

Progress SQL-89 Guide and Reference

© 2001 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Progress, Progress Results, Provision and WebSpeed are registered trademarks of Progress Software Corporation in the United States and other countries. Apptivity, AppServer, ProVision Plus, SmartObjects, IntelliStream, and other Progress product names are trademarks of Progress Software Corporation.

SonicMQ is a trademark of Sonic Software Corporation in the United States and other countries.

Progress Software Corporation acknowledges the use of Raster Imaging Technology copyrighted by Snowbound Software 1993-1997 and the IBM XML Parser for Java Edition.

© IBM Corporation 1998-1999. All rights reserved. U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Progress is a registered trademark of Progress Software Corporation and is used by IBM Corporation in the mark Progress/400 under license. Progress/400 AND 400® are trademarks of IBM Corporation and are used by Progress Software Corporation under license.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Any other trademarks and/or service marks contained herein are the property of their respective owners.

.

May 2001



Product Code: 4534
Item Number: 81103;9.1C

Contents

Preface . . . . . **ix**
Purpose ix
Audience ix
Organization of This Manual ix
Typographical Conventions x
Syntax Notation xi
Progress Messages xv
Other Useful Documentation xvii
Getting Started xvii
Development Tools xviii
Reporting Tools xix
4GL xx
Database xxi
DataServers xxi
SQL-89/Open Access xxi
SQL-92 xxii
Deployment xxii
WebSpeed xxiii
Reference xxiv

1. About Progress/SQL-89 . . . . . **1–1**
1.1 Terminology Differences Between SQL and Progress 1–2
1.2 Choosing a Tool for Creating and Maintaining Database Tables 1–2
1.3 Accessing Non-Progress Databases Through Progress/SQL 1–3
1.4 Progress 4GL Extensions to Progress/SQL 1–3
1.5 Differences Between Progress/SQL and Progress/ESQL 1–4
1.5.1 The Interactive SQL 1–4
1.5.2 Progress/ESQL 1–4

2. Progress/SQL-89 Components 2–1
2.1 The SQL Character Set 2–2
2.2 SQL Statements 2–2
2.3 Identifiers 2–2
2.4 Name Qualifiers 2–3
2.5 Literals 2–4
2.6 Operators 2–4
2.7 Null Values 2–5
2.8 Logical Expressions 2–5

3. Data Manipulation Language 3–1
3.1 Using the SELECT Statement 3–2
3.1.1 SELECT Statement Syntax 3–3
3.1.2 Specifying Search Conditions 3–4
3.1.3 Specifying Frame Properties 3–7
3.1.4 Grouping Data 3–8
3.1.5 Sorting Data 3–9
3.1.6 Exporting Data 3–10
3.1.7 Using Subqueries 3–10
3.1.8 Using Aggregate Functions 3–13
3.2 Selecting Data from Multiple Tables (Joins) 3–14
3.2.1 Understanding Join Types 3–16
3.2.2 Specifying Implicit Joins 3–18
3.2.3 Specifying Explicit Joins 3–19
3.2.4 Choosing Explicit Joins or Implicit Joins 3–22
3.3 Combining SELECT Statements (UNION) 3–23
3.4 Inserting Rows 3–24
3.5 Updating Rows 3–26
3.6 Deleting Rows 3–26
3.7 Using Cursors 3–27
3.7.1 Cursor Statements 3–28
3.7.2 Defining a Cursor 3–30
3.7.3 Opening a Cursor 3–33
3.7.4 Retrieving Values from the Retrieval Set 3–33
3.7.5 Positioned UPDATE 3–35
3.7.6 Positioned DELETE 3–36
3.7.7 Closing a Cursor 3–36
3.7.8 Cursors Within Internal Procedures 3–37
3.8 Selecting Single Rows 3–37
3.9 Transaction Processing 3–38
3.10 Database Triggers 3–40
3.11 Privilege Checking 3–41

4. Data Definition Language 4–1
4.1 Tables 4–2
4.1.1 Creating Tables 4–3
4.1.2 Altering Tables 4–5
4.1.3 Deleting Tables 4–6
4.2 Data Types 4–6
4.3 Views 4–8
4.3.1 Creating Views 4–8
4.3.2 Deleting Views 4–10
4.3.3 Updating Views 4–11
4.4 Indexes 4–12
4.4.1 Creating Indexes 4–12
4.4.2 Deleting Indexes 4–13
4.5 Access Privileges 4–13
4.5.1 Granting Privileges 4–14
4.5.2 Revoking Privileges 4–15
4.6 Creating a Schema 4–16

A. Progress/SQL-89 Reference A–1
ALTER TABLE Statement A–2
CLOSE Statement A–5
CREATE INDEX Statement A–6
CREATE SCHEMA Statement (ESQL Only) A–8
CREATE TABLE Statement A–10
CREATE VIEW Statement A–13
DECLARE CURSOR Statement A–15
DELETE FROM Statement A–17
DROP INDEX Statement A–18
DROP TABLE Statement A–19
DROP VIEW Statement A–20
FETCH Statement A–21
GRANT Statement A–23
INSERT INTO Statement A–26
OPEN Statement A–28
REVOKE Statement A–29
SELECT Statement A–32
UNION Statement A–39
UPDATE Statement A–42

B. Progress Metaschema Tables B–1
B.1 Schema Tables for Tables and Columns B–2
B.2 Schema Tables for Views B–4

Index Index–1

Figures

Figure 3–1: Inner Joins 3–15
Figure 3–2: Left Outer Joins 3–17
Figure 3–3: Cursor Execution Sequence 3–29

Tables

Table 1–1:	Equivalent SQL and Progress Terms	1–2
Table 2–1:	Column, Table, and View Name Qualifiers	2–3
Table 2–2:	Truth Table for AND	2–5
Table 2–3:	Truth Table for OR	2–5
Table 2–4:	Truth Table for NOT	2–6
Table 3–1:	Search Condition Values	3–12
Table 3–2:	SQL Aggregate Functions	3–13
Table 3–3:	Progress/SQL Cursor Statements	3–28
Table 3–4:	SQL Statements and Associated Record Locks	3–30
Table 3–5:	Progress Record-locking Phrases	3–31
Table 3–6:	SQL Statements and Database Schema Triggers	**3–40**
Table 4–1:	Progress/SQL Statements and Progress Privileges	4–14
Table A–1:	SQL Aggregate Functions	A–33
Table B–1:	Fields in the _File Schema Table	B–2
Table B–2:	Fields in the _Field Schema Table	B–3
Table B–3:	Fields in the _View Schema Table	B–5
Table B–4:	Fields in the _View-Col Schema Table	B–6
Table B–5:	Fields in the _View-Ref Schema Table	B–7

Procedures

r-cretab.p A–12
r-union1.p A–40
r-union2.p A–40

Preface

Purpose

This manual documents Progress/SQL-89. It serves as both a user guide and a reference for the programmer using interactive SQL-89.

Audience

This manual is written for programmers who want to use Progress/SQL-89 and who know Progress, Progress development tools, and SQL-89. If you are using embedded SQL-89, use this manual together with the *Progress Embedded SQL-89 Guide and Reference*.

Organization of This Manual

Chapter 1, "About Progress/SQL-89"

Covers the differences in terminology between Progress/SQL-89 and Progress 4GL, choosing between Progress/SQL-89 and the Progress Data Dictionary to create and maintain tables, accessing non-Progress databases through Progress/SQL-89, Progress4GL extensions to Progress/SQL-89, and the differences between Progress/SQL-89 and Embedded SQL-89 (ESQL).

Chapter 2, "Progress/SQL-89 Components"

Describes the building blocks of Progress/SQL-89. It covers the character set, statements, identifiers, name qualifiers, literals, operators, null values, and logical expressions.

Chapter 3, "Data Manipulation Language"

Describes the basic SQL-89 Data Manipulation Language (DML) statements which include SELECT, INSERT, UPDATE, and DELETE. This chapter also describes using cursors, transaction processing, and privilege checking.

Chapter 4, "Data Definition Language"

Describes the SQL-89 Data Definition Language (DDL) statements, and discusses tables, data types, views, indexes, access privileges, and schema.

Appendix A, "Progress/SQL-89 Reference"

Provides an alphabetical reference and description of each Progress/SQL-89 statement.

Appendix B, "Progress Metaschema Tables"

Describes the metaschema tables that Progress/SQL-89 uses.

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 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 a procedure.

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

- 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 ACCUM function above, the *aggregate* and *expression* options are defined with the syntax for the ACCUM function in the *Progress Language Reference*.

- You must end all statements (except for DO, FOR, FUNCTION, PROCEDURE, and REPEAT) with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon, as in this example:

```
FOR EACH Customer:
   DISPLAY Name.
END.
```

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

  In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

  SYNTAX

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

  In some instances, square brackets are not a syntax notation, but part of the language.

  For example, this syntax for the INITIAL option uses brackets to bound an initial value list for an array variable definition. In these cases, normal text brackets ([ ]) are used:

  SYNTAX

```
INITIAL [ constant [ , constant ] ... ]
```

  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 this example, you must specify the items BY and *expression* and can optionally specify the item DESCENDING, in that order:

  SYNTAX

```
{ BY expression [ DESCENDING ] }
```

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

For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. In these cases, normal text braces ({ }) are used:

SYNTAX

```
{ &argument-name }
```

- A vertical bar (**|**) indicates a choice.

  In this example, EACH, FIRST, and LAST are optional, but you can only choose one:

  SYNTAX

  ```
  PRESELECT [ EACH | FIRST | LAST ] record-phrase
  ```

  In this example, you must select one of *logical-name* or *alias*:

  SYNTAX

  ```
  CONNECTED ( { logical-name | alias } )
  ```

- 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 this example, you must include two expressions, but you can optionally include more. Note that each subsequent expression must be preceded by a comma:

  SYNTAX

  ```
  MAXIMUM ( expression , expression [ , expression ] ... )
  ```

In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP is allowed:

SYNTAX

```
MESSAGE { expression | SKIP [ (n) ] } ...
```

In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

SYNTAX

```
{ include-file
    [ argument | &argument-name = "argument-value" ] ... }
```

- In some 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, WITH is followed by several optional items:

  SYNTAX

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

In this example, ASSIGN requires one of two choices: either one or more of *field*, or one of *record*. Other options available with either *field* or *record* are grouped with braces and brackets. The open and close braces indicate the required order of options:

SYNTAX

```
ASSIGN {   { [ FRAME frame ]
               { field [ = expression ] }
               [ WHEN expression ]
           } ...
         | { record [ EXCEPT field ... ] }
       }
```

Progress Messages

Progress displays several types of messages to inform you of routine and unusual occurrences:

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, if Progress cannot find a record with a specified index field value).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to running it (for example, if a procedure references a table name that is not defined in the database).
- Startup messages inform you of unusual conditions detected while Progress is getting ready to execute (for example, if you entered an invalid startup parameter).

After displaying a message, Progress proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify, or that are assumed, as part of the procedure. This is the most common action taken following execution messages.
- Returns to the Progress Procedure Editor so that you can correct an error in a procedure. This is the usual action taken following compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

Progress messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table (200)
```

Use Progress online help to get more information about Progress messages. On the Windows platform, many Progress tools include the following Help menu options to provide information about messages:

- Choose Help→Recent Messages to display detailed descriptions of the most recent Progress message and all other messages returned in the current session.
- Choose Help→Messages, then enter the message number to display a description of any Progress message. (If you encounter an error that terminates Progress, make a note of the message number before restarting.)
- In the Procedure Editor, press the **HELP** key (**F2** or **CTRL-W**).

On the UNIX platform, you can use the Progress PRO command to start a single-user mode character Progress client session and view a brief description of a message by providing its number. Follow these steps:

1 ♦ Start the Progress Procedure Editor:

```
install-dir/dlc/bin/pro
```

2 ♦ Press **F3** to access the menu bar, then choose Help→Messages.

3 ♦ Type the message number, and press **ENTER**. Details about that message number appear.

4 ♦ Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose File→Exit.

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 Application Development Environment — Getting Started (Windows only)

A practical guide to graphical application development within the Progress Application Development Environment (ADE). This guide includes an overview of the ADE and its tools, an overview of Progress SmartObject technology, and tutorials and exercises that help you better understand SmartObject technology and how to use the ADE to develop applications.

Progress Language Tutorial for Windows and *Progress Language Tutorial for Character*

Platform-specific tutorials designed for new Progress users. The tutorials use a step-by-step approach to explore the Progress application development environment using the 4GL.

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.

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup commands and parameters in alphabetical order.

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 ADM 2 Guide

A guide to using the Application Development Model, Version 2 (ADM 2) application architecture to develop Progress applications. It includes instructions for building and using Progress SmartObjects.

Progress ADM 2 Reference

A reference for the Application Development Model, Version 2 (ADM 2) application. It includes descriptions of ADM 2 functions and procedures.

Progress AppBuilder Developer's Guide (Windows only)

A programmer's guide to using the Progress AppBuilder visual layout editor. AppBuilder is a Rapid Application Development (RAD) tool that can significantly reduce the time and effort required to create Progress applications.

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.

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

A guide for the Progress development toolset, including the Progress Procedure Editor and the Application Compiler.

Progress Debugger Guide

A guide for the Progress Application Debugger. The Debugger helps you trace and correct programming errors by allowing you to monitor and modify procedure execution as it happens.

Progress Help Development Guide (Windows only)

A guide that describes how to develop and integrate an online help system for a Progress application.

Progress Translation Manager Guide (Windows only)

A guide that describes how to use the Progress Translation Manager tool to manage the entire process of translating the text phrases in Progress applications.

Progress Visual Translator Guide (Windows only)

A guide that describes how to use the Progress Visual Translator tool to translate text phrases from procedures into one or more spoken languages.

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

An administration and development guide for generating Report Builder reports using the Progress Report Engine.

Progress Report Builder Tutorial (Windows only)

A tutorial that provides step-by-step instructions for creating eight sample Report Builder reports.

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

A guide for system administrators that describes how to set up and maintain the Results product in a graphical environment. This guide also describes how to program, customize, and package Results with your own products. In addition, it describes how to convert character-based Results applications to graphical Results applications.

Progress Results User's Guide for Windows and *Progress Results User's Guide for UNIX*

Platform-specific guides for users with little or no programming experience that explain how to query, report, and update information with Results. Each guide also helps advanced users and application developers customize and integrate Results into their own applications.

4GL

Building Distributed Applications Using the Progress AppServer

A guide that provides comprehensive information about building and implementing distributed applications using the Progress AppServer. Topics include basic product information and terminology, design options and issues, setup and maintenance considerations, 4GL programming details, and remote debugging.

Progress External Program Interfaces

A guide to accessing non-Progress applications from Progress. This guide describes how to use system clipboards, UNIX named pipes, Windows dynamic link libraries, Windows dynamic data exchange, Windows ActiveX controls, and the Progress Host Language Call Interface to communicate with non-Progress applications and extend Progress functionality.

Progress Internationalization Guide

A guide to developing Progress applications for markets worldwide. The guide covers both internationalization—writing an application so that it adapts readily to different locales (languages, cultures, or regions)—and localization—adapting an application to different locales.

Progress Language Reference

A three-volume reference set that contains extensive descriptions and examples for each statement, phrase, function, operator, widget, attribute, method, and event in the Progress language.

Progress Programming Handbook

A two-volume handbook that details advanced Progress programming techniques.

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.

DataServers

Progress DataServer Guides

These guides describe how to use the DataServers to access non-Progress databases. They provide instructions for building the DataServer modules, a discussion of programming considerations, and a tutorial. Each DataServer has its own guide, for example, the *Progress DataServer for ODBC Guide*, the *Progress DataServer for ORACLE Guide*, or the *Progress/400 Product Guide*.

MERANT ODBC Branded Driver Reference

The Enterprise DataServer for ODBC includes MERANT ODBC drivers for all the supported data sources. For configuration information, see the MERANT documentation, which is available as a PDF file in *installation-path*\odbc. To read this file you must have the Adobe Acrobat Reader Version 3.1 or higher installed on your system. If you do not have the Adobe Acrobat Reader, you can download it from the Adobe Web site at: http://www.adobe.com/prodindex/acrobat/readstep.html.

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

A guide to Progress Embedded SQL-89 for C, including step-by-step instructions on building ESQL-89 applications and reference information on all Embedded SQL-89 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-89 and the ANSI standards upon which it is based.

Progress Open Client Developer's Guide

A guide that describes how to write and deploy Java and ActiveX applications that run as clients of the Progress AppServer. The guide includes information about how to expose the AppServer as a set of Java classes or as an ActiveX server.

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.

Progress SQL-92 Guide and Reference

A user guide and reference for programmers who use Progress SQL-92. It includes information on all supported SQL-92 statements, SQL-92 Data Manipulation Language components, SQL-92 Data Definition Language components, and Progress functions. The guide describes how to use the Progress SQL-92 Java classes and how to create and use Java stored procedures and triggers.

Deployment

Progress Client Deployment Guide

A guide that describes the client deployment process and application administration concepts and procedures.

Progress Developer's Toolkit

A guide to using the Developer's Toolkit. This guide describes the advantages and disadvantages of different strategies for deploying Progress applications and explains how you can use the Toolkit to deploy applications with your selected strategy.

Progress Portability Guide

A guide that explains how to use the Progress toolset to build applications that are portable across all supported operating systems, user interfaces, and databases, following the Progress programming model.

WebSpeed

Getting Started with WebSpeed

Provides an introduction to the WebSpeed Workshop tools for creating Web applications. It introduces you to all the components of the WebSpeed Workshop and takes you through the process of creating your own Intranet application.

WebSpeed Installation and Configuration Guide

Provides instructions for installing WebSpeed on Windows and UNIX systems. It also discusses designing WebSpeed environments, configuring WebSpeed Brokers, WebSpeed Agents, and the NameServer, and connecting to a variety of data sources.

WebSpeed Developer's Guide

Provides a complete overview of WebSpeed and the guidance necessary to develop and deploy WebSpeed applications on the Web.

WebSpeed Version 3 Product Update Bulletin

A booklet 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.

Welcome to WebSpeed! (Hard copy only)

A booklet that explains how WebSpeed software and media are packaged. *Welcome to WebSpeed!* also provides descriptions of the various services Progress Software Corporation offers.

Reference

Pocket Progress (Hard copy only)

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

Pocket WebSpeed (Hard copy only)

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

1

About Progress/SQL-89

The Progress Structured Query Language (Progress/SQL-89) is a relational database language based on the 1989 SQL standard of the American Standards Institute (ANSI). It meets the Level 2 requirement of the standard. In addition, Progress/SQL-89 supports many Progress features.

This chapter covers the following topics:

- Terminology differences between SQL-89 and Progress
- Choosing a tool for creating and maintaining database tables
- Accessing non-Progress databases through Progress/SQL-89
- Progress 4GL extensions to Progress/SQL-89
- Differences between Progress/SQL-89 and Progress/ESQL-89

1.1 Terminology Differences Between SQL and Progress

Progress/SQL and the Progress 4GL use different terms to refer to the same database objects. Table 1–1 lists SQL terms and the Progress equivalents.

Table 1–1: Equivalent SQL and Progress Terms

SQL	Progress
Column	Field
Row	Record

1.2 Choosing a Tool for Creating and Maintaining Database Tables

To define the structure of a database, you can use Progress/SQL Data Definition Language (DDL), the Progress Data Dictionary tool, or both. However, you must maintain each table the same way you define it. If you create a table with Progress/SQL DDL statements, you can use only Progress/SQL statements to change the privileges and structure of that table-you cannot use the Progress Data Dictionary. However, you can use the Data Dictionary to set or alter schema information that SQL DDL does not maintain, such as the Description and Help fields. Similarly, you must use the Data Dictionary to make changes to the structure of a table created with the Data Dictionary.

Progress Software Corporation recommends that you define tables by using the Progress Data Dictionary, even if you plan to access the tables through Progress/SQL. This establishes a consistent way of maintaining the schema. But if you plan to modify the schema dynamically—perhaps by running an application containing DDL—you must define the schema by using Progress/SQL, and sacrifice some features of the Progress Data Dictionary. For example, if you define tables using SQL, you cannot deny the blank user ID access to the tables, since you define access rights of the blank user ID using the Data Dictionary. For more information on the blank user ID, see the *Progress Database Administration Guide and Reference* and the *Progress Client Deployment Guide*.

1.3 Accessing Non-Progress Databases Through Progress/SQL

You can use Progress/SQL to retrieve and update data in the SQL-based non-Progress databases that Progress supports. For example, Progress supports DataServers to access ORACLE, SYBASE, and Rdb databases. The DataServer processes the database requests from an application. The interface processing is transparent to the application. To use Progress/SQL with a non-Progress database, you only have to specify the appropriate Progress database parameters when you use the CONNECT statement to connect the databases.

You cannot use SQL to modify the schema of non-Progress databases.

1.4 Progress 4GL Extensions to Progress/SQL

Progress/SQL supports the following Progress 4GL language elements:

- The Progress format phrases FORMAT, LABEL, and COLUMN-LABEL in the Progress/SQL CREATE TABLE, ALTER TABLE, and SELECT statements. These phrases control the display format and labels for data displayed on the screen or in printed reports.
- The frame phrase (WITH) in Progress/SQL queries. This phrase determines frame attributes.
- The Progress DATE and LOGICAL data types.
- Progress procedure variables and field values (retrieved by previous queries) in Progress/SQL statements.
- Subscripts for access to existing array fields created with Progress. (You cannot create array fields with Progress/SQL.)
- The BEGINS operator in a WHERE clause.
- The plus sign (+) for character string concatenation.
- Identifiers greater than 18 characters; Progress/SQL supports identifiers of up to 32 characters.
- Enhancements to the CONNECT statements to support DataServers.

- The record locking descriptions (FOR READ ONLY and FOR UPDATE) on the DECLARE CURSOR statement.
- Several DDL statements: ALTER TABLE, CREATE INDEX, DROP INDEX, DROP TABLE, and DROP VIEW.

NOTE

- Progress/SQL's rules for conversions and references within ORDER BY, GROUP BY, and other clauses are not as restrictive as the ANSI standard rules.

1.5 Differences Between Progress/SQL and Progress/ESQL

Progress/SQL is the name of Progress Software Corporation's SQL product family.

Progress/SQL comprises

- An interactive SQL.
- An embedded SQL, called Progress/ESQL.

1.5.1 The Interactive SQL

You can use the interactive SQL to write Progress procedures, which might also contain Progress 4GL - the two integrate well. You can edit using the Progress procedure editor, and compile and run using the Progress Application Compiler.

You must end each interactive SQL statement with a period, which is the Progress 4GL statement terminator.

1.5.2 Progress/ESQL

You can embed Progress/ESQL within C source code. You can edit using your favorite text editor, then preprocess, compile, and link (against ESQL-LIB) to produce an executable, which you can run like any other executable.

You must end each Progress/ESQL statement with a semicolon, which is the C statement terminator.

In order to use a C data item in an Progress/ESQL statement, you must first declare the "host variable" to ESQL, and then you must prepend a colon to the C data item within the SQL statement.

Certain Progress 4GL extensions to SQL are not available in Progress/ESQL.

- You cannot use the SELECT statement's WITH option, as you can in the interactive SQL to access frame, STREAM, and EXPORT options.
- All SELECT statements must be either cursor SELECTs, or singleton SELECTs that return values to host variables.
- You must reference columns and host variables by name.
- You cannot use most format phrase options or array subrange expressions (*variable* [*expression* [FOR *n*]]) that apply to columns and variables in Progress/SQL.

NOTE

- You can use the COLUMN-LABEL, LABEL, and FORMAT options, especially in the CREATE TABLE statement. You can also qualify table names and column names for multiple schemas.

For more information on Progress/ESQL, see the *Progress Embedded SQL-89 Guide and Reference*.

2

Progress/SQL-89 Components

This chapter describes the building blocks of Progress/SQL-89. The building blocks are:

- The SQL-89 character set
- SQL-89 statements
- Identifiers
- Name qualifiers
- Literals
- Operators
- Null values
- Logical expressions

2.1 The SQL Character Set

The SQL character set consists of the uppercase and lowercase letters A through Z, the digits 0 through 9, and the following special characters:

```
% ( ) _ < > . + - = * , ' " /
```

2.2 SQL Statements

Each Progress/SQL statement must end with a period. You can include comments within the /* */ symbols; Progress/SQL ignores these statements. Any multiple spaces, tabs, and newline characters are treated as a single space.

When entering lists of items, such as a series of column names, separate the items with commas.

```
SELECT Name, Address, City, State FROM Customer.
```

2.3 Identifiers

Identifiers are names that you give to language elements. All of the following elements are identifiers:

- Table and view names
- Column names
- Index names
- Correlation names
- Host language variables

Identifiers can be up to 32 characters long and are not case sensitive. The first character must be an uppercase or lowercase letter. The remainder of the identifier can be any combination of letters (A-Z and a-z), digits (0-9), and underscore (_) characters. You cannot use any of the SQL special characters (except for the underscore character) as part of an identifier.

You cannot use SQL reserved words or Progress reserved words as identifiers, except for ORDER; ORDER is not a reserved word, so you can use it in the Sports demonstration database to refer to the Order table. For a list of SQL and Progress reserved words, see the *Progress Language Reference*.

A table or view name must be unique within a database. A column name must be unique within a table; however, you can have duplicate column names within a database. An index name must be unique within a table and within a database.

Progress/SQL allows you to access existing Progress tables, fields, and variables even if their names contain characters that are not valid in Progress/SQL identifiers, such as hyphens and percent signs. You cannot, however, create new SQL tables and columns with these characters. Similarly, you cannot create new views and view columns with characters that are not valid in SQL identifiers. However, if the view columns are created implicitly with column names from the table, Progress/SQL allows you to retain the invalid characters.

2.4 Name Qualifiers

Name qualifiers are necessary to avoid ambiguous references. When you use the same column name in more than one table and those columns are referenced in the same query, you must qualify the column names with their table names (for example, Customer.Cust-Num and Order.Cust-Num in the Sports database).

When accessing multiple or non-Progress databases, you should also qualify table names and view names. If you do not qualify your reference, Progress/SQL takes the name from the default database, if it exists there. In Progress, you receive an error message if the table or view name exists in more than one database.

Table 2–1 shows the general syntax for qualifying column, table, and view names.

Table 2–1: Column, Table, and View Name Qualifiers

Qualifier	Syntax
Column	[[*database*] *table.*]] *column*
Table or view	[*database.*] *table* or *view*

2.5 Literals

Literals are constants. Literals can be character string, numeric, date, or logical data.

A character string literal can consist of non-numeric or character data, or a combination of numeric and non-numeric data. You must enclose character string literals in either single quotation marks (' ') or double quotation marks (" "), as shown in the following examples.

```
'Jane Smith'  "$125.00"  '01754'  "15 Oak St."
```

To include a single or double quotation mark within a character string literal, either enter the mark twice or use the opposite mark. For example:

```
'Brown''s Plumbing Supply'  or   "Brown's Plumbing Supply"
```

Date literals require the format *mm/dd/yy*. For example: 09/13/57.

Logical literals are represented by yes/no or true/false.

The backslash (\) is a Progress escape character. To specify a backslash in a string literal, use a double backslash (\\).

Progress/SQL provides two wildcard characters that you can use in LIKE clauses to match character strings. The percent sign (%) matches zero or more characters. The underscore (_) matches any single character.

2.6 Operators

Operators are the symbols you use to perform calculations and comparisons. SQL operators are the same as Progress operators, except that you cannot use the alternative Progress inequality operators: GT (greater than), LT (less than), or equality operator EQ. Instead, use the standard symbolic operators (>, <, =) for these operations. For detailed information about operators, see the *Progress Language Reference*.

NOTES

- When you use the arithmetic operators +, -, *, and /, at least one blank space must precede and follow the operator. Omitting the spaces results in an error message.
- If you start Progress with an ANSI SQL startup parameter (-Q or -Q2) and then divide by zero, the offending Progress/SQL expression returns the UNKNOWN value and raises an SQL data exception.

2.7 Null Values

A null value indicates that the value for a particular column is unknown or not available. The Progress/SQL null value is equivalent to the Progress unknown value (?).

Any arithmetic expression that contains a null value evaluates to null. Any comparison expression that contains a null value (in WHERE and HAVING clauses) evaluates to unknown, neither true nor false. When a WHERE or HAVING clause evaluates to unknown for a particular row, Progress does not select or modify that row.

2.8 Logical Expressions

Logical expressions are expressions that evaluate to TRUE, FALSE, or UNKNOWN (equivalent to the Progress unknown value and to the Progress/SQL null value). You combine logical expressions with the logical operators AND and OR, and you negate a logical expression by prepending the logical operator NOT. The resulting logical expression has the value TRUE, FALSE, or UNKNOWN.

If you combine two logical expressions using AND, determine the truth value of the result from Table 2–2.

Table 2–2: Truth Table for AND

AND	TRUE	UNKNOWN	FALSE
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	UNKNOWN	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE

If you combine two logical expressions using OR, determine the truth value of the result from Table 2–3.

Table 2–3: Truth Table for OR

OR	TRUE	UNKNOWN	FALSE
TRUE	TRUE	TRUE	TRUE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	UNKNOWN	FALSE

If you prepend NOT to a logical expression, determine the truth value of the result from Table 2–4.

Table 2–4: Truth Table for NOT

NOT	
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

3

Data Manipulation Language

You can use the Progress/SQL-89 data manipulation language (DML) to retrieve and update data in Progress databases and non-Progress databases.

This chapter covers:

- Using the SELECT statement
- Selecting data from multiple tables (Joins)
- Combining SELECT statements (UNION)
- Inserting, updating and deleting rows
- Using cursors
- Selecting single rows
- Transaction processing
- Database triggers
- Privilege checking

You can use all of the examples in this chapter with the Progress Sports demonstration database.

3.1 Using the SELECT Statement

The SELECT statement retrieves data from the database. Because the SELECT statement is the most complex SQL statement, this explanation is divided into the following sections:

- SELECT statement syntax
- Specifying search conditions
- Specifying frame properties
- Grouping data
- Sorting data
- Exporting data
- Using subqueries
- Using aggregate functions
- Selecting data from multiple tables (joins)

3.1.1 SELECT Statement Syntax

The basic SELECT statement retrieves and displays as many rows of data as satisfy the selection criteria you specify. The basic SELECT statement is equivalent to the combination of the FOR EACH and DISPLAY statements in Progress.

The SELECT statement has the following syntax.

SYNTAX

```
SELECT
  [ ALL | DISTINCT ]
  { * | column-list }
  [ INTO variable-list ]
  FROM {   table-name [ correlation-name ]
         | implicit-join
         | explicit-join
       }
  [ WHERE search-condition ]
  [ GROUP BY column [ , column ] ... ]
  [ HAVING search-condition ]
  [ ORDER BY sort-criteria ]
  [ WITH [ frame-options ] [ STREAM stream ] [ EXPORT ] ]
```

To specify the columns to retrieve, you can either use an asterisk (*) to indicate all columns, or you can enter a *column-list*, which is one or more column names and aggregate functions separated by commas. (For information on aggregate functions, see the "Using Aggregate Functions" section.) The keyword ALL, the default, selects all of the values in the specified columns. The optional keyword DISTINCT retrieves only rows with unique values.

NOTES

- Progress/SQL does not support 4GL buffer names in SQL FROM phrases.

The following example selects all values for all columns in the customer table and displays them in a readable format.

```
SELECT * FROM Customer WITH 2 COLUMNS.
```

- The SELECT statement supports word indexing through the use of the CONTAINS and the MATCHES search conditions. For more information on search conditions, see the "Specifying Search Conditions" section below. For more information on word indexing, see the "Word Indexes" section of the Database Access chapter in the *Progress Programming Handbook*.

3.1.2 Specifying Search Conditions

The WHERE clause of the SELECT statement specifies search conditions for the data you want to display. If you omit the WHERE clause, all rows are included in the retrieval set. The WHERE clause can also specify join conditions. For more information, see the "Selecting Data from Multiple Tables (Joins)" section.

This next example selects the name, address, city, and state columns from the customer table. The WHERE clause specifies customers from Massachusetts only.

```
SELECT Name, Address, City, State
  FROM Customer
  WHERE State = 'MA'.
```

A WHERE clause consists of one or more search conditions connected by the logical operators AND, OR, and NOT. All of the following are search conditions:

SYNTAX

```
expression relational-operator expression
```

In the following example, the WHERE clause specifies customers whose customer number is greater than 10.

```
SELECT Cust-Num, Name
  FROM Customer
    WHERE Cust-Num > 10.
```

SYNTAX

```
expression1 [ NOT ] BETWEEN expression2 AND expression3
```

The BETWEEN search condition is equivalent to:

expression1 >= *expression2* AND *expression1* <= *expression3*

The following example uses BETWEEN and NOT BETWEEN to select items.

```
SELECT Item-Num, Item-Name
  FROM Item
    WHERE Item-Num BETWEEN 10 AND 20
       AND Item-Num NOT BETWEEN 15 AND 17.
```

SYNTAX

```
column-name IS [ NOT ] NULL
```

This search condition tests whether a column contains a null value. See Chapter 2, "Progress/SQL-89 Components," for information on null values.

The following SELECT statement uses the IS NULL phrase to find which customers' orders have not been shipped.

```
SELECT Name, City, State, Order-Date
  FROM Customer, Order
    WHERE Ship-Date IS NULL.
```

NOTE: This SELECT also joins the customer and order tables. For more information on joins, see the "Selecting Data from Multiple Tables (Joins)" section.

SYNTAX

```
column-name [ NOT ] LIKE "string" [ ESCAPE "character" ]
```

The LIKE search condition compares character values in a column to a character string that is enclosed in either single or double quotation marks. The string can be a pattern created with any number of characters, including the percent sign (%) and underscore (_) wildcard characters. The percent sign matches zero or more characters. The underscore matches any single character.

The following example selects all USA customers with postal-codes beginning with 02.

```
SELECT Cust-Num, Name
  FROM Customer
     WHERE Country = "USA" AND Postal-Code LIKE "02%".
```

To search for the percent sign or underscore, you must declare an escape character to disable the use of the character as a wildcard. To declare an escape character, use the keyword ESCAPE

followed by the escape character enclosed in either single or double quotation marks. In the search string, precede the percent sign or underscore with the declared escape character.

In the following example, the vertical bar (|) is used as the escape character to search for the percent sign in the last position of terms.

```
SELECT Name, Terms
  FROM Customer
    WHERE Terms LIKE "|%" ESCAPE "|".
```

To use a backslash (\) as the escape character, you must specify it as "\\", since "\" is already defined as an escape character in Progress.

SYNTAX

```
expression [ NOT ] IN ({ value-list | SELECT-statement } )
```

This search condition tests whether the expression is equal or not equal to any item in the *value-list* or any value that the SELECT statement returns. If you enter a list of values, separate them with commas.

The following query uses the IN syntax to select customers from New England.

```
SELECT Name, Address, City, State
   FROM Customer
      WHERE State IN ("MA", "NH", "VT", "ME", "CT", "RI").
```

SYNTAX

```
expression CONTAINS "string"
```

The CONTAINS search condition tests whether any portion of any item that the query returns contains the string. The test ignores case.

The following query uses the CONTAINS search condition to select customers who are on "credit hold."

```
SELECT * FROM Customer
      WHERE Comment CONTAINS 'credit hold'.
```

SYNTAX

```
expression MATCHES "string"
```

The MATCHES search condition tests whether any item that the query returns matches the string. The test ignores case.

Unlike the CONTAINS search condition (and = operator), the MATCHES search condition recognizes the wildcard (*) (zero or more of any character) and "." (one of any character).

The following query uses the MATCHES search condition to select customers whose "state" column begins with "m" and ends with "x," such as "Middlesex."

```
SELECT * FROM Customer
      WHERE State MATCHES 'm*x'.
```

3.1.3 Specifying Frame Properties

To modify a frame's properties with the SELECT statement, use a frame phrase. To modify the frame properties of an individual field, variable, or expression, use a format phrase.

NOTE

- You cannot use frame options in a SELECT statement within a UNION statement.

For more information on frame phrases and format phrases, see the *Progress Language Reference*.

3.1.4 Grouping Data

A group is a set of rows that has the same value for a specified column or columns. The optional GROUP BY clause of a SELECT statement results in a single row in the result table for each group of rows. This clause is useful for arranging a table into conceptual groups so that you can apply an operation or function, such as COUNT or SUM, to each group.

The GROUP BY clause has the following syntax:

SYNTAX

```
[ GROUP BY column [ , column ] ... ]
[ HAVING search-condition ]
```

The list of columns you specify determines the group. This list consists of a column name or a list of column names separated by commas. The optional HAVING clause excludes particular groups from the query results. The *search-condition* in the HAVING clause usually involves an aggregate function; a HAVING clause without a GROUP BY clause is valid only if the selection list in the SELECT statement consists of aggregate expressions. See the "Using Aggregate Functions" section for more information. If you specify a HAVING clause without a GROUP BY clause, the entire table is treated as a single group.

The following example displays the total number of items in a product line and the maximum cost of an item in that product line:

```
SELECT Cat-Page, Count(*), MAX(Price)
  FROM Item
  GROUP BY Cat-Page.
```

NOTE

- You cannot use the GROUP BY or HAVING clause with SELECT INTO.

3.1.5 Sorting Data

The ORDER BY clause of a SELECT statement sorts query results by the values in one or more columns. You can specify the columns by name or by order within the SELECT statement. The ORDER BY clause has the following syntax.

SYNTAX

```
ORDER BY
  { n | expression } [ ASC | DESC ]
  [ , n | , expression [ ASC | DESC ] ] ...
```

The following example of the ORDER BY clause sorts the data alphabetically by state and, within each state, alphabetically by city:

```
SELECT Name, Address, City, State
  FROM Customer
  ORDER BY State, City.
```

As an alternative to column names, you can specify integers in the ORDER BY clause. The integer refers to the position of the column in the column list. In the following example, the 2 refers to the SUM column.

```
SELECT Cat-Page, SUM(Price)
  FROM Item
  GROUP BY Cat-Page
  ORDER BY 2 DESC.
```

Ascending is the default sort order. You can specify the keyword DESC to sort the results in descending order.

Two null values are considered equal within the context of GROUP BY, ORDER BY, and DISTINCT clauses. In ORDER BY clauses, null values are also considered greater than all non-null values. When the sort order is ascending, null values are sorted last. When the sort order is descending, null values are sorted first.

NOTE

- If you repeatedly perform an SQL query that involves multiple indexes and that does not use the ORDER BY option, the order of rows that Progress returns might vary.

3.1.6 Exporting Data

In Progress/SQL, you can convert data to standard character format and display it to the current output destination or to a named output stream using the WITH EXPORT clause of the SELECT statement. This clause works exactly like the Progress EXPORT statement.

The following example converts all of the values in the customer table into a standard character format and sends them to the file `customer.d`.

```
OUTPUT TO customer.d.
SELECT * FROM Customer WITH EXPORT.
```

The code in this example is equivalent to the following Progress code.

```
OUTPUT TO customer.d.
FOR EACH Customer:
   EXPORT Customer.
END.
```

NOTE

- You cannot use WITH EXPORT or WITH STREAM in a SELECT statement within a UNION statement.

3.1.7 Using Subqueries

A subquery is a SELECT statement that is nested within a WHERE clause. The result of the subquery is used as a value in the WHERE clause. You must enclose a subquery in parentheses ( ), and it can have only one column in its SELECT list. EXISTS is the exception to this restriction; the aggregate expression in an EXISTS is not evaluated unless the subquery contains a HAVING clause.

There are two types of subqueries: singleton subqueries and set subqueries. A singleton subquery is preceded by a comparison operator and returns, at most, a single row.

You use a singleton subquery in the following context.

SYNTAX

```
expression operator ( SELECT-statement )
```

The following example contains a singleton subquery in the WHERE clause. The subquery returns the order number of the order with the maximum number of item 51 ordered.

```
SELECT Order-Mum
  FROM Order-Line
 WHERE Item-Num = 51 AND Qty =
       (SELECT MAX(Qty) FROM Order-Line WHERE Item-Num = 51).
```

A set subquery returns a set of one or more rows. You use set subqueries in the following contexts.

SYNTAX

```
{ expression operator
  {   ANY | ALL | SOME } ( SELECT-statement )
    | expression [ NOT ] IN ( SELECT-statement )
    |  [ NOT ] EXISTS ( SELECT-statement ) }
```

The set subquery options are search conditions that perform the following functions:

- Determine whether the subquery selected any rows.
- Determine whether the result of a subquery includes an expression.
- Compare the result of a subquery with an expression.

When you specify ALL, the search condition is true if the comparison is true for each of the values that the subquery returns. When you specify ANY, the search condition is true if the comparison is true for at least one of the values that the subquery returns. The keyword SOME is equivalent to the keyword ANY.

The subquery can return zero, one, or many values. If it returns zero values, the search condition has a value of TRUE if you specify ALL, FALSE if you specify ANY. If a NULL value is encountered, the condition has a value of FALSE if you specify ALL, UNKNOWN if you specify ANY and no other value matches. See Table 3–1 for a summary of these search condition values using the ANY, ALL, or SOME keywords.

Table 3–1: Search Condition Values

Subquery Return Value	ALL	ANY (or SOME)
0	TRUE	FALSE
NULL	FALSE	UNKNOWN

The [NOT] IN search condition tests whether the expression is equal, or not equal, to any value that the SELECT statement returns.

The EXISTS search condition determines whether the subquery returns any rows. The search condition is true if the subquery returns one or more rows. (This is similar to the Progress CAN-FIND function.) You can nest EXISTS searches within other EXISTS searches.

The following example lists information about customers who have orders in the order table.

```
SELECT Name, Cust-Num, City, State
  FROM Customer
  WHERE EXISTS
         (SELECT * FROM Order WHERE Order.Cust-Num = Customer.Cust-Num).
```

You can use SELECT * in subqueries only with the EXISTS search condition, even if the result has more than one column. You specify SELECT * for an EXISTS search condition, since no data is actually returned.

Correlated Subqueries

A correlated subquery is a subquery that depends on a value set in the outer query. This example finds the customer with the maximum credit-limit within each state using a correlated subquery.

```
SELECT Cust-Num, Credit-Limit, State
FROM Customer this_cust
WHERE Credit-Limit = (SELECT MAX(Credit-limit)
                      FROM Customer
                      WHERE this_cust.State = Customer.State).
```

The outer query accesses a customer record through the correlation name this_cust. The subquery references that variable to find the maximum credit-limit of any customer in the same state as customer this_cust. The subquery cannot be evaluated until a record is read into this_cust. Therefore, the subquery must be re-evaluated each time the value of this_cust.State changes.

3.1.8 Using Aggregate Functions

Aggregate functions allow you to evaluate groups of values. In general, each function operates on the set of values in one column or on an expression. The set of column values is determined by the WHERE clause of a SELECT statement. Table 3–2 lists the SQL aggregate functions.

Table 3–2: SQL Aggregate Functions

Function	Description
AVG([DISTINCT] *column*)	Calculates an average value for all rows in the result list (or for all distinct rows if you so specify). The *column* must be numeric. It can refer to a column or it can be a calculation.
COUNT(*)	Counts the number of rows in the result list (this count includes duplicate rows).
COUNT(DISTINCT *column*)	Counts the number of rows containing different values for *column*.
MAX([DISTINCT] *column*)	Returns the maximum value of *column* for all rows (or for all distinct rows that you specify).
MIN([DISTINCT] *column*)	Returns the minimum value of *column* for all rows (or for all distinct rows that you specify).
SUM([DISTINCT] *column*)	Calculates the sum of *column* for all rows (or for all distinct rows that you specify). The *expression* must evaluate to a numeric value.

The following example counts all rows in the customer table.

```
SELECT COUNT(*) FROM Customer.
```

You can include multiple aggregate functions in a single SELECT list. This example returns the average, maximum, and minimum price of items in the item table.

```
SELECT AVG(Price), MAX(Price), MIN(Price)
   FROM Item.
```

When aggregate functions encounter null values, they are not included in the aggregate total, with the exception of COUNT(*). COUNT(*) counts all rows, including rows that have null values.

Aggregates of columns or expressions (except for COUNT) containing no values yield NULL (the Progress unknown value), rather than 0 as in the Progress 4GL. In the following example, because no customers have negative cust-num values, the aggregate function AVG yields a value of NULL for the credit-limit field.

```
SELECT AVG(Credit-Limit) FROM Customer
   WHERE Cust-Num < 0.
```

The following example counts the number of different countries in the customer table and the total number of rows in the table.

```
SELECT COUNT(DISTINCT Country), COUNT(*)
   FROM Customer.
```

Aggregate functions can include arithmetic expressions, and you can use the results of aggregate functions in other operations, as shown in the following example.

```
SELECT Cat-Page, SUM(Price * On-Hand)
  FROM Item
  GROUP BY Cat-Page
  HAVING SUM(Price * On-Hand) > 0
  ORDER BY 2.
```

3.2 Selecting Data from Multiple Tables (Joins)

You can specify a single table name in the FROM clause of the SELECT statement to select data from one table. You can also select data from multiple tables and recombine rows within the same table by specifying a join in the FROM clause. A *join* is a binary operation that selects and combines the records from multiple tables so that each result in the results list contains a single record from each table. That is, a single join operation combines the records of one table with those of another table or combines the records of one table with the results of a previous join. Figure 3–1 shows how you might join three tables.

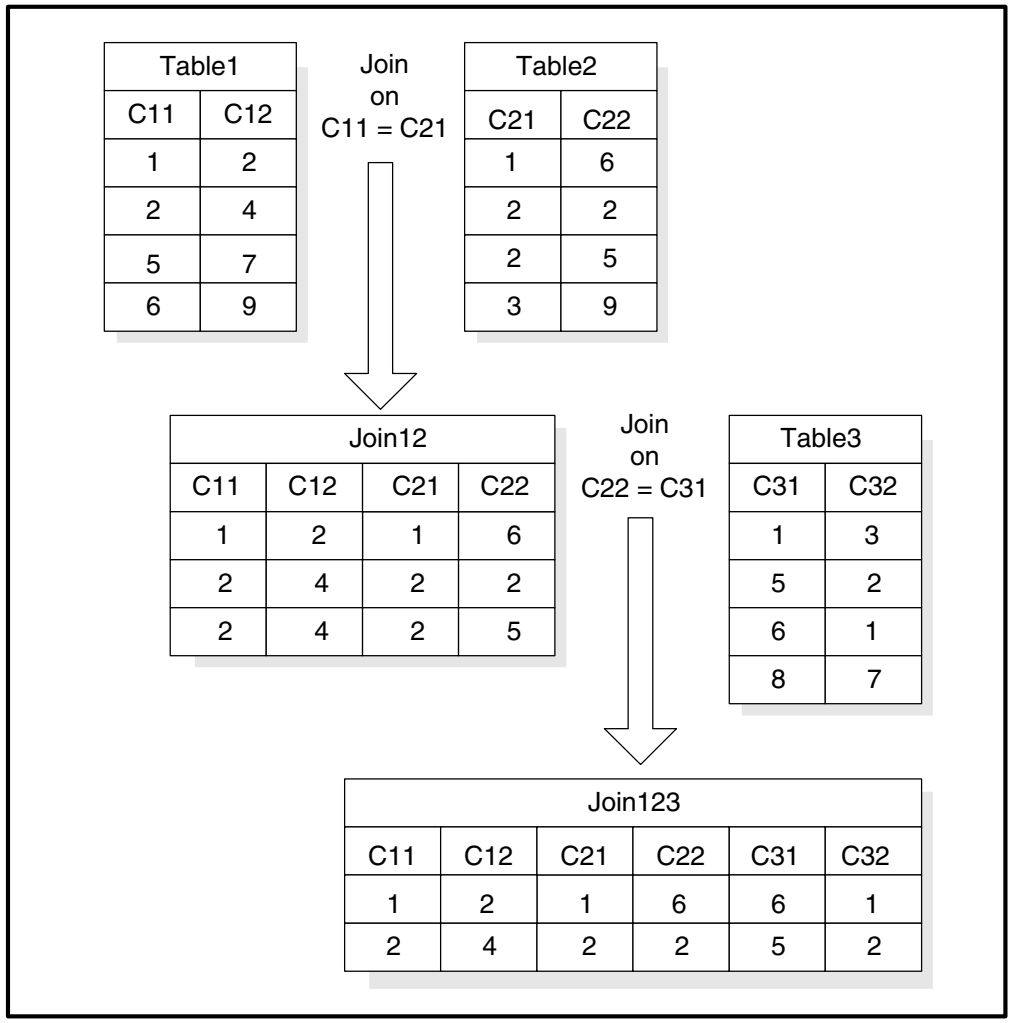

Figure 3–1: Inner Joins

A table or prior join can be either on the left- or right side of a join operation. Thus, the results of joining the three tables in Figure 3–1 depends on two join operations-one join between Table1 (left side) and Table2 (right side) and one join between the first join (left side) and Table3 (right side).The relations C11 = C21 and C22 = C31 represent *join conditions*, which determine how one table is related to the other (that is, which records selected from one table join with the records in the other table). How the records from joined tables are combined depends on the order of the tables in the join, the type of join operation used, and the selection criteria applied to each table.

3.2.1 Understanding Join Types

Progress/SQL supports two basic types of join: inner and outer. An *inner join* returns the records selected for the table (or join) on the left side combined with the related records selected from the table on the right. For any records not selected from the right table, the join returns no records from either the left or right sides of the join. Thus, only related records that are selected from both sides are returned for an inner join. Figure 3–1 shows an example of inner joins.

There are three types of outer join:

- Left
- Right
- Full

All outer joins return the records selected for an inner join. In addition, for each set of records selected from the table (or join) on the **left** side, a *left outer join* returns unknown values (?) from the table on the right where there is no record selected or otherwise related to the records on the left. That is, records from the left table (or join) are preserved for all unmatched records in the right table. A right outer join is the reverse of a left outer join. That is, for each set of records selected from the table (or join) on the **right** side, a *right outer join* returns unknown values from the tables on the left where there is no record selected or otherwise related to the records on the right. That is, records from the right table (or join) are preserved for all unmatched records in the left table. A *full outer join* returns the results of both a left and right outer join in one results list.

NOTE

- Progress/SQL does not currently support a full outer join. However, you can emulate a full outer join using a UNION of a left and right outer join.

Figure 3–2 shows an example of left outer joins. Thus, the results list contains all selected values for the left-most joined table and unknown values for all unmatched records from tables on the right. In a right outer join of the same three tables, the results list would show all selected values for the right-most joined table and unknown values for all unmatched records from tables on the left.

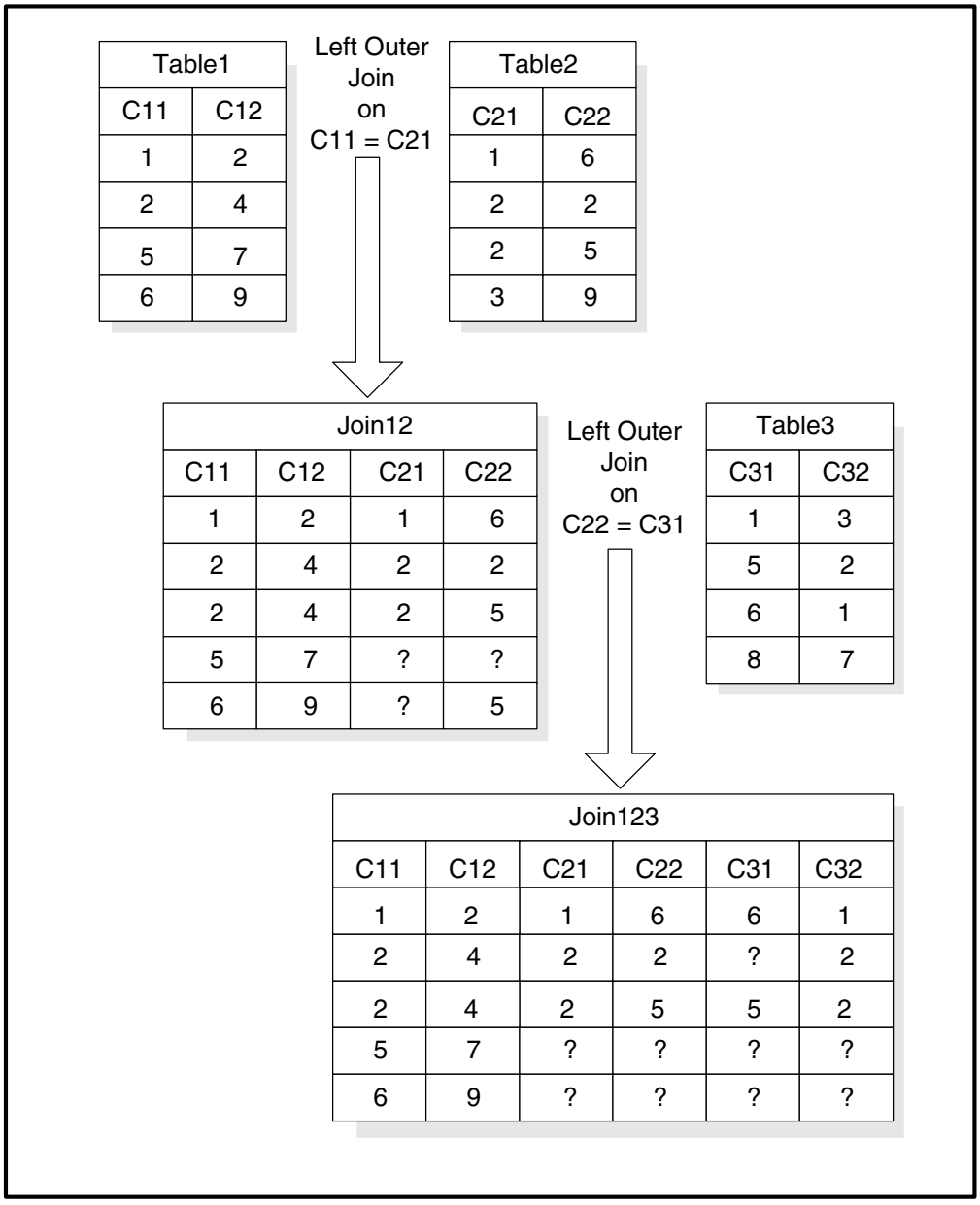

Figure 3–2: Left Outer Joins

Progress SQL provides support for inner, left outer, and right outer joins using two types of join specifications in the FROM clause: implicit and explicit joins. You can specify either implicit joins or explicit joins, but not both in the same FROM clause.

3.2.2 Specifying Implicit Joins

The following syntax of the FROM clause specifies an implicit join of two or more tables.

SYNTAX

```
FROM table-name [ correlation-name ]
  [ , table-name [ correlation-name ] ] ...
```

Thus, an *implicit join* is an inner join implied by a comma-separated list of tables in the FROM clause. Progress builds each inner join from left to right starting with the first table on the left, and uses the results list for each join on the left to make an inner join with each succeeding table on the right. You specify all of the join conditions and selection criteria for each table in an implicit join using the WHERE option.

Example

The following example joins the order table and the order-line table from the Sports database, and returns only those rows in the order table that have a row in the order-line table with the same order-num.

```
SELECT Cust-Num, Order.Order-Num, Line-Num, Item-Num
  FROM Order, Order-Line
    WHERE Order.Order-Num = Order-Line.Order-Num.
```

This SELECT statement retrieves the customer number, order number, line number, and item number for each order line of the customer. Since both the tables order and order-line have a column named order-num, you must use name qualifiers to avoid ambiguity.

Implicit Self-joins

You can use an implicit join to perform self-joins. A *self-join* combines information within a table by creating a result table or subset of information using data from the same table. To compare rows within the same table, you must specify a correlation name, which is an alias for a table name. Using this alias, you can refer to the same table as two different tables in the SELECT statement. In the following example, alt is a correlation name used to make a second reference to the customer table, joining different customer records with the same sales rep.

```
SELECT Customer.Name, Customer.Sales-Rep, alt.Name
  FROM Customer, Customer alt
    WHERE alt.Sales-Rep = Customer.Sales-Rep
       AND alt.Cust-Num <> Customer.Cust-Num.
```

3.2.3 Specifying Explicit Joins

The following syntax of the FROM clause specifies an explicit join.

SYNTAX

```
FROM table-name [ correlation-name ]
  [ INNER | LEFT [ OUTER ] ]
  JOIN table-name [ correlation-name ] ON search-condition
  [ [ INNER | LEFT [ OUTER ] ]
    JOIN table-name [ correlation-name ] ON search-condition
  ] ...
```

```
  FROM table-name [ correlation-name ] RIGHT [ OUTER ]
   JOIN table-name [ correlation-name ] ON search-condition
```

An *explicit join* is a join specification in which the type of join is specified by JOIN and its associated options, and the join conditions are specified by an ON clause. The ON clause can specify one or more join conditions connected by the logical operators AND, OR, and NOT.

You can also specify table selection criteria either with the ON clause or the WHERE clause of the SELECT statement. However, the ON clause offers better performance, because its search conditions can often be satisfied on the server. By default, the JOIN option specifies an inner join between the right table and the left table (or join). For more information, see the "Choosing Explicit Joins or Implicit Joins" section.

Inner and Left Outer Joins

You can mix inner and left outer joins, joining as many tables as you want. You cannot use parentheses or otherwise nest explicit joins. Multiple explicit joins always proceed in series from left to right, similar to implicit joins. Each successive inner or left outer join joins the next table on the right with the previous join.

The most efficient mix of inner and left outer joins specifies inner joins all together in one series on the left, and left outer joins all together in a second series on the right. This is because the results of any left outer joins followed by an inner join are identical to the results of all inner joins on the same tables. Specifying left outer joins prior to an inner join only makes the join process more lengthy because the final inner join must eliminate all joins containing null records that are generated by prior left outer joins.

Examples

The following example shows an explicit inner join of the order table and the order-line table, and returns only those rows in the order table that have a row in the order-line table with the same order-num.

```
SELECT Cust-Num, Order.Order-Num, Line-Num, Item-Num
  FROM Order JOIN Order-Line
    ON Order.Order-Num = Order-Line.Order-Num.
```

This example shows an inner join of four tables: customer, order, order-line, and item. It displays only those orders that contain line items with quantity sales extended to greater than two-thirds the customer credit limit.

```
SELECT Customer.Name, Customer.Credit-Limit, Order.Order-Num,
  Item.Item-Name
  FROM Customer JOIN Order ON Order.Cust-Num = Customer.Cust-Num
    JOIN Order-Line ON Order-Line.Order-Num = Order.Order-Num AND
     (Order-Line.Price * Order-Line.Qty) > (.667 * Customer.Credit-Limit)
    JOIN Item ON Order-Line.Item-Num = Item.Item-Num
WITH 10 DOWN.
```

This example shows a left outer join of the same four tables, which in addition to the inner join displays customers without orders (returning null order records) and all orders without sufficiently extended order lines (returning null order-line and item records).

```
SELECT Customer.Name, Customer.Credit-Limit, Order.Order-Num,
  Item.Item-Name
  FROM Customer LEFT JOIN Order ON Order.Cust-Num = Customer.Cust-Num
    LEFT JOIN Order-Line ON Order-Line.Order-Num = Order.Order-Num AND
     (Order-Line.Price * Order-Line.Qty) > (.667 * Customer.Credit-Limit)
    LEFT JOIN Item ON Order-Line.Item-Num = Item.Item-Num
WITH 10 DOWN.
```

This example shows an inner join of the customer and order tables followed by left outer joins of the order-line and item tables. The initial inner join ignores customers without orders (returning only valid orders), and the outer joins show all orders without sufficiently extended order lines (returning null order-line and item records).

```
SELECT Customer.Name, Customer.Credit-Limit, Order.Order-Num,
  Item.Item-Name
  FROM Customer INNER JOIN Order ON Order.Cust-Num = Customer.Cust-Num
    LEFT JOIN Order-Line ON Order-Line.Order-Num = Order.Order-Num AND
     (Order-Line.Price * Order-Line.Qty) > (.667 * Customer.Credit-Limit)
    LEFT JOIN Item ON Order-Line.Item-Num = Item.Item-Num
WITH 10 DOWN.
```

Right Outer Joins

In Progress/SQL you currently can specify a right outer join between no more than two tables in a single SELECT statement. This limitation originates in the implementation of right outer joins, which are based on reversed left outer joins. To properly associate multiple right outer joins using the current left outer join implementation, the Progress/SQL Compiler would have to allow nested joins, which it does not. Thus, you can combine only two tables at a time with a right outer join.

This example shows a right outer join of the Customer and Salesrep tables based on current customer balances. Thus, it shows sales reps and their customers for customers with balances greater than $50,000 and returns null customer records for sales reps that have no customers with balances that high.

This query demonstrates the Customer and Salesrep tables “right outer joined” on current balance. The query returns salesreps and, for each salesrep, customers with balances greater than $50,000. For salesreps with no customers with balances greater than $50,000, the query returns the salesrep and the “null” customer.

```
SELECT Customer.Name, Salesrep.Rep-Name
    FROM Customer RIGHT JOIN Salesrep
    ON Customer.Sales-Rep = Salesrep.Sales-Rep
        AND Customer.Balance > 50000.
```

Explicit Self-Joins

Like implicit joins, you can specify a self-join in an explicit join using a correlation name for the same table. This example lists customers on the left paired with all other customers with the same sales rep on the right that have a lower balance than the customer on the left.

```
SELECT alt1.Name, alt1.Balance, alt1.Sales-Rep, alt2.Name, alt2.Balance
  FROM Customer alt1 JOIN Customer alt2
    ON alt1.Sales-Rep = alt2.Sales-Rep
       AND alt1.Cust-Num <> alt2.Cust-Num AND alt1.Balance > alt2.Balance.
```

3.2.4 Choosing Explicit Joins or Implicit Joins

Explicit joins provide a greater variety of join operations than implicit joins. However, there are some trade offs and restrictions when using them.

Join Conditions and Selection Criteria

In the previous examples of explicit joins, the ON clause specifies both table join conditions (Customer.Sales-Rep = Salesrep.Sales-Rep) and selection criteria for individual tables participating in the join (Customer.Balance > 50000). For explicit joins, you must specify the join conditions using the ON clause, but you can alternatively specify the table selection criteria using the WHERE clause of the SELECT statement, just like implicit joins.

However, there is a difference in the way the ON and WHERE clauses apply selection criteria in an explicit join. The ON clause applies the selection criteria as the join is created and the WHERE clause applies the selection criteria after the join is created.

This difference can have two areas of performance impact:

- **Network traffic** — The ON clause is applied on the server whenever possible. This means that in a networked client/server application, the ON clause can return the selected join(s) to the client, causing less data to be returned over the network. This increases throughput, especially using high-performance servers.

- **Selection volume** — The WHERE clause must apply all of its selection criteria to the join that results from the full combination of tables in the SELECT. If you do not use the ON clause for selection, not only is more data returned to the client from the server, but the unselected join result can contain more records for the WHERE clause to select. For example, the same customer record might appear more than once in a multiple join, each instance of which must be selected by a WHERE clause. The ON clause, on the other hand, applies the selection criteria only to the records returned for a single join combination. Thus, for example, a single customer record can be selected once and eliminated from one join before it is combined with additional records from subsequent joins.

Thus, explicit joins afford maximum performance benefits wherever they can be used.

Restrictions on Explicit Joins

In Progress/SQL, explicit joins have the following restrictions:

- You can specify an explicit join only in a direct SELECT statement or in the cursor SELECT of a DECLARE CURSOR statement. Thus, you cannot specify an explicit join in a subquery or CREATE VIEW statement.
- You cannot specify view references in an explicit join. Only table references are allowed.

However, where you are unable to use an explicit join, you can use an implicit join.

3.3 Combining SELECT Statements (UNION)

You can use the UNION statement to combine the results of two or more SELECT statements. This allows you to create more complex queries than with just one SELECT statement.

SYNTAX

```
{ SELECT-statement | ( subunion ) }
  { UNION [ ALL ] { SELECT-statement | ( subunion ) } } ...
  [ ORDERED BY sort-criteria ]
```

NOTE

- You cannot use frame options, WITH EXPORT, or WITH STREAM in a SELECT statement within a UNION statement.

You can use the UNION statement to retrieve information from two or more tables and place it into a single retrieval set. The following example combines information from the Customer and Salesrep tables.

```
SELECT Name, State FROM Customer WHERE State = "MA"
UNION
SELECT Rep-Name, Region FROM Salesrep WHERE Region = "East".
```

You can also use the UNION statement to combine two or more queries into the same table. The following example combines three queries against the order table.

```
SELECT Order-Num, Cust-Num, Order-Date, "Order" FROM Order
      WHERE Order-Date < 2/1/93
UNION ALL
SELECT Order-Num, Cust-Num, Ship-Date, "Ship" FROM Order
      WHERE Ship-Date < 2/1/93
UNION ALL
SELECT Order-Num, Cust-Num, Promise-Date, "Promise" FROM Order
      WHERE Promise-Date < 2/1/93
ORDER BY Order-Num.
```

This example returns information about any orders with any date (order date, ship date, or promise date) before February 1, 1993. Because the ALL keyword combines each of the SELECT statements, if a single order has more than one date before February 1, 1993, it appears more than once in the result list. The fourth column of output contains a string constant that indicates which type of date is shown in the third column. The result list is sorted by order-num because of the ORDER BY clause.

3.4 Inserting Rows

The INSERT INTO statement inserts one or more rows into an existing table. The data you insert can be a list of values that you supply or values from another table.

The INSERT INTO statement has the following syntax.

SYNTAX

```
INSERT INTO table-name
  [ ( column-list ) ]
  { VALUES ( value-list ) | SELECT-statement }
```

The VALUES clause contains a list of values that you specify in your SQL statement. If you use the VALUES clause, you can insert only one row at a time into the table. The *value-list* can be one value or a series of values separated by commas. Each value can be a literal, the keyword NULL, a variable, a field defined and assigned a value elsewhere, or an expression. Each character string literal must be enclosed in either single or double quotation marks.

You can also insert values from another table by using a SELECT statement within the INSERT INTO statement.

This example inserts a new customer by specifying a list of values.

```
INSERT INTO Customer (Name, Address, City, State, Postal-Code, Cust-Num)
     VALUES ('Cycle Pro', '30 Boynton Street',
             'Jamaica Plain', 'MA', '02130', 101).
```

SQL inserts the first value into the first column in the list, the second value into the second column. Therefore, the number of columns must be the same as the number of values. If you specify a column list that does not include all columns of the table, SQL assigns the null or default value to each column omitted from the list. If you omit the column list entirely, SQL inserts the values into the columns in the order in which you created the columns. In this case, there must be a value for every column in the table.

If you use the SELECT statement with the INSERT INTO statement, all the rows that satisfy the SELECT statement are inserted into the table. The number and order of columns in the SELECT statement must match the implicit or explicit column list in the INSERT INTO statement. If you specify SELECT *, the asterisk is expanded into a list of all columns in the FROM clause table(s).

The following example inserts rows into a preferred customer table by selecting data from the customer table. It inserts a row for each customer whose balance exceeds $50,000.

```
INSERT INTO Prefer_Cust
    (Name, Address, City, State, Postal-Code, Cust-Num)
    SELECT Name, Address, City, State, Postal-Code, Cust-Num
    FROM Customer
    WHERE Balance > 50000.
```

When you use the INSERT INTO statement to add rows using a view, you can insert values only into columns that are defined in the view definition. (See Chapter 4, "Data Definition Language," for more information.) Any other columns in the underlying table for the view are set to their default value, or to null if there is no default.

3.5 Updating Rows

The searched UPDATE statement changes column values in one or more rows of a table. The searched UPDATE statement has the following syntax.

SYNTAX

```
UPDATE table-name
  SET column-name = { NULL | expression }
      [ , column-name = { NULL | expression } ] ...
  [ WHERE search-condition ]
```

The SET clause evaluates the *expression* and assigns it to the *column-name*. The expression can be a column name, a literal, an arithmetic operation, a procedure variable or field name, or any combination of these.

The WHERE clause determines the rows to update. If you omit the WHERE clause, all rows of the target table are updated. See the "Specifying Search Conditions" section for details on the search conditions used in WHERE clauses.

The following example changes the zip code for customers in Jamaica Plain, Massachusetts.

```
UPDATE Customer SET Postal-Code = '02130'
  WHERE City = 'Jamaica Plain' AND State = 'MA'.
```

See the "Positioned UPDATE" section for information on the positioned UPDATE statement.

3.6 Deleting Rows

The searched DELETE statement deletes one or more rows from a table. The DELETE statement has the following syntax.

SYNTAX

```
DELETE FROM table-name [ WHERE search-condition ]
```

The WHERE clause identifies the rows to delete. If you omit the WHERE clause, all rows of the target table are deleted. See the "Specifying Search Conditions" section for details on the search conditions used in WHERE clauses.

The following example deletes item number 51 from the item table.

```
DELETE FROM Item
     WHERE Item-Num = 51.
```

This example uses the LIKE search condition to specify the rows for deletion.

```
DELETE FROM Item
     WHERE Item-Name LIKE 'Croquet%'.
```

See the "Positioned DELETE" section for information on the positioned DELETE statement.

3.7 Using Cursors

In interactive SQL, the SELECT statement retrieves multiple rows of data and sends the results directly to the terminal. However, the basic SELECT statement cannot store and process multiple rows. Therefore, you must step through the rows that the SELECT statement chooses and manipulate them individually using a cursor.

A cursor is a pointer used to search through a retrieval set, pointing to each row in the set, one at a time. When a cursor is pointing to a row, it is positioned on that row. You can then update or delete the positioned row using the positioned forms of the UPDATE and DELETE statements.

Cursors let you retrieve column values from a row using a SELECT statement and let you assign the column values to procedure variables. To manipulate or update row values in the retrieval set, you must always use a cursor with each SELECT statement (except the singleton SELECT). Without cursors, there is no way to process individual rows.

Defining a cursor for a SELECT operation is a two-step process in which you declare the cursor with the DECLARE CURSOR statement and then open the cursor with the OPEN statement. Once you open the cursor, use the FETCH statement to retrieve the data. When you are finished with a cursor, close it with the CLOSE statement. You do not have to declare the cursor again if you reselect the data associated with it by reopening the cursor.

You can declare and open more than one cursor at a time to use with a single table, and there is no limit to the number of cursors you can declare in your procedure. However, you can use a cursor only in the procedure in which it is declared.

3.7.1 Cursor Statements

SQL cursor statements are used to define, open, and close cursors, as well as to retrieve individual rows and move the column values into procedure variables.

Table 3–3 lists the Progress/SQL cursor statements.

Table 3–3: Progress/SQL Cursor Statements

Statement	Description
DECLARE CURSOR	Associates a cursor name with a SELECT statement.
OPEN	Selects all rows that satisfy the DECLARE CURSOR statement and positions the cursor before the first row.
FETCH	Retrieves the next row and assigns column values from that row to procedure variables. Positions the cursor on the next row or, if there is no next row, after the last row.
Positioned UPDATE	Modifies the data in the row to which the cursor is positioned.
Positioned DELETE	Deletes the row to which the cursor is positioned.
CLOSE	Closes the cursor.

The following example shows how the basic SQL cursor statements are used:

```
DEFINE VARIABLE namevar LIKE Customer.Name.
DEFINE VARIABLE maxvar LIKE Customer.Credit-Limit.
DECLARE c1 CURSOR FOR
    SELECT Name, Credit-Limit
    FROM Customer
    WHERE Credit-Limit < 10000.
OPEN c1.
REPEAT:
    FETCH c1 INTO namevar, maxvar.
    UPDATE Customer
        SET Credit-Limit = maxvar + 1000
        WHERE CURRENT OF c1.
END
CLOSE c1
```

Figure 3–3 defines the sequence of the cursor statements when you execute the procedure.

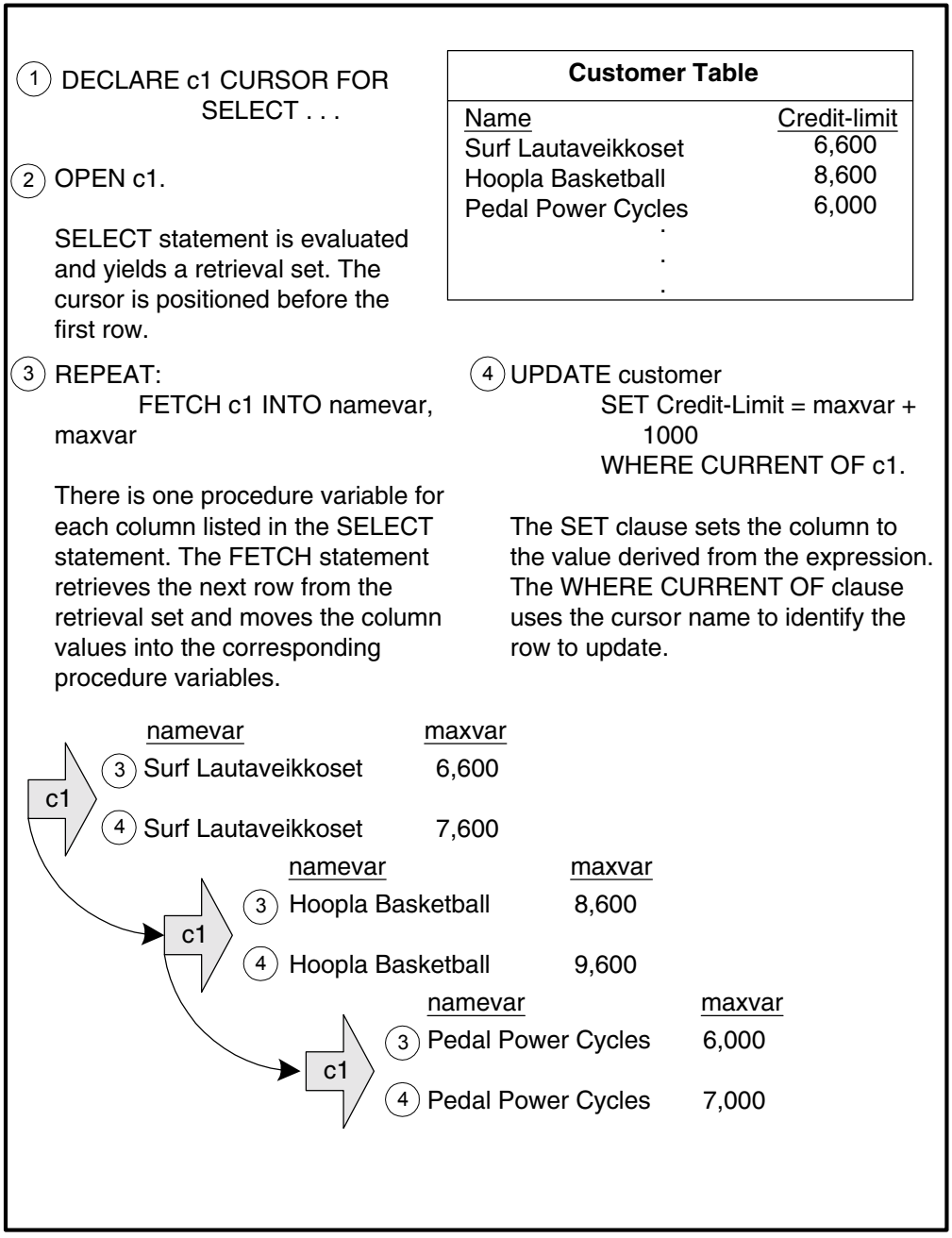

Figure 3–3: Cursor Execution Sequence

3.7.2 Defining a Cursor

The DECLARE CURSOR statement has the following syntax.

SYNTAX

```
DECLARE cursor-name CURSOR FOR
  { SELECT-statement | UNION-statement }
  [ FOR { READ ONLY | UPDATE } ]
```

The *cursor-name* can be any valid SQL identifier. The *SELECT-statement* is any valid SQL SELECT statement, including WHERE, GROUP BY, HAVING, and ORDER BY clauses, as well as subqueries, aggregates, joins, and all other valid SELECT statement syntax. The *SELECT-statement* specifies a retrieval set of rows that is accessible when the cursor is opened.

NOTE

- Progress does not support SQL cursor names in 4GL statements.

FOR READ ONLY allows you to read the selected rows but not update or delete them. FOR READ ONLY is performed with NO-LOCK to prevent lock table overflow. To coordinate concurrent access to records in multi-user applications, SQL statements use a default record-locking scheme based on the Progress record-locking phrases. You do not insert Progress record-locking phrases directly into ESQL programs. Progress applies the lock types shown in Table 3–4 implicitly.

Table 3–4: SQL Statements and Associated Record Locks *(1 of 2)*

Embedded SQL Statement	Associated Record Lock
Non-cursor SELECT statements (SELECT INTO)	NO-LOCK only. You can see any uncommitted changes to the database.
OPEN *cursor* statements[1]	SHARE-LOCK by default. Records retrieved through a cursor use SHARE-LOCK while being read and are converted to EXCLUSIVE-LOCK if updated or deleted. To read records with the NO-LOCK option, specify FOR READ ONLY in the associated DECLARE CURSOR statement.

Table 3–4: SQL Statements and Associated Record Locks *(2 of 2)*

Embedded SQL Statement	Associated Record Lock
Searched UPDATE statements (UPDATE WHERE *condition*) Positioned UPDATE statements (UPDATE WHERE CURRENT) Searched DELETE statements (DELETE WHERE *condition*) Positioned DELETE statements (DELETE WHERE CURRENT)	EXCLUSIVE-LOCK only.

[1] Cursor SELECT statements do not obtain locks until the cursor is opened.

Table 3–5 shows the Progress record locks and explains their effects.

Table 3–5: Progress Record-locking Phrases

Lock Type	Effect
NO-LOCK	Rows are not locked. A row is read even if another application holds an EXCLUSIVE-LOCK on it. You cannot update rows read with NO-LOCK.
SHARE-LOCK	Rows are read in anticipation of a possible update. A row is not read if another application holds an EXCLUSIVE-LOCK on it.
EXCLUSIVE-LOCK	Rows are locked until the end of the transaction. Another application cannot read rows using SHARE-LOCK or EXCLUSIVE-LOCK.

For more information on Progress record locking, see the *Progress Programming Handbook*. To guarantee serializable SQL transactions in ESQL, you can specify the ANSI SQL Client (-Q2) startup parameter. If you use this parameter, Progress/SQL conforms to the ANSI rules.

To define a cursor, you must have the SELECT privilege on all tables referred to in the DECLARE CURSOR statement (see the section on access privileges in Chapter 4, "Data Definition Language," for more information). The DECLARE CURSOR statement associates the cursor with the SELECT statement and assigns a Progress correlation name with the same name as *cursor-name*. Later, Progress/SQL fetches rows into the correlation table. If the cursor definition involves a join, Progress/SQL defines a correlation name for each table in the join. The first correlation name is called *cursor-name*. The other correlation names start with *cursor-name* followed by a hyphen and a letter, such as *cursor-name-a* and *cursor-name-b*.

You must use the DECLARE CURSOR statement in your Progress/SQL procedure before you open or refer to the cursor. You declare a cursor only once, even if you intend to close and reopen it several times in a procedure.

In the following example, the DECLARE CURSOR statement uses a cursor named c1 to retrieve the names and customer numbers for all Massachusetts customers.

```
DECLARE c1 CURSOR FOR
  SELECT Name, Cust-Num
    FROM Customer
    WHERE State = 'MA'.
```

The SELECT statement can refer to a procedure variable in its WHERE clause. The variable is evaluated when the cursor is opened, not when it is declared. The following example illustrates how a procedure variable is used in the WHERE clause.

```
DEFINE VARIABLE state-var AS CHARACTER INITIAL "MA".

DECLARE c1 CURSOR FOR
    SELECT Name, Address, City, State
      FROM Customer
      WHERE State = state-var ORDER BY State, City.
```

NOTE

- According to the SQL standard, a cursor is not updatable if an ORDER BY clause appears in the DECLARE CURSOR statement. Progress/SQL does not enforce this restriction. However, updatable cursors must obey the same restrictions as updatable views. For a list of these restrictions, see the section on updating views in Chapter 4, "Data Definition Language."

3.7.3 Opening a Cursor

The OPEN statement has the following general syntax.

SYNTAX

```
OPEN cursor-name
```

The OPEN statement positions the cursor before the first row of the retrieval set. The retrieval set is the set of records that results from executing SELECT in the DECLARE CURSOR statement. Progress/SQL uses current values for any procedure variables that you refer to in the SELECT statement to yield the retrieval set.

The following example opens the previously declared cursor.

```
OPEN c1.
```

You can open and close a cursor multiple times after you declare it. Each time you open the cursor, the SELECT statement is re-executed. Therefore, you do not have to declare the cursor more than once. With each execution, the retrieval set might contain different data if the values referenced in the SELECT statement have changed.

NOTE

- Progress/SQL does not support opening a cursor that is already open.

3.7.4 Retrieving Values from the Retrieval Set

The FETCH statement has the following general syntax.

SYNTAX

```
FETCH cursor-name INTO variable-list
```

The *variable-list* is a list of procedure variables separated by commas. There must be one variable for each column listed with SELECT in the DECLARE CURSOR statement. The data types of the variables and columns must be assignment compatible. Do not use the same name to specify procedure variables and SQL columns.

When referring to the values fetched into the list of variables, use the procedure variable name into which the list of variables was fetched. Do not use the column name from the DECLARE CURSOR statement.

NOTE

- Progress/SQL does not support FETCH INTO SQL variables.

The FETCH statement retrieves the next row from the retrieval set and moves each column value to the corresponding variable. There can be more than one FETCH statement for an open cursor, each with its own list of variables, as long as the number and data types of the variables correspond to the cursor definition.

In the following example, the customer name and number are fetched into the procedure variables, var1 and var2.

```
FETCH c1 INTO var1, var2.
```

If the DECLARE CURSOR statement specifies SELECT *, the number of variables in the list of variables must match the column count that the asterisk represents.

Each time you execute the FETCH statement, it retrieves a single row in the retrieval set. To fetch all the rows in a retrieval set, you must include the FETCH statement in a Progress looping block. For example, you could place the FETCH statement within a REPEAT block to fetch multiple rows, one row at a time.

When including FETCH in a repeat loop, the DISPLAY variables statement must appear inside of the loop to see the data for each row.

```
DEF VAR var1 LIKE Ncust.Cust-Num.
DEF VAR var2 LIKE Ncust.Name.

OUTPUT TO out.file PAGED PAGE-SIZE 60.
DECLARE c1 CURSOR FOR SELECT * FROM Ncust.
OPEN c1.
  REPEAT:
    FETCH c1 INTO var1, var2.
    DISPLAY var1 var2.
  END.

END.
```

3.7.5 Positioned UPDATE

The positioned form of the UPDATE statement is used with an open cursor. Once you have used the cursor statement FETCH to retrieve a row, you can individually update one or more columns in that row.

The positioned UPDATE statement has the following general syntax.

SYNTAX

```
UPDATE table-name
  SET column-name = { NULL | expression }
      [ , column-name = { NULL | expression } ] ...
  WHERE CURRENT OF cursor-name
```

The table in the positioned UPDATE statement is the same table associated with the cursor in the DECLARE CURSOR statement. The SET clause sets the specified columns to their new values, as derived from the expressions. The WHERE CURRENT OF clause uses the *cursor-name* to identify the cursor whose current row is to be updated.

You cannot update a cursor if it is defined for a SELECT statement that contains any of the following clauses or expressions:

- A join
- A GROUP BY or HAVING clause
- An expression or constant in the SELECT list
- The DISTINCT keyword
- A reference to a view that cannot be updated

3.7.6 Positioned DELETE

The positioned form of the DELETE statement is used with an open cursor. Once you have used the cursor statement FETCH to retrieve a row, you can delete that row.

The positioned DELETE statement has the following general syntax.

SYNTAX

```
DELETE FROM table-name WHERE CURRENT OF cursor-name
```

The table in the positioned DELETE statement is the same table associated with the cursor in the DECLARE CURSOR statement. The WHERE CURRENT OF clause uses the *cursor-name* to identify the cursor. When you delete the current row, the cursor is positioned before the next row, or after the last row if there is no next row.

The cursor restrictions listed for the positioned UPDATE also apply to positioned DELETE.

3.7.7 Closing a Cursor

The CLOSE statement has the following general syntax.

SYNTAX

```
CLOSE cursor-name
```

This statement closes the cursor. You cannot refer to the cursor again unless you open it. If you reopen it, Progress/SQL re-evaluates the selection criteria for the retrieval set and repositions the cursor before the first row. All open cursors are closed when you exit from the procedure.

The following statement closes cursor c1.

```
CLOSE c1.
```

3.7.8 Cursors Within Internal Procedures

Cursors within internal procedures are supported by the provision of two cursor styles:

- The DECLARE CURSOR statement is in the main procedure, and all other statements that refer to the cursor, such as OPEN, FETCH, UPDATE, DELETE and CLOSE, are in the main procedure or any subprocedures.

  The only restriction is that the DECLARE CURSOR statement must compile (that is, occur in the source file) before any other statement that refers to the cursor.

- The DECLARE CURSOR statement and all other statements that refer to the cursor are in a single internal procedure.

3.8 Selecting Single Rows

The SELECT INTO statement retrieves only one row of the table. This is also called a singleton SELECT. You can use the SELECT INTO statement instead of declaring a cursor and fetching one row through it. The SELECT INTO statement is commonly used for aggregate functions.

Here is the general syntax for the SELECT INTO statement.

SYNTAX

```
SELECT
  [ ALL | DISTINCT ]
  { * | column-list }
  INTO variable-list
  FROM
  {   table-name [ correlation-name ]
    | implicit-join
    | explicit-join
  }
  [ WHERE search-condition ]
  [ GROUP BY column [ , column ] ... ]
  [ HAVING search-condition ]
  [ ORDER BY sort-criteria ]
```

The SELECT INTO statement assumes that the WHERE clause evaluates to only one row. If you know that multiple rows exist with the same values, specify DISTINCT to prevent an error. ALL is the default. The *variable-list* is a list of procedure variables separated by commas. There must be one variable for each column or aggregate function in the *column-list*. The data types of the variables and columns must be compatible. You can use the same name to specify procedure variables and SQL columns. All other elements of the syntax are the same as for the SELECT statement.

In the following example, the SELECT INTO statement retrieves the name and state for customer number 10.

```
DEFINE VARIABLE namevar LIKE Customer.Name.
DEFINE VARIABLE statevar LIKE Customer.State.

SELECT Name, State INTO namevar, statevar
  FROM Customer
  WHERE Cust-Num = 10.
```

You can use a SELECT INTO statement instead of a cursor to retrieve aggregate values that are not grouped, because such queries return only a single row.

3.9 Transaction Processing

Because Progress has its own transaction management facilities, it is not necessary to terminate a transaction explicitly.

If you use SQL statements to update the database (INSERT, UPDATE, or DELETE), Progress/SQL performs the entire operation or nothing at all. This differs from Progress behavior. For example, if you update 50 rows with Progress statements and an error occurs on the fiftieth row, Progress only undoes that row and leaves the other 49 rows updated. If the same error occurs when you use SQL statements to update the database, Progress/SQL rolls back all 50 rows.

Because an SQL update is treated as a single transaction, Progress holds record locks for all affected rows until the end of the transaction to allow a rollback. To include an UNDO statement with your SQL INSERT, UPDATE, or DELETE, you must include an enclosing transaction to override the transaction that the SQL statement started. For example, in the following procedure the UNDO does not actually roll back any work; the UPDATE statement is treated as a transaction.

```
DO ON ERROR UNDO, LEAVE:
  UPDATE Customer SET Credit-Limit = 0 WHERE State = 'NH'
  UNDO, LEAVE.
END.
```

If you explicitly start a transaction, however, the UPDATE statement is treated as a subtransaction and the UNDO causes the UPDATE to be undone.

```
DO TRANSACTION ON ERROR UNDO, LEAVE:
  UPDATE Customer SET Credit-Limit = 0 WHERE State = 'NH'
  UNDO, LEAVE.
END.
```

3.10 Database Triggers

If a table has database triggers defined in the schema, then accessing rows in that table with Progress/SQL statements causes those triggers to execute. Table 3–6 lists the SQL statements that can cause schema triggers to execute.

Table 3–6: SQL Statements and Database Schema Triggers

SQL Statement	Triggers
DELETE FROM	FIND and DELETE
FETCH	FIND
INSERT INTO	CREATE, ASSIGN, and WRITE
SELECT	FIND
UPDATE	FIND, ASSIGN, and WRITE
UNION	FIND

Schema triggers execute in both interactive SQL and ESQL. You cannot define session triggers in Progress/SQL.

Progress manages SQL write triggers for INSERT and searched UPDATE statements by deferring execution of all write triggers until the entire searched UPDATE or INSERT statement has executed. However, write triggers are a part of the statement's transaction, so if a trigger fails, the entire searched UPDATE or DELETE statement is undone.

3.11 Privilege Checking

Progress automatically performs privilege checking at compile time for procedures containingProgress/SQL statements. This is done the same way in Progress 4GL procedures. You can view the table-level and column-level compile-time security privileges by choosing Admin→ Security in the Data Administration tool. However, you cannot use this menu to change privileges if you created the table you are viewing with the CREATE TABLE statement. Instead, you must use the GRANT and REVOKE statements.

In addition to providing compile-time security, you can also check privileges at run time. To prevent unauthorized users from running procedures that contain Progress/SQL statements, use the CAN-DO function to check the user ID established during login, or check the contents of the user ID directly with the Progress/SQL USER keyword or the Progress USERID function. The USER keyword is equivalent to the value of the USERID function for the current default (working) database.

If you use the CAN-DO or USERID function in Progress/SQL, you must modify and recompile the procedure whenever you want to change the user IDs that are allowed to execute it. Alternatively, you can set up an activities-based table to define the users who are permitted to run a particular procedure. You read the table in your procedure to check the permissions for the current user ID.

See the *Progress Language Reference* for descriptions of the CAN-DO and USERID functions. For an explanation of compile-time security and activities-based security checking, see the *Progress Programming Handbook*.

4

Data Definition Language

WithProgress, you can use SQL-89 data definition statements, the Progress Data Dictionary, or both to create SQL-89 tables and Progress schema.

You can use the SQL-89 Data Definition Language to create or modify a database. However, you must define a single table with a single mechanism.

Progress Software Corporation recommends that you use the Data Dictionary to define tables, even if you expect to use SQL-89 data manipulation statements against them. This establishes a consistent way to maintain the data for all the tables in your application. The only time you must use SQL-89 to define a table is to modify the privileges and change controls that SQL provides using the GRANT and REVOKE statements.

To access the data in a database, you can use SQL-89, Progress 4GL data manipulation statements, or both.

You must compile and run a Progress procedure containing SQL-89 data definition statements using the same user ID. This is checked at run time, and Progress returns an error message if different user IDs attempt to compile and run the procedure. You cannot compile or run data definition statements when the user ID is blank.

This chapter describes:

- Tables
- Data types
- Views
- Indexes
- Access privileges
- Schema creation

4.1 Tables

A table is a collection of information organized into named columns. The following SQL statements are used to define, modify, and delete tables. You can use the following SQL statements to add and drop indexes.

CREATE TABLE

Creates a new, empty table and defines the columns and their data types.

ALTER TABLE

Adds new columns, deletes existing columns, or changes characteristics of existing columns.

DROP TABLE

Deletes a table from the database.

When you create a new database, the security administrator in the Data Administration tool is automatically set to an asterisk (*), which means that all users can create tables. To limit the users who can create tables, replace the asterisk with a list of user names.

When you create a table, you become the owner of that table. The owner is the first name in the privilege fields of the schema tables and, by default, is the only user who can access the table. You cannot change the owner.

The owner of a table can grant and revoke access privileges using the GRANT and REVOKE statements. In addition to controlling access privileges, the owner of a table is the only user who can alter or delete the table, and create or delete associated indexes.

NOTE: If you use SQL data definition statements to define or modify the schema for a database which does not have security defined for it, you may receive an error. If you access the database without security (that is, you do not connect with -U and -P), you must have a USER environment variable already defined. If the USER environment variable is blank, you will receive the error.

4.1.1 Creating Tables

SYNTAX

```
CREATE TABLE table-name
  (  { column data-type [ column-options ]
       | UNIQUE ( column [ , column ] )  }
     [    , column data-type [ column-options ]
       | , UNIQUE ( column [ , column ] )
     ] ...  )
```

NOTE: You cannot create tables for non-Progress databases.

With the CREATE TABLE statement, you enter a name for the table and define one or more columns. No two columns can have the same name in the same table. For each column you name, you must enter a data type for that column. You can also enter column options for each column.

SYNTAX

```
  [ COLUMN-LABEL string ]
  [ DEFAULT initial-value ]
  [ FORMAT string ]
  [ LABEL string ]
  [ [ NOT ] CASE-SENSITIVE ]
  [ NOT NULL [ UNIQUE ] ]
```

Use the optional keywords NOT NULL to indicate that the column must have a value and UNIQUE to indicate that all values in the column must be unique.

The optional keyword CASE-SENSITIVE ensures the column is case sensitive, in accordance with the ANSI SQL89 Level 2 standard. If you indicate that the column is case sensitive, any comparisons to the column are also case sensitive. For example, the comparison “IF CONTACT = NAME” is case sensitive if either the column CONTACT or NAME is case sensitive.

If you attempt to use the CASE-SENSITIVE keyword with noncharacter columns, Progress returns a warning error and the keyword is ignored. Use NOT CASE-SENSITIVE to disable the case sensitive default created using the ANSI SQL (-Q) startup parameter.

Including the UNIQUE keyword in an individual column definition is equivalent to including that column in a separate UNIQUE clause. When you use the keyword UNIQUE to specify that a column or combination of columns must be unique, Progress automatically creates an index on the column or columns and adds records to the _Index and _Index-Field schema tables. The index is named sql-uniq*n*, where *n* is a number Progress assigns.

Consider the following points when creating a unique index:

- The first unique column you specify in the CREATE TABLE statement becomes the primary index. If you do not specify a unique constraint on any column, Progress creates a default index named sql-default, which orders rows by their internal RECID.
- If you do not specify any unique columns in the CREATE TABLE statement, you can create indexes later on.
- You cannot use the DROP INDEX statement to remove an index created with the UNIQUE option in a CREATE TABLE statement. This is because such an index is part of the basic table definition and its deletion removes the unique constraint check.
- If you include a column in a separate UNIQUE clause, you must also specify that the column is NOT NULL in the individual column definition.

The following example creates a table named cust_table. The cust_num field is indexed because of the UNIQUE option, and its column definition is specified as NOT NULL.

```
CREATE TABLE Cust_Table
  (Cust_Num INTEGER NOT NULL,
   Name CHARACTER (30),
   Address CHARACTER (30),
   City CHARACTER (15),
   State CHARACTER (2),
   Phone CHARACTER (10),
   Credit_Limit DECIMAL (6),
   Balance DECIMAL (6),
   UNIQUE (Cust_Num) ).
```

You can also describe how the data appears on the screen or in a printed report by using the Progress syntax for display formats and labels. For information on the FORMAT, LABEL, and COLUMN-LABEL format phrases, see the Format Phrase reference entry in the *Progress Language Reference*.

4.1.2 Altering Tables

The ALTER TABLE statement lets you add new columns to a table, delete columns from a table, or change the format and labels associated with an existing column. The ALTER TABLE statement has the following syntax.

SYNTAX

```
ALTER TABLE table-name
  {
     { ADD COLUMN column-name data-type
             [ FORMAT string ]
             [ LABEL string ]
             [ COLUMN-LABEL string ]
             [ DEFAULT initial-value ]
             [ [ NOT ] CASE-SENSITIVE ]    }
   |  { DROP COLUMN column-name }
   |  { ALTER COLUMN column-name
             [ FORMAT string ]
             [ LABEL string ]
             [ COLUMN-LABEL string ]
             [ DEFAULT initial-value ]
             [ [ NOT ] CASE-SENSITIVE ]    }
  }
```

You can enter the CASE-SENSITIVE, FORMAT, LABEL, DEFAULT, and COLUMN-LABEL keywords in any order. You cannot alter tables in non-Progress databases.

The following example adds a new column to a table named cust_table.

```
ALTER TABLE Cust_Table
  ADD COLUMN Sales_Rep CHARACTER (3)
    LABEL 'Sales Rep'.
```

When you add a new column to an existing table, Progress inserts null or default values in all existing rows for the new column. Therefore, you cannot use the NOT NULL clause in the ALTER TABLE statement unless you specify a default. You can, however, replace the column's null values by using an UPDATE statement following the ALTER TABLE statement. In addition, you cannot use the UNIQUE qualifier if the table already contains data rows, because all rows receive the same value for the new column.

You can also specify display formats and labels for column data by using the Progress syntax for display formats and labels. For information on the FORMAT, LABEL, and COLUMN-LABEL format phrases, see the Format Phrase reference entry in the *Progress Language Reference*.

4.1.3 Deleting Tables

The DROP TABLE statement deletes a table from the database. It also deletes all indexes defined on that table, all access privileges, and all data associated with the table. Following is the syntax of the DROP TABLE statement.

SYNTAX

```
DROP TABLE table-name
```

The following example deletes a table named cust_table.

```
DROP TABLE Cust_Table.
```

Only the owner of a table can delete that table. You cannot delete a table that is referenced in a view definition.

4.2 Data Types

When you define columns in a CREATE TABLE or ALTER TABLE statement, you must specify a data type for each new column. The data type determines the kind of data (such as characters, digits, or a date) that a column can store.

CHARACTER(*n*)

A character string of *n* characters. In interactive SQL, the length is used only to form the default display format (" *x* (*n*)"), since character values in Progress are variable-length. You can abbreviate CHARACTER to CHAR.

INTEGER

A whole number from -2,147,483,648 to 2,147,483,647, inclusive. You can abbreviate INTEGER to INT.

SMALLINT

Maps to the INTEGER data type.

DECIMAL [(*m* [, *n*])]

A decimal number up to 50 digits long. The precision (*m*) specifies the number of significant digits. The scale (*n*) determines the number of digits to the right of the decimal point. The scale can be up to 10 digits (15 in ANSI mode). Progress/SQL truncates the value to the specified scale before storing it in the column. You can abbreviate DECIMAL to DEC.

FLOAT [(*m*)]

Maps to the DECIMAL data type. By default, the FLOAT data type receives a sliding point format.

DOUBLE PRECISION

Maps to the DECIMAL data type. The DOUBLE PRECISION data type receives a sliding point format.

DATE

A date from 1/1/32768 BC through 12/31/32767 AD. You must enclose DATE values within either single or double quotation marks. You can specify dates in this century with the format *mm/dd/yy* or with *mm/dd/yyyy*. Dates in other centuries require the format *mm/dd/yyyy*.

LOGICAL

Yes/no or true/false values.

REAL

Maps to the DECIMAL data type.

NUMERIC [(*m* [, *n*])]

Equivalent to DECIMAL.

Progress/SQL rejects any violations of data type specifications. For example, you cannot insert a character-string value into a column defined as DECIMAL. Progress data typing restrictions are not as strict in ESQL. See the *Progress Embedded SQL-89 Guide and Reference* for more information.

4.3 Views

A view is a virtual table that consists of rows and columns from one or more tables in a database. To the user, a view looks like a table, but it does not exist as such. A view represents data in the table or tables used to define it, rather than physical data of its own.

CREATE VIEW

Creates a view from columns and rows of one or more existing tables or other views.

DROP VIEW

Deletes a view from the database.

To create a view you must have at least the SELECT privilege on the tables to which the view refers (see the "Access Privileges" section for more information). When you create a view, you become its owner. If the view is updatable, the owner's initial privileges are the same as the privileges on the underlying table. If the view is not updatable, the owner's initial privilege is SELECT only. The "Updating Views" section describes the conditions that determine whether a view is updatable.

The owner of a view has the GRANT OPTION on these privileges only if the owner also has the GRANT OPTION on the corresponding privileges of the underlying table. Privileges inherited from the underlying tables are those in effect at the time you create the view. Changes to privileges in the underlying tables do not affect the privileges in existing views.

Once you have created a view, you can use it as a table; the view follows the same semantics as a table. You can update the underlying table through the view if the view is updatable. You cannot create an index on a view or alter a view. In addition, only the owner of a view can delete that view.

To see information that is stored for a particular view, choose Admin→ Load Data and Definitions→ SQL Views from the Data Administration tool, or use SELECT statements to extract the fields from the _View schema table.

4.3.1 Creating Views

The CREATE VIEW statement creates a view based on one or more tables, views, or both. The CREATE VIEW statement has the following syntax.

SYNTAX

```
CREATE VIEW view-name [ ( column [ , column ] ... ) ]
  AS SELECT-statement [ WITH CHECK OPTION ]
```

The view name you specify must be unique among all existing table names and view names in the database. A view of a non-Progress database created with CREATE VIEW is stored in the schema holder, not in the non-Progress database.

The SELECT in the CREATE VIEW statement determines the tables and other views used to create the view and the columns that appear in the view. The view column names can be the same as or different from the table column names. To use the table column names as the view column names, omit the *column-list*. To use different column names in the view, enter the column names separated by commas. If you are using different column names, the names cannot contain any characters invalid in SQL. However, you must supply a *column-list* if there are any expressions or duplicate column names in the select list.

The following example creates a view that includes only customers from New York.

```
CREATE VIEW ny_view
  AS SELECT * FROM Cust_Table
    WHERE State = 'NY'.
```

The keywords WITH CHECK OPTION ensure that all updates to the view (for example, inserting a new row or updating an existing row) satisfy the view-defining condition. The view-defining condition is located in the WHERE clause of the SELECT statement in the view definition. The error message "VIEW INSERT OR UPDATE VIOLATES THE VIEW DEFINITION" occurs at run time if a view update is rejected.

The following example shows how the WITH CHECK OPTION clause selects only those customers from Massachusetts.

```
CREATE VIEW view_mass
  AS SELECT Name, City, State FROM Cust_Table
    WHERE State = 'MA' WITH CHECK OPTION.
```

Without the WITH CHECK OPTION clause, you can insert a row or update an existing row through this view with a value other than MA for the state column. Those rows would not appear in the view (although they would exist in the database). If you specify the WITH CHECK OPTION option, such inserts or updates are rejected.

The WITH CHECK OPTION option is especially useful when a view serves as a security feature. For example, assume a security administrator creates a view for clerical use. You must enter all new customers through this view.

```
CREATE VIEW view_num
  AS SELECT * FROM Customer
    WHERE Cust_Num BETWEEN 1000 AND 9999
    WITH CHECK OPTION.
```

The WITH CHECK OPTION option enforces that customer numbers must fall in the range 1000-9999. The WITH CHECK OPTION option prohibits you from changing an existing number or inserting a new number if it is outside the range.

NOTE

- You cannot specify the WITH CHECK OPTION option on a non-updatable view. If you try, the error message "CHECK OPTION NOT PERMITTED ON NON-UPDATABLE VIEWS" appears for the CREATE VIEW statement.

You can nest views in a view definition; you can define a view in terms of another view. If the top-level view contains the WITH CHECK OPTION option, then the WHERE clauses for all the nested views are enforced for INSERT and UPDATE. If you do not specify a WITH CHECK OPTION option for the top-level view, then the WHERE clauses for any underlying view with the WITH CHECK OPTION option are enforced instead.

4.3.2 Deleting Views

The DROP VIEW statement deletes one or more views from the database. The DROP VIEW statement has the following syntax.

SYNTAX

```
DROP VIEW view-name
```

The following example deletes a view named my_view.

```
DROP VIEW my_view.
```

If you have defined other views in relation to the view you are deleting, Progress/SQL does not permit the deletion. The underlying tables are not affected by deleting the view.

4.3.3 Updating Views

When you update a view, you update the underlying table for the view. The same SQL data manipulation statements update rows in views and tables. For example, to add new rows using a view, use the INSERT INTO statement. Substitute the view name for the table name to manipulate the rows.

Updates to views can be limited by the view definitions used to create them. Your choice of keywords, expressions, constants, and aggregate functions in the SELECT statement determines whether you can update a view. You can specifically impose restrictions on updating views using the CHECK OPTION keywords as a run-time security feature or by setting restricted privileges on the view.

You can update the underlying database through the view as long as the view is updatable. A view is updatable only if you observe the following restrictions:

- The view definition must not include the keyword DISTINCT, a GROUP BY clause, a HAVING clause, or contain expressions, constants, or aggregate functions in the SELECT list.
- The FROM clause of the SELECT statement in the view definition must specify only one table.
- The WHERE clause, if any, cannot include a subquery.
- If the FROM clause references a view, that view must be updatable.
- If a view is defined with the CHECK OPTION, you should know the criteria for the view-defining condition before attempting to update and insert rows. If you try to update a view that does not satisfy the view-defining condition, the update (or insert) is rejected, and Progress returns the error message, “VIEW INSERT OR UPDATE VIOLATES THE VIEW DEFINITION.”

4.4 Indexes

Indexes provide for more efficient retrieval of rows and optimize the performance of queries. In addition to using the UNIQUE option in a CREATE TABLE statement to create indexes, you can add and delete index definitions directly. You can use the following SQL statements to add and drop indexes.

CREATE INDEX

Creates an index on one or more columns.

DROP INDEX

Deletes an index.

Only the owner of a table can add and delete indexes on that table.

4.4.1 Creating Indexes

The CREATE INDEX statement creates an index on one or more columns. The CREATE INDEX statement has the following syntax.

SYNTAX

```
CREATE [ UNIQUE ] INDEX index-name
  ON table-name ( column [ , column ] )
```

The following example creates a unique index on the order-num and order-line columns of the order-line table.

```
CREATE UNIQUE INDEX po
  ON Order_Line (Order_Num, Line_Num).
```

The UNIQUE option places a restriction on the column or columns of the index. The restriction requires that the values in the column or combination of columns are unique; no duplicate entries are allowed. In the preceding example, the keyword UNIQUE ensures that each set of order-num and line-num values are unique within the table.

The index name must also be unique within the database.

4.4.2 Deleting Indexes

The DROP INDEX statement deletes one or more indexes from the database. The DROP INDEX statement has the following syntax.

SYNTAX

```
DROP INDEX index-name
```

The following example deletes an index named po.

```
DROP INDEX po.
```

NOTE

- You cannot use the DROP INDEX statement to delete an index created with the UNIQUE option in a CREATE TABLE statement.

4.5 Access Privileges

To perform operations on tables and views, you must have the appropriate privileges. Privileges consist of a combination of objects (table, view, or column) and one or more operations. The operations are SELECT, INSERT, UPDATE, DELETE, and ALL PRIVILEGES.

The owner (creator) of a table automatically holds all privileges on that table. The owner's privileges of a view depend on the privileges the view owner holds on the underlying tables and views. If the owner holds INSERT, UPDATE, and DELETE privileges on the underlying tables and views, and the view is updatable, then the owner holds those same privileges on the view. Otherwise, the owner holds only the SELECT privilege on the view.

As the owner of a table or view, you can grant privileges to other users using the GRANT statement. Using the WITH GRANT OPTION clause, you can allow the recipients of the privileges to grant those same privileges to other users. The owner and any user of the GRANT privilege can revoke privileges using the REVOKE statement. However, no privileges can be revoked from the owner.

4.5.1 Granting Privileges

The GRANT statement grants access privileges to users of a table or view, and optionally grants the recipient the right to grant those privileges to other users. The GRANT statement has the following syntax.

SYNTAX

```
GRANT
  {    ALL [ PRIVILEGES ]
     | [ SELECT ] [ INSERT ] [ DELETE ]
       [ UPDATE  [ ( column-list ) ] ]
  }
  ON table-name TO { grantee-list | PUBLIC }
    [ WITH GRANT OPTION ]
```

The *grantee-list* is either a list of user names separated by commas, or the keyword PUBLIC. The keyword PUBLIC grants the specified privileges to all users. It is equivalent to the use of the asterisk (*) in Progress privilege fields.

Table 4–1 lists the Progress/SQL statements and their associated Progress privilege fields.

Table 4–1: Progress/SQL Statements and Progress Privileges

SQL Statement	Progress Privilege Field
SELECT	_Can-read
INSERT	_Can-create
DELETE	_Can-delete
UPDATE	_Can-write

To allow the recipients of the privileges the right to grant those privileges to other users, use the WITH GRANT OPTION keyword. In the privilege lists of the schema tables, Progress uses a pound sign (#) preceding the user name to indicate that the user does not have the GRANT privilege. The absence of the pound sign indicates that the user holds the GRANT privilege and can grant the privilege to other users.

The following statement grants all privileges on the cust_table to users lewis and agganis. It also allows them to grant any of the privileges to other users.

```
GRANT ALL PRIVILEGES
  ON Cust_Table
  TO lewis, agganis
  WITH GRANT OPTION.
```

The following example grants user garcia the SELECT privilege on all columns, and the UPDATE privilege on only the credit_limit column of the cust_table.

```
GRANT SELECT, UPDATE (Credit_Limit)
  ON Cust_Table
  TO garcia.
```

4.5.2 Revoking Privileges

The REVOKE statement revokes either particular privileges or all privileges previously granted to one or more users. The owner and any user with the GRANT privilege can revoke privileges. The REVOKE statement has the following syntax.

SYNTAX

```
REVOKE
  { ALL [ PRIVILEGES ]
    | [ SELECT ] [ INSERT ] [ DELETE ]
      [ UPDATE [ ( column-list ) ] ]
  }
ON table-name FROM { grantee-list | PUBLIC }
```

The *grantee-list* is either a list of user names separated by commas, or the keyword PUBLIC. The keyword PUBLIC revokes the specified privileges from all users, except those users to whom privileges have been individually granted. No privileges can be revoked from the owner.

The following example revokes DELETE and UPDATE privileges on the cust_table from users jones and brown.

```
REVOKE DELETE, UPDATE
  ON Cust_Table
  FROM jones, brown.
```

4.6 Creating a Schema

Progress/ESQL provides access to all DDL statements available in Progress/SQL. In addition, Progress/ESQL provides the CREATE SCHEMA statement that allows you to create an entire schema in a single statement. The CREATE SCHEMA statement lets you combine a set of table and view definitions and grant privileges on them.

SYNTAX

```
EXEC SQL CREATE SCHEMA AUTHORIZATION db-user
  {   CREATE-TABLE-statement
    | CREATE-VIEW-statement
    | GRANT-statement
  } ... ;
```

The substatement options allow you to create tables and views, and grant table and column privileges in a connected database. The value *db-user* must be the logical name of the database. The substatement options can refer to tables and views created by earlier substatement options specified in the same CREATE SCHEMA statement.

This statement uses the logical database name *db-user* as a user ID and password to allow your application to create multiple schemas, possibly with the same table and view names, associated with different users. To distinguish the schemas in Progress, each db-user corresponds to a different logical database.

In the following example, the application creates a customer table in a schema with the user name joe, and attempts to access the table with the following SELECT statement fragment.

```
CREATE SCHEMA AUTHORIZATION joe
    CREATE TABLE Customer (Cust_Num INT, Name CHAR(20))
    CREATE UNIQUE INDEX Cust_Num FOR Customer(Cust_Num)
    CREATE TABLE Order  (Order_Num INT, Cust_Num INT);
    SELECT * FROM Customer ...
```

To resolve any ambiguity between the customer table in the joe schema and a table by the same name in other schemas, ESQL/C interprets the SELECT statement fragment, as follows.

```
    SELECT * FROM joe.Customer ...
```

The CREATE SCHEMA statement is supported in ESQL only. For more information on the CREATE SCHEMA statement, see Appendix A, "Progress/SQL-89 Reference" and the *Progress Embedded SQL-89 Guide and Reference*.

A

Progress/SQL-89 Reference

This appendix contains a reference entry for each Progress/SQL-89 statement.

ALTER TABLE Statement

Adds new columns to a table, deletes columns from a table, or changes the format or labels associated with an existing column.

SYNTAX

```
ALTER TABLE table-name
  {
    {  ADD COLUMN column-name data-type
            [ FORMAT string ]
            [ LABEL string ]
            [ COLUMN-LABEL string ]
            [ DEFAULT initial-value ]
            [ [ NOT ] CASE-SENSITIVE ]
     }
       |  { DROP COLUMN column-name }
       |  { ALTER COLUMN column-name
            [ FORMAT string ]
            [ LABEL string ]
            [ COLUMN-LABEL string ]
            [ DEFAULT initial-value ]
            [ [ NOT ] CASE-SENSITIVE ]
          }
  }
```

table-name

Represents the name of the table you want to change.

ADD COLUMN *column-name data-type*

Adds a new column to the specified table. You must specify the data type of the new column you want to add. The Progress/SQL data types are CHARACTER, INTEGER, SMALLINT, DECIMAL, FLOAT, DATE, REAL, NUMERIC, and LOGICAL.

FORMAT *string*

Specifies the display format for the column. You must enclose the *string* in single or double quotation marks. The data type determines the storage format of the column.

LABEL *string*

Specifies a label for the column. You must enclose the *string* in single or double quotation marks.

COLUMN-LABEL *string*

Specifies a label for the column when its values are displayed vertically (in columns) on the screen or in a printed report. You must enclose the *string* in single or double quotation marks.

DEFAULT *initial-value*

Assigns a default value for the column. This is the same as setting the default value for a field in the Progress Data Dictionary.

[NOT] CASE-SENSITIVE

Indicates whether the values in a character column and the comparisons made to it must be case sensitive. The default is case sensitive if you used the ANSI SQL (-Q) parameter. Otherwise, the default is not case sensitive. You cannot specify the [NOT] CASE-SENSITIVE option on ALTER COLUMN if the column being altered already participates in any indexes.

DROP COLUMN *column-name*

Deletes the specified column from the table.

ALTER COLUMN *column-name*

Changes the display format, label, default value, or column label for the specified column. (You can also change all four items for the specified column with ALTER COLUMN.)

EXAMPLES

The following examples show how you can add or change a column for the cust_table table.

```
ALTER TABLE Cust_Table
  ADD COLUMN Contact CHARACTER (30).
```

```
ALTER TABLE Cust_table
  DROP COLUMN Sales_Rep.
```

```
ALTER TABLE Cust_Table
  ALTER COLUMN Contact
    LABEL 'Customer Contact'.
```

NOTES

- Only the owner of the table can use the ALTER TABLE statement.
- When adding a new column to a table, you cannot use the NOT NULL and UNIQUE clauses unless you use the DEFAULT clause. If you do not use the DEFAULT clause, Progress/SQL inserts null values into all existing rows for the new column.
- When you add a new column through the ALTER TABLE statement, the update privilege is set to the owner of the table. You have to grant update privileges to other users explicitly.
- If you use the ANSI SQL (-Q) parameter to enforce strict ANSI SQL conformance, all columns added with the ALTER TABLE statement are case sensitive by default. To override the default, use the NOT CASE-SENSITIVE option for each specific column. See the *Progress Startup Command and Parameter Reference* for more information about the -Q startup parameter.
- Use the CASE-SENSITIVE option only to distinguish between uppercase and lowercase values entered for a character column. For example, use CASE-SENSITIVE to define a column for a part number that contains mixed uppercase and lowercase characters.

SEE ALSO

CREATE SCHEMA Statement (ESQL Only), CREATE TABLE Statement, DROP TABLE Statement, GRANT Statement, REVOKE Statement

CLOSE Statement

Closes an open cursor.

SYNTAX

```
CLOSE cursor-name
```

cursor-name

Represents the name given to the cursor when defined in the DECLARE CURSOR statement.

EXAMPLE

The following example shows how to close the c01 cursor.

```
CLOSE c01.
```

NOTES

- If you reopen the cursor, Progress obtains the retrieval set (which might have changed) and repositions the cursor to the first row.
- You can use the CLOSE statement in interactive SQL and ESQL.

SEE ALSO

DECLARE CURSOR Statement, DELETE FROM Statement, FETCH Statement, INSERT INTO Statement, OPEN Statement, UPDATE Statement

CREATE INDEX Statement

Creates an index on a table that was created using the SQL Data Definition Language.

SYNTAX

```
CREATE
  [ UNIQUE ] INDEX index-name
  ON table-name ( column [ , column ] )
```

[UNIQUE]

Indicates that all values in the specified column or combination of columns must be unique.

INDEX *index-name*

The name you want to assign to the index. The index name must be unique within the database, not just within the table.

ON *table-name*

The name of the table that contains the column or columns you want to index. This table must have been created with the SQL Data Definition Language.

column

The name of a column that you want to index. You can define an index on one or more columns.

EXAMPLE

This example creates the num index.

```
CREATE UNIQUE INDEX num
  ON Cust_Table (Cust_Num).
```

NOTES

- The index is built immediately for all existing rows in the table.
- If you specify UNIQUE, and all values in the existing rows are not unique, Progress/SQL returns an error message and does not create the index.
- In a unique index, there can be only one record with unknown values in index fields. Therefore, the index fields in an SQL-managed table must all have the _mandatory flag set. If you attempt to create a unique index on an SQL table without all key fields set to _mandatory, Progress/SQL returns the error message:

```
All columns in a UNIQUE index must be NOT NULL.
```

SEE ALSO

DROP INDEX Statement

CREATE SCHEMA Statement (ESQL Only)

Creates tables and views and grants privileges on them. The database in which the schema is created must be connected when you execute the CREATE SCHEMA statement.

SYNTAX

```
CREATE SCHEMA AUTHORIZATION db-user
  {   CREATE-TABLE-statement
    | CREATE-VIEW-statement
    |  GRANT-statement
  } ...
```

db-user

This term serves as a logical database name, a user ID, and a password. When you execute the CREATE SCHEMA statement, Progress searches for a connected database with the logical name *db-user*. It then establishes *db-user* as the user ID and password for that database connection.

CREATE-TABLE-statement

A CREATE TABLE statement describing a table to be created in the database.

CREATE-VIEW-statement

A CREATE VIEW statement describing a view to be created in the database.

GRANT-statement

A GRANT statement granting privileges on a table or view to the owner of the table or view (or any user who holds the GRANT OPTION on the table or view).

NOTES

- You can execute the CREATE SCHEMA statement only from ESQL. You cannot use it in interactive SQL.
- When the CREATE SCHEMA statement is executed, Progress searches for a database with the specified logical name. It creates the alias DICTDB for that database. It then sets the user ID for that database connection to the logical name you specified. It also uses that name as a password.
- Each CREATE TABLE, CREATE VIEW, and GRANT statement within a CREATE SCHEMA statement is executed separately. If an error occurs in one statement, that specific statement is undone and the CREATE SCHEMA is terminated. However, all previous schema changes are committed.

SEE ALSO

CREATE TABLE Statement, CREATE VIEW Statement, GRANT Statement

CREATE TABLE Statement

Creates a new table containing the columns you specify.

SYNTAX

```
CREATE TABLE table-name
  ( { column data-type [ column-options ]
    | UNIQUE ( column [ , column ] )
    }
    [   , column data-type [ column-options ]
      | , UNIQUE ( column [ , column ] )
    ] ...
  )
```

table-name

The name of the base table you want to create.

column

The name of a column for the table.

data-type

The data type of the column you want to define. The data types are CHARACTER, INTEGER, SMALLINT, DECIMAL, FLOAT, DATE, and LOGICAL.

column-options

Options that describe the field. This is the syntax for *column-options:*

SYNTAX

```
[ COLUMN-LABEL string ]
[ DEFAULT initial-value ]
[ FORMAT string ]
[ LABEL string ]
[ [ NOT ] CASE-SENSITIVE ]
[ NOT NULL [ UNIQUE ] ]
```

COLUMN-LABEL *string*

Specifies a new label for the column when its values are displayed vertically (in columns) on the screen or in a printed report. You must specify the label as a constant enclosed in single or double quotation marks. You can create stacked or multi-line column labels. Indicate a line break within the label by inserting an exclamation point.

DEFAULT *initial-value*

Assigns a default value for the column. This is the same as setting the initial value for a field in the Progress Data Dictionary.

FORMAT *string*

Specifies a new display format for the column. You must enclose the *string* in single or double quotation marks. The data type determines the storage format of the column.

LABEL *string*

Specifies a new label for the column. You must enclose the *string* in single or double quotation marks.

[NOT **]** CASE-SENSITIVE

Indicates whether the values in a character column, and comparisons made to them, must be case sensitive. The default is case sensitive if you used the ANSI SQL (-Q) parameter. Otherwise, the default is not case sensitive. (See the -Q startup parameter information in the "NOTES" section.)

NOT NULL **[** UNIQUE **]**

Indicates that the column must have a value. The UNIQUE option indicates that all values in the column must be unique.

UNIQUE (*column-name* **[** *, column-name* **]** **...**)

Indicates that all values in the column or combination of columns must be unique. If you use the UNIQUE option, Progress/SQL automatically creates a unique index. The index is named sql-uniq*n*, where *n* is a number that makes the index name unique within the table. Each *column-name* you specify must be defined with the NOT NULL option within the CREATE TABLE statement.

EXAMPLE

This example creates a new base table with the specific columns for name, job, dept, etc. It creates a unique index on the combination of the name and startdate fields.

r-cretab.p

```
CREATE TABLE Employee (Emp_Num INTEGER NOT NULL UNIQUE,
                       Name CHARACTER(30) NOT NULL,
                       Job CHARACTER(15),
                       Dept CHARACTER(15),
                       Startdate DATE NOT NULL,
                       Salary DECIMAL(8,2),
                       UNIQUE(Name, Startdate)).
```

NOTES

- The user who creates a table can grant privileges on that table to other users. See the GRANT statement reference entry for more information.
- You cannot use the DROP INDEX statement to remove an index created with the UNIQUE option in a CREATE TABLE statement because the index is part of the base table definition. To create an index that you can later remove, use CREATE INDEX.
- You cannot use CREATE TABLE to create a table in an ORACLE database. However, you can use it to create a table in a schema holder database.
- Use the CASE-SENSITIVE option only to distinguish between uppercase and lowercase values entered for a character column. For example, use CASE-SENSITIVE to define a column for a part number that contains mixed uppercase and lowercase characters.
- If you use the ANSI SQL (-Q) parameter to enforce strict ANSI SQL conformance, all columns that the CREATE TABLE statement creates are case sensitive by default. Use the NOT CASE-SENSITIVE option for each column that you want to override the default. See the *Progress Startup Command and Parameter Reference* for more information about the -Q startup parameter.
- When you use the -Q parameter to start a Progress session, case sensitivity is not enforced for columns that are already defined as case insensitive.

SEE ALSO

ALTER TABLE Statement, CREATE SCHEMA Statement (ESQL Only), DROP TABLE Statement, GRANT Statement, REVOKE Statement

CREATE VIEW Statement

Creates a view from one or more base tables, other views, or both.

SYNTAX

```
CREATE VIEW view-name [ ( column [ , column ] ... ) ]
  AS SELECT-statement [ WITH CHECK OPTION ]
```

view-name

The name of the view you want to create.

column [, *column*] ...

The names of the columns in the view. If you do not specify column names, Progress/SQL uses the column names in the SELECT statement. If the select list includes expressions or duplicate column names, you must specify column names for the view.

AS *SELECT-statement*

Defines how to derive the data for the view. The SELECT statement within a CREATE VIEW statement cannot contain an ORDER BY clause or a WITH clause. If you use the following syntax elements, you cannot update data in the view:

- Format options for columns
- Using a DISTINCT, GROUP BY, or HAVING clause
- Expressions (other than simple column references) in the select-list
- More than one table in the FROM clause
- A subquery in the WHERE clause

WITH CHECK OPTION

Ensures that all updates to the view satisfy the view-defining condition in the WHERE clause of the *SELECT-statement*. You cannot specify this option if the view is non-updatable.

EXAMPLE

This example creates a table view called doc from an employee in the Documentation department.

```
CREATE VIEW Doc
  AS SELECT Emp_Num, Name, Job
    FROM Employee
    WHERE Dept = 'Documentation'.
```

NOTES

- You can use the CREATE VIEW statement in interactive SQL or ESQL. However, the SQL Preprocessor does not recognize the CREATE VIEW statement.
- You must have the SELECT privilege on all tables and views to which the view that you are creating refers.
- You cannot use CREATE VIEW to create a view in an ORACLE database. However, you can create a view in the schema holder that refers to ORACLE objects.
- You cannot use a UNION statement within the CREATE VIEW statement. You can use only a simple SELECT statement.
- The following restrictions apply to updating tables through views:
  - The view definition must not include the keyword DISTINCT, a GROUP BY clause, a HAVING clause, an expression, or an aggregate function.
  - The FROM clause of the SELECT statement must specify only one table.
  - The WHERE clause must not include a subquery.
  - If the FROM clause refers to a view, the view must be updatable.
  - Only an updatable view can include the keyword WITH CHECK OPTION in its definition.

SEE ALSO

CREATE SCHEMA Statement (ESQL Only), DROP VIEW Statement, GRANT Statement, REVOKE Statement

DECLARE CURSOR Statement

Associates a cursor name with a SELECT statement.

SYNTAX

```
DECLARE cursor-name CURSOR FOR
  { SELECT-statement | UNION-statement }
  [ FOR { READ [ ONLY ] | UPDATE } ]
```

`cursor-name`

Identifies the name of the cursor.

NOTE

- Progress does not support SQL cursor names in 4GL statements.

CURSOR-FOR {*SELECT-statement* | *UNION-statement*}

SELECT-statement specifies a retrieval set of rows that is accessible when you open the cursor. For more information, see the SELECT Statement reference entry.

UNION-statement specifies a retrieval set of rows that is accessible when you open the cursor. For more information, see the reference entry for the UNION statement.

FOR READ [ONLY]

Specifies that you cannot modify the rows retrieved for the cursor.

FOR UPDATE

Specifies that you can modify the rows retrieved for the cursor. You cannot use FOR UPDATE if you associate the cursor with a UNION statement.

EXAMPLE

This example associates the c01 cursor with a SELECT statement.

```
DECLARE c01 CURSOR FOR SELECT * FROM Customer
  WHERE Cust-Num < 5.
```

NOTE

- You can use the DECLARE CURSOR statement in interactive SQL and ESQL.

SEE ALSO

CLOSE Statement, DELETE FROM Statement, FETCH Statement, OPEN Statement, UPDATE Statement

DELETE FROM Statement

Deletes one or more rows from a table.

SYNTAX

```
DELETE FROM table-name
  [ WHERE { search-condition | CURRENT OF cursor-name } ]
```

`FROM table-name`

The name of the table from which to delete rows.

`WHERE search-condition`

Identifies the rows to delete. If you omit the WHERE clause, all rows of the target table are deleted.

`WHERE CURRENT OF cursor-name`

Identifies the cursor that points to the row to delete (positioned delete).

EXAMPLE

This example deletes records with customer numbers greater than 100 from the customer table.

```
DELETE FROM Customer
  WHERE Cust-Num > 100.
```

NOTES

- You can use the DELETE FROM statement in both interactive SQL and ESQL.
- See Chapter 3, "Data Manipulation Language." for more information about this statement.

SEE ALSO

CLOSE Statement, DECLARE CURSOR Statement, FETCH Statement, INSERT INTO Statement, OPEN Statement, UPDATE Statement

DROP INDEX Statement

Deletes an index.

SYNTAX

```
DROP INDEX index-name
```

index-name

The name of the index to delete.

EXAMPLE

This example deletes the item_num index.

```
DROP INDEX item_num.
```

NOTES

- You can use the DROP INDEX statement in interactive SQL or ESQL.
- Only the owner of an index can delete that index.

SEE ALSO

CREATE INDEX Statement

DROP TABLE Statement

Deletes a table from the database. It also deletes all indexes defined on that table and all access privileges, as well as all data.

SYNTAX

```
DROP TABLE table-name
```

table-name

The name of the table to remove.

EXAMPLE

This example deletes the employee table from the database.

```
DROP TABLE Employee.
```

NOTES

- You can use the DROP TABLE statement in interactive SQL or ESQL.
- Only the owner of a table can delete that table.
- Progress/SQL does not let you delete a table that is referenced by a view definition.

SEE ALSO

ALTER TABLE Statement, CREATE SCHEMA Statement (ESQL Only), CREATE TABLE Statement, GRANT Statement, REVOKE Statement

DROP VIEW Statement

Deletes a view from the database.

SYNTAX

```
DROP VIEW view-name
```

view-name

The name of the view to delete.

EXAMPLE

This example deletes the doc view from the database.

```
DROP VIEW doc.
```

NOTES

- You can use the DROP VIEW statement in interactive SQL or ESQL.
- Only the owner of a view can delete that view.
- Progress/SQL does not let you delete a view that is referenced by another view definition.

SEE ALSO

CREATE SCHEMA Statement (ESQL Only), CREATE VIEW Statement, GRANT Statement, REVOKE Statement

FETCH Statement

Retrieves the next row from the retrieval set that the OPEN statement accesses.

SYNTAX

```
FETCH cursor-name INTO variable-list
```

cursor-name

Identifies the name of the cursor defined in the DECLARE CURSOR statement.

INTO *variable-list*

Lists the procedure variables that receive the column values.

The *variable-list* consists of a comma-separated list of variable references. Each variable reference has the following syntax:

SYNTAX

```
variable [ [ expression [ FOR n ] ] ]
  [ [ INDICATOR ] indicator-variable
    [ [ expression [ FOR n ] ] ]
  ]
```

Each *indicator-variable* must be an integer. If a NULL value is fetched for a field, the indicator variable is set to -1. If a character field must be truncated to fit in the variable, the indicator variable is set to the original (untruncated) length of the value. Otherwise, if the selection succeeds, it is set to 0. You cannot use indicator variables for array fields.

Each *expression* is an integer array subscript. To specify a subrange of an array you can specify FOR *n*, where *n* is an integer constant.

EXAMPLE

This procedure retrieves c03.

```
FETCH c03 INTO pname, pnum.
```

NOTES

- You can use the FETCH statement in both interactive SQL and ESQL.
- Progress/SQL does not support FETCHing INTO SQL variables.

SEE ALSO

CLOSE Statement, DECLARE CURSOR Statement, DELETE FROM Statement, OPEN Statement, UPDATE Statement

GRANT Statement

Allows the owner (or any user who holds the GRANT OPTION on a table or view) to grant privileges on that table or view.

SYNTAX

```
GRANT
  {    ALL [ PRIVILEGES ]
     | [ SELECT ] [ INSERT ] [ DELETE ]
       [ UPDATE  [ ( column-list ) ] ]
  }
  ON table-name TO { grantee-list | PUBLIC }
    [ WITH GRANT OPTION ]
```

`ALL [ PRIVILEGES ]`

Grants all privileges that the granting user has (SELECT, INSERT, DELETE, and UPDATE) to the specified users.

`SELECT`

Grants the SELECT (_can-read) privilege to the specified users.

`INSERT`

Grants the INSERT (_can-create) privilege to the specified users.

`DELETE`

Grants the DELETE (_can-delete) privilege to the specified users.

`UPDATE [ (column-list) ]`

Grants the UPDATE (_can-write) privilege to the specified users. You can list the columns that the user can update. If you specify the keyword UPDATE but omit the column list, the grantees can update all columns of the specified table.

The syntax of *column-list* is as follows.

SYNTAX

```
column-name [ , column-name ] ...
```

ON *table-name*

Represents the name of the table or view where you want to grant privileges.

TO { *grantee-list* | PUBLIC }

Represents the users to whom you want to grant privileges. You can specify a list of user names or the keyword PUBLIC, which grants the privileges to all users.

The syntax of *grantee-list* is as follows.

SYNTAX

```
user-name [ , user-name ] ...
```

WITH GRANT OPTION

Grants the recipient of the privileges the right to grant those privileges to other users, and to revoke the privileges from anyone except the owner of the table. If this option is omitted, the recipient cannot grant the privileges to other users.

EXAMPLES

The following example grants privileges to retrieve, insert, and update employee information to user nancy. This also gives nancy the ability to grant privileges on the table to other users.

```
GRANT ALL PRIVILEGES
  ON Employee
  TO nancy
  WITH GRANT OPTION
```

The following example grants the privilege to retrieve doc information to public users.

```
GRANT SELECT
  ON Doc
  TO PUBLIC.
```

The following example updates the grant of the employee information to users alan, bob, and kathy.

```
GRANT SELECT, INSERT, UPDATE
  ON Employee
  TO alan, bob, kathy.
```

NOTE

- You can use the GRANT statement in interactive SQL or ESQL or within the CREATE SCHEMA statement. However, the GRANT and REVOKE statements only work on tables that you created using an SQL CREATE TABLE statement; tables created using the Data Dictionary do not accept SQL DDL statements.

SEE ALSO

ALTER TABLE Statement, CREATE SCHEMA Statement (ESQL Only), CREATE TABLE Statement, CREATE VIEW Statement, DROP TABLE Statement, DROP VIEW Statement, REVOKE Statement

INSERT INTO Statement

Adds new rows to a table.

SYNTAX

```
INSERT INTO table-name
  [ ( column-list ) ]
  { VALUES ( value-list ) | SELECT-statement }
```

table-name

The name of the table where you want to insert new rows.

(*column-list*)

The names of one or more columns where you want to insert values.

The syntax of *column-list* is as follows.

SYNTAX

```
column [ [ expression [ FOR n ] ] ]
  [ , column [ [ expression [ FOR n ] ] ] ]
```

If you specify a column list that omits one or more columns in the table, the omitted columns have null values. If you omit the column list, Progress/SQL assumes you want to insert the data into all columns in the target table in left-to-right order.

Each *expression* is an integer array subscript. To specify a subrange of an array you can specify FOR *n*, where *n* is an integer constant.

VALUES *value-list*

Specifies the values to insert into the table. Each value can be a literal, a defined Progress variable or field name, an expression, or the keyword NULL. You must enclose each character string literal in single or double quotation marks. Progress inserts the *n*th value in the VALUES clause into the *n*th column position in the INTO clause.

SELECT-statement

Selects values from columns in another table for insertion into the target table. For more information, see the SELECT StatementSELECT Statement reference entry.

EXAMPLE

This example adds new rows into the employee table.

```
INSERT INTO Employee
  (Emp_Num, Name, Job, Dept, Hiredate, Salary)
  VALUES ('1204', 'Robins, Lynn', 'Mgr', 'Research', NULL, NULL).
```

NOTE

- You can use the INSERT INTO statement in interactive SQL and ESQL.

SEE ALSO

DELETE FROM Statement, FETCH Statement, SELECT Statement, UNION Statement, UPDATE Statement

OPEN Statement

Selects the retrieval set from the execution of the SELECT clause in a DECLARE CURSOR statement.

SYNTAX

```
OPEN cursor-name
```

cursor-name

The name of the cursor defined in the DECLARE CURSOR statement.

EXAMPLE

This example opens cursor c01.

```
OPEN c01.
```

NOTES

- You can use the OPEN statement in interactive SQL and ESQL.
- You must declare a cursor before you open it.
- Progress/SQL does not support opening a cursor that is already open.

SEE ALSO

CLOSE Statement, DECLARE CURSOR Statement, DELETE FROM Statement, FETCH Statement, UPDATE Statement

REVOKE Statement

Allows the owner or any user who holds the GRANT OPTION on a table or view to revoke privileges on that table or view.

SYNTAX

```
REVOKE
  {   ALL [ PRIVILEGES ]
    | [ SELECT ] [ INSERT ] [ DELETE ]
      [ UPDATE [ ( column-list ) ] ]
  }
ON table-name FROM { grantee-list | PUBLIC }
```

`ALL [ PRIVILEGES ]`

Revokes all privileges that the revoking user has (SELECT, INSERT, DELETE, and UPDATE) from the specified users. This only revokes the privileges that the user has.

`SELECT`

Revokes the SELECT privilege from the specified users.

`INSERT`

Revokes the INSERT privilege from the specified users.

`DELETE`

Revokes the DELETE privilege from the specified users.

`UPDATE [ (column-list) ]`

Revokes the UPDATE privilege from the specified users. You can list the columns for which to revoke the UPDATE privilege. If you specify the keyword UPDATE, but omit the column list, the UPDATE privilege is revoked on all columns of the specified table.

The syntax of *column-list* is as follows.

SYNTAX

```
column-name [ , column-name ] ...
```

ON *table-name*

Indicates the name of the table or view to revoke privileges.

FROM { *grantee-list* | PUBLIC }

Identifies the users from which you want to revoke privileges. You can specify a list of user names or the keyword PUBLIC which revokes the privileges from all users.

The syntax of *grantee-list* is as follows.

SYNTAX

```
user-name [ , user-name ] ...
```

EXAMPLES

The following example revokes SELECT permissions for the public on the doc view.

```
REVOKE SELECT
  ON Doc
  FROM PUBLIC.
```

The following example revokes INSERT and UPDATE permissions for kathy on the employee table.

```
REVOKE INSERT, UPDATE
  ON Employee
  FROM kathy.
```

NOTES

- You cannot revoke privileges from the owner of a table or view.
- The GRANT and REVOKE statements only work on tables that you created using an SQL CREATE TABLE statement; tables created using the Data Dictionary do not accept SQL DDL statements.

SEE ALSO

ALTER TABLE Statement, CREATE SCHEMA Statement (ESQL Only), CREATE TABLE Statement,CREATE VIEW Statement, DROP TABLE Statement, DROP VIEW Statement, GRANT Statement

SELECT Statement

Retrieves and displays data from a table or join. For more information on the SELECT statement, see Chapter 3, "Data Manipulation Language."

SYNTAX

```
SELECT
  [ ALL | DISTINCT ]
  { * | column-list }
  [ INTO variable-list ]
  FROM {   table-name [ correlation-name ]
         | implicit-join
         | explicit-join
       }
  [ WHERE search-condition ]
  [ GROUP BY column [ , column ] ... ]
  [ HAVING search-condition ]
  [ ORDER BY sort-criteria ]
  [ WITH [ frame-options ] [ STREAM stream ] [ EXPORT ] ]
```

ALL

Retrieves all values in the specified columns. ALL is the default.

DISTINCT

Retrieves only unique sets of values (tuples) in the specified columns.

*

Indicates all columns of the specified table(s).

column-list

Specifies the names of the columns to retrieve. The syntax of the *column-list* is as follows.

SYNTAX

```
    column [ format-phrase ]
      [ , column [ format-phrase ] ] ...
```

For the syntax of *format-phrase*, see the Format Phrase reference entry in the *Progress Language Reference*.

Each *column* can also be an aggregate function. Table A–1 lists the aggregate functions Progress/SQL supports.

Table A–1: SQL Aggregate Functions

Function	Description
AVG([DISTINCT] *expression*)	Calculates an average value for all rows in the result list. The *expression* can refer to a column or a calculation.
COUNT(*)	Counts the number of rows in the result list. This count includes duplicate rows.
COUNT(DISTINCT *expression*)	Counts the number of rows with different values for *expression*.
MAX([DISTINCT] *expression*)	Returns the maximum value of *expression* for all rows (or for all distinct rows that you specify).
MIN([DISTINCT] *expression*)	Returns the minimum value of *expression* for all rows (or for all distinct rows that you specify).
SUM([DISTINCT] *expression*)	Calculates the sum of *expression* for all rows (or for all distinct rows that you specify). The *expression* must evaluate to a numeric value.

`INTO` *`variable-list`*

Lists the local program variables to receive the column values (singleton SELECT).

The *variable-list* consists of a comma-separated list of variable references. Each variable reference has the following syntax.

SYNTAX

```
variable [ [ expression [ FOR n ] ] ]
  [ [ INDICATOR ] indicator-variable
      [ [ expression [ FOR n ] ] ] ]
```

Each *indicator-variable* must be an integer. If a NULL value is fetched for a field, the indicator variable is set to -1. If a character field must be truncated to fit in the variable, the indicator variable is set to the original (untruncated) length of the value. Otherwise, if the selection succeeds, it is set to 0. You cannot use indicator variables for array fields.

Each *expression* is an integer array subscript. To specify a subrange of an array you can specify FOR *n*, where *n* is an integer constant.

You cannot use the GROUP BY or HAVING clauses with SELECT INTO.

FROM

Specifies the table or join from which to select data.

NOTE: Progress/SQL does not support 4GL buffer names in SQL FROM phrases.

table-name

Specifies the name of a table or view from which to select data.

correlation-name

Specifies an alias for a table name, useful for specifying joins between a table and itself.

implicit-join

Specifies multiple tables from which to select data using an implied inner join. This is the syntax for an *implicit-join*.

SYNTAX

```
table-name [ correlation-name ]
  [ , table-name [ correlation-name ] ] ...
```

Any SELECT statement can specify an *implicit-join*, including those used in subqueries. You can specify both table join conditions and selection criteria for each table in the join using the SELECT statement WHERE option. A *table-name* in an *implicit-join* can specify either a table or a view.

For more information on implicit joins, see Chapter 3, "Data Manipulation Language."

explicit-join

Specifies multiple tables from which to select data using a specific type of join. This is the syntax for an *explicit-join*.

SYNTAX

```
FROM table-name [ correlation-name ]
  [ INNER | LEFT [ OUTER ] ]
  JOIN table-name [ correlation-name ] ON search-condition
  [ [ INNER | LEFT [ OUTER ] ]
    JOIN table-name [ correlation-name ] ON search-condition
  ] ...
```

```
  table-name [ correlation name ] RIGHT [ OUTER ]
    JOIN table-name [ correlation-name ]
      ON search-condition ...
```

An *explicit-join* can contain either a combination of inner and left outer joins of multiple tables or a single right outer join between two tables. You can specify an *explicit-join* only in a direct SELECT statement or in a cursor SELECT specified in a DECLARE CURSOR statement. Thus, you cannot specify an *explicit-join* in a subquery (SELECT in a WHERE clause), CREATE VIEW statement, or INSERT INTO statement.

NOTE: A *table-name* in an *explicit-join* cannot specify a view.

JOIN by default or with the INNER option specifies an inner join. JOIN with the LEFT [OUTER] option specifies a left outer join, and JOIN with the RIGHT [OUTER] option specifies a right outer join.

The ON clause specifies both table join conditions and any additional selection criteria for each table in the join. The *search-condition* is one or more relational or logical expressions connected by a logical operator (AND, OR, or NOT). For an *explicit-join*, the WHERE clause of the SELECT statement specifies additional selection criteria on the complete join result and plays no part in building the join itself.

NOTE

- Unlike the WHERE clause, the *search-condition* for the ON clause cannot include a subquery.

For more information on explicit joins, see Chapter 3, "Data Manipulation Language."

`WHERE search-condition`

Specifies the conditions for selecting data from tables and joins. For an *implicit-join*, these conditions can specify table join criteria and the selection criteria for each table in the join. For an *explicit-join*, these conditions specify selection criteria on the join result and play no part in building the join itself.

The *search-condition* is one or more relational or logical expressions connected by a logical operator (AND, OR, or NOT). Each expression can include a subquery (nested SELECT). For information on subqueries, see Chapter 3, "Data Manipulation Language."

`GROUP BY column [ , column ]`

Groups rows that have the same value for the specified column or columns. Each *column* can be a reference to a column in the table or an expression. The GROUP BY clause produces a single row for each group of rows sharing the same *column* values.

`HAVING search-condition`

Specifies one or more qualifying conditions, normally for the GROUP BY clause. The HAVING clause allows you to exclude groups from the query results.

`ORDER BY sort-criteria`

Sorts the query results by the values in one or more columns. The syntax of *sort-criteria* is as follows.

SYNTAX

```
{ n | expression } [ ASC | DESC ]
[ , n | , expression [ ASC | DESC ] ] ...
```

In this syntax, *n* represents an integer that specifies the position of a column or expression in the select list. Use this option to sort on arithmetic expressions included in the select list. Each *expression* represents a column name or other value by which to sort the results. This might be a calculated expression involving one or more table columns.

NOTE

- If you repeatedly perform an SQL query that involves multiple indexes and that does not use the ORDER BY option, the order of rows returned might vary.

`ASC`

Sorts the query results in ascending order. Ascending order is the default.

`DESC`

Sorts the query results in descending order.

`frame-options`

These options specify the overall layout and processing properties of a frame. See the Frame Phrase reference entry in the *Progress Language Reference* for information about the options.

You cannot use frame options in a SELECT statement within a UNION statement.

`STREAM` *`stream`*

Specifies the name of a stream. If you do not name a stream, Progress uses the unnamed stream. See the DEFINE STREAM Statement reference entry in the *Progress Language Reference* and the *Progress Programming Handbook* for more information about streams.

You cannot use STREAM in a SELECT statement within a UNION statement.

`EXPORT`

Converts data to a standard character format and displays it to the current output destination (except when the current output destination is the screen) or to a named output stream.

You cannot use EXPORT in a SELECT statement within a UNION statement.

EXAMPLES

This example retrieves each distinct (unique) country from the customer table.

```
SELECT DISTINCT Country FROM Customer.
```

This example retrieves certain fields from the order table if the ship-date field has a null value.

```
SELECT Cust-Num, Order-Num, Order-Date
  FROM Order
  WHERE Ship-Date IS NULL.
```

This example retrieves information about customers whose balance is greater than that of each customer for sales rep JAL. The procedure lists the customers retrieved in descending order of balance.

```
SELECT Cust-Num, Name, Sales-Rep, Balance
  FROM Customer WHERE Balance > All
    (SELECT Balance FROM Customer WHERE Sales-Rep = "JAL")
  ORDER BY Balance DESC.
```

NOTES

- You can use the SELECT statement in interactive SQL and ESQL.
- In ESQL only, you must use a cursor with the SELECT statement unless you specify the INTO option to perform a singleton SELECT.
- The SELECT statement can also have one or more subqueries.
- For each row selected, the server returns the subset of the available columns specified by *column-list* plus any additional columns that the client requires to complete the query. For more information on how Progress might determine the additional columns in this subset, see the section on field lists in the *Progress Programming Handbook* and the *Progress DataServer Guide* for any DataServers you use.

SEE ALSO

CREATE VIEW Statement, DECLARE CURSOR Statement, INSERT INTO Statement, UNION Statement, UPDATE Statement

UNION Statement

Combines two or more SELECT statements to create a complex query.

SYNTAX

```
{ SELECT-statement | ( subunion ) }
  { UNION [ ALL ] { SELECT-statement | ( subunion ) } } ...
  [ ORDERED BY sort-criteria ]
```

SELECT-*statement*

An SQL SELECT statement that retrieves one or more rows from one or more tables. The SELECT statements within a UNION cannot use the INTO option. Any formatting options (such as LABEL or FORMAT) you specify in the SELECT statement are ignored. You cannot use the frame options, STREAM or EXPORT, in a SELECT statement within a UNION statement. For more information, see the SELECT StatementSELECT Statement reference entry.

subunion

Two SQL SELECT statements joined by the UNION keyword. A subunion might optionally use the ALL keyword. The syntax of *subunion* is as follows.

SYNTAX

```
SELECT-statement UNION [ ALL ] SELECT-statement
```

ALL

Specifies that the union of two SELECT statements includes any duplicate rows. If you do not specify ALL, then only one copy of each row is returned in the union.

ORDER BY *sort-criteria*

Sorts the query results by the values in one or more columns. The syntax of *sort-criteria* is as follows.

SYNTAX

```
{ n | expression } [ ASC | DESC ]
  [ , n | , expression [ ASC | DESC ] ] ...
```

In this syntax, *n* represents an integer that specifies the position of a column or expression in the select list. Use this option to sort on arithmetic expressions included in the select list. Each *expression* represents a column name or other value by which to sort the results. This can be a calculated expression involving one or more table columns.

EXAMPLES

You can use the UNION statement to retrieve information from two or more tables into a single results list. For example, the following procedure takes information from the Customer and Salesrep tables.

r-union1.p

```
SELECT Name, State FROM Customer WHERE State = "MA"
 UNION
SELECT Rep-Name, Region FROM Salesrep WHERE Region = "East".
```

You can also use the UNION statement to combine two or more queries into the same table. The following procedure combines three queries into the result table.

r-union2.p

```
SELECT Order-Num, Cust-Num, Order-Date, "Order" FROM Order
      WHERE Order-Date < 2/1/93
UNION ALL
SELECT Order-Num, Cust-Num, Ship-Date, "Ship" FROM Order
      WHERE Ship-Date < 2/1/93
UNION ALL
SELECT Order-Num, Cust-Num, Promise-Date, "Promise" FROM Order
      WHERE Promise-Date < 2/1/93
ORDER BY Order-Num1.
```

This procedure returns information about any orders with any date (order date, ship date, or promise date) before February 1, 1993. Because the ALL keyword combines each of the SELECT statements, if a single order has more than one date before February 1, 1993, then it appears more than once in the result list. The fourth column of output contains a string constant that indicates which type of date is shown in the third column. The result list is sorted by order-num because of the ORDER BY clause.

NOTES

- The columns you specify in each SELECT statement must be compatible. Each SELECT statement must refer to the same number of columns, and the *n*th column of each list must be assignment-compatible with the *n*th column of the other lists.
- The columns in the output of a UNION statement take the labels of the field specified in the first SELECT statement.

SEE ALSO

SELECT Statement, UPDATE Statement

UPDATE Statement

Changes values in one or more rows of a table.

SYNTAX

```
UPDATE table-name
  SET column-name = { NULL | expression }
  [ , column-name = { NULL | expression } ] ...
  [ WHERE { search-condition | CURRENT OF cursor-name } ]
```

`table-name`

Represents the name of the table in which to update rows.

`SET column-name = { NULL | expression } [ , column-name = { NULL | expression } ] ...`

Assigns new values to each column named. The *column-name* can be qualified with a table name or a database name and table name. Each *expression* has the following syntax.

SYNTAX

```
value-expression [ INDICATOR ] variable
  [ [ expression [ FOR n ] ] ]
```

The *value-expression* can involve a literal, column name, arithmetic operation, a defined Progress variable or field name, or any combination of these. Each indicator variable must be an integer. You can set a column to the NULL value by setting the indicator variable to -1. Otherwise, set the indicator variable to 0. You can use an indicator variable with an expression, but not with the literal NULL.

`WHERE search-condition`

Identifies the conditions under which the rows are updated. The search condition compares the values in one column to the values in another column or to a literal. If you omit the WHERE clause, all rows of the target table are updated. For more detailed information about WHERE clause search conditions, see Chapter 3, “Data Manipulation Language.”

`WHERE CURRENT OF cursor-name`

Identifies the cursor that points to the row to be updated (positioned update).

EXAMPLE

```
UPDATE Item
  SET Price = Price * 1.25
  WHERE Item-Num = 52.
```

NOTE

- You can use the UPDATE statement in interactive SQL and ESQL.

SEE ALSO

CLOSE Statement, DECLARE CURSOR Statement, DELETE FROM Statement, FETCH Statement, INSERT INTO Statement, OPEN Statement, SELECT Statement, UNION Statement, UPDATE Statement

B

Progress Metaschema Tables

This chapter describes the metaschema tables that Progress/SQL-89 uses. The metaschema tables are hidden database tables that hold the definitions of database objects such as tables, fields, indexes, and views. The information includes indication of table ownership, privileges, indexes, and the status of various flags. When SQL-89 data definition statements are processed, syntax is generated to create the appropriate schema records and to give values to certain schema fields.

The schema records created for SQL-89 tables have the same format as the schema records that Progress uses. However, some fields in the _File and _Field schema tables are not used with SQL-89-created tables. Views, however, have their own associated schema tables: _View, _View-Col, and _View-Ref.

Schema table names all begin with an underscore (_). All fields within the schema tables also begin with an underscore.

Schema table privilege fields (such as _Can-create) contain a pound sign (#) in front of the user name if the user has that privilege, but cannot grant it to or revoke it from other users. In all other respects, the form of the privilege fields is the same as in Progress.

You can update certain schema fields in the Progress Data Dictionary even if you create them with the SQL-89 CREATE TABLE statement. These include the description and help fields and other fields SQL-89 does not use. The Data Dictionary screens are automatically adjusted for SQL-89 tables to indicate which schema fields you can update using the Data Dictionary.

B.1 Schema Tables for Tables and Columns

The SQL data definition statements that you specify in your procedures directly update the fields in the schema tables _File, _Field, _Index, and _Index Field. Table B–1 and Table B–2 list the information contained in the _File, and _Field schema tables. The _Index and _Index-Field schema tables primarily contain system information.

Table B–1 lists the fields in the _File schema table.

Table B–1: Fields in the _File Schema Table

Field	Description
_File-Name	The table name, as specified in the CREATE TABLE statement. You cannot change this value.
_Can-Create	Initially set to the user ID of the user who first compiles and runs the CREATE TABLE statement for this table. You cannot remove or change this first user ID. You can modify the rest of the field using the GRANT and REVOKE statements.
_Can-Read	Initially set to the owner. You can modify this value using the GRANT and REVOKE statements.
_Can-Write	Initially set to * (all users). Write privilege (through the UPDATE statement) is checked at the column level. You cannot modify this value.
_Can-Delete	Initially set to the owner. You can modify this value using the GRANT and REVOKE statements.
_DB-Lang	Set to 0 for tables created using the Data Dictionary, and 1 for tables created with SQL. You cannot modify this value.

Table B–2 lists and describes the fields in the _Field schema table. The _Field table contains one record for each column in a table.

Table B–2: Fields in the _Field Schema Table *(1 of 2)*

Field	Description
_Field-Name	The column name, as specified in the CREATE TABLE statement. You cannot change this value.
_Data-Type	The data type of the column, as specified in the CREATE TABLE statement. You cannot change this value.
_Label	The label used for the field. The default value is a question mark (?), which means that the _Field-Name is used as the label. You can modify this value using the ALTER TABLE statement.
_Decimals	Set to the number of digits to the right of the decimal point for columns declared as DECIMAL or NUMERIC in the CREATE TABLE statement. Also used for character columns to store the declared length of the column. This field is set to the maximum for the FLOAT and REAL data types.
_Order	Set to correspond to the order of the column in the CREATE TABLE statement. This indicates the default order, a left-to-right display of the columns in the table. The lowest number is the leftmost column and highest number is the rightmost. You cannot modify this value.
_Can-Read	Initialized to all users by the CREATE TABLE statement because the read privilege (through the SELECT statement) is checked at the table level. You cannot modify this value.
_Can-Write	Initialized to the owner in the CREATE TABLE statement. You can modify this value using the GRANT and REVOKE statements.
_Col-Label	The label for the field when column labels are used. This overrides the _Label value when column labels are in effect. Column labels that appear at the top of a column can contain exclamation points (!) to indicate line breaks. You can modify this value using the ALTER TABLE statement.
_Initial	Contains the default value, if there is one. You can modify this value using the ALTER TABLE statement.

Table B–2: Fields in the _Field Schema Table *(2 of 2)*

Field	Description
_Fld-Case	Indicates whether the field is case sensitive. For tables created with the CREATE TABLE statement, the default is NO unless you use the ANSI SQL (-Q) startup parameter. You cannot change the field if the record is in an index. You can modify this value using the ALTER TABLE statement.
_Extent	Always set to 0 from the CREATE TABLE statement. Progress array fields can be referred to, but not created in, SQL statements.
_Format	Set through the CREATE TABLE or ALTER TABLE statement. If not set explicitly, an appropriate default is used, depending on the declared size of the column. You can modify this value using the ALTER TABLE statement.
_Mandatory	Set to YES for a column declared as NOT NULL in the CREATE TABLE statement. Otherwise, set to NO. You cannot modify this value.

B.2 Schema Tables for Views

Three schema tables store information on views: _View, _View-Col, and _View-Ref. The view schema tables translate all references into fully qualified base table terms.

The _View schema table contains view definitions, including view name, the names of base tables to which the view refers, base table translations of the WHERE and GROUP BY clauses (if included), and the actual text of the view definition from the CREATE VIEW statement. Four privilege fields contain read, write, create, and delete privileges. The _Updatable field indicates whether the view is updatable.

The _View-Col schema table is the equivalent of the _Field schema table. There is a record for each column in the view definition. Each record contains the view name, column name, equivalent base table column or expression, and two privilege fields.

The _View-Ref schema table is a cross-reference table used to connect views with their underlying tables. A record is created in this table for each base table column to which the view refers. This table provides information on which views are defined in terms of other views, as well as which base tables are used. This schema table prevents you from deleting tables, columns, or views that are referred to in view definitions. For example, you can use this table to look up view dependencies to see all the view definitions that refer to a particular view, base table, or column. When operating in a multi-database environment, only the view relationships within a single database are stored in that database's _View-Ref table. Dependencies on other databases are not maintained.

Table B–3, Table B–4, and Table B–5 list and describe fields that the CREATE VIEW statement creates. Never modify fields in the view schema tables.

Table B–3: Fields in the _View Schema Table *(1 of 2)*

Field	Description
_Auth-id	The owner (creator) of the view.
_View-Name	The name of the view.
_Base-Tables	A list of the base tables referred to directly or indirectly in the FROM clause of the view definition. If the view is defined in terms of other views, these view definitions are resolved to base tables when the CREATE VIEW statement is compiled.
_Where-Cls	The WHERE clause of the view definition, in terms of base tables. All column references are fully qualified and references to other views are resolved when the CREATE VIEW statement is compiled.
_Group-By	The GROUP BY clause of the view definition, also in base table terms. If the view definition contains the CHECK OPTION, the CHECK OPTION WHERE clause is stored here.
_View-Def	The original view definition as entered in the CREATE VIEW statement. This field is included primarily for documentation purposes so you can see the original view definition. Never alter this field. Any quotation marks in the view definition are not included in the _View-Def field. Otherwise, the original text is unchanged, but might be truncated when displayed on the screen.

Table B–3: Fields in the _View Schema Table *(2 of 2)*

Field	Description
_Can-Read _Can-Write _Can-Create _Can-Delete	These fields are initialized as base tables and are modified using the GRANT and REVOKE statements.
_Updatable	When the view is created, SQL determines whether it is updatable according to the ANSI standard. Set to YES or NO.

Table B–4: Fields in the _View-Col Schema Table

Field	Description
_Auth-id	The owner (creator) of the view.
_View-Name	The name of the view.
_Col-Name	The name of the column. Can be specified explicitly in the view definition, or, if no column list is included, is the same as the name of the corresponding column in the SELECT list.
_Base-Col	The fully qualified base table column name or expression to that this view column references. Fully resolved when the CREATE VIEW statement is compiled.
_Can-Write	Initialized to the owner in the CREATE VIEW statement. You can modify this value using the GRANT and REVOKE statements.
_Can-Create	Reserved for future use.
_Vcol-Order	Provides the default, left-to-right display order of the columns. The lowest number represents the leftmost column; the highest number represents the rightmost column.

Table B–5: Fields in the _View-Ref Schema Table

Field	Description
_Auth-id	The owner (creator) of the view.
_View-Name	The name of the view whose definition contains the reference.
_Ref-Table	The referenced table or view. This field is not resolved to the base table if the immediate reference is to a view.
_Base-Col	The referenced base table column. This field is resolved to the base table, even if the immediate reference is to a view.

Index

Symbols

(?) unknown value 2–5

A

ADD COLUMN option
  ALTER TABLE statement A–2

Aggregate functions 3–13

ALL option
  GRANT statement A–23
  privilege operations 4–13
  REVOKE statement A–29
  SELECT statement 3–3, A–32
  UNION statement A–39

ALTER TABLE statement 4–2, 4–5, A–2

AND logical operator 2–5, 3–4

AS SELECT statement, CREATE VIEW statement A–13

ASC option
  SELECT statement A–37

Ascending sort order 3–9

ASSIGN trigger 3–40

Audience ix

AVG
  aggregate function 3–13

B

BETWEEN search condition 3–4

Bold typeface, as typographical convention x

C

CAN-DO function 3–41

CASE-SENSITIVE option
  ALTER TABLE statement A–3
  CREATE TABLE statement A–11

CHARACTER data type 4–6

Character set
  SQL
    See SQL statements.

Character sets
  in SQL 2–2

Character strings
  literals 2–4
  matching 2–4

CLOSE statement A–5

syntax 3–36

Column names 2–2
as identifiers. *See* Identifiers.
duplicate 2–3
qualifiers 2–3
views 4–9

COLUMN-LABEL option
ALTER TABLE statement A–3
CREATE TABLE statement A–11

Columns
adding to tables 4–5
ordinal positions 3–9

Comments, in Progress/SQL 2–2

Compiling procedures, with SQL statements 4–1

CONNECT statement, Data Server support 1–3

CONTAINS search condition 3–6

Correlated subqueries 3–12

Correlation names 3–18, 3–22
as identifiers
See Identifiers.
cursors 3–32

COUNT, aggregate function 3–13

CREATE INDEX statement 4–12, A–6
syntax 4–12

CREATE SCHEMA statement 4–16, A–8

CREATE TABLE statement 4–2, 4–3, A–8, A–10

CREATE triggers 3–40

CREATE VIEW statement 4–8, A–8, A–13
fields B–5
syntax 4–8
WITH CHECK OPTION option 4–9

Cursors 3–27
closing 3–36
declaring 3–28, 3–30
defining 3–27, 3–32
opening 3–33
restrictions 3–32
statements 3–28

D

Data
exporting 3–10
grouping 3–8
maintaining 4–1
retrieval using cursors 3–27
sorting query results 3–9
types 4–6

Data Definition Language 4–1
statements 1–4
in Progress/SQL 4–16

Data Dictionary, defining database structure 1–2

Data Manipulation Language 3–1

Database objects B–1

Database triggers, with Progress/SQL 3–40

Databases
defining with the Data Dictionary 1–2
non-Progress and SQL 1–3
schema user names 4–16

DATE data type 4–7

DECIMAL data type 4–7

DECLARE CURSOR statement A–15
record locking 1–4
syntax 3–30
UNION statement A–15

DEFAULT option
ALTER TABLE statement A–3
CREATE TABLE statement A–11

DELETE FROM statement A–17

DELETE option
GRANT statement A–23
REVOKE statement A–29

DELETE statement
positioned form 3–36
privilege operations 4–13
syntax 3–26

DELETE triggers 3–40

DESC option
SELECT statement A–37
SELECT statement. *See also* Descending sort order.

Descending sort order 3–9

DISTINCT option
SELECT statement 3–3, A–32

DOUBLE PRECISION data type 4–7

DROP COLUMN option
ALTER TABLE statement A–3

DROP INDEX statement 4–12, 4–13, A–18

DROP TABLE statement 4–2, A–19
syntax 4–6

DROP VIEW statement 4–8, A–20
syntax 4–10

E

Embedded SQL 1–4

Error messages
displaying descriptions xv

Escape characters 3–6

ESQL.*See* Embedded SQL

EXISTS, search condition 3–12

Explicit joins
compared to implicit joins 3–22
examples 3–20, 3–21
explained 3–19
restrictions 3–23

EXPORT option
SELECT statement A–37

Exporting data 3–10

Expressions, logical 2–5

Extensions, with Progress/SQL 1–3

F

FETCH statement 3–33, A–21
using local variables 4–14

Field lists A–38

_File schema table B–2

Files
schema B–1

FIND trigger 3–40

FLOAT data type 4–7

FOR READ option, DECLARE CURSOR statement A–15

FOR UPDATE option, DECLARE CURSOR statement A–15

FORMAT option
ALTER TABLE statement A–2
CREATE TABLE statement A–11

Format phrases, in Progress/SQL 1–3

Frame phrase
WITH phrase
extension to Progress/SQL 1–3

Frame properties, specifying with SELECT statement 3–7

FROM option
DELETE FROM statement A–17
REVOKE statement A–30

SELECT statement 3–14, A–34

Full outer joins, defined 3–16

Functions, aggregate 3–13

G

GRANT statement 4–14, A–23
CREATE SCHEMA statement A–8

GROUP BY option
, ELECT statement 3–8

GROUP BY option, SELECT statement A–36

H

HAVING option, SELECT statement 3–8, A–36

Help
Progress messages xv

I

Identifiers 1–3
Progress/SQL 2–2
reserved words 2–2

Implicit joins
compared to explicit joins 3–22
examples 3–18
explained 3–18

IN search condition 3–6

Index names, as identifiers. *See* Identifiers.

INDEX option, statement A–6

Indexes 4–12
created by UNIQUE 4–4
creating 4–12
definition statements 4–12
deleting 4–13

Inner joins
defined 3–16
explicit A–35
implicit A–34
sample tables 3–15
specifying
explicit 3–19
implicit 3–18

INSERT INTO statement 3–24, A–26
syntax 3–24

INSERT option
GRANT statement A–23
privilege operations 4–13
REVOKE statement A–29

INTEGER data type 4–6

INTO option
FETCH statement A–21
SELECT statement A–33

Italic typeface, as typographical convention
x

J

JOIN option, SELECT statement A–35

Joins
conditions
defined 3–15
explicit 3–19
explicit compared to implicit 3–22
implicit 3–18
defined 3–14
explicit A–35
explained 3–19
restrictions 3–23
explicit compared to implicit 3–22
implicit A–34
explained 3–18
inner
explicit A–35
implicit A–34
inner explicit, specifying 3–19
inner implicit, specifying 3–18
left outer A–35

specifying 3–19
right outer A–35
specifying 3–21
self joins
explicit 3–22
implicit 3–18
types 3–16
using SELECT statement 3–14

Joins (SQL) 3–4

K

Keystrokes xi

L

LABEL option
ALTER TABLE statement A–3
CREATE TABLE statement A–11

Left outer joins A–35
defined 3–16
sample tables 3–17
specifying 3–19

LIKE clause 2–4

LIKE option, search conditions 3–5

Literals
character string 2–4
constants 2–4
in SQL statements 2–4
logic 2–4

LOGICAL data type 4–7

Logical expressions 2–5

Logical operators 3–4

M

Manual
organization of ix
syntax notation xi

MATCHES search condition 3–7

Messages, displaying descriptions xv

Metaschema tables B–1

MIN
aggregate function 3–13

Monospaced typeface, as typographical convention x

N

Name qualifiers, in SQL statements 2–3

Names
correlation 3–18, 3–22
cursors 3–32

NOT CASE-SENSITIVE option
ALTER TABLE statement A–3
CREATE TABLE statement A–11

NOT logical operator 2–5, 3–4

NOT NULL option
CREATE TABLE statement 4–3

NOT NULL option, CREATE TABLE statement A–11

NULL option
UPDATE statement A–42

NULL search condition 3–5

Null values 2–5

NUMERIC data type 4–7

O

Objects, terminology 1–2

ON option
CREATE INDEX statement A–6
GRANT statement A–24
SELECT statement A–35

ON TABLE option, REVOKE statement
A–30

ONLY option, DECLARE CURSOR
statement A–15

OPEN statement A–28
syntax 3–33

Operators 2–4
spacing 2–4

Options
ALL, SELECT statement 3–3
ASC, SELECT statement 3–9
DESC, SELECT statement 3–9
DISTINCT, SELECT statement 3–3
FROM, SELECT statement 3–14
GROUP BY, SELECT statement 3–8
HAVING, SELECT statement 3–8
LIKE, WHERE option
search conditions 3–5
NOT NULL, CREATE TABLE
statement 4–3
ON, SELECT statement 3–19
ORDER BY, SELECT statement 3–9
null values 3–9
SET, UPDATE statement 3–26
UNIQUE, CREATE TABLE statement
4–3
WHERE, SELECT statement 3–26
WITH EXPORT, SELECT statement
3–10

OR logical operator 2–5, 3–4

ORDER BY option
null values 3–9
SELECT statement 3–9, A–36
UNION statement A–39
with integers 3–9

Outer joins A–35
defined 3–16

P

Positioned DELETE statement
syntax 3–36

Positioned UPDATE statement
syntax 3–35

Predicates (SQL)
See Search conditions

Privileges
access 4–13
checking 3–41
GRANT statement 4–14
granting 4–14
objects and operations 4–13
of views 4–8
REVOKE statement 4–15
revoking 4–15

PRIVILEGES option, GRANT statement
A–23

Progress/ESQL
CREATE SCHEMA statement 4–16
DDL statements 4–16

Progress/SQL
statement syntax 2–2
terminology 1–2
transaction processing 3–38

PUBLIC option
GRANT statement 4–14
REVOKE statement A–30

Q

Qualifiers
column names 2–3
syntax 2–3
table names 2–3
view names 2–3

Queries, sorting results, ORDER BY option
3–9

R

REAL data type 4–7

Record locking

DECLARE CURSOR statement 1–4
phrases 3–31
SQL statements 3–30

Reserved words, as identifiers 2–2

Retrieval sets 3–33

Retrieving values 3–33

REVOKE statement A–29
syntax 4–15

Right outer joins A–35
defined 3–16
specifying 3–21

Rows
deleting 3–26
inserting 3–24
updating 3–26

S

Schema
creating 4–16
records B–1
tables B–2
for views B–4
triggers 3–40

Search conditions
BETWEEN 3–4
CONTAINS 3–6
for set subqueries 3–11
LIKE 3–5
MATCHES 3–7
NULL 3–5
SELECT statement, ON option 3–19

Security 3–41

SELECT INTO statement
See Singleton SELECT.

SELECT option
GRANT statement A–23
REVOKE statement A–29

SELECT statement 3–2, A–32
ALL option 3–3
ASC option,, see also Ascending sort order.
DECLARE CURSOR statement A–15
DESC option,, see also Descending sort order.
DISTINCT option 3–3
frame properties 3–7
FROM option 3–14
GROUP BY option 3–8
INSERT INTO statement A–26
join operations 3–14
ORDER BY option 3–9
privilege operations 4–13
syntax 3–3
WHERE option 3–4
WITH EXPORT option 3–10

Selection criteria
explicit compared to implicit 3–22
explicit joins 3–19
implicit join 3–18

Self joins
explicit 3–22
implicit 3–18

SET option, UPDATE statement 3–26, A–42

Set subqueries 3–11
search conditions 3–11

Singleton SELECT
syntax 3–37

Singleton subqueries 3–10

SMALLINT data type 4–6

Sorting
in ascending order 3–9
in descending order 3–9
query results 3–9

SQL ANSI standards 1–1

SQL character set 2–2

SQL schema, user names 4–16

SQL statements
ALTER TABLE 4–5, A–2
CLOSE 3–36, A–5
CREATE INDEX 4–12, A–6
CREATE SCHEMA 4–16, A–8
CREATE TABLE 4–3, A–10
CREATE VIEW 4–8, A–13
DECLARE CURSOR 3–30, A–15
DELETE 3–26, 3–36
DELETE FROM A–17
DROP INDEX 4–13, A–18
DROP TABLE 4–6, A–19
DROP VIEW 4–10, A–20
FETCH 3–33, A–21
GRANT 4–14, A–23
INSERT INTO 3–24, A–26
OPEN 3–33, A–28
REVOKE 4–15, A–29
SELECT 3–2, A–32
SELECT INTO 3–37
UNION 3–23, A–39
UPDATE 3–26, 3–35, A–42

Standards, ANSI 1–1

STREAM option
SELECT statement A–37

Subqueries 3–10
correlated 3–12
in SELECT statement 3–10
search condition values 3–12
set 3–11
singleton 3–10

Subqueries, nested in WHERE clauses, of SELECT statement 3–10

Subscripts, array field with Progress/SQL 1–3

SUM, aggregate function 3–13

Symbols, operators 2–4

Syntax notation xi

Syntax, Progress/SQL statements 2–2

T

Table names
as identifiers
See Identifiers.
qualifiers 2–3
schema B–1

Tables
altering 4–5
and SQL statements 4–2
creating 4–3
definition statements 4–2
deleting 4–6
owners 4–2
privileges 4–2
selecting from multiple,, see also Join operation.
virtual
views 4–8

TO option, GRANT statement A–24

Transaction processing 3–38

Triggers
schema 3–40
with Progress/SQL 3–40

Typographical conventions x

U

UNION statement 3–23, A–39
DECLARE CURSOR statement A–15
SELECT statement A–39

Unique indexes 4–4

UNIQUE option
CREATE INDEX statement A–6
CREATE TABLE statement 4–3, A–11
creating an index 4–4

Unknown values
See Null values

UPDATE option
GRANT statement A–23

REVOKE statement A–29

UPDATE statement A–42
positioned form 3–35
privilege operations 4–13
searched form 3–26
SET option 3–26
WHERE option 3–26

USERID function 3–41

V

VALUES option, INSERT INTO statement A–26

Variables, host language, as identifiers 2–2

View names
as identifiers,, see also Identifiers.
qualifiers 2–3

_View schema table B–4

_View-Col schema table B–4

_View-Ref schema table B–5

Views 4–8
CREATE VIEW statement 4–8
creating 4–8
deleting 4–10
displaying 4–8
DROP VIEW statement 4–10
explicit joins A–35
initial privileges for owners 4–8
updating 4–8, 4–11
updating WITH CHECK OPTION 4–9

W

WHERE CURRENT OF option
DELETE FROM statement A–17

WHERE CURRENT OF, UPDATE statement A–42

WHERE option
DELETE FROM statement A–17
SELECT statement 3–4, A–36
UPDATE statement 3–26, A–42

Wildcard characters 2–4

WITH CHECK OPTION option, CREATE VIEW statement 4–9, A–13

WITH EXPORT option, SELECT statement 3–10

WITH GRANT OPTION option, GRANT statement 4–14, A–24

Progress/SQL
with non-Progress databases 1–3

WITH phrase, extension to Progress/SQL 1–3

WRITE trigger 3–40