 of 4)*

SQL-92 Statement	SQL-92	ODBC SQL Grammar	Progress Extension	Notes
UPDATE STATISTICS	–	–	√	–
WHENEVER	√	–	`SQLWARNING STOP ACTION`	Embedded SQL-92 only

Glossary

Add [an ODBC Data Source]

Makes a data source available to ODBC through the Add operation of the ODBC Administrator utility. Adding a data source tells ODBC where a specific database resides and which ODBC driver to use to access it. Adding a data source also invokes a setup dialog box for the particular driver so you can provide other details the driver needs to connect to the database.

Alias

A temporary name for a table or column specified in the FROM clause of an SQL query expression. Also called **Correlation Name**. Derived tables and search conditions that join a table with itself must specify an alias. Once a query specifies an alias, references to the table or column must use the alias and not the underlying table or column name.

Applet

A special kind of Java program that a Java-enabled browser can download over the network and execute.

ASCII

(American Standard Code for Information Interchange) seven-bit character set that provides 128 character combinations.

Bytecode

Machine-independent code generated by a Java compiler and executed by a Java interpreter.

Cardinality

Number of rows in a result table.

Cartesian Product

Also called **Cross-Product**. In a query expression, the result table generated when a FROM clause lists more than one table but specifies no join conditions. In such a case, the result table is formed by combining every row of every table with all other rows in all tables. Typically, Cartesian products are not useful and are slow to process.

Client

Generally, in client server systems, the part of the system that sends requests to **Servers** and receives the results produced by acting on those requests.

Collation

The rules used to control how character strings in a character set compare with each other. Each character set specifies a collating sequence that defines relative values of each character for comparing, merging, and sorting character strings.

Column Alias

An alias specified for a column. See **Alias**.

Constraint

Part of an SQL table definition that restricts the values that can be stored in a table. When you insert, delete, or update column values, the constraint checks the new values against the conditions specified by the constraint. If the value violates the constraint, it generates an error. Along with **Triggers**, constraints enforce **Referential Integrity** by ensuring that a value stored in the foreign key of a table must either be null or be equal to some value in the matching unique or primary key of another table.

Correlation Name

Another term for **Alias**.

Cross Product

Another term for **Cartesian Product**.

Data Dictionary

Another term for **System Catalog**.

Data Source

See **ODBC Data Source**.

Derived Table

A **Virtual Table** specified as a query expression in the FROM clause of another query expression.

Driver Manager

See **JDBC Driver Manager** and **ODBC Driver Manager**.

Form of Use

The storage format for characters in a character set. Some character sets, such as **ASCII**, require one byte (**Octet**) for each character. Others, such as **Unicode**, use multiple bytes, and are called multi-octet character sets.

Java Snippet

See **Snippet**.

JDBC

(Java Database Connectivity) A database-independent SQL interface that allows Java programs to access relational databases. JDBC is quite similar to ODBC.

JDBC Driver

Database-specific software that receives calls from the **JDBC Driver Manager**, translates them into a form that a database server can process, and then returns data to the application.

JDBC Driver Manager

A Java class that implements methods to route calls from a **JDBC** application to the appropriate **JDBC Driver** for a particular JDBC URL.

Join

A relational operation that combines data from two tables.

Input Parameter

In a **Stored Procedure** specification, an argument that an application must pass when it calls the stored procedure. In an SQL statement, a **Parameter Marker** in the statement string that acts as a placeholder for a value that will be substituted when the statement executes.

Interface

In Java, a definition of a set of methods that one or more objects will implement. Interfaces declare only methods and constants, not variables.

Metadata

Data that describes the objects (tables, columns, views, and indexes) that are stored in the database. Metadata is "data about data." It is stored in a collection of tables called **System Tables**.

Octet

A group of eight bits. Synonymous with byte, and often used in descriptions of character set encoding format.

ODBC Application

Any program that calls ODBC functions and uses them to issue SQL statements.

ODBC Data Source

In ODBC terminology, a specific combination of a database system, the operating system it uses, and any network software required to access it. Before applications can access a database through ODBC, you use the ODBC Administrator to add a data source-register information about the database and an ODBC driver that can connect to it-for that database. More than one data source name can refer to the same database, and deleting a data source does not delete the associated database.

ODBC Driver

Software that processes ODBC function calls for a specific data source. The driver connects to the data source, translates the standard SQL statements into syntax the data source can process, and returns data to the application. Progress SQL-92 includes an **ODBC Driver**.

ODBC Driver Manager

A Microsoft-supplied program that routes calls from an application to the appropriate ODBC driver for a data source.

Output Parameter

In a stored procedure specification, an argument in which the stored procedure returns a value after it executes.

Package

A group of related Java classes and interfaces, like a class library in C++. The Java development environment includes many packages of classes that procedures can import. The Java run-time system automatically imports the *java.lang* package. Stored procedures must explicitly import other classes by specifying them in the IMPORT clause of a CREATE PROCEDURE statement.

Parameter Marker

A question mark (?) character in a procedure call or SQL statement string that acts as a placeholder for a parameter's value. The actual value for the input or output parameter will be supplied at run time when the procedure executes. The CALL statement (or corresponding ODBC or JDBC escape clause) uses parameter markers to pass parameters to stored procedures, and the *SQLIStatement*, *SQLPStatement*, and *SQLCursor* objects use them within procedures.

Primary Key

A subset of the columns in a table, characterized by the constraint that no two records in a table can have the same primary key value, and that no columns of the primary key can have a null value. Primary keys are specified in a CREATE TABLE statement.

Procedure Body

In a Stored Procedure, the Java code between the BEGIN and END keywords of a CREATE PROCEDURE statement.

Procedure Result Set

In a stored procedure, a set of data rows returned to the calling application. The number and data types of columns in the procedure result set are specified in the RESULT clause of the CREATE PROCEDURE statement. The procedure can transfer data from an **SQL Result Set** to the procedure result set or it can store data generated internally. A stored procedure can have only one procedure result set.

Procedure Specification

In a CREATE PROCEDURE statement, the clauses preceding the procedure body that specify the procedure name, any input and output parameters, any result set columns, and any Java packages to import.

Procedure Variable

A Java variable declared within the body of a stored procedure, as compared to a procedure **Input Parameter** or **Output Parameter**, which are declared outside the procedure body and are visible to the application that calls the stored procedure.

Query Expression

An important element of the SQL languages. Query expressions specify a result table derived from some combination of rows from the tables or views identified in the FROM clause of the expression. Query expressions are the basis of the SELECT, UPDATE, DELETE, and INSERT statements, and can be used in some expressions and search conditions.

Referential Integrity

The condition where the value stored in a database table's foreign key must either be null or be equal to some value in another table's matching unique or primary key. SQL provides two mechanisms to enforce referential integrity: constraints specified as part of CREATE TABLE statements prevent updates that violate referential integrity, and **Triggers** specified in CREATE TRIGGER statements execute a stored procedure to enforce referential integrity.

Repertoire

The set of characters allowed in a character set.

Result Set

In a **Stored Procedure**, either an **SQL Result Set** or a **Procedure Result Set**.

More generally, another term for **Result Table**.

Result Table

A virtual table of values derived from columns and rows of one or more tables that meet conditions specified by an SQL query expression.

Row Identifier

Another term for **Tuple** identifier.

Search Condition

The SQL syntax element that specifies a condition that is true or false about a given row or group of rows. Query expressions and UPDATE statements can specify a search condition. The search condition restricts the number of rows in the result table for the query expression or UPDATE statement. Search conditions contain one or more predicates. Search conditions follow the WHERE or HAVING keywords in SQL statements.

Selectivity

The fraction of a table's rows returned by a query.

Server

Generally, in client server systems, the part of the system that receives requests from **Clients** and responds by sending back the results produced by acting on them.

Snippet

In a stored procedure, the sequence of Java statements between the BEGIN and END keywords in the CREATE PROCEDURE (or CREATE TRIGGER) statement. The Java statements become a method in a class that SQL creates and submits to the Java compiler.

SQL Diagnostics Area

A data structure that contains information about the execution status (success, error, or warning conditions) of the most recent SQL statement. The SQL-92 standard specified the diagnostics area as a standardized alternative to widely varying implementations of the SQLCA. Progress SQL-92 supports both the SQLCA and the SQL diagnostics area. The SQL GET DIAGNOSTICS statement returns information about the diagnostics area to an application, including the value of the SQLSTATE status parameter.

SQLCA

SQL Communications area. A data structure that contains information about the execution status (success, error, or warning conditions) of the most recent SQL statement. The SQLCA includes an SQLCODE field. The SQLCA provides the same information as the SQL diagnostics area, but is not compliant with the SQL-92 standard. Progress SQL-92 supports both the SQLCA and the SQL diagnostics area.

SQLCODE

An integer status parameter whose value indicates the condition status returned by the most recent SQL statement. An SQLCODE value of zero means success, a positive value means warning, and a negative value means an error status. SQLCODE is superseded by SQLSTATE in the SQL-92 standard. Applications declare either **SQLSTATE** or SQLCODE, or both. SQL returns the status to SQLSTATE or SQLCODE after execution of each SQL statement.

SQL Result Set

In a stored procedure, the set of data rows generated by an SQL statement (SELECT and, in some cases, CALL).

SQLSTATE

A five-character status parameter whose value indicates the condition status returned by the most recent SQL statement. SQLSTATE is specified by the SQL-92 standard as a replacement for the **SQLCODE** status parameter (which was part of SQL-89). SQLSTATE defines many more specific error conditions than SQLCODE, which allows applications to implement more portable error handling. Applications declare either SQLSTATE or **SQLCODE**, or both. SQL returns the status to SQLSTATE or **SQLCODE** after execution of each SQL statement.

Stored Procedure

A snippet of Java source code embedded in an SQL CREATE PROCEDURE statement. The source code can use all standard Java features as well as use Progress SQL-92-supplied Java classes for processing any number of SQL statements.

System Catalog

Tables created by SQL to store descriptions of objects (tables, columns, views, and indexes) that are stored in the database.

System Table

Another term for **System Catalog**.

Tid

Another term for **Tuple Identifier**.

Transaction

A group of operations whose changes can be made permanent or undone only as a unit.

Trigger

A special type of Stored Procedure that helps ensure referential integrity for a database. Like stored procedures, triggers also contain Java source code (embedded in a CREATE TRIGGER statement) and use Progress SQL-92 Java classes. However, triggers are automatically invoked ("fired") by certain SQL operations (an insert, update, or delete operation) on the trigger's target table.

Trigger Action Time

The BEFORE or AFTER keywords in a CREATE TRIGGER statement. The trigger action time specifies whether the actions implemented by the **Trigger** execute before or after the triggering INSERT, UPDATE, or DELETE statement.

Trigger Event

The statement that causes a trigger to execute. Trigger events can be SQL INSERT, UPDATE, or DELETE statements that affect the table for which a trigger is defined.

Triggered Action

The Java code within the BEGIN END clause of a CREATE TRIGGER statement. The code implements actions to be completed when a triggering statement specifies the target table.

Tuple Identifier

A unique identifier for a tuple (row) in a table. The SQL scalar function ROWID and related functions return tuple identifiers to applications.

Unicode

A superset of the **ASCII** character set that uses multiple bytes for each character rather than ASCII's seven-bit representation. Able to handle 65,536 character combinations instead of ASCII's 128, Unicode includes alphabets for many of the world's languages. The first 128 codes of Unicode are identical to ASCII, with a second-byte value of zero.

View

A **Virtual Table** that re-creates the result table specified by a SELECT statement. No data is stored in a view, but other queries can refer to it as if it were a table containing data corresponding to the result table it specifies.

Virtual Table

A table of values that is not physically stored in a database, but instead derived from columns and rows of other tables. SQL generates virtual tables in its processing of query expressions: the FROM, WHERE, GROUP BY, and HAVING clauses each generate a virtual table based on their input.

Virtual Machine

The Java specification for a hardware-independent and portable language environment. Java language compilers generate code that can execute on a virtual machine. Implementations of the Java virtual machine for specific hardware and software platforms allow the same compiled code to execute without modification.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y