

Progress DataServer for Microsoft SQL Server Guide

© 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: 4610
Item Number: 81069;9.1C

Contents

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

1. Introduction **1–1**
1.1 DataServer Components 1–3
1.1.1 DataServer for SQL Server™ Logic 1–6
1.1.2 The Schema Holder 1–7
1.1.3 Security 1–8
1.2 DataServer Utilities 1–9
1.3 DataServer Demonstration Databases 1–10
1.4 DataServer Configurations 1–10
1.4.1 The Local DataServer Configuration 1–11

1.4.2 The Remote DataServer Configuration 1–12
1.4.3 Configuring Distributed DataServer Applications Using ProBroker 1–14
1.4.4 Configuring Distributed DataServer Applications Using the Progress Explorer 1–15
1.5 Software Requirements 1–16
1.6 Guidelines for Using the DataServer 1–17
1.6.1 Using the DataServer for SQL Server™ for the First Time 1–18
1.6.2 Migrating a Progress Database to an MSS Data Source 1–18
1.6.3 Documentation Resources 1–19

2. Programming Considerations 2–1
2.1 Database Design Issues 2–3
2.1.1 Progress Database Objects and MSS Database Objects 2–3
2.1.2 Naming Conventions 2–4
2.1.3 MSS Data Source Limits 2–6
2.1.4 Code Pages 2–6
2.1.5 Indexes 2–9
2.1.6 Case Sensitivity 2–10
2.1.7 MSS Data Source Views 2–11
2.1.8 Triggers 2–12
2.1.9 Progress Sequences 2–12
2.2 Data Types 2–13
2.2.1 User-defined Data Types 2–15
2.2.2 Arrays 2–16
2.2.3 Unknown Values 2–16
2.2.4 Zero-length Character Strings 2–16
2.3 Record Creation 2–17
2.4 Data Source Record Locking 2–19
2.4.1 Share Locks 2–20
2.4.2 Exclusive Locks 2–21
2.4.3 NO–LOCK 2–21
2.4.4 Locking Examples 2–21
2.4.5 The NO–WAIT Option 2–23
2.5 Transactions 2–24
2.6 Error Handling 2–24
2.7 Cursors 2–27
2.7.1 Forward and Backward Scrolling 2–27
2.8 Progress 4GL Issues 2–28
2.8.1 ROWID Function 2–28
2.8.2 RECID Function 2–29
2.8.3 DEFINE BROWSE Statement 2–30
2.8.4 Field Lists 2–30
2.8.5 FIND Statements 2–31

2.8.6 Compiling Progress Procedures 2–32
2.8.7 FOR Statements 2–33
2.8.8 Unsupported 4GL Statements 2–34
2.9 Stored Procedures 2–35
2.9.1 Defining Native Stored Procedures to Progress 2–35
2.9.2 Running a Stored Procedure 2–38
2.9.3 Retrieving Return Codes 2–40
2.9.4 Retrieving Output Parameter Values 2–40
2.9.5 Retrieving Data Source Results 2–41
2.9.6 Defining a View to Use As a Buffer 2–42
2.9.7 Sending SQL Statements Directly to the Data Source 2–46
2.9.8 Handling Errors from Stored Procedures 2–47
2.10 Enhancing DataServer Performance 2–47
2.10.1 Progress Query Types 2–48
2.10.2 Query Tuning 2–48
2.10.3 Caching Records 2–52
2.10.4 Join by SQLDB 2–53
2.10.5 Writing Queries for Performance 2–56
2.10.6 Modifying the Run-time Schema Check 2–57

3. Configuring the DataServer 3–1
3.1 DataServer Components 3–2
3.2 Configuring the ODBC Driver and Registering the Data Source 3–4
3.2.1 Configuring the ODBC Driver 3–4
3.2.2 Registering Your Data Source 3–4
3.3 Configuring a Local DataServer 3–5
3.4 Configuring the Remote DataServer 3–5
3.4.1 Configuring with the Progress Explorer 3–6
3.4.2 Configuring with ProControl 3–8
3.4.3 Configuring from the Command Line 3–9
3.5 Creating a Schema Holder 3–16
3.5.1 Establishing Permissions 3–16
3.5.2 Preparing to Create the Schema Holder 3–17
3.5.3 Creating an Empty Progress Database 3–18
3.5.4 Creating a Schema Holder 3–19
3.6 Maintaining a Schema Holder 3–23
3.6.1 Updating a Deployed Schema Holder 3–23

4. Connecting the DataServer 4–1
4.1 Starting a Local DataServer 4–2
4.2 Starting a Remote DataServer 4–3
4.2.1 Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client 4–3
4.2.2 Starting and Stopping a Broker Process Using MSSMAN . . . 4–5

4.2.3 Starting and Stopping a Broker Process with ProControl 4–6
4.2.4 Starting and Stopping a Broker Process from the Command Line 4–6
4.2.5 Starting the NT or Windows Client Process 4–7
4.2.6 Starting the UNIX Client Process 4–7
4.3 Connection Guidelines 4–8
4.3.1 Connections and Security on NT 4–8
4.4 Connecting a Schema Holder 4–9
4.4.1 Connecting a Schema Holder at Startup. 4–10
4.4.2 Unsupported Connection Parameters. 4–15
4.4.3 Special Connection Issues 4–16
4.4.4 Query Tuning with Connection and Startup Parameters. 4–17
4.4.5 Analyzing Performance 4–20
4.4.6 Local Schema Caching 4–21
4.5 Connection Failures and Progress Responses 4–22
4.5.1 Connection Troubleshooting 4–23
4.5.2 Managing Connections to a SQL Server™ Database. 4–24
4.5.3 Accessing the DataServer Log 4–25

5. The DataServer Tutorial 5–1
5.1 Demonstration Databases for MSS DataServers 5–2
5.2 Preparing to Use the DataServer Utilities 5–2
5.3 Updating a Schema Holder 5–6
5.4 Verifying a Schema Holder 5–9
5.5 Changing Connection Information in a Schema Holder 5–14
5.6 Modifying a Schema Holder 5–15
5.6.1 Modifying Table-level Information 5–15
5.6.2 Modifying Field-level Information 5–16
5.6.3 Defining the ROWID 5–18
5.7 Changing a Code Page in a Schema Holder 5–20
5.8 Deleting a Schema Holder 5–21
5.9 Adding Extended 4GL Support 5–22
5.9.1 Modifying Tables to Support Arrays 5–22
5.9.2 Modifying Tables to Support the ROWID Function 5–22
5.10 Migrating a Progress Database to SQL Server™ 5–23
5.10.1 Preparing a Data Source for the Utility 5–24
5.10.2 Running the Progress-to-MSS Utility 5–24
5.11 Adjust Schema Utility 5–29

6. Troubleshooting 6–1
6.1 Tuning Your Environment with the –Dsrv Startup Parameter 6–2
6.1.1 ODBC Options 6–3
6.1.2 DataServer Options 6–4
6.1.3 Using SQL Server™ and DataServer Options 6–5
6.2 ODBC Driver Problems on Windows Platforms 6–7

A. Migrating from ODBC DataServer A–1
A.1 Upgrading to SQL Server 7™ or Later A–2
A.2 Creating a Schema Holder A–2
A.3 Modifying Your Application A–2

B. Stored Procedure Reference B–1
CLOSE STORED–PROCEDURE Statement B–2
PROC–HANDLE Function B–4
PROC–STATUS Function B–5
RUN STORED–PROCEDURE Statement B–6

C. Building DataServer Executables C–1
C.1 Overview C–2
C.2 PROBUILD and the DataServer C–3
C.2.1 PROBUILD Requirements C–3
C.3 Building Executables on the UNIX Client C–4
C.4 Setting Environment Variables C–5
C.4.1 Linking DataServer Executables C–6

D. Server Related Command Line Utilities and Startup Parameters D–1
D.1 Progress Explorer Command Line Utilities for the DataServer D–2
MSSCONFIG Utility D–2
MSSMAN Utility D–5
NSCONFIG Utility D–7
NSMAN Utility D–10
D.2 Non-Progress Explorer Command-line Utilities for the DataServer D–12
PROBRKR Command D–12
PROSHUT Command D–14
D.3 DataServer Startup Parameters D–17

Index Index–1

Figures

Figure 1–1:	Architecture for DataServer for SQL Server™	1–5
Figure 1–2:	DataServer Logic	1–6
Figure 1–3:	The Schema-loading Process	1–7
Figure 1–4:	The Local DataServer for SQL Server™	1–11
Figure 1–5:	The Remote DataServer for SQL Server™	1–12
Figure 1–6:	The Distributed DataServer for SQL Server™ Using ProBroker	1–14
Figure 2–1:	DataServer Processes and Code Pages	2–6

Tables

Table 1–1: DataServer Architecture Components 1–4
Table 1–2: Supported Configurations 1–13
Table 1–3: How to Use This Manual 1–19
Table 1–4: DataServer-related Topics 1–19
Table 2–1: ProgressDatabase Objects and MSS Database Objects 2–3
Table 2–2: Database Naming Restrictions 2–4
Table 2–3: SQL Server™ Data Type Equivalents 2–13
Table 2–4: Progress and Data Source Locking 2–19
Table 2–5: N Values in the –Dsrv Parameter 2–20
Table 2–6: 4GL Differences Between Progress Databases and MSS Data Sources 2–34
Table 2–7: Argument Data Types for Stored Procedures 2–36
Table 2–8: Returning Values from Stored Procedures 2–37
Table 2–9: Query-tuning Options 2–50
Table 2–10: Controlling Join by SQLDB Behavior 2–55
Table 3–1: Installing the DataServer Components 3–2
Table 3–2: Environment Variables for the Remote DataServer 3–10
Table 3–3: DataServer for SQL Server™ Sections of the ubroker.properties File 3–12
Table 3–4: SQL Server™ and Progress Code Pages 3–20
Table 4–1: DataServer Connection Parameters 4–11
Table 4–2: Connection Query-tuning Options 4–18
Table 4–3: Diagnostic Options 4–20
Table 4–4: Failure Responses 4–22
Table 5–1: Progress-to-MSS Utility 5–3
Table 5–2: DataServer Utilities 5–5
Table 5–3: Verify Utility Report 5–9
Table 5–4: Progress-to-MSS Utility 5–26
Table 5–5: Progress-to-MSS Utility Batch Parameters 5–28
Table 6–1: ODBC Options 6–3
Table 6–2: DataServer Options 6–4
Table C–1: Building DataServer Processes C–2
Table D–1: DataServer Parameters for UNIX and Windows D–17

Preface

Purpose

This manual explains how to use the Progress DataServer for Microsoft® SQL Server™. It provides startup instructions and a brief tutorial that introduces the utilities that support the DataServer. Additionally, it discusses database design and programming issues to consider when creating applications that access the Progress and SQL Server™ database management systems.

Audience

This book is intended for programmers who want to develop Progress applications that run with SQL Server™ databases. It assumes a fundamental knowledge of Progress and SQL Server™.

Organization of This Manual

Chapter 1, "Introduction"

Describes the DataServer architecture (components and configurations) and software requirements, and provides guidelines for using the DataServer.

Chapter 2, "Programming Considerations"

Discusses the differences between the Progress database and SQL Server™ and the DataServer strategies for resolving these differences. Includes database design issues, application issues, Progress 4GL issues, stored procedures, and DataServer performance enhancement.

Chapter 3, “Configuring the DataServer”

Presents instructions for configuring the DataServer and for creating, maintaining, and deploying a schema holder.

Chapter 4, “Connecting the DataServer”

Presents various methods for starting and connecting the DataServer and describes connecting a schema holder. In addition, it provides connection guidelines and lists connection failures and Progress responses.

Chapter 5, “The DataServer Tutorial”

Provides an overview of the demonstration databases and the opportunity to work with the DataServer utilities for maintaining a schema holder. In addition, it describes the Progress-to-MSS migration utility.

Chapter 6, “Troubleshooting”

Describes common problems and how to work around them, including tuning your environment and resolving ODBC driver problems.

Appendix A, “Migrating from ODBC DataServer”

Provides information on how to migrate from the DataServer for ODBC to the DataServer for SQL Server™.

Appendix B, “Stored Procedure Reference”

Describes the Progress 4GL statements and functions that support running stored procedures.

Appendix C, “Building DataServer Executables”

Describes how to build DataServer executables.

Appendix D, “Server Related Command Line Utilities and Startup Parameters”

Describes the utilities you use to configure, manage, start, and stop the DataServer host and client.

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. 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**).

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, such as:

- *Progress/400 Product Guide*
- *Progress DataServer for ODBC Guide*
- *Progress DataServer for ORACLE 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.

Progress SQL-89 Guide and Reference

A user guide and reference for programmers who use interactive Progress/SQL-89. It includes information on all supported SQL-89 statements, SQL-89 Data Manipulation Language components, SQL-89 Data Definition Language components, and supported Progress functions.

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 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

Introduction

The Progress DataServer for Microsoft® SQL Server™ allows you to develop and deploy Progress 4GL or WebSpeed applications that access a SQL Server™ database. The DataServer for SQL Server™ supports Microsoft SQL Server™ Versions 7 and later. The Progress DataServer for ODBC supports earlier versions of SQL Server™. In this manual, any reference to the term MSS or data source refers specifically to the SQL Server™ database. The DataServer itself is fully ODBC-2.0 compliant. It also complies partially with the ODBC 3.0 standard by supporting extensions for running stored procedures.

The DataServer for SQL Server™ allows you to access your MSS data source with the Progress 4GL and develop applications within the Progress Application Development Environment (ADE). The ADE is a set of tools that helps you to maintain data sources and develop applications with graphical user interfaces. In addition to providing you with the ADE tools, the DataServer allows you to implement the Progress database features and Progress 4GL extensions in applications that run with the data source. Some of these tools and features are:

- **AppBuilder or WebSpeed applications**. Use the AppBuilder to design and generate code for your graphical user interfaces. For example, you can use the AppBuilder to create a browse widget for viewing data stored in your foreign data source.

- **Progress event-driven programming model**. Use this programming model to develop applications that respond to choices your end users make.

- **Data Dictionary**. Use the Data Dictionary to modify database schema; create indexes; and define database triggers, validation expressions, and help messages.
- **Database triggers**. Use a trigger to fire a block of Progress 4GL code whenever a specific data source event occurs; for example, when creating or deleting a record or assigning a value to a field.

Note that using the DataServer to access an MSS data source does not provide you with access to all Progress 4GL, WebSpeed, and database features. For details, see Chapter 2, "Programming Considerations."

Two versions of the DataServer are available:

- The Enterprise DataServer for SQL Server™ is a fully featured DataServer with a wide range of deployment flexibility and scalability. Its features include the following:
  - It is available for use by multiple users.
  - It is available on NT.
  - It can be configured on remote servers.
  - The ODBC driver for the data source is bundled with the Enterprise version.

NOTE: To install and use an ODBC driver successfully, your system must meet the driver system requirements for connecting to data sources. These system requirements are documented in the *DataDirect Connect Reference* available on the Progress product CD in Adobe Acrobat PDF format. You can view and print this book using Acrobat Reader, version 3.0 or higher, licensed from Adobe Systems. You can download the Acrobat Reader software from the Adobe Web site at no charge. The URL for the Acrobat Reader is: `http://www.adobe.com/products/acrobat/readstep.html`.

- The Personal DataServer for SQL Server™ provides the same features as the Enterprise version except for the following restrictions:
  - It is available for use by a single user only.
  - It is available for purchase only on Windows and NT; however, the Personal DataServer for MSS functionality is supplied with all Enterprise DataServer for MSS products.
  - The ODBC driver is not bundled with the Personal version. The driver must be purchased separately from the vendor.

This manual documents both versions and uses the term DataServer for SQL Server™ to refer to features and functionality that are common to both. Features and functions that are specific to the Personal DataServer or the Enterprise DataServer are noted as such.

This chapter describes the DataServer for SQL Server™ and discusses how Progress and WebSpeed applications work with it to access data sources through the ODBC standard. It discusses the following topics:

- DataServer components, including DataServer logic, the schema holder, security, utilities, the demonstration database, and configurations
- Software requirements
- Guidelines for using the DataServer
- Related DataServer information in other Progress and WebSpeed manuals

Subsequent chapters provide additional information about using the DataServer. If you are using the DataServer with WebSpeed and with applications written in SpeedScript, all information regarding Progress 4GL applies to your application.

NOTE: Do not attempt to use the DataServer without first reading the programming guidelines in Chapter 2. Also, be sure to follow the step-by-step instructions in Chapter 3, “Configuring the DataServer,” and Chapter 4, “Connecting the DataServer,” for installing, configuring, and connecting to a data source.

1.1 DataServer Components

The DataServer is a set of software components that allows Progress and WebSpeed applications to access data in the data source. The DataServer enables access by translating standard Progress and WebSpeed code into calls appropriate to the data source. The application can thus manipulate the data as though it were Progress-based data.

The DataServer consists of the components described in Table 1–1.

Table 1–1: DataServer Architecture Components

Component	Description
Enterprise DataServer for SQL Server™ Personal DataServer for SQL Server™	A Progress software module that allows you to use Progress or WebSpeed.
ODBC Driver Manager	A software module that manages the connection between an application and ODBC drivers. It is implemented on Windows by default as a Microsoft-supplied dynamic link library (DLL) module.
Schema holder	A repository for foreign data definitions.
ODBC driver	A software module that implements the ODBC Application Programming Interface (API) specification. The driver interfaces between the ODBC Driver Manager and the data source.
ODBC data source	A name that identifies a specific set of data and how to obtain it. You must register a SQL Server™ database as an ODBC data source.
ODBC Administrator (Windows platforms only)	A Microsoft program for registering and configuring data sources. You *register* a data source by providing information that the ODBC driver needs through the ODBC Administrator. Note that this registration method is used only in Windows platforms.
DataServer MSS utilities	A set of utilities that allows you to perform certain tasks related to the DataServer. You access them from the Progress Data Administration tool.

Figure 1–1 illustrates how the DataServer components are organized for both the Enterprise and the Personal DataServer for SQL Server™.

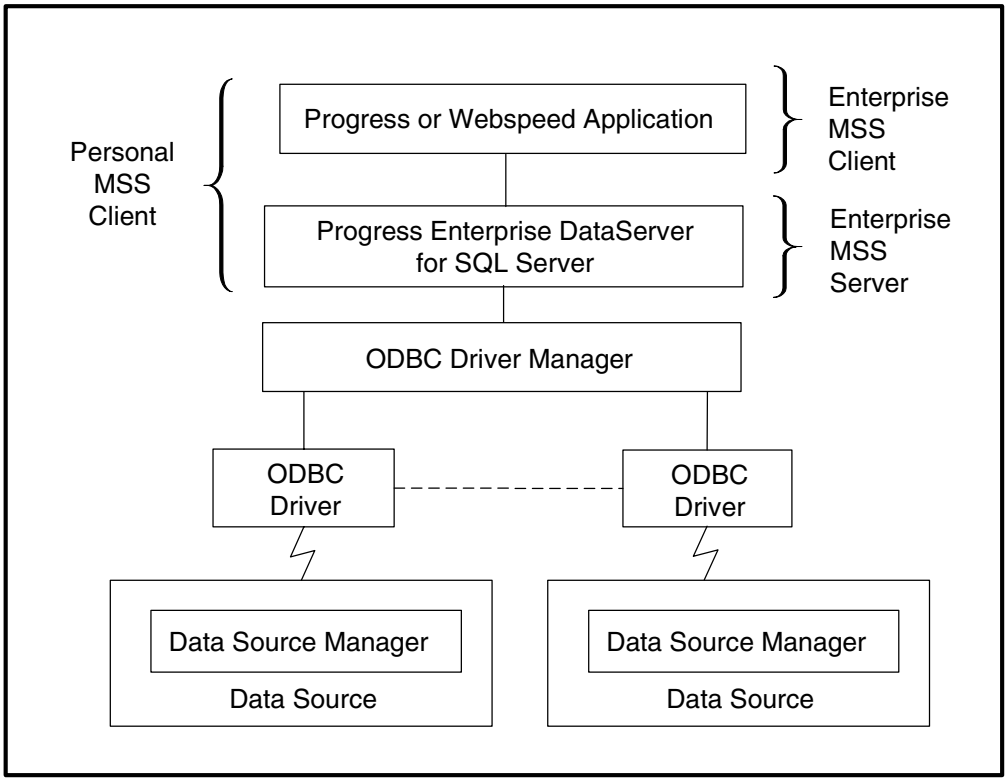


Figure 1–1: Architecture for DataServer for SQL Server™

As shown in Figure 1–1, the components for the Personal DataServer for SQL Server™ are organized similarly to the Enterprise edition except that the DataServer is embedded in the client rather than executing separately.

In either case, a Progress or WebSpeed application uses the DataServer to work with the ODBC Driver Manager, which accesses the desired ODBC driver. The driver then accesses the SQL Server™ database.

Some of the DataServer components are linked with elements of the standard Progress architecture, as described in the “DataServer Configurations” section.

1.1.1 DataServer for SQL Server™ Logic

The flow chart in Figure 1–2 illustrates the DataServer's internal logic.

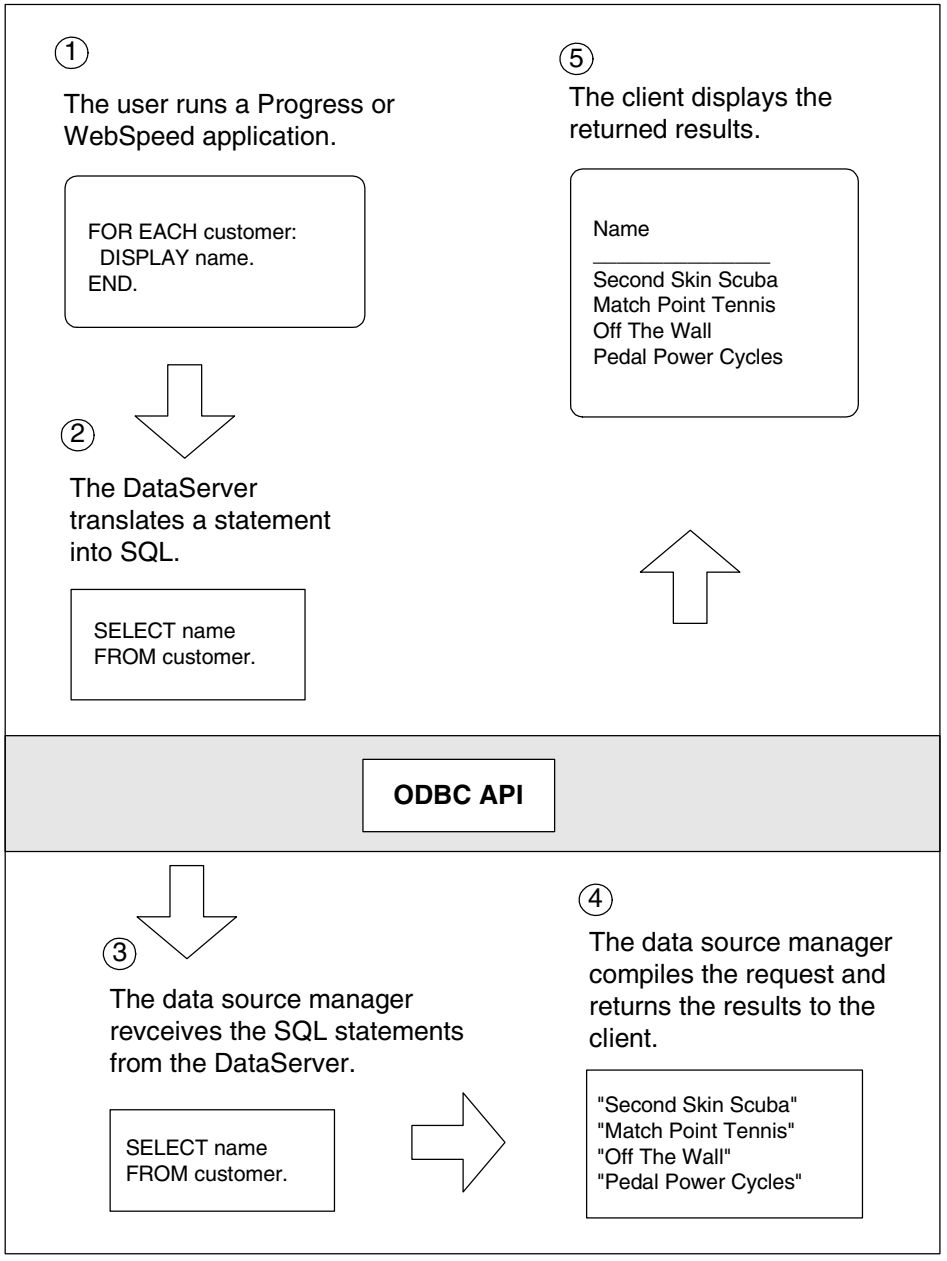


Figure 1–2: DataServer Logic

As shown in Figure 1–2, when you execute an application that accesses a SQL Server™ database, the Compiler translates Progress 4GL or SpeedScript statements into their SQL equivalents. The DataServer then issues the SQL statements to the ODBC driver through the ODBC API. The driver, which provides the software mechanisms for accessing and controlling the data source, processes the SQL statements, transfers them to the data source manager, and returns the results to Progress through the ODBC API.

In order to facilitate the flow of statements between client and server, the DataServer places Progress equivalents for data definitions from a supported data source into a schema holder (a repository for data definitions for one or more MSS data sources). When the Progress client or WebSpeed Agent executes statements and retrieves information from the data source, it relies on data definitions in the schema holder.

1.1.2 The Schema Holder

As noted in the previous section, a *schema holder* contains information about the MSS data sources. This information, called the *schema* of the data source, is a description of its structure, its tables, the fields within its tables, and its indexes.

Progress accesses the schema holder only when it compiles procedures and at the beginning of a run-time session for *schema caching* (the loading of data definitions into memory). Typically, the schema holder is not accessed during a run-time session after the initial schema caching.

Before a Progress client can access data in the MSS data source, you must create a schema holder and load the required data source definitions into the schema holder. You can then use the Progress Data Dictionary to add Progress database features such as validation expressions and messages.

Figure 1–3 illustrates the schema-loading process.

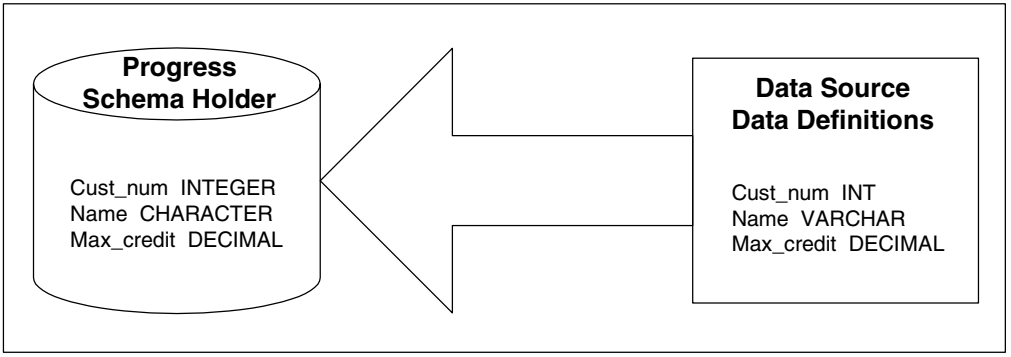


Figure 1–3: The Schema-loading Process

If you plan to use the DataServer to send SQL statements directly to the data source using only the RUN STORED–PROCEDURE syntax and you do not expect returned data, you need not load data definitions into the schema holder; however, you must do the following:

- Load the stored procedure into the schema holder.
- Connect to an empty data source.

For information on using RUN STORED–PROCEDURE, see the "Stored Procedures" section in Chapter 2, "Programming Considerations."

1.1.3 Security

Using the DataServer for SQL Server™ involves following the security guidelines required by both the Progress database and the MSS data source. By default, Progress allows unrestricted access to data sources, so at minimum, you should follow the guidelines that the data source requires for your applications.

Progress Security

The Progress database management system has no minimum security requirements. You can, however, impose security features on any Progress database or schema holder. There are four levels of application security that you can impose:

- Database-connection security
- Schema security
- Compile-time security
- Run-time security

For more information on Progress security, see the data security chapter in the *Progress Programming Handbook*.

SQL Server™ Database Security

As noted previously, you should follow the security guidelines that your MSS data source has established for your applications. The SQL Server™ database might require that all users supply a valid login name and password to access it. Data source access security typically has four levels—system administrator, database owner, database object owner, and public:

- System administrators can grant or revoke permissions to other users to create or own a wide type of objects; for example, databases.
- Database owners can grant other users permission to access or modify a database or its objects.
- A user can be the owner of objects in a database owned by another user.
- Public database objects can be accessed by any users without restriction.
- The DataServer for SQL Server™ will allow you to use either the Windows NT or SQL Server™ authentication model for your connection to the SQL Server™ data source. For more information on database security, see Microsoft's SQL Server™ documentation.

NOTE: There are specific security requirements for accessing data with the DataServer that relate to creating a schema holder. For details, see the "Creating a Schema Holder" section in Chapter 3, "Configuring the DataServer."

1.2 DataServer Utilities

Progress provides a set of utilities that allows you to perform certain tasks related to the DataServer. There are utilities for:

- Creating a schema holder
- Updating a schema holder
- Verifying that the definitions in the schema holder match the current data source information
- Editing connection information for a schema holder
- Changing the code page for a schema holder

- Deleting the schema
- Migrating an existing Progress database to a SQL Server™ database.

In addition, you can use the Progress Data Dictionary to modify data definitions at the field level; for example, to change display formats, add help messages, or add validation expressions. You can also use the Progress Data Administration tool to manipulate data definition files for the schema holder.

1.3 DataServer Demonstration Databases

As part of the DataServer product, Progress Software Corporation provides you with the ability to create demonstration data that you can use to experiment with the DataServer. Note that you do not install the demonstration databases as part of the Progress installation media. Instead, you create them by running the Progress-to-MSS utility, which migrates the Progress Sports database—objects and data—to your MSS data source. For instructions, see the "Demonstration Databases for MSS DataServers" section in Chapter 5, "The DataServer Tutorial."

1.4 DataServer Configurations

The Enterprise DataServer for SQL Server™ can run in a variety of configurations. Depending on its configuration, it is either a single Progress executable or a set of Progress executables that you can distribute across operating systems. There are two general types of configurations—local and remote—with variations on each type:

- **Local DataServer** — All of the DataServer software components, the schema holder, the ODBC software, and SQL Server™ client software run on one machine.
- **Remote DataServer** — The DataServer software components run on different machines. Typically, the Progress client component runs on one machine, while the Progress server component runs on another machine called the host machine. Note that the ODBC software and client software required run on the same machine as the DataServer software modules. In the remote DataServer configuration, this is the host machine. The host machine and the machine on which the client runs communicate through Progress networking (TCP/IP). A remote configuration exists within the framework of a distributed application, whereby DataServers are spawned on the host machine using the Progress ProBroker executable or broker in the Progress Explorer administration framework.

The Personal DataServer for SQL Server™ runs only in a local DataServer environment. It cannot be run in the remote (or distributed) DataServer configuration.

1.4.1 The Local DataServer Configuration

Figure 1–4 shows the local DataServer configuration. You can configure both the Personal and Enterprise DataServers for SQL Server™ in this way; however, the Personal DataServer for SQL Server™ is restricted to single-user Windows platforms only.

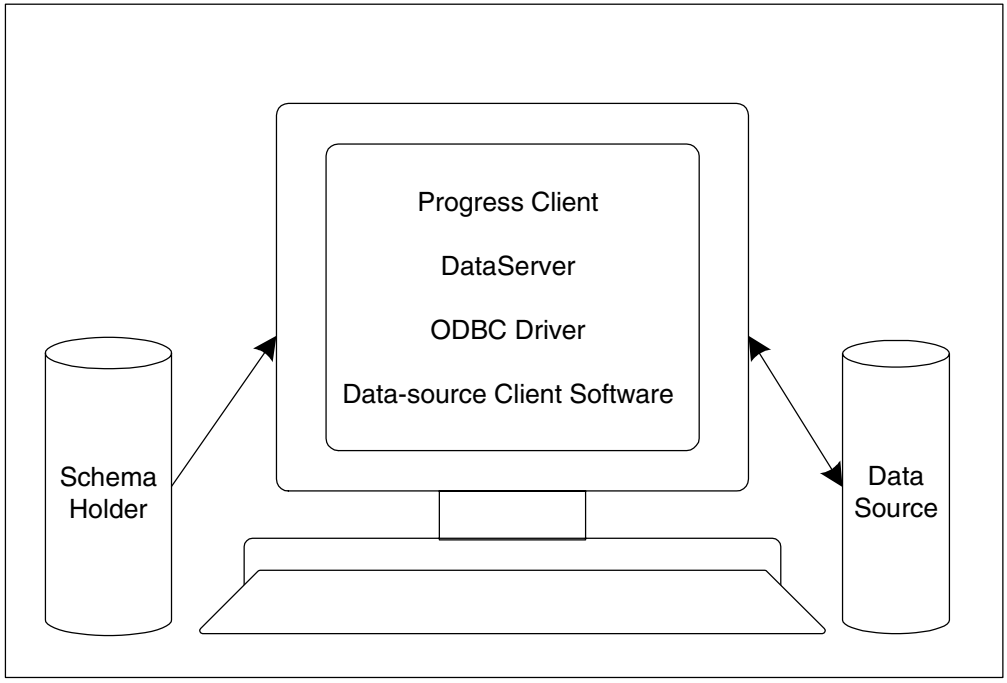


Figure 1–4: The Local DataServer for SQL Server™

In the local DataServer configuration, all of the DataServer software components, the schema holder, and data source client software run on the same machine. Depending on the ODBC and data source client software implementation, the actual target database may be local or remote to the machine where the local Progress DataServer for SQL Server™ executes. With the SQL Server™ client software-supplied modules that manage networked database communications, a remote database configuration is possible. The DataServer cares only about the local MSS data source definition. Remote database access is transparent to the DataServer.

1.4.2 The Remote DataServer Configuration

Figure 1–5 shows the remote DataServer configuration.

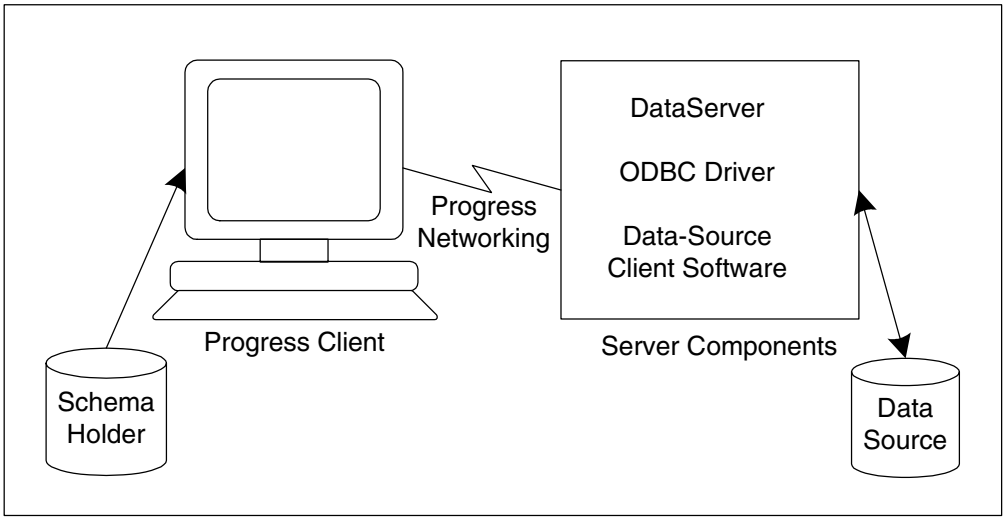


Figure 1–5: The Remote DataServer for SQL Server™

In the remote configuration, the Progress client and schema holder run on a client machine and the Progress server components and any SQL Server™ client software run on a remote server machine. The actual target database may be local or remote to the machine where the Progress DataServer for SQL Server™ executes. With the SQL Server™ client software supplied modules that manage networked database communications, a remote database configuration is possible. The DataServer cares only about the data source definition on its machine. Remote database access is transparent to the DataServer.

In remote DataServer configurations, Progress handles the communication between the client and the server. The client and server processes that make up the DataServer adapt to a variety of network configurations.

Table 1–2 lists the supported configurations for setting up a remote DataServer. It contains possible client-server combinations and networking options.

Table 1–2: Supported Configurations

Client	Networking	Server
Progress on Windows and NT	None	The server, client, ODBC client, and data source are on the same machine. You can configure both the Personal and Enterprise DataServers for SQL Server™ in this way. (The Personal DataServer for SQL Server™ is restricted to single-user Windows platforms only.)
Progress on Windows, NT, and UNIX	Progress	The client can reside on Windows, NT, or Unix. The server, ODBC driver, and data source are on a NT server machine. This configuration is possible only with the Enterprise DataServer for SQL Server™.

1.4.3 Configuring Distributed DataServer Applications Using ProBroker

Figure 1–6 shows an example of a distributed DataServer configuration that uses ProBroker. (This configuration is available only for the Enterprise DataServer for SQL Server™.) The figure illustrates how to combine clients and servers that run on different platforms.

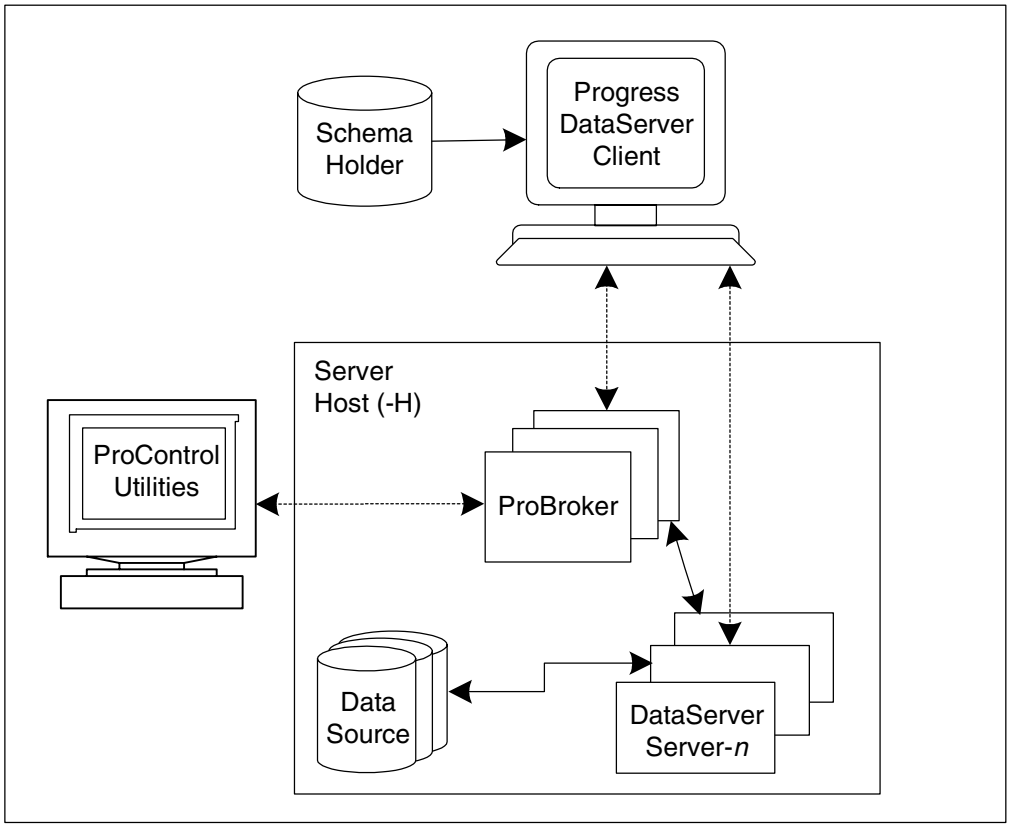


Figure 1–6: The Distributed DataServer for SQL Server™ Using ProBroker

In Figure 1–6, the Progress client runs on a client machine (either Windows or UNIX) and can access a remote NT server. Multiple brokers may be running simultaneously on one server machine. The client connects initially to the ProBroker executable by locating the host and port of the executing broker. It is then automatically reconnected to an Enterprise DataServer established for it by the broker. Each executing broker may spawn a multitude of DataServer processes. A spawned DataServer process uses the ODBC driver to locate and connect to the data source. The actual target database might be either local or remote to the host machine. Note that in this example, the schema holder also runs on the Windows client; however, you can optionally locate it on any host that is accessible on your network. Each spawned DataServer

can service database requests for the same database or for a different database than those of other spawned servers.

In remote DataServer configurations, Progress handles the communication between the client and the server. The client and server processes that make up the DataServer adapt to a variety of network configurations.

1.4.4 Configuring Distributed DataServer Applications Using the Progress Explorer

The Progress Explorer tool is a graphical user interface that provides an easy way for you to manage Progress servers. Progress Explorer is one of the primary components of the Progress administration framework.

The Progress Explorer tool runs as a Windows client and works with another administration component, the Progress AdminServer, to provide a client/server framework for managing the following Progress servers:

- AppServer
- Database
- DataServer for ODBC
- DataServer for ORACLE
- DataServer for SQL Server™
- NameServer
- WebSpeed Messenger
- WebSpeed Server
- SonicMQ

Using the Progress Explorer tool, you can:

- Create new instances of Progress servers and configure their property settings
- Modify property settings of existing Progress server instances
- Start and stop Progress servers
- Monitor the status of Progress servers

For more information about working with the Progress Explorer tool, see the Progress Explorer online help.

The Progress client runs on a client machine (either Windows or UNIX) and accesses a remote server on any supported DataServer for SQL Server™ platform. Multiple NameServers and/or brokers may be running simultaneously on one server machine. The DataServer client connects to a broker for the Progress Explorer either directly or through a controlling NameServer. (See the important note that follows.) It is then automatically reconnected to an Enterprise DataServer established for it by the broker. Each executing broker may spawn a multitude of DataServer processes. A spawned DataServer process uses the ODBC driver to locate and connect to the data source. The actual target database might be either local or remote to the host machine. Note that in this example, the schema holder also runs on the Windows client; however, you can optionally locate it on any host that is accessible on your network. Each spawned DataServer can service database requests for the same database or for a different database than those of other spawned servers.

In remote DataServer configurations, Progress handles the communication between the client and the server. The Progress Explorer supports only the TCP/IP network configuration.

NOTE: In a run-time configuration, **all** DataServer clients should attach consistently either to a set of NameServers or to a set of brokers. **Do not** run brokers under controlling NameServers for one client while another client simultaneously attaches directly to a broker.

For more information, about configuring and connecting the DataServer, see the following sections:

- "Configuring with the Progress Explorer" in Chapter 3, "Configuring the DataServer"
- "Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client" in Chapter 4, "Connecting the DataServer"

1.5 Software Requirements

This section describes the software requirements for using the DataServer.

NOTE: For the Enterprise DataServer, Progress supplies the ODBC drivers for the supported data sources.

Progress-supplied software:

- Personal DataServer for SQL Server™ or Enterprise DataServer for SQL Server™
- WebSpeed Transaction Server Version 3.0 or later or Progress Version 9.1B or later

Data-source server software:

- Microsoft SQL Server™ Version 7.0 or higher

Data source client software:

- Client connectivity component software from your SQL Server™ installation

  For information on exact client software compatibilities with Microsoft SQL Server™ Version 7 or later, see the Microsoft Web Site at `www.microsoft.com`.

ODBC software:

- For the Personal DataServer one of the following can be purchased and installed separately:

  – MERANT DataDirect ODBC driver Version 3.50 or later (for SQL Server™ Version 7.0)

  – Microsoft ODBC Driver (for SQL Server™ Version 7.0 only)

  For information on exact driver version compatibilities with the Microsoft SQL Server™ client and server data source software, see the Microsoft Web Site at `www.microsoft.com`.

- For the Enterprise DataServer, Progress supplies the ODBC driver, which is automatically installed when you install the DataServer for SQL Server™. See the *DataDirect Connect ODBC Reference* on the Progress CD for additional information.

1.6 Guidelines for Using the DataServer

Progress supports many capabilities not found in other database management systems, such as backward scrolling cursors and the ability to find the previous or last record in a table. The DataServer supports these and other programming and database features to ensure that your applications work with both Progress databases and SQL Server™ databases.

The DataServer allows you to use Progress features as extensions to your data source. Some of the Progress programming and database design techniques that you can implement on your MSS data source using the DataServer are:

- ROWID function
- Arrays

For access to some of these features, you might have to make minor modifications to how your MSS data-source or application is organized. For a discussion of these issues and instructions for modifying your data source, see Chapter 2, "Programming Considerations."

If you create an MSS data-source from an existing Progress database with the Progress-to-MSS migration utility and select the Create RECID Field option, you can use the FIND PREV/LAST statements in addition to taking advantage of Progress-like cursor behavior.

How you use the DataServer depends on whether you plan to access information in a data source through a Progress application or migrate a Progress database to an MSS data source. The following sections summarize what you must do and point you to the information you need in this manual.

1.6.1 Using the DataServer for SQL Server™ for the First Time

You must make these preparations before using the DataServer:

1. Install the DataServer modules on the machines your configuration requires.
2. For the Personal DataServer, purchase and install the ODBC driver software.

   For the Enterprise DataServer, the ODBC driver software is automatically installed when you install the DataServer software on the machine that will execute the server component.
3. Create a local schema holder on the client or server machine, as appropriate. Schema holders cannot be transferred between different host machines.

Chapter 3, "Configuring the DataServer," provides information about where to install DataServer modules and how to create a schema holder.

1.6.2 Migrating a Progress Database to an MSS Data Source

You must follow these steps when setting up and using the DataServer:

1. Install the DataServer components on the machines that your configuration requires.
2. Run the Progress-to-MSS migration utility.

See the "Migrating a Progress Database to SQL Server™" section in Chapter 5, "The DataServer Tutorial," for specific instructions.

1.6.3 Documentation Resources

Table 1–3 suggests paths through this manual that accommodate different approaches to using the DataServer for SQL Server™.

Table 1–3: How to Use This Manual

If You Are . . .	This . . .
New to the DataServer for SQL Server™	Chapter 2, "Programming Considerations" Chapter 3, "Configuring the DataServer" Chapter 4, "Connecting the DataServer" Chapter 5, "The DataServer Tutorial"
Migrating a Progress database to an MSS data source	"Running the Progress-to-MSS Utility" section in Chapter 5, "The DataServer Tutorial" Chapter 2, "Programming Considerations" Chapter 3, "Configuring the DataServer" Chapter 4, "Connecting the DataServer"
Upgrading from an earlier version of SQL Server™	Appendix A, "Migrating from ODBC DataServer" Chapter 3, "Configuring the DataServer"

Table 1–4 lists manuals from the Progress and WebSpeed documentation sets that contain useful information on different aspects of DataServer usage.

Table 1–4: DataServer-related Topics *(1 of 2)*

Topic	Progress Manuals
Starting Progress	*Progress Installation and Configuration Guide Version 9 for Windows* *Progress Installation and Configuration Guide Version 9 for UNIX*
Using the Progress Data Dictionary	*Progress Language Tutorial for Character* *Progress Language Tutorial for Windows*
Defining security for a Progress database	*Progress Database Administration Guide and Reference*

Table 1–4: DataServer-related Topics *(2 of 2)*

Topic	Progress Manuals
Writing applications in the Progress 4GL	*Progress Application Development Environment — Getting Started* *Progress Language Tutorial for Character* *Progress Language Tutorial for Windows*
Connecting your data source with WebSpeed	*WebSpeed Installation and Configuration Guide*
Writing WebSpeed applications	*WebSpeed Developer's Guide*

2

Programming Considerations

An application developed in the Progress Application Development Environment (ADE) can use the DataServer for SQL Server™ to access data from multiple sources transparently. With the DataServer, your applications can access information in both Progress databases and SQL Server™ databases. This chapter discusses the differences between Progress databases and SQL Server™ databases that you must consider when you plan your applications and design your databases, as well as the DataServer strategies for resolving these differences, which your application might need to accommodate. Specifically, it discusses the following topics:

- Database design issues including database objects, naming conventions, limitations, code pages, indexes, views, triggers, and sequences
- Application issues including data types (user defined, arrays, unknown values, zero-length character string), record creation and record locking, transactions, error handling, and cursors
- Progress 4GL issues including the COMPILE statement, field lists, the FIND statements, the ROWID and RECID functions, as well as 4GL statements and functions that the DataServer does not support
- Stored procedures including running native stored procedures from a Progress procedure, retrieving return codes, setting parameter values, defining buffers and receiving database results, and sending SQL directly to the data source
- Enhancing DataServer performance

Follow these guidelines carefully when you develop your application to ensure that it can access Progress databases and SQL Server™ databases transparently.

The material in this chapter is also of interest to users who plan to migrate a Progress database to an MSS data source. However, such a migration raises additional issues that you must consider when designing your application. For details, see the "Running the Progress-to-MSS Utility" section in Chapter 5, "The DataServer Tutorial."

2.1 Database Design Issues

When you create or modify the Progress databases or SQL Server™ databases that your applications access, you must consider general database design issues such as Progress and MSS database objects, naming conventions, code pages, indexes, and data types. The following sections describe how Progress databases and MSS databases differ in such areas and how the DataServer resolves the differences.

2.1.1 Progress Database Objects and MSS Database Objects

The Progress database and the SQL Server™ database share the structural elements common to relational databases, but each system has its own elements. These elements, called *database* or *data source* objects, are components of the Progress database or MSS databases's logical structure. Table 2–1 lists Progress database objects and the MSS database equivalents.

Table 2–1: ProgressDatabase Objects and MSS Database Objects *(1 of 2)*

Progress	SQL Server™
Table/File	Table
Field	Column
Record	Row
Index	Index
Unique index	Primary key
Nonunique index	Foreign key
Sequences	Native equivalent[1]
Validation expression	Integrity constraint
Validation message	No equivalent
Initial value	Default

Table 2–1: ProgressDatabase Objects and MSS Database Objects *(2 of 2)*

Progress	SQL Server™
Schema trigger	Trigger
No equivalent to stored procedures	Stored procedure
View	View

[1] The DataServer supports Progress sequences through native stored procedures and triggers.

2.1.2 Naming Conventions

When planning for maximum transparency across Progress databases and MSS data sources, be sure to consider the restrictions that each places on naming data source objects when you perform a Progress-to-MSS conversion. Table 2–2 describes these restrictions, and the notes that follow it discuss how the DataServer resolves differences between Progress and SQL Server™ naming conventions.

Table 2–2: Database Naming Restrictions *(1 of 2)*

Category	Progress	SQL Server™
Alphanumeric characters	A–Z or a–z 0–9	All alphanumeric characters from the character set that you defined for your SQL Server™.
Special characters	Allows $, &, #, %, –, and _[1]	Allows @, $, #, _, ¥, and £[2]
Initial character	Alphabetic only	Alphabetic only

Table 2–2: Database Naming Restrictions *(2 of 2)*

Category	Progress	SQL Server™
Maximum length of object name	32 characters	24 characters[3]
Keywords	Not allowed[4]	Not allowed[4]
Unique table names	Required[5]	Required[6]

1 Although Progress allows you to use the ampersand (&) in database-object names, the Progress ADE tools use this character internally to denote preprocessor directives.

2 When the DataServer encounters characters not supported by Progress in a table, index, or field name, it converts them to underscores (_) in the schema holder. For example, the data source table name emp@status becomes emp_status in the schema holder. If the unsupported character is at the beginning of an object name, the DataServer substitutes a lowercase z. Also, the DataServer strips @ symbols from stored procedure parameter names.

3 Object names are limited to 32 characters. When you use the Progress-to-MSS utility to perform a database conversion, however, the utility truncates object names to 24 characters in order to accommodate the suffix characters that it appends when it finds nonunique names, reused Progress keywords, array fields, sequences, triggers, and so forth. If you intend to use the Progress-to-MSS utility, be sure to limit object names to 24 characters to accommodate this name resolution during conversion.

4 Progress and SQL Server™ databases contain restrictions against using keywords as database object names. If a data source object name consists of a Progress keyword, the DataServer appends an underscore character (_) to the name. For example, an object named “each” becomes “each_”.

5 SQL Server™ qualifies nonunique table names to make them unique (see your data source documentation for details); however, Progress still regards these qualified names as nonunique. The DataServer resolves nonunique table names for you as follows: when Progress encounters matching table names while creating or updating a schema holder, it names the second and subsequent tables *table-1*, *table-2*, etc. For example, if Progress encounters a table named lee.employee (the employee table owned by the user lee) that is the second instance of the name in the data source, it names the corresponding table in the schema holder employee-1.
NOTE: When resolving naming conventions, the DataServer can append up to three characters to an object name: two in the case of a nonunique name and one in the case of a Progress keyword. Therefore, you should limit your object names to 29 characters so that you do not exceed the Progress 32-character limit for object names in the schema holder.

6 SQL Server™ allows you to qualify an object name by adding the following:
- the database name
- the owner’s name
- (for columns) the table name

2.1.3 MSS Data Source Limits

An MSS data source has upper limits on various components; for example, the number of columns per table, the number of columns per SELECT statement in a view, the maximum size of a data row, and the maximum length of character data types. These restrictions might affect how Progress applications run. See your Microsoft® SQL Server™ documentation for full details.

2.1.4 Code Pages

Flexibility in managing code pages, also called *character sets*, is critical for international applications. When you access an MSS data source through the DataServer, the DataServer retrieves character data as determined by the code page that the data source uses. For Progress databases, the default code page is iso8859–1; however, Progress allows you to designate code pages for the data source and for the client.

Figure 2–1 shows a possible configuration of code pages for the DataServer components and processes.



Figure 2–1: DataServer Processes and Code Pages

In the configuration shown in Figure 2–1, all components use the same code page. (Please note that the Progress iso8859–1 code page is a subset of the iso-1 code page used by the default installation of SQL Server™. For information on setting character sets for your data source, see the administration guide supplied by the vendor.) On the Progress side, if the client and the schema holder use different code pages, a conversion takes place between them.

In order for DataServer applications to manipulate data from an MSS data source accurately, you must specify the correct code page in the schema holder. For Progress applications accessing the DataServer, the schema holder identifies the code page of the character data. The DataServer sends the data source name for the code page to the data source to indicate the character set for the data that the data source returns.

Be sure to set the code page in the schema holder to match a code page that the MSS data source supports. To minimize the number of translations, specify the default code page that the data source uses. If Progress does not support the code page, you can specify instead a compatible code page that is available for your data source. The directory `%DLC%/prolang/convmap` contains conversion tables for all of the code pages that Progress supports. Check to see whether any of them match your code page.

The default code page setting in the schema holder is iso8859–1. You can specify a different code page for the schema holder at the following times:

- When you create the DataServer schema for the MSS data source.
- When you load a new schema with a specified code page into an existing schema holder. In this case, the newly loaded schema's code page overrides the schema holder's original code page.

NOTE: It is possible to change the code page at other times. However, because changing the code page does not affect the data already in the database, writing new data to your database using a different code page can corrupt the data in your database. You cannot use the PROUTIL utility to change the code page used by the DataServer.

Keep in mind that your MSS software configuration might have local requirements for defining the proper language interface between the ODBC drivers and the data source. See your Microsoft® SQL Server™ database documentation for details.

Client Code Page

The Internal Code Page (-cpinternal) startup parameter determines the code page that the Progress client uses when it manipulates data in memory. If the Progress client uses a different code page from the code page set in the schema holder, the DataServer translates between the two code pages, so you must verify that the `convmap.cp` file contains a conversion table for the client and the code page setting in the schema holder. Suppose, for example, that you set the schema holder to code page ibm850 and the client uses code page iso8859–1. The `convmap.cp`

file must include a table that converts from ibm850 to iso8859–1 and from iso8859–1 to ibm850. If `convmap.cp` does not include the appropriate table, you can define your own conversion table.

Progress also allows you to define your own collation tables; however, customized collation tables only take effect after data source collation when you use the DataServer to access an MSS data source. The data source collation tables, not the Progress collation tables, have first priority when you perform comparisons and sorts. After performing comparisons and sorts, the Progress client may sort out records that do not conform to the requirements of your customized collation tables.

For example, if you use the default code page 1252 and the default case insensitive sort ordering with a SQL Server™ data source, collation of the tilde character (~), which is ASCII character 126, sorts before all the alphanumeric characters (0–9, a–z). If you were to select data from a character column "name" with the following:

```
select * from table where name <=CHR(126) (the equivalent of ~)
```

SQL Server™ returns records with columns that begin with a tilde character but no alphanumeric records, since ASCII character 126 is sorted below alphanumeric characters in this SQL Server™ collation.

Normally, the default Progress collation sorts a tilde character above all alphanumeric characters. Therefore, in order for the above example to exhibit Progress-like behavior and return alphanumeric records as well as records beginning with the tilde, the SQL Server™ sort order for this code page would need to be modified accordingly.

Conversely, if you execute the opposite:

```
select * from table where name >=CHR(126)
```

SQL Server™ returns records with columns that began with a tilde character followed by all that begin with alphanumeric characters. However, the default Progress collation, which sorts the tilde higher than all the alphanumeric characters, would omit records beginning with alphanumeric characters and only return records beginning with the tilde character.

To get the full result set returned from SQL Server™ from the Progress client would require modifying the collation table associated with the Progress code page and weighting it to match the SQL Server™ sort order.

For a complete discussion of how Progress handles code-page issues, see the *Progress Internationalization Guide*.

2.1.5 Indexes

You create and maintain all indexes from within the MSS data source, using native tools, rather than with the Progress Data Dictionary. A data source index uses a logical pointer to the physical locations of table rows in order to sequence data access. You can add and drop indexes but you cannot use their names in queries. The data source alone ultimately decides when and how to use indexes; its decisions are not affected by the DataServer.

Using index definitions in the MSS data source, the DataServer builds index information in the schema holder. Progress index definitions for the data source schema serve two purposes:

- They allow you to use the OF option in Progress with the FOR EACH and FIND statements. Using the OF option improves the readability of your code. The OF keyword is equivalent to the SQL WHERE clause. You can use OF only when you have a field of the same name in two tables and the field is an index in at least one of the tables. Therefore, since the cust-num field is common to both the order and customer tables, you could write the following statement:

```
FOR EACH order OF customer:
```

- Progress translates USE–INDEX to SQL ORDER BY for DataServer operations. For example, if you define city-dept as an MSS data source primary key on the city and department fields, it is a unique index in the schema holder. In this case, the following Progress statements are equivalent when accessing the data source:

```
FOR EACH employee USE-INDEX city-dept:
```

```
FOR EACH employee BY city BY department:
```

NOTE: If you do not specify a USE–INDEX or BY clause, your query will return records in an unpredictable order. Your application might not require predictable ordering, but if it does, be sure to include USE–INDEX or BY clause in your query definition.

Dummy Indexes for Sort Order

You can add dummy index definitions to your schema holder independent of the actual index definitions in your MSS data source to provide a sort order. These dummy indexes can improve the readability of your Progress 4GL code. An index in the schema holder need not match an

index in the MSS data source. However, the absence of an index definition in the data source can impact performance when you retrieve data with selection on the dummy indexes.

Unique Indexes

If your MSS data source tables have at least one unique index, they can be used to support operations such as backward and forward scrolling and accurate cursor positioning through the FIND CURRENT, PREV, and LAST statements. If a table does not have a unique index, you can only scroll forward through its data.

If an MSS data source table does not have a unique index, you can designate an index to serve as the unique index for the schema holder. An index that you designate as unique in the schema holder must be unique with respect to the data in the data source, otherwise you receive run-time errors. See Chapter 5, "The DataServer Tutorial," for instructions on using the Progress Data Dictionary to designate unique indexes.

MSS data source views and result sets from stored procedures do not have unique indexes. Just as for tables, you can use the Progress Data Dictionary to create a unique index in the schema holder based on fields in a view or result set so that you can browse data accessed through views or stored procedures.

NOTE: Do not change the designated ROWID key of a record while an application is running. Suppose, for example, that cust-num is a unique key and has been designated the Progress ROWID. If a user changes the value of cust-num for a customer from 1 to 111, other users receive an error message when they try to access the record for customer 1.

2.1.6 Case Sensitivity

By default, a Progress database is case insensitive; however, you can set the attributes of a field to define it as either case sensitive or case insensitive. If you intend to do this, consider the following:

- Schema pull criteria is case sensitive against SQL database qualifiers.
- Pattern-matching literals in data source access statements retrieve case sensitive data.
- Progress considers the user ID and password submitted at connection time to be case sensitive.

If an indexed field is case insensitive, Progress does not distinguish between uppercase and lowercase letters for that index when sorting or matching data. In general, this flexibility in an application makes data entry easier for end users because they can enter lowercase or uppercase versions of an index. However, if you want to enforce an uppercase/lowercase distinction in your applications, set the attribute to case sensitive.

If you are using a case sensitive page, the DataServer can make this feature compatible across Progress and MSS data sources. To support case insensitivity, an extra column must be added to the data source immediately before the indexed column. In most cases, this column is named _S#_*column* (exceptions are noted in the following sections). See the "Adding Extended 4GL Support" section in Chapter 5, "The DataServer Tutorial," for instructions on adding this column automatically with the Progress-to-MSS utility.

NOTE: By default, sort order in SQL Server™ is not case sensitive.

2.1.7 MSS Data Source Views

MSS data source schema objects include views. A *view* is a presentation of data in one or more tables. Views appear as tables, not as views, in the Data Dictionary's table list for the schema holder. In addition, the schema holder contains no unique index information for views. Because views do not have unique indexes, you cannot modify any of the data that a view contains; however, you can access a view with the FOR EACH, FIND NEXT, and GET NEXT Progress statements. Furthermore, because views do not have index definitions, the DataServer cannot reposition the cursor to retrieve individual records. Thus, you must be sure to get all of the data that you need in a single database request.

The following Progress code example retrieves data from a view:

```
OPEN QUERY query-name FOR EACH view-name.
REPEAT:
  GET NEXT query-name.
  DISPLAY view-name.
.
.
.
END.
```

If a view has a unique combination of columns, you can simulate a unique index using the Progress Data Dictionary. You can then access a view that has a simulated unique index just as you do a table; that is, you can scroll backward and forward, and update, create, and delete data. See the "Modifying Field-level Information" section in Chapter 5, "The DataServer Tutorial," for information on how to do this.

Some views are the results of joins and contain data from more than one table. You can also provide unique index information for these views if they have a unique combination of columns. You can then scroll backward and forward, but the MSS data source does not allow you to create or delete data in a multi-table view. You can, however, update data in some views.

The DataServer does not support access to columns in views that are the results of aggregates or computations unless the calculated column has a name associated with it. You assign a

specific name to a calculated column when you define a data source view. For example, the following SQL statement names a computed column in a view definition:

```
create view view-name as select cc = count(cust_num) from customer
```

You can also access those views by using the RUN STORED–PROC send–sql–statement option to send a SQL statement to select the data from the view. In this case, you can access the view without adding index definitions for the view to the schema holder.

Although the schema holder contains your views, the Data Dictionary's SQL View Report does not list them, nor can you access them through the PRO/SQL menu functions.

2.1.8 Triggers

Triggers are code that an application associates with a data source object and an action. For example, writing a record might cause code associated with that object or action to execute. The DataServer allows an application to execute triggers for both Progress databases (including the schema holder) and MSS data sources. In an application that executes both types, the Progress trigger (CREATE, FIND, UPDATE, DELETE) executes first. If processing a Progress trigger results in a data source request, the DataServer passes the request to the MSS data source and the operation (INSERT, UPDATE, DELETE) executes.

Triggers for Progress databases and MSS data sources are independent of each other. A data source trigger that rolls back does not affect Progress triggers. Defining a trigger in Progress does not create a data source trigger definition. A Progress trigger that rolls back does so independently of the data source's transaction scope. Note, however, that although triggers for Progress databases and MSS data sources are independent, they might affect each other based on the kind of transaction your application is executing.

2.1.9 Progress Sequences

A *sequence generator* is a database object that provides incremental values within any integer range. (You can specify any positive or negative increment.) During a Progress-to-MSS conversion, the DataServer internally simulates support for Progress sequences using native MSS features. You cannot create sequences for SQL Server™ tables directly through the Progress Data Dictionary.

2.2 Data Types

SQL Server™ data types differ from Progress data types. However, each data type supported by the DataServer has at least one Progress equivalent.

The DataServer translates SQL Server™ data types into Progress equivalents and places the mapping into the schema holder. You can access this mapping information using the Data Dictionary. For example, the Data Dictionary lists the SQL Server™ datetime and smalldatetime data types as timestamp, which is the equivalent ODBC SQL data type and **not** the SQL Server™ timestamp data type. There may also be some restrictions in data source compatibility with Progress. For example, the Progress database cannot support a numeric or decimal field defined in SQL Server™ with greater than 28 digits.

You can also modify these definitions using the Data Dictionary. For example, the DataServer maps the SQL Server™ tinyint data type to the Progress equivalent, INTEGER. Suppose, however, that your application uses the tinyint field in such a way that the LOGICAL data type is a more suitable equivalent. In this case, you would change the data type from INTEGER to LOGICAL in the schema holder. If you do change a data-type mapping, be sure to select a data type that accommodates the data in the column, otherwise conversion errors might occur at run time. Also, remember to specify a display format that is appropriate for the new data type. See the "Modifying a Schema Holder" section in Chapter 5, "The DataServer Tutorial," for an explanation of how to use the Data Dictionary to change Progress data types in the schema holder.

The table in the following section lists the data types supported by the DataServer for SQL Server™ and the default Progress equivalents. Table 2–3 lists these data types, their ODBC SQL equivalents, and their default Progress equivalents. The data types in parentheses are alternative data types that you can specify in the schema holder for your MSS data source.

Table 2–3: SQL Server™ Data Type Equivalents *(1 of 3)*

SQL Server™ Data Type	SQL-ODBC[1] Data Type	Progress Default
integer	SQL_INTEGER	INTEGER (DECIMAL)
smallint	SQL_SMALLINT	INTEGER (DECIMAL or LOGICAL)
tinyint	SQL_TINYINT	INTEGER (DECIMAL or LOGICAL)
decimal	SQL_DECIMAL	DECIMAL (INTEGER)

Table 2–3: SQL Server™ Data Type Equivalents *(2 of 3)*

SQL Server™ Data Type	SQL-ODBC[1] Data Type	Progress Default
numeric[2]	SQL_DECIMAL	DECIMAL (INTEGER)
float[3]	SQL_FLOAT	DECIMAL (INTEGER)
double precision	SQL_DOUBLE	DECIMAL (INTEGER)
real	SQL_REAL	DECIMAL (INTEGER)
char[4]	SQL_CHAR	CHARACTER
varchar[4]	SQL_VARCHAR	CHARACTER
nchar[4,5]	SQL_CHAR	CHARACTER
nvarchar[5]	SQL_VARCHAR	CHARACTER
text ntext	SQL_LONGVARCHAR	CHARACTER[6]
money	SQL_DECIMAL	DECIMAL (INTEGER)
smallmoney	SQL_DECIMAL	DECIMAL (INTEGER)
datetime	SQL_TIMESTAMP	CHARACTER[7] (DATE[8])
smalldatetime	SQL_TIMESTAMP	CHARACTER[7] (DATE[8])
binary[4]	SQL_BINARY	CHARACTER
varbinary	SQL_VARBINARY	CHARACTER
image	SQL_LONGVARBINARY	CHARACTER[6]
bit	SQL_BIT	LOGICAL
timestamp[9]	SQL_VARBINARY	Unsupported

Table 2–3: SQL Server™ Data Type Equivalents *(3 of 3)*

SQL Server™ Data Type	SQL-ODBC[1] Data Type	Progress Default
identity	NA	You can only display these values.[10]

1 The SQL-ODBC data types demonstrate the mapping of native SQL Server™ data types to the ODBC standard.

2 The DataServer truncates values in SQL Server™ decimal or numeric columns defined with a scale greater than 10. In the case of float columns, it reliably translates up to 10 places only.

3 Do not use the float or real data types in joins, in primary keys, or with the equality operator

4 When you define a binary, char, or nchar column to allow nulls, SQL Server™ stores the data type definitions as varbinary, varchar, and nvarchar, respectively. This does not affect how the DataServer maps the SQL Server™ data types to Progress data types.

5 You can access nchar and nvarchar data types as SQL Server™ objects and bypass their conversion to CHARACTER by running a SQL Server ™ stored procedure or using the send–sql–statement option supported by the DataServer.

6 Although the SQL Server™ text and image fields can hold up to 2MB, Progress retrieves only up to 32K. If you are using a Progress Format phrase, there might be additional limits on the size of text and image fields. See the Format Phrase entry in the *Progress Language Reference*. You can use the DataServer (-Dsrv MAX_LENGTH) startup parameter to limit the amount of returned text data.

7 By default, the initial value of a SQL Server™ datetime or smalldatetime column is unknown ("?"). The default initial values for binary and varbinary are also unknown (“?”). The SQL Server™ datetime and smalldatetime data types contain both date and time information. The DataServer maps these to the Progress DATE data type; however, you can change the DATE data type to CHARACTER in the schema holder. If you do this, remember to change the format to match the new data type. For example, if you change the data type to CHARACTER, specify a character format, such as x(26).

8 When mapping of the SQL Server™ datetime or smalldatetime data types is to the Progress DATE data type, Progress truncates the time portion of the date.

9 The DataServer considers a timestamp data type to be a hidden value. It is not visible to the Progress user, but you can still access a SQL Server™ table that contains a timestamp column.

10 You can display values in identity columns, but you cannot insert or update them.

2.2.1 User-defined Data Types

SQL Server™ allows you to define your own data types that map to native SQL Server™ data types. When the DataServer reads the schema information for a user-defined data type, it reads the SQL Server™ base data type and maps it to the equivalent Progress data type. Suppose, for example, that you create a data type named phone_number and map it to the char data type. In the schema holder, the DataServer represents your phone_number data type as a CHARACTER data type. If you make any changes to a user-defined data type, you must update the schema holder to reflect those changes.

2.2.2 Arrays

The Progress database allows you to define fields as arrays, also called *field extents*. Progress interprets specially named data source columns of the same data type as a Progress field with the same number of array elements. You must name the data source columns *column-name*##1, *column-name*##2, and so forth. The DataServer creates a single field definition in the schema holder for the field extents. See the "Adding Extended 4GL Support" section in Chapter 5, "The DataServer Tutorial," for instructions on adding these columns automatically with the Progress-to-MSS utility.

2.2.3 Unknown Values

The DataServer supports null values. Procedures that use a null value behave exactly as they do when accessing an unknown value ("?") in a Progress database, except for one difference—you cannot compare a field to the unknown value if the field is not allowed to hold the unknown value (i.e., is not null-capable). For example, if the cust-num field is not null-capable, the following statement fails at run time:

```
FIND cust WHERE cust-num <> ?
```

A column that is not null-capable is marked "mandatory" in the schema holder.

In a DataServer application, you assign the unknown value to a column by using the question mark operator (?), which the DataServer translates to the appropriate null-value representation. For example, the following procedure assigns the unknown value to the address2 field of the customer table:

```
FIND FIRST customer.
address2 = ?.
```

2.2.4 Zero-length Character Strings

When you use the unknown value in a WHERE clause with the DataServer, the unknown value satisfies only the equals (=) or not equal (<>) operator. You can also use a zero-length character string in a WHERE clause. The unknown value and zero-length character string are not the same. The unknown value translates to a NULL, which is a special marker in a data source that supports NULLs used to represent missing information. On the other hand, zero-length strings and blank columns contain actual values. Both of the following statements find the first customer record with a zero-length string in the address2 field. Notice the space between the quotation marks in the first statement:

```
FIND FIRST customer WHERE address2 = " ".

FIND FIRST customer WHERE address2 = "".
```

Although “” and “ ” evaluate the same way in a WHERE clause, they have different results when you use them with the BEGINS function. For example, the following statement retrieves all customer names except those that have the unknown value:

```
FOR EACH customer WHERE names BEGINS "":
```

The following statement uses “ ” to retrieve only those names that begin with a space:

```
FOR EACH customer WHERE names BEGINS " ":
```

2.3 Record Creation

Record creation is handled differently for Progress databases and MSS data sources accessed through the DataServer. The difference occurs because Progress code run against a Progress database follows different record-scoping rules than Progress code run against an MSS data source. The code fragments in this section illustrate these differences.

Suppose that you have a table called cust with a field called cust-num that is defined as an indexed field, and you write the following procedure:

```
DO TRANSACTION:
    CREATE customer.
    name = "SMITH".
    cust-num = 10.
    address = "1 Main St".
END.
```

When you run this procedure:

- Progress does not create the record at the CREATE statement. Instead, it writes it to the database at the end of the record scope or when the index information is supplied, whichever occurs first. In this example, Progress writes the record after executing the statement cust-num = 10.

- The DataServer writes the record later than Progress does, at the end of the record scope. In this example, it writes the record after executing the statement END.

The following procedure, which uses multiple buffers for the same record, illustrates the differences between Progress and DataServer record creation:

```
DEFINE BUFFER xcust FOR customer.
CREATE customer.
cust-num = 111.
FIND xcust WHERE xcust.cust-num = 111.
DISPLAY xcust.
```

In this procedure, the code creates a customer, sets cust-num equal to 111, then finds and displays the customer record using cust-num (the unique index). In this case:

- Progress displays the customer 111 record.
- The DataServer fails to find customer 111 because it has not yet written the record for customer 111 to the data source.

To get a consistent response from the DataServer, use this procedure instead:

```
DEFINE BUFFER xcust FOR customer.
CREATE customer.
cust-num = 111.
VALIDATE customer. /* or RELEASE customer. */
FIND xcust WHERE xcust.cust-num = 111.
DISPLAY xcust.
```

The VALIDATE or RELEASE statement causes the DataServer to write the customer 111 record to the database before the FIND statement occurs.

NOTE: If you set the default value when creating a record, you must change the value before you create another record with the default value if the field is part of a unique key. Otherwise, the second record will cause a duplicate key error.

Record updates are handled similarly to record creation. A record is updated in an MSS data source at the end of record scope or at the end of a transaction, whichever comes first. For example, when you run the following procedure, the newly updated record is not found:

```
FIND customer WHERE cust-num = 12.
DO TRANSACTION.
  ASSIGN address = "14 Oak Park".
  FIND customer WHERE address = "14 Oak Park".
END.
```

To send the record to the data source sooner, use the VALIDATE statement, as follows:

```
FIND customer WHERE cust-num = 12.
DO TRANSACTION:
    ASSIGN address = "14 Oak Park".
    VALIDATE customer.
    FIND customer WHERE address = "14 Oak Park".
END.
```

For more information about record scoping and transaction behavior, see the *Progress Programming Handbook*.

2.4 Data Source Record Locking

In a DataServer application, SQL Server™ handles all of its own locking issues. The Progress 4GL locking rules are modified when you access information from an MSS data source. As a result, the Progress phrases NO–LOCK and SHARE–LOCK have isolation-level dependencies. The EXCLUSIVE-LOCK behaves the same in SQL Server™ as in Progress.

Table 2–4 provides data source specific comparisons.

Table 2–4: Progress and Data Source Locking

Progress	Data Source
NO–LOCK	Can support the NO–LOCK option in a manner consistent with Progress when transaction isolation level is set to read uncommitted.
SHARE–LOCK	May support shared locks at the table, page, and record level. However, the scope and duration of Progress vs. SQL Server™ shared locks may differ depending on how data source cursors behave at a transaction boundary and how isolation levels are set. The repeatable read isolation level emulates Progress SHARE-LOCK behavior most closely. For more information, see your SQL Server™ documentation.
EXCLUSIVE–LOCK	Can support the EXCLUSIVE-LOCK option in a manner consistent with Progress using any available isolation level. However, the SQL Server™ optimizer might produce locks at either the table, page, and/or the record level.

The Merant drivers provide for transaction isolation levels. In a multi-user configuration, you can isolate users from each other in your data source by setting the isolation level. In your

Progress schema holder, use the –Dsrv TXN_ISOLATION,*n* connection parameter (where n = 1, 2, 4, or 8) to set the isolation level in ODBC. See the *Microsoft ODBC Programmer's Reference* and the SQL Server™ documentation for more information.

NOTE: SQL Server™ might use page-level or table-level locking rather than record-level locking, if its optimizer determines this is the best choice. This can affect data access when two or more users attempt to read or update different records that are on the same page. See your SQL Server™ documentation for details.

Table 2–5 shows the possible –Dsrv TXN_ISOLATION,*n* values with the respective meaning.

Table 2–5: N Values in the –Dsrv Parameter

Value	Meaning
1	Read uncommitted
2	Read committed
4	Repeatable read
8	Serializable

2.4.1 Share Locks

The different isolation levels will determine what kind of share locks will take effect. In SQL Server™, a repeatable read and serializable isolation level are synonymous.

The DataServer ignores the 4GL SHARE–LOCK option when used in Progress statements. Instead, share locks are governed by the data source and the available ODBC isolation levels. If you wish to change the share lock behavior, you may be able to do so by changing the isolation level at connection time using the -Dsrv parameter.

When you read records with a 4GL statement, regardless of whether you include the SHARE–LOCK option, the MSS data source typically behaves as follows:

- It puts some form of share lock on the record, page, or table if the ODBC isolation level is anything other than read uncommitted. This occurs regardless of whether the share lock is specified in the Progress 4GL statement.
- After the data source reads the record, it releases the share lock if the isolation level is read uncommitted or read committed. It may hold share locks until the completion of a transaction if the isolation level is repeatable read or serializable.

If you hold a record with a share lock, other users can usually access that record and apply a share lock, but this is dependent on the isolation level they have selected. Refer to the transaction and locking references in the *Microsoft ODBC Programmer's Reference* or data source reference manuals for more information.

2.4.2 Exclusive Locks

When you update, delete, or create a record, SQL Server™ puts an exclusive lock on the record; however, the data source does not apply the exclusive lock to a record until all share locks on it are released. Therefore, you cannot perform an update on a record until other users release it. If a record has an exclusive lock on it, no other user can access it until it is released at the end of a transaction. In a Progress transaction block, the data source always holds an exclusive lock until the end of a transaction's scope if the data source driver supports commitment control boundaries and the ODBC AUTOCOMMIT feature is not turned on.

2.4.3 NO–LOCK

The DataServer can emulate the Progress NO–LOCK behavior if the isolation level is set to read uncommitted at connection time and you specify the NO–LOCK syntax in your 4GL statement.

2.4.4 Locking Examples

The following example illustrates how the end of a transaction affects Progress and MSS data source locks differently:

```
DO TRANSACTION:
    FIND customer WHERE cust-num = 10.
    UPDATE customer.
END.

/* Do a second transaction with the same customer record. */
```

- When you access a Progress database with this procedure, the customer record is share-locked when the first transaction ends.
- When you access an MSS data source with the DataServer, the customer record is released when the first transaction ends.

This example illustrates how Progress and MSS data source share locks differ in scope and duration:

```
FIND customer WHERE cust-num = 10 SHARE-LOCK NO-WAIT NO-ERROR.
IF AVAILABLE customer THEN DO:
  DISPLAY customer.
  PROMPT-FOR customer.

  tx:
  DO TRANSACTION ON ERROR UNDO tx, RETRY tx:
    FIND customer WHERE cust-num = 10 EXCLUSIVE-LOCK NO-WAIT NO-ERROR.
    IF LOCKED customer THEN DO:
      Message "customer locked - retrying".
      UNDO tx, RETRY tx.
      END.
    ELSE DO:
      ASSIGN customer.
      LEAVE.
      END.
  END.
END.
```

In this example, the SHARE–LOCK syntax is ignored because the share lock behavior is determined by SQL Server™ and the isolation level. If isolation level read uncommitted or read committed is selected, the first record might be updated by another user before the second FIND statement executes. In this case, the record that the second FIND statement fetches might be different from the record fetched by the first FIND statement. This procedure might cause update information to be lost because the procedure applies updates based on the first record, and these updates will overwrite the values in the second record.

Using the DataServer to access a SQL Server™ database ensures that locks are upgraded in the data source in the same way as in a Progress database. For example, the following procedure causes the same behavior whether you access a Progress database or an MSS data source:

```
FIND customer WHERE cust-num = 10.
DISPLAY customer.
PROMPT-FOR customer.

DO TRANSACTION:
   ASSIGN customer.
END.
```

The record is share-locked when it is fetched. The DataServer upgrades the share lock to an exclusive lock inside the transaction by locking the record, reading it, and checking whether the record has changed since it was first fetched. If it has changed, the lock upgrade fails and you receive an error message.

You might have to wait to access a record in the following circumstances:

- You try to update a record when another user is reading it (it is share-locked). This also depends on the isolation level.
- You try to read or update a record when another user is updating it (it is exclusive-locked).

When this happens, Progress uses a time-out loop, checking periodically to see whether the record is available. You can choose Cancel at any time to abort the request.

The MSS data source notifies the DataServer if it cannot perform a requested operation within a given period of time. Under unusual system or network loads, the DataServer might receive notification that a request has not been completed. In this case, it returns a message that the record the request was accessing is locked, even though no other user has a lock on the record.

One type of locking behavior that you might encounter is a deadlock, or "deadly embrace." A *deadlock* occurs when two users want to access each other's table, page, or record, and the table, page, or record that they want has 1) an exclusive lock on it, or, 2) one of the users needs to put an exclusive lock on it. Neither table, page, or record will give up its lock until the other table, page, or record is available. When an MSS data source detects this situation:

- The data source kills the transaction that has accumulated the least amount of CPU time and releases the table, page, or record for the other user.
- The Progress 4GL displays a message that the transaction was killed.
- The system responds as if you had chosen Cancel.

For details on how Progress locks work, see the *Progress Programming Handbook*. See the ODBC and data source documentation for more information about locks.

2.4.5 The NO–WAIT Option

The NO–WAIT option works for DataServer applications in the same way that it works for Progress applications: the DataServer uses a time-out mechanism. If SQL Server™ does not return a record, the DataServer considers the record to be locked. It then cancels the request to the SQL Server™ and sets the "locked" and "not available" conditions.

During a period of heavy demand, you might encounter situations where the "not available" condition is set although the record is not currently locked by a user. In this case, you might want to increase the time-out interval by using the -Dsrv RESP_TIMEOUT parameter.

2.5 Transactions

With DataServer operations, an MSS data source handles its own transaction roll back and recovery operations. However, the Progress transaction scoping rules apply: a transaction ends when the code exits the outermost block that performs an update. With the DataServer:

- The Progress 4GL sends a COMMIT to the data source, when a transaction that updates an MSS data source ends successfully.
- The Progress 4GL sends a ROLLBACK to the data source if you interrupt the transaction.

See the *Progress Programming Handbook* for details on how Progress handles transactions and error conditions.

2.6 Error Handling

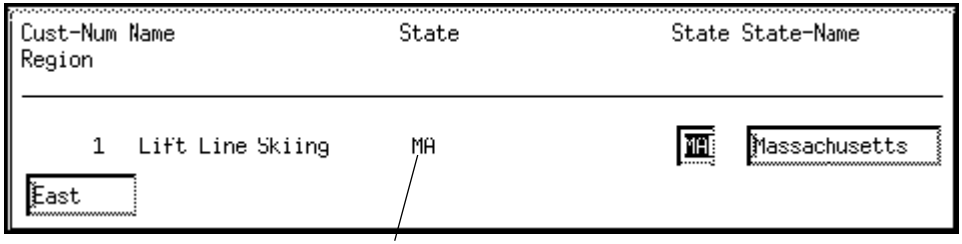
One common data-entry error is attempting to add a duplicate record; that is, trying to create a record using a unique key that already exists in the database. Suppose, for example, that a user tries to add a customer with Cust-Num = 1 (where Cust-Num is a unique key), but a customer record with this Cust-Num value already exists. The attempt fails and Progress generates an error.

When this type of error occurs, Progress tries to resolve it by working back through the procedure, looking at each block header until it finds the closest block with the error-handling property, and then undoing and retrying the block. (See the *Progress Programming Handbook* for more information about error handling.) However, because the DataServer is accessing a non-Progress data source, Progress cannot detect duplicate-key errors until the end of a transaction block. Therefore, if an error occurs in a subtransaction, Progress cannot detect it until the end of the entire transaction block and must perform default error handling for the entire transaction block.

The following example illustrates Progress and DataServer error handling:

```
rep-blk:
REPEAT:
    PROMPT-FOR customer.cust-num. /* User input */
    FIND customer USING cust-num NO-ERROR.
    IF AVAILABLE customer THEN
      UPDATE customer.cust-num name customer.state. /* User input */
    do-blk:
    DO ON ERROR UNDO do-blk, RETRY do-blk:
      FIND state WHERE st.state = customer.state.
      DISPLAY state.
      SET state. /* User input */
    END.
END.
```

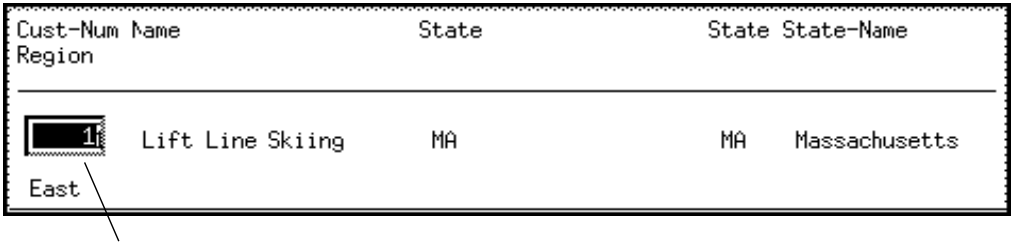
This procedure displays the following screen, in which the user is prompted to enter data into the Cust-Num field and then the State field:



Progress prompts for the state abbreviation.

Suppose that the user enters an existing state (for example, NH) while Progress is processing the DO block. When this duplicate-key entry occurs for a Progress database, Progress returns control to the DO block, displays a message that the record exists, and reprompts the user for a state abbreviation.

However, with the DataServer, if a duplicate key entry occurs in the DO block, Progress returns control to the REPEAT block rather than the DO block. As a result, the procedure reprompts the user for a customer number after the inner transaction completes:



The DataServer prompts for the customer number.

If you use NO–ERROR to do your own error handling, you must account for the fact that an MSS data source creates or updates a record later than Progress does. For example, the following code does not trap data source errors, because the requests to perform the operations have not yet been sent to the data source:

```
CREATE customer NO-ERROR.
ASSIGN cust-num = 45 NO-ERROR.
ASSIGN name = "Smith" NO-ERROR.
```

The VALIDATE statement causes the DataServer to send requests to your MSS data source, so incorporate it into your error-handling technique, as in the following example:

```
DEFINE VAR j AS INTEGER.
DO TRANSACTION:
  CREATE customer NO-ERROR.

  ASSIGN cust-num = 45 NO-ERROR.
  VALIDATE customer.
  IF ERROR-STATUS:ERROR THEN DO:
    MESSAGE "error: number of messages = " ERROR-STATUS:NUM-MESSAGES.
    DO j = 1 TO ERROR-STATUS:NUM-MESSAGES:
      MESSAGE "error" ERROR-STATUS:GET-NUMBER(j)
          ERROR-STATUS:GET-MESSAGE (j).
    END.
    UNDO, LEAVE.
  END.

  ASSIGN name = "Smith" NO-ERROR.
  VALIDATE customer.
  IF ERROR-STATUS:ERROR THEN . . .
END.
```

This code returns data-source errors after the VALIDATE statement.

Another difference in behavior occurs when two users simultaneously attempt to create records with duplicate keys. Progress raises an error immediately, but the data source raises an error after the first transaction commits and only if the second transaction does not roll back. To avoid this difference, either change the scope of the transaction so that it completes more quickly or make the key nonunique and enforce uniqueness at the application level. Another technique is to use a RELEASE or VALIDATE statement when you check for the key's uniqueness.

2.7 Cursors

A *cursor* is like a pointer that points to consecutive records in a table. Progress uses cursors to keep track of where it is in a table; for example, when it processes FOR EACH statements. Suppose that you are reading records from the customer table using the cust-num index, and your current record is customer number 50. This means that Progress has a cursor positioned at cust-num 50. Note that Progress maintains cursor positioning across queries.

The DataServer allows applications that access MSS data sources to imitate Progress cursor behavior for FIND cursors. FOR EACH and OPEN QUERY statements do not retain cursor position against a FIND statement.

2.7.1 Forward and Backward Scrolling

With the DataServer, forward and backward scrolling works with an SQL Server™ just as it does with a Progress database **except** when the data source fails to find a record. In Progress, the cursor position can change after a failed search; however, with the MSS DataServer, a failed search does not affect the cursor. For example, if a cursor is positioned at cust-num 50 and a request for the next customer from a Progress table fails, the cursor moves to the next sequential record beyond cust-num 50. The same failed request from a data source leaves the cursor at cust-num 50. Your DataServer applications should not assume a certain cursor position after a failed find.

You can, however, still use the AVAILABLE function to determine whether a record is available. The AVAILABLE function behaves consistently whether it accesses a Progress database or an MSS data source.

2.8 Progress 4GL Issues

The following sections describe how to use various 4GL statements in DataServer applications and list which 4GL statements and functions the DataServer does not support.

2.8.1 ROWID Function

The ROWID data type provides a unique record identifier that is compatible across Progress and SQL Server™ databases. Applications that use the ROWID function behave in the same way whether they access records in a Progress databases or in an MSS data source. The ROWID function is more consistent than RECID across data sources. Progress Software Corporation recommends that you replace the RECID function with ROWID in existing applications.

The DataServer supports the ROWID function for MSS data source tables that have a unique index. The Progress DataServer utilities use an index that meets this criterion to provide values for the ROWID function. If you build your schema holder using the Progress-to-MSS utility and you select to create the RECID field, it is automatically designated as the ROWID index. However, if you do not create the RECID field, you can select a unique index to support ROWID. See the "Defining the ROWID" section in Chapter 5, "The DataServer Tutorial," for instructions.

The ROWID value in an MSS data source differs from the ROWID value in a Progress database in the following ways:

- In Progress, you can create a ROWID without creating a record. In DataServer applications, creating a ROWID creates a record. The following statement illustrates the difference in behavior:

```
CREATE customer.
a=ROWID(customer).
```

  The DataServer creates a customer record using default values.

- The ROWID changes if the value of the unique keys in the designated index changes.

- The DataServer considers a single instance of the unknown value to be unique, so it is possible to have a ROWID with the unknown value. However, the Progress ROWID function fails if you search for an unknown value, because a Progress ROWID will never have the unknown value. For example, the following FIND statement fails even if there is a row with the unknown value in the designated column:

```
FIND customer WHERE ROWID(customer) = ?.
```

- If you force the creation of a record before entering the value for the designated column (for example, by committing a transaction or releasing or validating a record), the creation fails if the column cannot have NULL values. If the column can have NULL values, the DataServer assigns the new record a ROWID of NULL. However, if the column has an initial value, the DataServer creates the row with that initial value as the ROWID.

Follow these guidelines when using ROWID in applications that you want to deploy across multiple Progress databases and/or MSS data sources:

- Do not try to get a record's ROWID value before the user assigns values to the unique keys of the record.
- Refresh the ROWID value if a value of a unique key might have changed.
- Refresh the ROWID value after you undo a DELETE. The ROWID value might be different after the record is recreated.
- ROWID values are stable for a session, but you cannot rely on them to be the same across sessions.

For a complete description of the ROWID function, see its reference entry in the *Progress Language Reference*.

2.8.2 RECID Function

For backward compatibility, the DataServer supports the RECID function for MSS data source tables that have a unique 4-byte integer column.

When create RECID field is switched on, the Progress-to-MSS migration utility creates an indexed NUMBER column with unique values for each row called PROGRESS_RECID. You can also add this column to tables manually if you are using an existing SQL Server™ database or if you ported a Progress database without the create RECID field switched on.

If the Progress _RECID field does not exist in the table, the DataServer utility automatically designates the index that meets the unique key criteria. For a complete description of the RECID function, see its reference entry in the *Progress Language Reference*.

You can make an existing application that includes RECID behave more consistently across data sources by replacing RECID with ROWID. See the "ROWID Function" section for more information.

2.8.3 DEFINE BROWSE Statement

The DEFINE BROWSE statement relies on a unique record identifier for forward and backward scrolling. If your MSS data source table does not support the Progress ROWID function (through either a PROGRESS_RECID column or an indexed NUMBER column with unique values), you can write code that explicitly requests the default Progress browse scrolling behavior, similar to this:

```
DEFINE QUERY q FOR customer FIELDS (cust-num name address) SCROLLING NO-LOCK.
DEFINE BROWSE b QUERY q DISPLAY cust-num name address with 10 down.
DEFINE BUTTON upd.
OPEN QUERY q FOR EACH customer NO-LOCK.
ENABLE upd b WITH FRAME x.
ON CHOOSE OF upd DO:
  DEFINE VAR n as INT.
  n = CURRENT-RESULT-ROW("q").
  GET PREV q.
  GET NEXT q EXCLUSIVE-LOCK.
  IF CURRENT-RESULT-ROW("q") = n THEN
    UPDATE customer.address WITH FRAME z VIEW-AS DIALOG-BOX.
    /* else, indicate that an error occurred: the record was deleted in
    the meantime. */
    DISPLAY address WITH BROWSE b.
  END.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

2.8.4 Field Lists

The DataServer fully supports the use of field lists in queries (DEFINE QUERY, FOR EACH, PRESELECT, and SQL SELECT statements). For example, the following statement returns the same results for a Progress database and an MSS data source:

```
DEFINE QUERY myquery FOR customer FIELDS (cust_num name) SCROLLING NO-LOCK.
```

Include the SCROLLING option to enable GET PREVIOUS. You must include the NO–LOCK option when you open queries that are defined with field lists.

Similarly, you must include the NO–LOCK option in FOR EACH statements that include field lists, as in the following example:

```
FOR EACH customer FIELDS (cust_num name) NO-LOCK:
```

Field lists are effective only when you also specify the NO–LOCK option. This option ensures that the DataServer does not have to refetch rows, which can slow performance.

Use field lists to retrieve only those fields that your application requires. (For performance reasons, the DataServer retrieves the first index field even if you do not include it in the field list. In cases where the DataServer can predict that a query will require a refetch, it retrieves the entire record.) The DataServer allocates memory based on the maximum size defined for a field in a record. Omitting larger fields from a query can enhance performance. In addition, combining lookahead cursors and field lists greatly improves a query's performance.

When you specify a field that has an extent, the query returns the entire array.

When the DataServer processes a query with a field list, it caches the fields that are part of the field list and any other fields that the query specified, which you can then access without making another call to ODBC. For example, the DataServer fetches the name and the zip field to process the following query:

```
FOR EACH customer FIELDS (name) WHERE zip = 01730 NO-LOCK:
```

NOTE: Cached fields might have performance implications if you modify the record later, as the DataServer must refetch the record to place a lock on it.

If you specify a field list in a join, you might have to adjust the cache size for lookahead cursors, either with the CACHE–SIZE option in a QUERY–TUNING phrase or at the session level with the -Dsrv qt_cache_size startup parameter.

Any performance gained through field lists is lost if you use nonlookahead cursors.

See the Record Phrase entry in the *Progress Language Reference* for more information on the FIELDS option.

2.8.5 FIND Statements

The DataServer fully supports the FIND FIRST, CURRENT, LAST, PREV, and NEXT statements for MSS data source tables that have unique indexes. If a table has no unique index, only the FIND FIRST and FIND NEXT statements are available for that table. The following section provides additional information about specific FIND statements.

FIND PREV and FIND LAST Statements

Progress applications that use the FIND PREV or FIND LAST statements work on tables in a manner consistent with Progress. The only exception occurs when the FIND PREV or FIND LAST statement fails. In Progress, the cursor is located after the last record that was scanned. In the MSS data source, the cursor behaves as if the failed FIND had never occurred. To support these statements, a table must include support for the Progress ROWID function (through either a PROGRESS_RECID column or an indexed NUMBER column with unique values). See the "ROWID Function" section for more information.

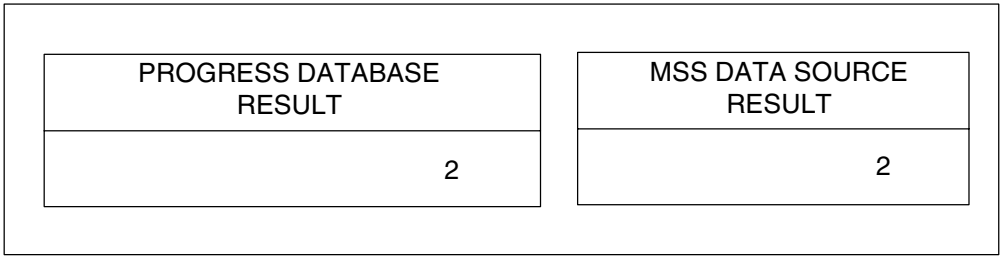
For example, the procedure find.p accesses Progress and SQL Server™ databases using the same FIND and FIND PREV statements in each case:

find.p

```
/* This code accesses a PROGRESS database. */
FIND demo.customer WHERE demo.customer.cust-num = 3.
FIND PREV demo.customer.
DISPLAY demo.customer.cust-num
  LABEL "PROGRESS DATABASE RESULT" with FRAME new-frame.

/* This code accesses an MSS data source. */
FIND mssdemo.customer WHERE mssdemo.customer.cust-num = 3.
FIND PREV mssdemo.customer.
DISPLAY mssdemo.customer.cust-num
  LABEL "MSS DATA SOURCE RESULT" with COL 29.
```

When you run find.p with a Progress table and an MSS table, you get these results (assuming that the database has records for customer numbers 1 through 4):



If the FIND PREV statement fails, the cursor remains located after customer.cust-num 3 in the Progress table, which was the last record scanned. In the data source table, the cursor is positioned at cust-num 2. Failed finds do not affect cursor position in data source tables.

2.8.6 Compiling Progress Procedures

The COMPILE statement lets you compile Progress procedures and save the compilations to disk. This speeds up your application execution time since Progress does not have to recompile it every time you want to run a procedure.

To compile procedures that access an MSS data source, start up Progress and connect to the schema holder using the schema holder's logical database name, then use the COMPILE statement. If you change the name of the schema holder after you compile a procedure, you must connect to the renamed schema holder and recompile the procedure. For more information, see the COMPILE Statement reference entry in the *Progress Language Reference*.

NOTE: You do not have to connect to the SQL Server™ database to compile a procedure. The schema holder contains all the information that the compiler requires.

R-code

Progress generates r-code when it compiles a 4GL procedure. The compiled r-code is portable among machines. For example, r-code that you compile on a Sun machine can run on any other UNIX machine. R-code is also compatible across platforms. See the *Progress Portability Guide* for information on designing compatible user interfaces for applications.

R-code is **not** portable among windowing systems; that is, r-code compiled for a character application will not run under Windows and r-code compiled for Windows will not run under a character application.

R-code is also **not** portable among database management systems. Progress generates calls that are specific to a database. For example:

- Code that you compile for a database named sports will not run with a database named mysports.
- Code that you compile for SQL Server™ will not run on Progress.

The size of r-code grows when you compile procedures against an MSS data source as compared to compiling against a Progress database. The r-code for a DataServer application contains as text portions of SQL statements that the DataServer passes to the data source.

2.8.7 FOR Statements

FOR FIRST and FOR LAST statements that cause the DataServer to sort data before returning it to the client behave differently when returning data from an MSS data source than when returning data from a Progress database. The data source does not sort data like Progress does, so the order in which records return might differ.

2.8.8 Unsupported 4GL Statements

The DataServer supplies you with complete Progress functionality when accessing MSS data sources. Nearly all 4GL language elements (statements, functions, and so forth) and Progress Data Dictionary features work the same whether your application accesses a data source through the DataServer or a Progress database.

Use the DBRESTRICTIONS function to find out which Progress features your MSS data source does not support. For the DataServer for SQL Server™, DBRESTRICTIONS can return SETUSERID and COUNT–OF. In certain circumstances, it can return additional values. For example, it can return READ–ONLY if you connect to the schema holder in read-only mode. See the DBRESTRICTIONS function reference entry in the *Progress Language Reference* for information on syntax.

Table 2–6 summarizes the 4GL differences between Progress databases and MSS data sources.

Table 2–6: 4GL Differences Between Progress Databases and cMSS Data Sources *(1 of 2)*

Progress Feature	MSS Data Source Difference
CONTAINS operator	This operator relates to word indexing, which the DataServer does not support.
COUNT–OF function	The DataServer does not support this function.
CREATE statement	Records that you create after opening a cursor might be invisible to that cursor.
FIND statements FOR EACH statements OPEN QUERY statement	To reduce the number of records included in the results set, qualify your FIND statements and queries with a WHERE clause.
MATCHES function BEGINS function	The DataServer does not support using the percent (%) or underscore (_) character with the MATCHES and BEGINS functions. Do not use these functions with a pattern that is not an expression. It is theoretically possible to do this with a Progress database, but using this kind of criteria results in poor performance.

Table 2–6: 4GL Differences Between Progress Databases and cMSS Data Sources *(2 of 2)*

Progress Feature	MSS Data Source Difference
NO–LOCK option[1] SHARE–LOCK option EXCLUSIVE–LOCK option	A Progress EXCLUSIVE–LOCK is emulated in the SQL Server™ DataServer. A Progress NO-LOCK can be emulated in the SQL Server™ database when the isolation level is set to read-uncommitted. SHARE-LOCK behavior is data source and isolation level dependent. See the section "Data Source Record Locking" for more information.
NULL	SQL Server™ NULL = Progress unknown value ("?") Progress empty string ("") = a one space string in SQL Server™.
Record creation	A record is created at the end of a record's scope and **not** when the required index information becomes available.
SETUSERID function	You cannot use this function to change the login name and password.

[1] For more information, see the "Data Source Record Locking" section.

2.9 Stored Procedures

From the viewpoint of an ODBC-compliant data source, a *stored procedure* is a predefined procedure or set of statements or program that is defined for a specific data source according to that data source's rules. By executing the stored procedure, you execute these statements or programs without having to enter their individual statements or code at each execution.

2.9.1 Defining Native Stored Procedures to Progress

The first time you run a stored procedure, the data-source management system creates an execution plan for it and stores the plan in the database. The next time you run the stored procedure, it runs as a precompiled procedure. This makes access to the database much quicker and more efficient than when you access it with new queries each time.

The DataServer allows you to use the Progress 4GL to run SQL Server™ stored procedures. The procedures must be defined according to the rules of the data source. All stored procedures are executed on the server machine by the native data source.

When you create or update your schema image, the stored procedures, functions, and packages appear in the list of accessible objects along with tables, view, and sequences. Progress allows you to run the stored procedures that you create in data sources from within Progress procedures. See Microsoft's SQL Server™ documentation for complete information about creating and using stored procedures.

Stored procedures called from within Progress applications cannot return Boolean types. Table 2–7 lists issues that occur when you pass other data types as parameters.

Table 2–7: Argument Data Types for Stored Procedures

Progress	MSS Data Source
DECIMAL FLOAT INTEGER	The DataServer represents all three data types as the Progress INTEGER data type in the schema image. To preserve the scale and precision of these data types, you must manually update the information in the schema image for these parameters. Use the Progress Data Dictionary to update the data type and format information in the field property sheet for the parameter.
CHAR	The data source represents this type as a VARCHAR parameter. Its size cannot exceed the VARCHAR size limit for the associated data source. If the VARCHAR parameter exceeds this limit, it causes an error.
DATE	If you pass a DATE data type as an input parameter and use it in an equality test, the test might fail. In this case, use DATEPART() or DATEDIFF().

If you are running several stored procedures, run them serially and process all the results from one stored procedure before you run a second one. The DataServer allows only one active request for running a stored procedure. However, you can process results from several stored procedures concurrently if you specify the DataServer startup parameter (-Dsrv PRGRS_PROC_TRAN) when you start your Progress session. When you run stored procedures concurrently, the DataServer uses one connection to the data source per procedure. If the stored procedures attempt to update the same record from a single client's requests, the connections could block each other or a deadlock might occur.

You run stored procedures from within Progress procedures by using the 4GL RUN STORED–PROCEDURE statement. A stored procedure that you run with this statement can return three types of values:

- An integer return code, which could be a success code or a value returned by the stored procedure (defined by the data source)
- Values of output parameters that you define when you create the procedure
- Results from the database

Progress has statements and functions that allow you to use the return codes and the values of the output parameters. Table 2–8 lists these statements and functions.

Table 2–8: Returning Values from Stored Procedures

Progress	Description
CLOSE STORED–PROCEDURE statement	Retrieves the values from the output parameters you defined for the stored procedure and tells Progress that the stored procedure has ended.
PROC–HANDLE function	Allows you to specify a handle to identify a stored procedure.
PROC–STATUS function	Reads the return value.
RUN STORED–PROCEDURE statement	Executes a stored procedure.

Note that you can substitute the abbreviations CLOSE STORED–PROC and RUN STORED–PROC for the full names CLOSE STORED–PROCEDURE and RUN STORED–PROCEDURE, respectively. The remainder of this guide generally uses the abbreviated form.

See Appendix B, "Stored Procedure Reference," for reference entries for the statements and functions described in Table 2–8.

Progress provides two techniques for accessing the results returned from the data source by the stored procedure. You can:

- Use the proc-text-buffer that Progress supplies for the rows of results.
- Define a buffer for the rows of results.

After you read the values into the buffer, you can operate on them in a variety of ways. You can use the database data just as you would use information from a Progress database—format it and use it for calculations.

The following sections describe how to do the following:

- Run stored procedures
- Retrieve return codes
- Retrieve output parameter values
- Define and use buffers to retrieve database results

All of the following examples of how to run stored procedures in Progress use a stored procedure created in Transact-SQL for SQL Server™:.

```
CREATE PROCEDURE pcust (@num INT, @orders INT OUT, @states INT OUT) AS
BEGIN
SELECT customer.cust_num, customer.name, order_num FROM customer, order_
  WHERE customer.cust_num = order_.cust_num AND customer.cust_num > @num
SELECT @orders = @@rowcount
SELECT cust_num, state.state FROM customer, state WHERE
  customer.state = state.state AND customer.cust_num > @num
SELECT @states = @@rowcount
END
RETURN 0
GO
```

This Transact-SQL code creates the stored procedure pcust and defines three parameters: `num`, which is an input parameter, and `orders` and `states`, which are output parameters. The procedure returns values for the output parameters to the caller after processing the results of the pcust SELECT statements. You can think of output parameters as temporary fields; that is, you can access the data in these columns using the standard notation of *tablename.fieldname*. (Note that although pcust is a stored procedure, its syntax is that of a table and it is stored in a table definition.) For example, you can access the data in the orders and states fields by specifying pcust.orders and pcust.states. All the parameters in the example have an integer data type.

2.9.2 Running a Stored Procedure

The Progress 4GL statement RUN STORED–PROC allows you to run a stored procedure in the MSS data source. You must also indicate the end of a stored procedure in your Progress procedure by using the CLOSE STORED–PROC statement.

This is the syntax for the RUN STORED–PROC statement:

SYNTAX

```
RUN STORED-PROC procedure [ NO-ERROR ]
  [ ( [ OUTPUT ] parameter , ... [ OUTPUT ] parameter ) ]
```

This is the partial syntax for the CLOSE STORED–PROC statement:

SYNTAX

```
CLOSE STORED-PROC procedure
```

For example, the following Progress 4GL code runs the stored procedure pcust:

```
DEFINE VAR handle1 AS int.
RUN STORED-PROC pcust handle1 = PROC-HANDLE (20, OUTPUT 0, OUTPUT 0).
FOR EACH proc-text-buffer:
 DISPLAY proc-text-buffer.
END.
CLOSE STORED-PROC pcust WHERE PROC-HANDLE = handle1.
DISPLAY pcust.orders pcust.states.
```

This code first defines an integer variable named handle1 that serves as a handle for identifying the stored procedure. If you have only one active stored procedure, you do not have to specify a handle. However, it is good programming practice to use handles to identify all of your stored procedures.

It then runs the stored procedure as follows:

- The pcust stored procedure passes the values 20, 0, and 0 to the three parameters (specifying orders and states as output parameters).
- Using a FOR EACH statement, it reads the results into the Progress-supplied buffer proc–text–buffer.
- It displays the results.

The Progress procedure next uses the CLOSE STORED–PROC statement to fetch the orders and states output parameters and then displays them. Note that the stored procedure does not return output parameter values unless you request them with the keyword OUTPUT or INPUT–OUTPUT when you execute the procedure.

You can close all stored procedures at once with the following statement:

```
RUN STORED-PROC closeallprocs.
```

NOTE: For SQL Server™, the DataServer typically maintains one connection to SQL Server™. If your application requires that you process other queries while a stored procedure is open, use the -Dsrv qt_separate_connection parameter or the QUERY–TUNING (SEPARATE–CONNECTION) option to specify that the DataServer use a separate connection for each statement that requires a cursor.

See Appendix B, "Stored Procedure Reference," for a description of the complete syntax for the Progress statements and functions that support running stored procedures.

2.9.3 Retrieving Return Codes

A stored procedure might return a code that provides information. For example, it might indicate whether the stored procedure was successful or whether it encountered an error condition. The following example of 4GL code runs the stored procedure pcust. It uses the PROC–STATUS function and the CLOSE STORED–PROC statement to retrieve the return code and assign the value to the variable stat. The meaning of this return code is defined by the underlying data source:

```
/* Return status */

DEFINE VAR stat AS integer.
RUN STORED-PROCEDURE pcust (20, output 0, output 0).
FOR EACH proc-text-buffer:
 DISPLAY proc-text-buffer.
END.
CLOSE STORED-PROC pcust stat = PROC-STATUS.
DISPLAY pcust.orders pcust.states stat.
```

2.9.4 Retrieving Output Parameter Values

When you call a stored procedure, you can specify the ordered list of positional parameters or you can name them individually. To retrieve output parameter values from a stored procedure, request them with the keyword OUTPUT or INPUT–OUTPUT when you execute the procedure. When you run a stored procedure in a DataServer application, the parameters are supplied and passed using Progress data types.

The following 4GL procedure uses the second option for passing parameters—it passes them by name with the PARAM option:

```
/* Parameters by name */

RUN STORED-PROC pcust (PARAM num = 20, output PARAM states = 0,
 output PARAM orders = 0).
FOR EACH proc-text-buffer:
 DISPLAY proc-text-buffer.
END.
CLOSE STORED-PROC pcust.
DISPLAY pcust.orders pcust.states.
```

When you use PARAM to specify parameter names, you do not have to specify all parameters for the stored procedure. Instead, you can include only those parameters you want to use, in any order you choose. If the stored procedure names a default value for the parameter, you do not have to name that parameter at run time. However, you must explicitly name parameters that do not have defaults or name them when you want to pass values that are different from the default.

2.9.5 Retrieving Data Source Results

Progress provides two techniques for accessing results returned from a data source by the stored procedure. You can:

- Use the proc–text–buffer that Progress supplies for the rows of results
- Define a special view on the MSS data source to use as a buffer for the rows of results

The following 4GL procedure reads the database results from the stored procedure into the proc–text–buffer supplied by Progress:

```
RUN STORED-PROC pcust (20, output 0, output 0).
FOR EACH proc-text-buffer:
 DISPLAY proc-text-buffer.
END.
CLOSE STORED-PROC pcust.
 DISPLAY pcust.orders pcust.states.
```

The Progress-defined buffer, proc–text–buffer, has one character field named proc–text. The buffer accepts the returned database results, converts them to CHARACTER data type, and concatenates them into one string. The advantage of using the proc–text–buffer is that you do not have to worry about what kind of data the procedure returns. The buffer accepts any type of data, in any order. The disadvantage is that it is much more difficult to manipulate the data after

you receive it. To act on anything but CHARACTER data, you must extract the data from the buffer and convert it to its original data type before you can use it.

Another benefit of the proc–text–buffer is that it holds the results from all of the SQL statements included in a stored procedure. However, a buffer that you create can hold the results of only one SQL statement.

This is the partial syntax for the DEFINE BUFFER statement that you use to create a buffer with the same characteristics of the proc–text–buffer:

SYNTAX

```
DEFINE BUFFER buffer-name FOR proc-text-buffer
```

For a complete description, see the DEFINE BUFFER entry in the *Progress Language Reference*.

2.9.6 Defining a View to Use As a Buffer

If you do not use the buffer proc–text–buffer defined by Progress, you must define your own. Defining a view in the data source that can serve as a buffer allows you to retrieve database results in their original data types.

While a stored procedure can include multiple SQL statements, a buffer that you define holds the results of only a single SQL statement.

Follow these steps to define a buffer:

1 ♦ Define a view in the MSS data source:

- With the naming convention _BUFFER_*buffername*
- With the same number of columns and data types that the stored procedure returns in the results set
- With the columns in the order that the stored procedure returns them

For example, to return two columns with two types of values, an integer and a character string, use an SQL utility to define the following view in the data source:

```
CREATE VIEW _BUFFER_custlist AS SELECT customer.cust_num,
customer.name FROM customer WHERE 1 = 0
GO
```

Notice that these views are defined to ensure that they never return any results. This indicates that you will not use the views as views, but as buffers. It is not necessary to define views that you will use as buffers this way, but it does allow you to distinguish quickly between views and buffers.

2 ♦ Update your schema image using the Update/Add Table Definitions DataServer utility. The utility adds the view to the list of accessible objects in the schema holder. The DataServer defines the view as a buffer that Progress can use.

See the "Updating a Schema Holder" section in Chapter 5, "The DataServer Tutorial," for instructions on using this utility.

This buffer defines two returned values for a stored procedure—an INTEGER and a CHARACTER value—in that order. If the data types do not match those returned by the stored procedure, the procedure returns more than two types of values, or returns the values in a different order than you specified, you receive a run-time error.

The easiest way to create a buffer that accepts data from stored procedures is to use the text of the SQL SELECT statement from the stored procedure. This ensures that you define your data types correctly and in the correct order. Use a native process such as sp_helptext to view the stored procedure from SQL Server™, or view procedures in the system tables.

The next example does not use the supplied buffer. Instead, it defines buffers by creating views in the data source, using the following syntax:

SYNTAX

```
CREATE VIEW _BUFFER_buffer-name
```

The following examples are of the views created in your MSS data source that you can use as buffers to store the results from the stored procedure pcust:

```
CREATE VIEW _BUFFER_pcust_orders AS SELECT customer.cust_num,
   customer.name, order_num FROM customer, order_ WHERE 1 = 0
GO
```

```
CREATE VIEW _BUFFER_pcust_states AS SELECT cust_num, state.state
FROM customer, state WHERE 1 = 0
GO
```

The following 4GL procedure shows the results of the stored procedure pcust being written into the new buffers pcust_orders and pcust_states:

```
/* Typed buffers */

RUN STORED-PROC pcust (20, output 0, output 0).
FOR EACH pcust_orders:
  DISPLAY pcust_orders.
END.
FOR EACH pcust_states:
 DISPLAY pcust_states.
END.
CLOSE STORED-PROC pcust.
 DISPLAY pcust.orders pcust.states.
```

Because two different buffers have been defined, the returned values maintain their data types instead of being converted to character strings and stored in the Progress-defined buffer proc–text–buffer. You can then use the returned values in calculations without first converting them back to their original data types. In addition, the two separate buffers make your output look cleaner, allowing Progress to build a new default frame for the two different types of output. Reading your results into an explicitly defined buffer also allows you to manipulate the data just as you would manipulate data from a Progress database; for example, with Frame phrases and FORM statements.

The next example accesses the stored procedure pcust twice; procedure handles (through the PROC–HANDLE function) identify the different results from your data source:

```
/* Procedure handles */

DEFINE VAR handle1 AS integer.
DEFINE VAR handle2 AS integer.
RUN STORED-PROCEDURE pcust handle1 =
 PROC-HANDLE (20, output 0, output 0).
RUN STORED-PROCEDURE pcust handle2 =
 PROC-HANDLE (20, output 0, output 0).
FOR EACH pcust_orders WHERE PROC-HANDLE = handle1:
 DISPLAY pcust_orders.
END.
FOR EACH pcust_states WHERE PROC-HANDLE = handle1:
 DISPLAY pcust_states.
END.
FOR EACH pcust_orders WHERE PROC-HANDLE = handle2:
 DISPLAY pcust_orders.
END.
FOR EACH pcust_states WHERE PROC-HANDLE = handle2:
 DISPLAY pcust_states.
END.
CLOSE STORED-PROC pcust WHERE PROC-HANDLE = handle1.
CLOSE STORED-PROC pcust WHERE PROC-HANDLE = handle2.
```

In this example, the results look the same as in the previous example. However, because you are running a stored procedure twice, Progress uses the procedure handles to identify the different instances. If you run more than one stored procedure in your application, you must explicitly define procedure handles for each.

The next example shows how to use standard Progress syntax to join the results of the stored procedures with other tables in the database:

```
/* Join with procedure results */

RUN STORED-PROC pcust (20, output 0, output 0).
FOR EACH pcust_orders, EACH order-line
    WHERE pcust_orders.order-num = order-line.order-num:
    DISPLAY order-line.order-num order-line.item-num.
END.
FOR EACH pcust_states:
    DISPLAY pcust_states.
END.
CLOSE STORED-PROC pcust.
```

This example joins the order information returned from the stored procedure with the order–line information in the same database.

2.9.7 Sending SQL Statements Directly to the Data Source

Progress uses stored-procedure syntax to allow you to send extended SQL statements directly to a data source. The DataServer uses the RUN STORED–PROCEDURE statement with the send–sql–statement option to pass SQL statements to the data source. Although Progress SQL allows you to use SQL statements, this option gives you access to Transact–SQL for SQL Server™. For example, you can issue DDL statements from within Progress procedures. You can send multiple SQL statements by concatenating them and passing the string as a single parameter to the send–sql–statement option.

You use the RUN STORED–PROC statement with the send–sql–statement option and pass the SQL statements as a parameter. The syntax of the statement must be valid SQL syntax and must follow SQL naming and case conventions. For example, this code passes a SELECT statement as a parameter:

```
DEFINE VAR handle1 AS INTEGER.
RUN STORED-PROC send-sql-statement handle1 = PROC-HANDLE
("SELECT name, address, city, state, postal_code
FROM customer WHERE credit_limit >= 500").
FOR EACH proc-text-buffer WHERE PROC-HANDLE = handle1:
    DISPLAY proc-text-buffer.
END.
CLOSE STORED-PROC send-sql-statement WHERE PROC-HANDLE = handle1.
```

This example returns the name, address, city, state, and postal code for all customers whose max–credit is greater than or equal to $500. You must read the results into a buffer as you would with a stored procedure called by a Progress procedure. You can read the results into the proc–text–buffer defined by Progress, as in the example above, or you can define your own buffer from within your data source that accepts a data type other than the CHARACTER data type.

The following example illustrates returning database results into the proc–text buffer and converting the results to the INTEGER data type:

```
DEFINE VAR handle1 AS INTEGER.
DEFINE VAR imax as INT.
RUN STORED-PROC send-sql-statement handle1 = PROC-HANDLE
("SELECT max (cust_num) FROM customer").
FOR EACH proc-text-buffer:
    imax = int (substring (proc-text, 1, 3)).
    DISPLAY imax.
END.
CLOSE STORED-PROC send-sql-statement WHERE PROC-HANDLE = handle1.
```

The DataServer passes the SQL statement directly to the MSS data source. The Progress Compiler does not process it, so errors occur only at run time and not when you compile a procedure.

2.9.8 Handling Errors from Stored Procedures

The RUN STORED–PROC statement supports the NO–ERROR option. The following example shows how to trap errors within a procedure:

```
DEFINE VAR j AS INTEGER.
RUN STORED-PROC send-sql-statement NO-ERROR
    ("select count (*) from xxx.customer where name between 'A' and 'Z' ").

IF ERROR-STATUS:ERROR THEN DO:
    DO j = 1 TO ERROR-STATUS:NUM-MESSAGES:
      MESSAGE "error" ERROR-STATUS:GET-NUMBER(j)
           ERROR-STATUS:GET-MESSAGE(j).
    END.
END.

CLOSE STORED-PROC send-sql-statement.
```

2.10 Enhancing DataServer Performance

When you develop a DataServer application, you can design it either to emphasize portability across data sources or to optimize the strengths of the DataServer's interaction with an SQL Server™ database. For example, you might write a query that gives you consistent results across databases or one that takes advantage of SQL Server's™ cursor management functionality.

In addition to influencing how DataServer applications perform through queries, you can control how the DataServer processes queries on a statement-by-statement basis. Some of the DataServer's default behavior might not be optimal for the application you are designing. The QUERY–TUNING phrase and startup and connection parameters give you the ability to control query processing.

Information on query tuning appears in the following locations:

- The remaining sections of this chapter document the QUERY–TUNING phrase.
- For information on tuning queries at compile time and run time, see the "Query Tuning with Connection and Startup Parameters" section in Chapter 4, "Connecting the DataServer."

2.10.1 Progress Query Types

The DataServer provides several ways to submit a Progress query to an MSS data source:

- **Progress 4GL —** This approach applies to the DEFINE QUERY and FOR EACH statements. The DataServer generates SQL for each of these statements. You can use the QUERY–TUNING option to customize the queries that the DataServer passes to ODBC.
- **Progress SQL SELECT —** This approach applies to the SQL SELECT statement. When you use this statement in a Progress procedure, the DataServer passes the SQL directly to the data source. This can improve performance, especially when counting records, and can also allow you to access certain types of data more effectively, such as aggregates.
- **Vendor-specific SQL —** This approach applies to stored procedures. If you want to use specialized query syntax supported only by Transact-SQL extensions, you can use RUN–STORED–PROC send–sql–statement to send the syntax to SQL Server™. You might use a stored procedure to include BEGINS as a search criterion; this SQL query can result in better performance. For more information, see the "Sending SQL Statements Directly to the Data Source" section.

Whether your application can take advantage of the strengths of a particular approach depends on the kind of query you are writing and the kind of data you are accessing. Another factor to keep in mind when you decide which technique to use for issuing queries is whether a query is better served by being processed by the client or by the server. Progress 4GL queries are processed by the client (except in the cases of most joins); SQL SELECT statements and Transact-SQL extensions are processed by the server.

2.10.2 Query Tuning

How you structure a query determines how efficiently you access a database. Using your MSS data source efficiently enhances the performance of DataServer applications. The standard approach to enhancement is using selection criteria to refine access to data, but you can further optimize the DataServer's execution of a query by specifying the Progress QUERY–TUNING phrase.

You can include the QUERY–TUNING phrase in these Progress statements:

- FOR EACH

  SYNTAX

```
FOR EACH table QUERY-TUNING(query-tuning-option
  query-tuning-option...)
```

- OPEN QUERY

  SYNTAX

```
OPEN QUERY query QUERY-TUNING(query-tuning-option
  query-tuning-option...)
```

- DO PRESELECT

  SYNTAX

```
DO PRESELECT table QUERY-TUNING(query-tuning-option
  query-tuning-option...)
```

- REPEAT PRESELECT

  SYNTAX

```
REPEAT PRESELECT table QUERY-TUNING(query-tuning-option
  query-tuning-option...)
```

You must place the QUERY–TUNING phrase after the last record phrase. For example, place it near the end of the statement where you also place block modifier phrases such as BREAK, ON ERROR, and TRANSACTION.

You can include multiple query-tuning options in a single statement; simply separate each option from the previous one by a single space.

Table 2–9 describes the query-tuning options.

Table 2–9: Query-tuning Options *(1 of 2)*

Option	Description
ARRAY–MESSAGE NO–ARRAY–MESSAGE	Specifies whether the DataServer sends multiple result rows in a single logical network message, thereby reducing network traffic. Default: ARRAY–MESSAGE, if the query uses a lookahead cursor
CACHE–SIZE *integer*	Specifies the size in bytes of the cache used by lookahead cursors. A larger cache size can improve performance for queries that return a large number of records because the DataServer might need fewer SQL statements to get the results. Minimum: The DataServer always caches at least one record. Maximum: None Default: 30000
DEBUG EXTENDED DEBUG SQL NO–DEBUG	Specifies whether the DataServer should print to the `dataserv.lg` file the debugging information that it generates for a query. Specify DEBUG SQL to print only the SQL that the DataServer executes against the ODBC data source. Specify DEBUG EXTENDED to print additional information, such as cursor statistics. Specify DEBUG *option* to override the NO–DEBUG default. In addition to EXTENDED and SQL, there are other options that can assist you in analyzing performance. Default: NO–DEBUG

Table 2–9: Query-tuning Options *(2 of 2)*

Option	Description
JOIN–BY–SQLDB NO–JOIN–BY–SQLDB	Specifies whether the DataServer allows its data source to perform a join (this usually improves performance). JOIN–BY–SQLDB implies SEPARATE–CONNECTION queries that include joins. Default: JOIN–BY–SQLDB JOIN–BY–SQLDB is a compile-time option. A query must be compiled to use or not use this option. You can turn off the JOIN–BY–SQLDB default globally at compile time by specifying the Server Join (-nojoinbysqldb) startup parameter when you start a Progress session. This parameter does not override the explicit use of JOIN–BY–SQLDB in the QUERY–TUNING phrase.
LOOKAHEAD NO–LOOKAHEAD	Specifies whether the DataServer uses lookahead or standard cursors. Lookahead cursors fetch as many records as fit in the allocated cache (see the CACHE–SIZE entry in this table). This reduces the number of SQL statements and network messages that are required, thereby improving performance. Using lookahead cursors results in behavior that is different from Progress because changes made to the records in the cache might not be immediately visible. Specify NO–LOOKAHEAD for behavior that is consistent with Progress. Default: LOOKAHEAD, only when statements use NO-LOCK.
SEPARATE–CONNECTION NO–SEPARATE–CONNECTION	Specifies whether each cursor should use a separate database connection. Executing cursors in separate connections might improve performance because the DataServer does not have to restart the cursors and sort the results. Do not specify SEPARATE–CONNECTION if you require behavior that is consistent with Progress. Default: NO–SEPARATE–CONNECTION except in certain cases. For details, see "Managing Connections to a SQL Server™ Database" in Chapter 4, "Connecting the DataServer."

All but two of the QUERY–TUNING options take effect at both compile time and run time. The exceptions are JOIN–BY–SQLDB and NO–JOIN–BY–SQLDB, which apply only at compile time. You can override query-tuning defaults (except JOIN–BY–SQLDB) at run-time by specifying the appropriate startup parameters.

The following example shows how to use the QUERY–TUNING phrase to enhance performance. It includes a join that the DataServer instructs the MSS data source to perform by default:

```
FOR EACH customer, EACH order OF customer WHERE order-num > 20
    BY customer.cust-num
    QUERY-TUNING(NO-LOOKAHEAD DEBUG EXTENDED)
    TRANSACTION:
```

The QUERY–TUNING options in this example specify the following:

- Lookahead cursors are not used (the NO–LOOKAHEAD option).
- The DataServer writes an extended report on the SQL statements that it executes (the DEBUG EXTENDED option).

When the DataServer constructs queries for an MSS data source, it uses the QUERY–TUNING options that you specify as guidelines. This is because there might be syntax considerations that prevent the DataServer from applying the QUERY–TUNING options as specified. In such a case, the DataServer executes the query using the most appropriate options.

NOTE: The DataServer does **not** issue errors or warnings if it does not apply the QUERY–TUNING options that you specify.

2.10.3 Caching Records

The DataServer caches results sets from an MSS data source to enhance performance. It caches as much data as fits in its allocated cache size. Depending on what kind of cursor a query is using, the DataServer caches row identifiers or records:

- **Standard cursors** — The DataServer caches row identifiers (PROGRESS_RECID column or other unique index) for the results set. If you use the Progress_RECID, each identifier requires 4 bytes of cache , therefore, a results set of 100 records requires 400 bytes of cache. If you do not use the Progress_RECID field, the cache size might be greater if the selected unique index has a greater length than this field.
- **Lookahead cursors** — The DataServer caches complete records or partial records as specified by a field list. It uses the maximum length allowed for a row as defined in the MSS data source to calculate the record length, not the actual contents of the record. In

addition to the defined row length, the record consists of a row identifier field; therefore, a row with a defined maximum length of 100 bytes requires 104 bytes of cache. If a column is longer than 256 bytes, the DataServer refetches it.

In the case of joins, each record in the cache is a result of the fields selected in the join. In addition to the record, there is a row identifier field (4 bytes) for each table involved in the join. For example, a three-way join adds 12 bytes to the cache for each record.

You can affect the performance of a query by controlling the size of the cache when lookahead cursors are used. As queries generate different results, they benefit from different cache sizes. Generally, the larger the cache, the faster the performance. However, you must balance cache size against other memory requirements for your system. Consider also that continually adjusting cache size in an application might decrease performance as each adjustment requires the DataServer to make several calls to the data source.

To determine the optimal cache size for a query, experiment with different values for CACHE–SIZE and use DEBUG EXTENDED to generate cursor statistics in the `dataserv.lg` file that you can examine. Aim for minimal cursor activity. The following example sets an optimal cache size for a particular query against the sports database:

```
FOR EACH customer, EACH order OF customer WHERE ord-num > 20
  QUERY-TUNING(CACHE-SIZE 20 DEBUG EXTENDED):
```

2.10.4 Join by SQLDB

For queries that include joins issued in FOR EACH and OPEN QUERY statements, the DataServer evaluates the queries and in some cases instructs the MSS data source to perform the joins. A join performed by the data source, called a *join by SQL DB*, can improve performance; however, you receive the associated query results in an order consistent with the data source, not with Progress. To get results that are consistent with Progress, turn off JOIN–BY–SQLDB, either with the QUERY–TUNING phrase at the query level or with the Server Join (-nojoinbysqldb) startup parameter when you compile. If the order of returned records is important to your application, specify a sort order on the query.

For each join, the DataServer evaluates whether the MSS data source can perform it and estimates whether doing so improves performance. To determine whether a join by SQL DB is possible, the DataServer assesses whether the following criteria are true:

- All tables in the join are in the same logical Progress database; that is, they are contained in the same DataServer schema.
- Every table, except the innermost one, has a unique record identifier (ROWID or RECID support).

- The query does not include a USING phrase for any of the inner tables. For example, a join by SQL DB will not occur for this query:

```
FOR EACH customer, EACH order OF customer USING order-num:
```

- The query does not include a BY phrase that contains expressions or array fields.
- The query does not include a request for an EXCLUSIVE LOCK on any of the tables in the join.
- The join does not exceed 10 levels.

To estimate whether performing a join by the data source might improve performance, the DataServer assesses whether these additional criteria are true:

- The join uses an OF clause or a WHERE clause for each of the inner table loops. For example, the following query requires a field-to-field correspondence between two tables:

```
FOR EACH customer, EACH order OF customer:
```

- The WHERE clause includes either an operator or the AND option. The following example includes the equals (=) operator:

```
FOR EACH customer, EACH order
   WHERE customer.cust_num = order.cust_num
```

The DataServer also performs a join by SQL DB for the following query:

```
FOR EACH customer, EACH order
   WHERE customer.cust_num = order.cust_num AND cust_num > 100:
```

For the following query, however, the DataServer instructs the client to perform the join because of the OR option:

```
FOR EACH customer, EACH order
   WHERE customer.cust_num = order.cust_num OR cust_num > 100:
```

By default, the DataServer instructs an MSS data source to perform a join when possible and when desirable. However, you can control the default behavior by using either the QUERY–TUNING NO–JOIN–BY–SQLDB phrase or the Server Join (-nojoinbysqldb) startup parameter. The QUERY–TUNING phrase controls the behavior for a single query. The -nojoinbysqldb parameter controls it at the session level. The query-level setting overrides the session-level setting.

Table 2–10 describes how these controls interact and affect the behavior.

Table 2–10: Controlling Join by SQLDB Behavior

QUERY–TUNING	Startup Parameter	Behavior
JOIN–BY–SQLDB	-nojoinbysqldb	The MSS data source performs the join if possible.
JOIN–BY–SQLDB	None	The MSS data source performs the join if possible.
NO–JOIN–BY–SQLDB	-nojoinbysqldb	The client performs the join.
NO–JOIN–BY–SQLDB	None	The client performs the join.
None	-nojoinbysqldb	The client performs the join.
None	None	The MSS data source performs the join if possible and the join contains the selection criteria described previously.

A join by SQL DB does not occur by default for the following query because the DataServer determines that it does not increase performance:

```
FOR EACH customer, EACH order:
```

You receive a warning if you specify JOIN–BY–SQLDB when the MSS data source cannot perform the join and the DataServer performs the join instead. You receive a warning at compile time if you specify JOIN–BY–SQLDB when the data source can perform the join but it is not optimal for it to do so.

2.10.5 Writing Queries for Performance

This section provides a collection of tips and guidelines to follow when writing queries. For example, a query that processes a large number of rows performs best if it uses NO–LOCK, lookahead cursors, a large cache size, and a small field list.

These tips and guidelines might improve the performance of your DataServer applications. To assess the usefulness of a particular suggestion, apply it, then use the DEBUG diagnostic options to gather statistics on how your application runs. (See the "Analyzing Performance" section in Chapter 4, "Connecting the DataServer," for information on collecting statistics.)

- Use FOR EACH, GET, and OPEN QUERY statements rather than FIND statements, which generally perform more slowly. Consider using the FOR FIRST statement instead of FIND FIRST.

  The only exception is that FIND LAST is faster than GET LAST. This is because GET LAST causes the client to process all of the records; the FIND LAST statement allows the server to retrieve the last record.

- Use field lists.

- Use the QUERY–TUNING options.

- Use lookahead cursors.

- Use NO–LOCK where possible.

- Avoid specifying lock upgrades. Instead, allow the DataServer and the MSS data source to handle lock upgrades.

- Do not ask for a particular ordering of results with USE–INDEX or BY clauses unless your application requires it. Instead, allow the DataServer and the MSS data source to determine the most efficient index (if any) for processing a query and avoid the overhead of sorting results.

- If you use a BY clause that will sort a large amount of data, make sure a corresponding index exists in your data source to make sorting efficient. In some cases it may also be desirable to have indexes over columns used in WHERE clause selection criteria.
- For aggregates, use either the RUN–STORED–PROCEDURE send–sql–statement syntax or a Progress SQL statement. If you use a Progress SQL statement with a cursor, declare the cursor read-only.
- When you test for the existence of a record, use the CAN–FIND FIRST function, which does not retrieve the record if the DataServer passes the entire WHERE clause to the MSS data source for processing. However, avoid nesting CAN–FIND functions.
- Avoid using the RECID function. Instead, use the ROWID function.

2.10.6 Modifying the Run-time Schema Check

At runtime, the DataServer for SQL Server™ performs a schema check to match a table definition in the data source against the schema image in the schema holder. It performs this task once for each table utilized in a Progress procedure. In order to retrieve table attributes without having to access data, the DataServer executes the following SQL statement against the data source:

```
SELECT * FROM table-name WHERE 1 = 2.
```

Skip Schema Verification

When r-code runs (each time a table, view, or buffer is opened), the DataServer checks the data definitions of the MSS data source to ensure the data definitions match the schema definitions in the schema holder. If the definitions do not match, the DataServer returns an error.

Unmatched definitions can cause corruption of data. For this reason, checking the integrity of data definitions at run time ensures the data corruption due to unmatched definitions will not occur. The skip schema check feature can be used to bypass this check at run time. Because definition verification is time consuming in a production environment, you might consider using the -Dsrv skip_schema_check startup parameter if your environment allows. You might consider using this option to increase performance, but only if you are certain that the data definitions in the data source match your schema holder definitions.

NOTE: The dataserv.lg log file denotes when the DataServer skips the schema check.

The following example shows how to use the -Dsrv parameter with the skip schema check option in the CONNECT statement.

SYNTAX

```
CONNECT data-source-name -ld logical-name -dt MSS

-Dsrv skip_schema_check
```

CAUTION: If you use the skip schema check option, the DataServer skips the schema check and does not detect discrepancies between the schema definitions and the data definitions. If there are discrepancies and the DataServer continues to process queries, inserts, and deletions, your data may become corrupted. Progress Software Corporation recommends that you weigh carefully the performance benefit against the risk to your database before deciding to use -Dsrv skip_schema_check.

3

Configuring the DataServer

Configuring the DataServer for SQL Server™ involves starting executables for several processes. This chapter provides step-by-step instructions for initially setting up the DataServer. Specifically, it describes how to:

- Configure the ODBC drivers and data sources
- Configure Local DataServer components
- Configure Remote DataServer components
- Create, maintain, and deploy a schema holder

Before you configure a DataServer, make sure that you have installed all of the required software. For details, see the “Software Requirements” section in Chapter 1, “Introduction.”

3.1 DataServer Components

The DataServer for SQL Server™ is available in two versions:

- The Enterprise DataServer for SQL Server™ can run in a variety of configurations. Some configurations for this version of the DataServer involve a single process running on one machine. Others involve multiple processes running on different machines across multiple platforms.
- The Personal DataServer for SQL Server™ runs only locally on NT or Windows.

For more details on the differences between the two versions, see Chapter 1, "Introduction."

Before you configure a DataServer, you must register your SQL Server™ database as an ODBC data source. For details, see the "Registering Your Data Source" section.

To set up a DataServer configuration, determine which components you need on which platforms, then set up the appropriate executables on those platforms. Table 3–1 lists the possible combinations and describes which executables you must set up on each machine. In this table, the term local indicates that the DataServer component runs on the same machine as the client, while the term remote indicates that the component runs on a different machine than the client.

Table 3–1: Installing the DataServer Components *(1 of 2)*

Client	DataServer	Installing and Configuring
NT	Local NT	Use the default Progress client executable (`prowin32.exe`). The server component is embedded in the client executable.
Windows	Local Windows	Use the default Progress client executable (`prowin32.exe`). The server component is embedded in the client executable.

Table 3–1: Installing the DataServer Components *(2 of 2)*

Client	DataServer	Installing and Configuring
NT or Windows	Remote NT	On the **client machine**, use the default Progress client executable (`prowin32.exe`). On the **host machine**, do one of the following to make a connection to the default DataServer executable (`_msssrv.exe`): – Use the default broker executable (`_probrkr.exe`) from the command line or configure it using ProControl. – Use the Java-based broker executable (mssman) from the command line or configure it using the Progress Explorer. This configuration is available only on the Enterprise DataServer for SQL Server™.
UNIX	Remote NT	On the **client machine**, use the default Progress client executable (`_progres`). On the **host machine**, do one of the following to make a connection to the default DataServer executable (`_msssrv.exe`): – Use the default broker executable (`_probrkr.exe`) from the command line or configure it using ProControl. – Use the Java-based broker executable (mssman) from the command line or configure it using the Progress Explorer. This configuration is available only on the Enterprise DataServer for SQL Server.

NOTE: You can optionally build custom DataServer executables using the Progress PROBUILD utility. See Appendix C, "Building DataServer Executables," for instructions. This is typically useful only on UNIX platforms.

For instructions on setting up your DataServer configuration, see the sections that apply to the platforms that you will use. For example, if you are configuring a local DataServer, see the "Configuring a Local DataServer" section. If you are building a remote DataServer configuration host, see the "Configuring the Remote DataServer" section.

3.2 Configuring the ODBC Driver and Registering the Data Source

Before you can configure the Progress DataServer:

- You might need to install your ODBC drivers and the client connectivity component from SQL Server™.
- You must set up your data sources.

If you installed the Enterprise DataServer for SQL Server™, the ODBC driver for SQL Server™ is bundled with the DataServer and installed automatically.

If you installed the Personal DataServer for SQL Server™, you must obtain the drivers from the vendor and install them separately.

3.2.1 Configuring the ODBC Driver

The DataServer relies on the ODBC driver to complete the connection to the data source, so it is important that you configure the driver as specified by the vendor.

For the Enterprise DataServer, refer to the *DataDirect Connect ODBC Reference* on the Progress CD for driver configuration information.

For the Personal DataServer, refer to the configuration information provided by your ODBC driver vendor.

3.2.2 Registering Your Data Source

On NT, before you can use your data source with the DataServer for SQL Server™, you must register it as an ODBC data source using the ODBC administration tool. To do this, follow these steps:

1 ♦ Start the ODBC administration tool for your data source. This can be either the administration tool provided by Microsoft or a repackaging of that tool by a non-Progress data source vendor. Also, some vendors might provide a similar administration tool.

2 ♦ Select the appropriate ODBC driver.

3 ♦ Select your SQL Server™ database.

4 ♦ Specify a name for your data source.

5 ♦ Set other configuration options required or optionally allowed through the driver vendor's configuration utilities against the target database.

6 ♦ Test connect.

7 ♦ Exit the ODBC administration tool.

You can now configure the DataServer, as described in the following sections.

3.3 Configuring a Local DataServer

In a local configuration, all DataServer components—client and server—run on the same machine. (The server component is embedded in the client executable.) This section describes how to configure a local DataServer.

Windows 95 and Windows 98-based local configurations are available only with the Personal DataServer for SQL Server. Windows NT-based local configurations are available with either the Personal or Enterprise DataServer for SQL Server™.

To configure the local DataServer on Windows, follow these steps:

1 ♦ Install the ODBC software and the client software on the system where your local DataServer resides.

NOTE: If you are configuring the Personal version of DataServer for SQL Server™ with either WebSpeed or AppServer, your local DataServer system is the system where the WebSpeed or AppServer broker resides and executes. You must install your Personal version of the DataServer for SQL Server™ at that location.

2 ♦ Make sure that you registered the data source with the ODBC driver correctly on the appropriate machine.

Once you have set up your environment, you can build the schema holder for your SQL Server datatbase™ and connect using the client executable. See the "Creating a Schema Holder" section for instructions.

3.4 Configuring the Remote DataServer

In configurations that include a remote host, **two** DataServer processes run on the host machine:

- **Broker** — The DataServer broker (`_probrkr.exe`) or the broker for the Progress Explorer on the host machine determines the types of requests coming over the network and starts (spawns) the appropriate DataServer (`_msssrv.exe`) for the client process.
- **DataServer** — The DataServer (`_msssrv.exe`) on the host machine accesses the SQL Server™ database and communicates with the client process.

NOTES:

- The Progress SQL Server™ DataServer component is an ODBC **client** with respect to the SQL Server database™ configuration even though from the Progress point-of-view it resides on the **server** machine. As a result, your SQL Server™ client software must reside on the machine from which the DataServer and the broker processes will execute. In the remote DataServer configuration, the Progress client component requires no special software or configuration; it only requires a standalone Progress NT or UNIX client.
- There are no Progress restrictions regarding the location of the actual SQL Server™ database.

Before you can run the server components, you must configure the DataServer by setting the required environment variables on the host machine. On NT, you configure using ProControl or the Progress Explorer tool. See the "Configuring with ProControl" or the "Configuring with the Progress Explorer" section for details.

NOTE: Configurations that use a remote DataServer on NT are available only with the Enterprise DataServer for SQL Server™.

3.4.1 Configuring with the Progress Explorer

On NT, you can use the Progress Explorer tool to configure the DataServer for SQL Server™.

NOTE: This section describes how to access the Progress Explorer to **configure** the DataServer for SQL Server™ on NT. It does **not** explain how to **connect** the DataServer. For connection instructions, see the "Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client" section in Chapter 4, "Connecting the DataServer." For information about the Progress Explorer, see the Progress Explorer online help.

Follow these steps to access the Progress Explorer:

1 ♦ Make sure that the Progress AdminServer is running.

2 ♦ Start the Progress Explorer.

From the Windows Desktop choose Start→ Programs→ Progress→ Progress Explorer Tool. The Progress Explorer appears in the MMC framework.

3 ♦ Connect to localhost.

4 ♦ From the Progress Explorer's left pane, select the MSS DataServer folder (where MSS stands for Microsoft® SQL Server™) and double-click. The list of existing DataServer brokers for SQL Server™ appears in the right pane.

5 ♦ Select the DataServer instance whose properties you want to create or edit, and right-click. A pop-up menu appears.

NOTE: The DataServer for SQL Server™ installation provides one predefined DataServer Broker (mssbroker1) and one predefined NameServer (NS1). Each broker is referred to as an instance. See the Progress Explorer online help for more information. You can use these predefined components as a starting point for creating and configuring additional DataServer Brokers, and, if needed, NameServers. (See the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX* for information about the NameServer's role in a configuration.)

6 ♦ Choose the Properties option from the pop-up menu. The Properties dialog box for that instance appears.

7 ♦ Configure the DataServer broker by setting general DataServer properties, owner information properties, application service properties, logging settings, environment variables, and others. For details on these settings, see the Progress Explorer online help.

NOTE: By default, your DataServer for SQL Server™ broker instances are defined with a controlling NameServer and are provided with a default Data Service. Progress Software Corporation recommends using a NameServer configuration at all times. In such cases, the DataServer client's initial connection is to the NameServer. However, you can alternatively connect the DataServer directly to the broker instance by setting the -DataService value to **none** in the connection parameters of your schema holder. If you will always use a -DataService value of **none**, you should remove the controlling NameServer from your broker instance definition. See the "Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client" section in Chapter 4, "Connecting the DataServer" for more information about connecting the DataServer to the NameServer and the broker.

Do not simultaneously run some DataServers for SQL Server™ under brokers with controlling NameServers and others directly under brokers (that is, without controlling NameServers). This defeats the purpose of using a NameServer to control brokers. If you do this, the benefits of the NameServer are lost and load balancing is ineffective. Progress Software Corporation recommends that you always use a NameServer, with one exception: you can choose initially to connect directly to a broker to simplify confirming an initial connection. Once you establish a connection, Progress recommends that you reintroduce the NameServer into your configuration.

3.4.2 Configuring with ProControl

This section describes how to configure the remote DataServer on the NT host using the ProControl utility. The Progress ProControl utility provides a graphical interface that allows you to manage and configure the DataServer host environment interactively.

If you have not used ProControl before, see the ProControl online help for security issues. For example, depending on whether a drive is configured as FAT, you might need to specify in the NT Services utility the account that users will use to log on.

To configure the DataServer components on the NT host using ProControl, follow these steps:

1 ♦ Access ProControl by double-clicking its icon in the NT Control Panel. The ProControl dialog box opens.

2 ♦ Choose Detail. Tabbed folders appear.

3 ♦ Choose the Options tab and activate the DataServer Page toggle box.

4 ♦ Choose OK and then choose Detail on the ProControl main window. The DataServers tab appears.

5 ♦ Choose the DataServers tab and choose Insert to add an entry for the broker. The General and Environment tabs appear.

6 ♦ Enter the appropriate information in the following fields in the General folder:

- **Identifiers** — Provide a unique name to identify the broker. The name must be unique to the host. The identifier can be up to 30 characters long. For example, you might name it mssbroker.
- **Executable Name** — Specify the broker executable. By default, the `_probrkr.exe` executable displays. If you want to use a customized executable that you built using the PROBUILD utility, specify it here.

- **Working Directory** — Specify the directory that the broker uses, for example, to create the `dataserv.lg` file.
- **Service Name** — Specify the value for the Service Name (-S) startup parameter. Look in your *install-path*`\system32\drivers\etc\services` file for an available service that the broker can use.
- **Network Protocol**— Specify TCP for the Network (-N) startup parameter.

7 ♦ Activate the Auto Start toggle box if you want the broker to start automatically when you boot the host machine. To start the broker automatically, you must also use the NT Services utility to set the Progress ProService utility to start automatically.

8 ♦ Choose the Environment tab in the DataServers folder.

9 ♦ In the Environment tab, set the MSSSRV variable to the DataServer executable that you are using. The default is `%DLC%/bin/_msssrv.exe`.

10 ♦ Choose OK.

11 ♦ Start the broker by selecting the identifier and choosing Start on the DataServers folder.

Once you have completely set up your environment, you can build the schema holder for your SQL Server™ database. See the "Creating a Schema Holder" section for instructions.

3.4.3 Configuring from the Command Line

You may configure a remote DataServer broker process from the command line. You may use the _probrkr.exe executable (the command line equivalent of using ProControl) or use the `mssman.bat` Progress Explorer utility (the command line equivalent of using the Progress Explorer).

Before you start the configuration tasks, make sure that you have installed your ODBC software and any data source-specific client software on your host machine. Also make sure that you have registered all of your data sources. See the "Configuring the ODBC Driver and Registering the Data Source" section for details.

To configure from the command line:

- When using the Progress broker (_probrkr.exe), you set the environment variables described in this section from the command line using environment-variable commands at the DOS shell prompt.
- When using the `mssman.bat` Progress Explorer utility, set the environment variables described in this section in the environment section of your broker properties file for the specific broker instance definition.
- You must set and export environment variables in the same environment (DOS shell) from which you plan to run the DataServer.

Table 3–2 describes the environment variables that you must set.

Table 3–2: Environment Variables for the Remote DataServer

Variable	Description
DSLOGDIR (Optional)	The pathname of the log file that Progress uses to keep track of DataServer processes and error messages. By default, Progress writes to `$DSLOGDIR/dataserv.lg` in the current directory.
MSSSRV	The pathname of the Progress server executable. This variable is required only when the DataServer is spawned using the Progress broker (`_probrkr.exe`). When using mssman, this setting is satisfied by the `srvrExecFile` setting in the broker properties file.
PROBRKR	The pathname of the broker executable. When using the Progress broker (`_probrkr.exe`), set this variable on the host machine to the broker executable name. When using `mssman.bat`, this setting is satisfied by the broker instance definition in the broker properties file.

Once you have completely set up your environment, you can build the schema holder for your SQL Server™ database and connect using the client executable. See the "Creating a Schema Holder" section for instructions.

The ubroker.properties File

When using the Progress Explorer administration framework, you configure the DataServer for SQL Server™ by editing the `%DLC%/properties/ubroker.properties` file. This file stores configuration definitions for all instances of the following Progress products:

- NameServers
- AppServers
- DataServers
- Databases
- WebSpeed Transaction Servers

When you use this file to configure the DataServer for SQL Server™, you provide information that enables the host to start a broker that spawns the appropriate DataServer process (`_msssrv.exe` or a custom executable that you built with PROBUILD).

Each configuration definition contains the environment variable and property settings for a broker instance. The command-line utilities use this file to store, validate, and manage the configurations for these brokers. A single copy of this file maintains all supported broker configurations for each Progress installation.

Editing the ubroker.properties File

NOTE: Be sure to preserve the original `%DLC%/properties/ubroker.properties` file. Rename the original file and work with a copy of the file. You must name the copy of the file `ubroker.properties`.

The simplest way to make configuration changes to the `ubroker.properties` file is to copy an existing broker or NameServer definition and then modify the copied values as required for your configuration. You must ensure that all related properties and sections of the file are properly specified for each broker or NameServer instance.

Table 3–3 describes the sections in the ubroker.properties file that apply to the DataServer for SQL Server™. The file configures a default NameServer named NameServer.NS1 and a default broker named mssbroker1, which you can use either without editing or as templates for your own configuration specifications.

Table 3–3: DataServer for SQL Server™ Sections of the ubroker.properties File *(1 of 2)*

Product	Section	Description
All products	ParentGroup	Defines the name of each NameServer and product broker parent entity.
	Environment	Default environment variable settings for all NameServers and product brokers.
	UBroker	Defines default property settings for all product brokers.
NameServer	NameServer	The parent entity of NameServers. It defines default property settings for all NameServer instances.
	NameServer.NS1	A sample property entity of a NameServer instance. It defines property settings for this NameServer instance.
	Environment.NS1	A sample environment entity of a NameServer instance. It defines environment variable settings for this NameServer instance.

Table 3–3: DataServer for SQL Server™ Sections of the ubroker.properties File *(2 of 2)*

Product	Section	Description
DataServer	UBroker.MS	The parent entity of DataServer for SQL Server™ brokers. It defines default property settings for all of these broker instances.
	UBroker.MS.mssbroker1	A sample property entity of a DataServer for SQL Server™ broker instance. It defines default property settings for the broker instance named mssbroker1. Note that although many of the settings in this section can work in your environment, some of them are for demonstration purposes only. You must specify the appropriate settings for the following: srvrExecFile: Specify the pathname of the DataServer executable that the broker runs. The default is %DLC%/bin/_msssrv.exe, which runs the default DataServer from your Progress install path. srvrStartupParam: Specify the Progress startup parameters for the DataServer. Do not modify the default parameters, although you can add parameters if necessary. The -svub switch is required for a broker connection in the Progress Explorer administration framework.
	Environment.mssbroker1	A sample environment entity of a DataServer for SQL Server™ broker instance. It defines environment variable settings that apply to the broker instance named mssbroker1. Be sure to set the variables to the appropriate values; the values in the file are for demonstration purposes only. Add to this section any environment variables that you want to apply to the DataServer's environment. See the "Configuring the Remote DataServer" section for the environment variable settings required to configure the DataServer for SQL Server™.

The following example illustrates the DataServer sections of the ubroker.properties file:

```
#
#  Default properties for broker instances serving MSS DataServer
#
[UBroker.MS]
    srvrExecFile="@{Startup\DLC}\bin\_msssrv.exe"
    srvrStartupParam=-svub -S X -N TCP -U X -P X -hs 0 -s 40
    operatingMode=State-aware
    classMain=com.progress.ubroker.broker.ubroker
    defaultService=0
    initialSrvrInstance=0
    minSrvrInstance=0
    maxSrvrInstance=256
    brkrLoggingLevel=3
    description=MSS DataServer Broker
#
# Sample MSS DataServer Broker definition
#
[UBroker.MS.mssbroker1]
    srvrExecFile="@{Startup\DLC}\bin\_msssrv.exe"
    srvrStartupParam=-svub -S X -N TCP -U X -P X -hs 0 -s 40
     srvrLogFile=@{WorkPath}\mssbroker1.server.log
     brokerLogFile=@{WorkPath}\mssbroker1.broker.log
     portNumber=4446
     defaultService=1
     appserviceNameList=mssbroker1
     controllingNameServer=NS1
     environment=mssbroker1
     uuid=172.18.103.53:1f415c:d6330e5d24:-7f1e
     description=A sample MSS DataServer Broker

#
# Environment for MSS Dataserver Broker: odbbroker1
#
[Environment.mssbroker1]
    DSLOGDIR={WorkPath}
```

For a complete description of the parameters included in each of these sections, see the comments in the %DLC%/properties/ubroker.properties file.

The ubroker.properties file is read on startup of the AdminServer process. For changes in any used environment variables to take effect, the AdminServer must be restarted.

Validating ubroker.properties Content

Whenever you create your own version of the `ubroker.properties` file, you should use the relevant validation utility to validate your changes and make sure that there are no syntax errors or conflicts. When configuring the DataServer for SQL Server™, you validate the file by using the `mssconfig.bat` utility. See the "MSSCONFIG Utility" entry in Appendix D, "Server Related Command Line Utilities and Startup Parameters," for more information.

Configuring Multiple Brokers

You can configure multiple brokers by adding more `UBroker.MS.`*`broker-name`* and `Environment.`*`broker-name`* sections. Each broker instance must have a unique name. The broker instances inherit the properties that you define in the `UBroker.MS` (parent entity) section, but can override them.

If you want to access multiple SQL Server™ databases and configure one or more brokers, each broker must run in a separate directory and a distinct environment.

Using the NameServer

By default, your DataServer for SQL Server™ broker instances are defined with a controlling NameServer and are provided with a default Data Service. Progress Software Corporation recommends using a NameServer configuration at all times. In such cases, the DataServer client's initial connection is to the NameServer. However, you can alternatively connect the DataServer directly to the broker instance by setting the -DataService value to **none** in the connection parameters of your schema holder. If you will always use a -DataService value of **none**, you should remove the controlling NameServer from your broker instance definition. See the "Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client" section in Chapter 4, "Connecting the DataServer", for more information about connecting the DataServer to the NameServer and the broker. See the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX* for more information about the NameServer's role in a configuration.

NOTE: **Do not** simultaneously run some DataServers for SQL Server™ under brokers with controlling NameServers and others directly under brokers (that is, without controlling NameServers). This defeats the purpose of using a NameServer to control brokers. If you do this, the benefits of the NameServer are lost and load balancing is ineffective. Progress Software Corporation recommends that you always use a NameServer, with one exception: you can choose initially to connect directly to a broker to simplify confirming an initial connection. Once you establish a connection, Progress recommends that you reintroduce the NameServer into your configuration.

3.5 Creating a Schema Holder

A schema holder contains a schema image of the SQL Server™ database that your DataServer application accesses. The schema image contains the schema for your data source—that is, its data definitions—in a Progress format. Although Progress applications can access a data source directly by sending SQL statements through the DataServer, they cannot directly access its data definitions and must use the schema-image information instead.

When you decide where to build your schema holder, consider your business needs and deployment strategy. The schema holder can reside on the client machine or on any host in an enterprise configuration. Client schema holders improve performance for some applications. Host schema holders give you the flexibility of having them run as NT services. Additionally, host schema holders might be easier to maintain. In a WebSpeed configuration, placing the schema holder on the Transaction Server machine optimizes performance.

In general, you must do the following to create a schema holder:

1. Establish the appropriate permissions for pulling the schema image into the schema holder.
2. Perform some preparatory steps.
3. Create and then connect an empty Progress database. This database becomes your schema holder and contains your schema image.
4. Create the schema holder, which involves specifying connection parameters and pulling the schema from the data source.

The following sections describe these steps in detail.

3.5.1 Establishing Permissions

When you use the DataServer to create a schema image in a schema holder, it accesses the associated SQL Server™ database. This section describes the data source permissions required when you create, update, or connect to a schema image.

Permissions for Creating or Updating a Schema Holder

You must have **select** privileges on syscolumns, sysindexes, sysobjects and sysusers' tables in your SQL Server™ database to create or update a schema image. This is because the DataServer utilities must access these tables, but they cannot do so without **select** privileges. However, this is the only time that you need this level of privilege when using the DataServer.

Permissions for Connecting a Schema Holder

When you connect to a schema holder, you can optionally also connect to your SQL Server™ database. To connect to the data source, you must provide a valid login name and password combination. Use the User ID (-U) parameter to provide the user ID and the Password (-P) parameter to provide the password.

Application-specific Permissions

In addition to the SQL Server™ database permissions required by the DataServer, the required permissions for users depend on the applications that they are using. For example, a user who is running a Progress application that queries but does not update the employee table in the data source must connect to the data source with a login name and password combination that provides at least **select** privileges for the employee table. For users who will manipulate data at runtime, the appropriate read, insert, update, and delete permissions must be granted as administered by the foreign (target) data source.

In summary, the login name (or user ID) and password combination required to run a particular application depends on the following:

- The tables that the application accesses
- The type of access required to those tables

3.5.2 Preparing to Create the Schema Holder

To prepare to create the schema holder, follow these steps:

1 ♦ Verify that your SQL Server™ database is accessible and that you can connect to it. Transact-SQL and the SQL Server™ query analyzer can serve as a test for connectivity.

2 ♦ Verify that you have installed the ODBC drivers.

3 ♦ Once you have configured your data source (DSN), make sure that you can establish a connection independent of using the Progress DataServer. From your DSN configuration, select to test the connection to SQL Server™ using the ODBC interface.

4 ♦ Start the DataServer as described in either the "Starting a Local DataServer" section or the "Starting a Remote DataServer" section in Chapter 4, "Connecting the DataServer," whichever is appropriate for your configuration.

5 ♦ Open the Progress Data Administration tool or the character Data Dictionary.

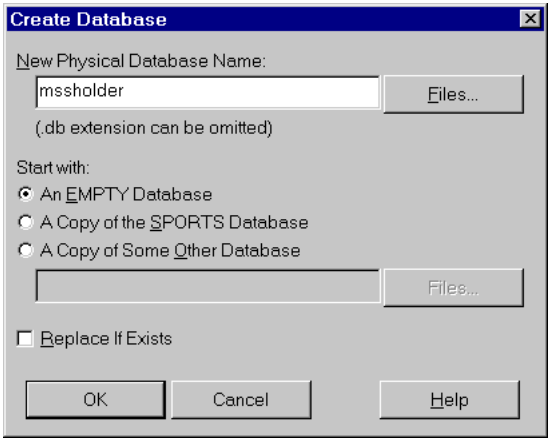
3.5.3 Creating an Empty Progress Database

The DataServer uses the empty Progress database as a holder for the schema for your SQL Server™ database. The simplest way to create an empty Progress database is to do it from the Data Administration tool, as described in this section. For information on other ways to create an empty Progress database, see the *Progress Database Administration Guide and Reference*.

To create and connect an empty Progress database from the Data Administration tool, follow these steps:

1 ♦ Start the Data Administration tool.

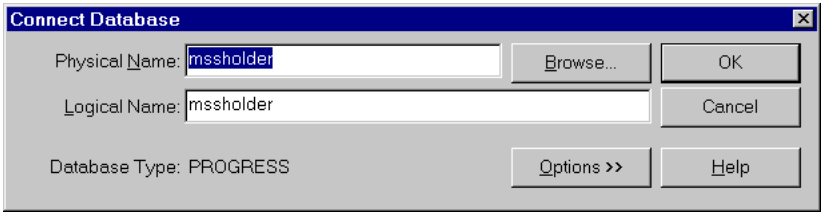
2 ♦ Select the Create Database option. The Create Database dialog box appears:



3 ♦ Type the schema-holder name (for example, mssholder) in the New Physical Database Name field.

4 ♦ Select the An EMPTY Database option.

5 ♦ Choose OK. The following dialog box appears. By default, the name of the newly created data source appears in the Physical Name field:



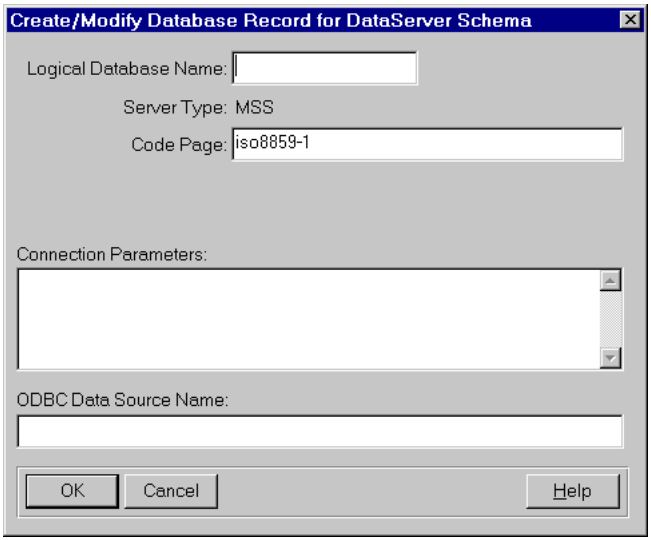
You do not have to provide any additional connection information. You can add connection parameters when you create the data source or edit connection information later. See the online help for a complete description of the Connect Database dialog box.

6 ♦ Choose OK to connect the empty Progress database and return to the Data Admin main window.

3.5.4 Creating a Schema Holder

Follow these steps to create the schema holder:

1 ♦ From the Data Administration main menu, select DataServer→ MS SQL Server Utilities→ Create DataServer Schema. The following dialog box appears:



2 ♦ In the Logical Database Name field, type the name that you will use to connect to your data source and refer to it in your programming applications. This name must be different from the schema holder name. For more information on database names, see the database access chapter in the *Progress Programming Handbook*.

NOTE: If you place the schema from a second SQL Server™ database into a schema holder, the second schema must have a different logical database name from the first schema. The schema holder has one physical name, but each schema that it contains must have a different logical name.

3 ♦ In the Code Page field, type the name of the code page for the schema holder. The name must be the Progress name for the code page that the data source uses. The default is iso8859–1.

Table 3–4 lists the most common SQL Server™ database code pages and the equivalent Progress names.

Table 3–4: SQL Server™ and Progress Code Pages

SQl Server™ Code Page	Progress Equivalent
iso_1	iso8859–1 (default schema-holder code page)
cp850	ibm850

If you use a code page that Progress does not support, you must supply a conversion table that translates between the Progress client code page and the code page that your data source uses. For a complete discussion of code pages, see the *Progress Internationalization Guide*.

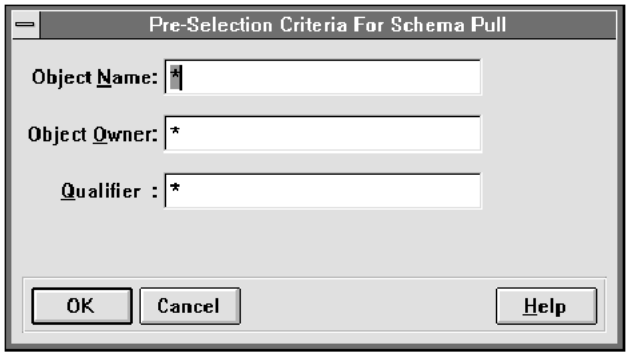
4 ♦ Type the connection parameters in the Connection Parameters field.

See Chapter 4, "Connecting the DataServer," for a description of the required and optional connection parameters.

5 ♦ In the ODBC Data Source Name field, type the name that you used when you registered the data source with the ODBC administration tool.

6 ♦ Choose OK. The utility prompts you for your data source user ID and password. Enter them if you did not provide them in the Connection Parameters field (see Step 4).

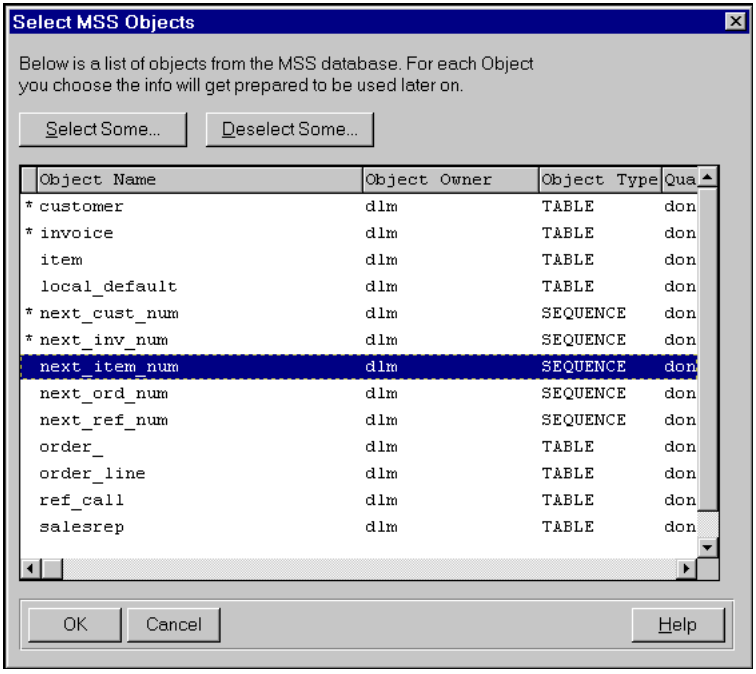
7 ♦ Choose OK. When the DataServer connects to the SQL Server™ database, it reads information about data source objects. The following dialog box appears:



You can select tables based on the object name, owner name, and qualifier. For example, you can specify A* in the Object Name field to list all the tables whose names begin with A.

NOTE: Progress Software Corporation recommends that you do not specify an entry that consists exclusively of wild cards for each of the three entry fields in the dialog box. An entry that consists exclusively of wild cards might degrade the performance of the database when you perform a schema pull. (It will include system catalog files from the data source not typically included in user databases.)

8 ♦ Choose OK. Progress displays a list of the data source objects that you can include in the schema holder:



If you specified all wild cards as your table-selection criteria, the list might also include system-owned objects, which you do not have to include in the schema holder.

9 ♦ Select the objects that you want to include in the schema holder and choose OK. The DataServer reads information about the objects that you select and loads their data definitions into the schema holder. The time that this process takes depends on the size and number of objects that you select.

For each table, the DataServer attempts to select an index to support the Progress ROWID. If an appropriate index does not exist, the DataServer issues the warning, "Please check warnings and messages in the file ds_upd.e." The `ds_upd.e` file lists the objects that do not support ROWID. You can change the DataServer's selection of an index to support ROWID by using the Progress Data Dictionary. See the "Defining the ROWID" section in Chapter 5, "The DataServer Tutorial," for instructions. For additional information, see the "Indexes" and "ROWID Function" sections, respectively, in Chapter 2, "Programming Considerations."

3.6 Maintaining a Schema Holder

The Progress Data Administration tool provides a set of DataServer utilities that you can use to maintain a schema holder. Chapter 5, "The DataServer Tutorial," describes these utilities. Recall that you must have select privileges on certain data source objects to update a schema holder; see the "Establishing Permissions" section for details.

If you make changes to a SQL Server™ database, make sure to update the associated schema holder to reflect those changes if you want them to be accessible to a DataServer application. Note that you do not need to update the schema holder if the application will never access data objects affected by the change. For example, if you add a table object that a DataServer application will never access, you do not need to update the schema holder.

Each time that you update the schema holder, you must recompile your DataServer application (`.p` and `.w` files) to generate new r-code.

3.6.1 Updating a Deployed Schema Holder

The guidelines and techniques that apply to deploying a Progress database also apply to deploying a schema holder for a SQL Server™ database. However, if you make any changes to the data source schema, make sure to update its schema holder. There are two techniques for updating a deployed schema holder:

- Allow your users to use the DataServer Update/Add Table Definitions utility.
- Send a new data definition file for the schema holder. Your users can use the DataServer Delete Schema utility to empty the original schema holder. They can then load the new data-definition file into the schema holder.

Follow these general steps to update a deployed DataServer application:

1 ♦ Run a SQL script to make changes to the SQL Server™ database.

2 ♦ Run the Update/Add Table Definitions utility on one schema holder.

3 ♦ Recompile code against the updated schema holder to build new r-code.

4 ♦ Send out copies of the new `.r` files to your users.

5 ♦ Distribute copies of the new schema holder `.db` and `.bi` files to your users. You must use the Progress PROCOPY utility to distribute them.

4

Connecting the DataServer

You can start and connect a DataServer using one of the following methods:

- Progress Explorer tool (Windows platforms only)
- Progress Explorer Command line utilities (MSSMAN and MSSCONFIG)
- ProControl utility (Windows platforms only)
- Command line statements (`_probrkr.exe`)

This chapter describes:

- Starting a local or remote DataServer using one of the above listed methods
- Connection guidelines
- Connecting a schema holder
- Connection failures and Progress responses

4.1 Starting a Local DataServer

This section describes how to start a local DataServer for SQL Server™ from the startup command.

Starting the Progress client starts the local DataServer. You include information about the ODBC data source, user ID, and password in the startup command.

To start the client, follow these steps:

1 ♦ Make sure that your ODBC drivers are installed and configured properly.

2 ♦ Start your SQL Server™ database.

3 ♦ Set any environment variables required for your configuration as described in the relevant section of Chapter 3, "Configuring the DataServer" (for example, DSLOGDIR). You must set them in the shell from which you are starting the DataServer.

NOTE: If you change the values of any environment variables, you must shut down the DataServer processes and restart them.

4 ♦ Enter the following command:

```
prowin32 schema-holder-name -db datasrc-name -dt MSS
  -ld logical-datasrc-name -U userID -P password
```

For example, the following command starts Progress with the local DataServer, connects a local schema holder named mssholder in single-user mode, and connects the SQL Server™ demo database with the user bob whose password is bobpass:

```
prowin32 mssholder -db demo -dt MSS -ld demo -U bob -P bobpass
```

4.2 Starting a Remote DataServer

This section describes how to start and stop a remote DataServer for SQL Server™. (Note that these procedures apply only to an Enterprise DataServer for SQL Server™.) Starting a remote DataServer involves starting the processes that make up the remote DataServer, as follows:

1 ♦ Start the DataServer broker process on your host machine. For details, see one of the following sections:

- "Starting and Stopping a Broker Process with ProControl"
- "Starting and Stopping a Broker Process from the Command Line"
- "Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client"

2 ♦ Start a Progress client process on a UNIX machine or on a PC running Windows that connects to the schema holder and the SQL Server™ database.

4.2.1 Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client

You can use the Progress Explorer to administer the server module (`_mssrv.exe`) of the DataServer on NT.

On the NT Host

Before you attempt to start the DataServer in the Explorer, be sure that you have configured it completely. After starting the broker from the Progress Explorer, you start your Progress client as you would in any remote DataServer configuration.

To start and stop the DataServer from the Explorer, see the Progress Explorer online help.

On the Client

After you start the broker on the host machine from the Progress Explorer, you can connect your UNIX or Windows client. Use the same parameters that you would use to connect to the schema holder and SQL Server™ database in a standard probrkr configuration. In addition:

- Include the -Dsrv SVUB,1 parameter. This parameter allows you to connect to the broker administered by the Explorer.
- Include the -DataService *data–service* parameter to connect through a NameServer to the broker. The value for *data–service* must specify a valid name from the DataService list registered to this NameServer as defined by your *appServiceNameList* entry in the broker

properties file. If a default DataService has been defined for your broker instance, you can omit this parameter and connect using the default service.

NOTE: **For diagnostic purposes only,** it is acceptable to bypass the NameServer connection and connect directly to the broker instance. To do this, specify the reserved DataService name “none” as follows:

```
-DataService none
```

- Set the -S parameter to one of the following:
  - The port number assigned to the controlling NameServer (when the -DataService value is not “none”) or the port number of the broker instance that you started in the Explorer (when the -DataService value is “none”)
  - The service name in your `services` file whose associated port matches the port of the controlling NameServer (when the -DataService value is not “none”) or the broker instance that you started in the Explorer (when the -DataService value is “none”)
- Set the -H parameter to the name of the NT machine where the NameServer and/or broker instance are running.

If you do not set the required -Dsrv SVUB,1 and optional -DataService *data–service* connection parameters as described in this section, the client is assumed to be configured for a standard Progress broker and the -H and -S parameters are used to locate a probrkr executable on the appropriate host machine. By setting the SVUB parameter on, you redirect the -H and -S parameters to locate the appropriate NameServer and/or broker on the host machine. The following example illustrates how to use these connection parameters for a client that connects to a NameServer:

```
CONNECT mssholder -db mssdemo -dt MSS -U bob -P bobpass -H host1
-S mssport -DataService mssbroker1 -Dsrv SVUB,1
```

4.2.2 Starting and Stopping a Broker Process Using MSSMAN

You can use the MSSMAN utility to start a DataServer broker on an NT host. To start and shut down the DataServer, follow these steps:

1 ♦ To start the DataServer broker, enter this command at the system prompt on the machine where the broker will run:

```
mssman -name broker-name -start
```

In this command, *broker-name* is the name that you specified for the broker when you configured it. Optionally, you can indicate a user account by specifying `-user` *user-name*.

If you want to run the broker from a remote machine, you must specify additional options that identify the remote host, as follows:

```
mssman -name broker-name -start -host host-name -user user-name
```

In this command:

- *broker-name* is the name that you specified for your DataServer broker instance when you configured your `ubroker.properties` file.
- *host-name* is the name of the host machine on which you want the broker to run.
- *user-name* is the user ID of the system account under which the broker will run.

2 ♦ To stop the DataServer broker, enter this command at the system prompt on the machine where the broker will run:

```
mssman -name broker-name -stop
```

You can stop a broker on a remote machine by adding the `-host` and `-user` options.

See the "MSSMAN Utility" entry in Appendix D, "Server Related Command Line Utilities and Startup Parameters," for a description of all the command options.

4.2.3 Starting and Stopping a Broker Process with ProControl

If you configured a broker for the DataServer on the NT host using the ProControl utility, you can use ProControl to start and stop the broker process. To do this, follow these steps:

1 ♦ Start the broker by choosing Start on the DataServers folder. The broker spawns the DataServer process when a client requests a connection.

To start the broker subsequently, select it from the selection list and choose Start.

2 ♦ Stop the broker by choosing Stop on the DataServers folder.

Alternatively, you can start and stop an NT broker process from the command line. For details, see the "Starting and Stopping a Broker Process from the Command Line" section.

4.2.4 Starting and Stopping a Broker Process from the Command Line

You may start and stop a broker process from the command line. To do this, follow these steps at the DOS shell prompt:

1 ♦ Set the environment variable MSSSRV to the name of the executable (including the path) of the DataServer for SQL Server™. Be sure to set this variable on the host machine.

2 ♦ To start the DataServer broker process, enter the following command at the system prompt on your NT host machine. Select a value for *service–name* from the list of available services in your *windows-install-dir*\system32\drivers\etc file:

```
brokername -H host-name -S service-name -N TCP
```

For example, the following command uses the default broker executable. The service name demosv is a service listed in that file:

```
_probrkr -H paris -S demosv -N TCP
```

3 ♦ To stop the broker process, enter this command:

```
proshut -Gw -H host-name -S service-name -N TCP
```

After you start the NT broker process, you are ready to start a Progress client on a PC running Windows or NT or on a UNIX machine. See the "Starting the NT or Windows Client Process" section or the "Starting the UNIX Client Process" section for instructions.

4.2.5 Starting the NT or Windows Client Process

Start the Progress client process on your Windows machine by running the `prowin32.exe` executable. The Windows executable includes support for the DataServer.

You can create a program icon for the Progress client process. On the command line for the program icon, enter the following information:

1. The executable.
2. The schema holder name. If you have multiple schema holders, create a program icon for each schema holder.
3. The connection parameters required by the remote DataServer configuration.

For example, a command line for a Progress client process that you use to access a SQL Server™ database might look like this:

```
prowin32 mssholder -RO -db demo -dt MSS -ld demo -H host1 -S oserviceA
  -N TCP -U bob -P bobpass
```

See the "Connecting a Schema Holder at Startup" section for command-line information and more examples.

4.2.6 Starting the UNIX Client Process

After starting the remote broker, you start the Progress client process on your UNIX machine by running this executable:

```
pro
```

You can supply the connection parameters required by the DataServer when you start the client process, or you can include them in the Connection Parameters field when you create a schema holder.

For example, this command starts the Progress client, connects a read-only schema holder named mssholder, and connects the SQL Server™ demo database with the user bob whose password is bobpass:

```
pro mssholder -RO -db demo -dt ODBC -ld mssdemo -H host1 -S oserviceA
  -N TCP -U bob -P bobpass
```

See the "Connecting a Schema Holder at Startup" section for descriptions of the required command line.

4.3 Connection Guidelines

When you create a schema holder, you specify certain connection information, such as the physical and logical names for your SQL Server™ database and the user and password information. Progress stores this information so that you do not have to provide it each time you connect.

After you consider your security requirements, follow these general steps for connecting a schema holder and a SQL Server™ database:

1 ♦ Connect the schema holder.

2 ♦ Connect the data source.

4.3.1 Connections and Security on NT

When you configure an ODBC data source on NT, you can choose one of three levels of security:

- **Standard** — Requires that a client pass a user ID and password that the SQL Server™ database validates against the list of users in the syslogins table. The request typically comes from a nontrusted connection, such as through TCP/IP. The Progress client or WebSpeed agent passes this information with the User ID (-U) and Password (-P) connection parameters.

- **Integrated —** Relies on the NT operating system to validate user information. The SQL Server™ database accepts user information from the process that requests a connection. The request must come from a trusted connection, such as one using Named Pipes. The SQL Server™ database performs no additional validation.

- **Mixed** — Accepts requests from trusted or nontrusted connections.
  - If the connection is trusted and the client provides no user ID, a user ID that consists entirely of spaces, or a user ID that matches the user that started the process, the SQL Server™ database accepts the connection.
  - If the connection is nontrusted, the Progress client must provide the user ID and password.

Progress Software Corporation recommends the following guidelines for working with a SQL Server™ database and NT security:

- Configure a SQL Server™ database to use Standard or Mixed security if you are using remote Progress clients.
- If you are using Mixed security, always have the clients specify the -U and -P connection parameters.

4.4 Connecting a Schema Holder

As described previously, connection can take place automatically using the information that you enter when you create a schema holder. In addition, Progress provides the following techniques for connecting to a schema holder:

- Use the Progress Data Dictionary or Data Administration tool. From the main menu, select Database→ Connect and supply the schema holder's physical name and the appropriate connection parameters. You can then select your SQL Server™ database as your working database and connect to it.
- Use the Progress 4GL CONNECT statement (see its reference entry in the *Progress Language Reference*). A CONNECT statement must first list the schema holder and related connection parameters, then the SQL Server™ database and related parameters.

This command connects a schema holder named mssholder and SQL Server™ database named mssdemo:

```
CONNECT mssholder -db mssdemo -dt MSS
-Dsrv qt_debug,EXTENDED
```

- Use connection parameters when you start Progress. You can include these parameters on the program item command line or in a parameter file that you specify when you start Progress. A parameter file is portable and easy to maintain. For information on creating a parameter file, see the *Progress Startup Command and Parameter Reference*.

You can use combinations of different connection techniques. For example, you can connect the schema holder at startup, then connect to the DataServer using the Progress CONNECT statement. If the proper connect information is stored in the schema holder, referencing a table in the SQL Server™ database will cause the DataServer to issue the connect statement automatically. Any combination of connection techniques works, as long as you follow these steps in the order shown:

1 ♦ Connect the schema holder first. If you are not updating the schema holder, you can specify the Read-only (-RO) connection parameter to enhance DataServer performance.

2 ♦ Connect to the SQL Server™ database.

If you connect to the schema holder and the SQL Server™ database in a single startup command or connection statement, be sure to specify parameters that affect the schema holder **before** the Database Name (-db) parameter. Specify only those parameters that affect the SQL Server™ database connection after the -db parameter.

The following section explains how to connect both a schema holder and a SQL Server™ database when you start up Progress.

4.4.1 Connecting a Schema Holder at Startup

Progress supports connection parameters that you can use to connect both the Progress schema holder and a SQL Server™ database at startup. These parameters control how your system connects to a database. If the DataServer runs in a remote configuration, your startup command or parameter file always includes parameters that control networking options.

Table 4–1 describes the database connection parameters that you use when you connect to a schema holder and a SQL Server™ database through the DataServer.

Table 4–1: DataServer Connection Parameters *(1 of 2)*

Parameter	Status	Description
Database Type MSS (-dt MSS)	Optional	Specifies that the type of the target data source is MSS (MicroSoft® SQL Server™).
Physical Database Name (-db)	Required	Indicates the name by which Progress recognizes the SQL Server™ database to which you want to connect. This name must match the name that you used when you registered the data source as a SQL Server™ database.
Logical Database Name (-ld)	Optional	Specifies the logical name of the SQL Server™ database. This is the name that you use to refer to the data source in your applications. You must use this parameter only when the logical data source name differs from its physical name. This name should match the logical database name that you defined in your schema holder. For example, your applications might refer to the SQL Server™ demo database as mydemo. In this case, the physical name is demo, and the logical name is mydemo.
Host Name (-H)	Required for remote DataServer	Indicates the name of the NT host machine in the network.
Service Name (-S)	Required for remote DataServer	Indicates the name of the service you are calling. If you use NameServer with Progress Explorer, specify the service name or IP address of the host machine where the NameServer resides. If you are using probrkr, ProControl, or the Progress Explorer without a NameServer, specify the service name or IP address of the host machine where the broker resides.
Network Type (-N)	Optional	Indicates the network type. TCP is the only possible value.
User ID (-U)	Required	Supplies the login name that the DataServer uses to log into the SQL Server™ database.

Table 4–1: DataServer Connection Parameters *(2 of 2)*

Parameter	Status	Description
Explorer Redirection (-SVUB)	Required for Progress Explorer connections	Redirects connection logic to the Progress Explorer instead of probrkr or ProControl. For more information see the "Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client" section.
Data Service (-DataService)	Required for Progress Explorer connections	Specifies the data service the NameServer uses. For more information, see the "Starting and Stopping a Broker Process from the Progress Explorer and Connecting a Client" section.
Password (-P)	Required	Supplies the password that the DataServer uses to log into the SQL Server™ database. Different login name and password combinations allow for different levels of user privileges.
Single–User Mode (-1)	Optional	Specifies that a schema holder is used in single-user mode. Single–User Mode is the default unless a server is started for the schema holder.
Read-Only (-RO)	Optional	Specifies that a schema holder is read-only. Connecting a schema holder as read-only increases processing speed at client startup time. It also allows multiple client processes on the same machine to access the schema holder without starting additional server processes.
Local Cache (-cache)	Optional	Specifies that you are using a local cache file for the schema holder. Create the cache file with the SAVE CACHE COMPLETE statement.
DataServer (-Dsrv)	See note	Specifies options with which you control your ODBC Driver and DataServer environment. See the "Query Tuning with Connection and Startup Parameters" section for more information. **NOTE:** When you specify a list of -Dsrv parameters, be sure not to include any spaces anywhere in this list.
Server Join (-nojoinbysqldb)	Optional	Specifies that the client evaluates and performs queries that have joins. This might slow performance, but it provides results that are consistent with Progress behavior. Use -nojoinbysqldb at compile time. It has no effect at run time.

You can create a parameter file for each database:

- For a local DataServer, the parameter file must contain the -db parameter and can optionally contain the -Dsrv, -U, and -P connection parameters, depending on the requirements of the data service.
- For a remote DataServer, the same parameter conditions apply as for a local DataServer. In addition, a remote connection must contain the -H and -S connection parameters.

You can add more startup and connection parameters than the ones listed—these are the typical parameters. For a complete list of parameters and for information on how to create a parameter file, see the *Progress Startup Command and Parameter Reference*.

Using a Local DataServer Configuration: Examples

Use the following general command syntax to start Progress:

- In single-user mode
- In a local DataServer configuration
- With a local schema holder connected
- With a SQL Server™ database connected

```
prowin32 schema-holder-name -1 -db datasrc-name -dt MSS
  -ld logical-datasrc-name -U userID -P password -Dsrv
  qt_debug,EXTENDED
```

You can type these commands on the command line of a program item property box.

The following examples start Progress in a local DataServer configuration. In these examples:

- The schema holder's physical name is mssholder and it is read-only.
- The physical data source name (and the SQL Server™ database name) is sports.
- The data source type is MSS (Microsoft® SQL Server™).
- The logical data source name is mysport.

- The user ID is bob.
- The password is bobpass.
- Assorted -Dsrv options are specified.

```
prowin32 mssholder -RO -db sports -dt MSS -ld mysport -U bob -P bobpass
-Dsrv qt_debug,EXTENDED
```

Using a Remote DataServer Configuration: Examples

This configuration is available only with the Enterprise DataServer for SQL Server™.

Use the following command syntax to start Progress:

- In single-user mode
- In a remote-DataServer configuration
- With a local schema holder connected
- With a SQL Server™ database connected

On a Windows Client:

```
prowin32 schema-holder-name -1 -db datasrc-name -dt MSS
  -ld logical-datasrc-name -H hostname -S service-name
  -N TCP -U userID -P password -Dsrv
  qt_debug,EXTENDED
```

On a UNIX Client:

```
pro schema-holder-name -db datasrc-name -dt MSS
  -ld logical-datasrc-name -H hostname -S service-name
  -N TCP -U userID -P password -Dsrv
  qt_debug,EXTENDED
```

The following examples start Progress in a remote DataServer configuration. In these examples:

- The schema holder's physical name is mssholder and it is read-only.
- The data-source name is sports.
- The data source type is MSS (Microsoft® SQL Server™).

- The logical data source name is mydemo.
- The host name is host1.
- The service name is mssport.
- The network type is TCP.
- The user ID is bob.
- The password is bobpass.

On a Windows Client:

```
prowin32 mssholder -RO -db sports -dt MSS -ld mydemo -H host1
-S mssport -N TCP -U bob -P bobpass
-Dsrv qt_debug,EXTENDED
```

On a UNIX Client:

```
pro mssholder -RO -db sports -dt MSS -ld mydemo -H host1
-S mssport -N TCP -U bob -P bobpass
-Dsrv qt_debug,EXTENDED
```

4.4.2 Unsupported Connection Parameters

You cannot use the following Progress connection parameters when connecting to a SQL Server™ database through the DataServer. If you do specify them, Progress and WebSpeed ignore them:

- Blocks in Database Buffers (-B)
- Before–image Filename (-g)
- Lock-table Entries (-L)
- Number of Users (-n)
- Buffered I/O—UNIX only (-r)
- Version 6 Query (-v6q)
- Raw I/O (-R)

4.4.3 Special Connection Issues

The DataServer (-Dsrv) parameter uses the PRGRS_CONNECT option to allow you to pass ODBC-specific information to the ODBC driver. A DataServer connection string contains all of the information needed to establish a connection. It consists of a series of keywords/value pairs separated by semicolons. The Progress DataServer for SQL Server™ passes the connection string specified by the PRGRS_CONNECT option directly through to the ODBC driver without modification. For more information, refer to the *Microsoft ODBC Programmer's Reference*. The parameter has the following syntax:

SYNTAX

```
-Dsrv PRGRS_CONNECT,connection-string;
```

ODBC-specific connection information is passed in *connection-string*. The connection string is separated from the option by a comma (,) and ends with a semicolon (;).

Value pairs within the connect string are delimited by a semicolon (;).

NOTE: PRGRS_CO is a valid abbreviation for PRGRS_CONNECT.

Use the PRGRS_CONNECT parameter in the following cases:

- To establish complex connections that require more than the Physical Database Name (-db), User ID (-U), and Password (-P) parameters, as follows:

  ```
  CONNECT datasrc-name -ld logical-datasrc-name -dt MSS
        -Dsrv PRGRS_CO,DSN=datasrc-name;server=servername;.
  ```

  For *datasrc-name*, supply the name of the SQL Server™ database. `Server` is a driver-specific keyword. The -Dsrv connection string is passed directly to the data source. The DataServer does not modify this value.

- To connect to a SQL Server™ database whose name has a blank space, which is not allowed by Progress, substitute the characters &^ for the illegal characters in the data source name. Progress ignores *datasrc-name* when you use PRGRS_CONNECT; however, you must supply it to pass syntax validation. Supply the name as part of the connection string for PRGRS_CONNECT, as follows:

  ```
  CONNECT datasrc-name -ld logical-database-name -dt MSS
        -Dsrv PRGRS_CO,data-&^source&^name;.
  ```

- To connect to the SQL Server™ database using the ODBC driver as a guide, specify an empty PRGRS_CONNECT, which tells the ODBC driver to handle the entire connection process interactively. For example:

```
CONNECT datasrc-name -ld logical-name -dt MSS
      -Dsrv PRGRS_CO,;.
```

4.4.4 Query Tuning with Connection and Startup Parameters

You can control aspects of DataServer query handling not only programmatically within 4GL statements, but also through startup and connection parameter options as described in this section.

NOTE: Startup and connection parameters override query-tuning defaults. However, options set in the QUERY–TUNING phrase take precedence over startup and connection parameters. For example, if you specify NO–DEBUG for a query within your application, specifying qt_debug,SQL at connection time overrides the default application behavior but does not override the NO–DEBUG option that you specified for the query. See the "Query Tuning" section in Chapter 2, "Programming Considerations," for more information.

You override query-tuning defaults with the DataServer (-Dsrv) connection parameter when you connect to a SQL Server database. The syntax is:

SYNTAX

```
CONNECT data-source-name -dt MSS
  -Dsrv query-tuning-option1,value1
  -Dsrv query-tuning-option2,value2.
```

An alternate syntax for the -Dsrv parameter is as follows:

SYNTAX

```
CONNECT data-source-name -dt MSS -Dsrv
  query-tuning-option1,value1,query-tuning-option2,value2.
```

Table 4–2 describes the query-tuning options that you can specify with the -Dsrv parameter. Unless otherwise indicated, these options apply at both compile time and run time. See Table 4–2, "Query-tuning Options" for more details.

Table 4–2: Connection Query-tuning Options *(1 of 2)*

Option	Description
qt_no_debug qt_debug,SQL qt_debug,EXTENDED qt_debug,CURSOR qt_debug,PERFORMANCE qt_debug,CALL_SUMMARY qt_debug,VERBOSE	Specifies whether the DataServer prints debugging information that it generates for the query to the `dataserv.lg` file. The default is qt_no_debug, to supply no debugging information. To override the default, specify qt_debug,*option* as follows: – Specify qt_debug,SQL to print the SQL statements that the DataServer executes. – Specify qt_debug,EXTENDED to print information such as cursor statistics in addition to the SQL statements executed by the DataServer. – Specify qt_debug,CURSOR to print information about the cursors that the DataServer uses for internal calls and for opening queries. – Specify qt_debug,PERFORMANCE to print information on the amount of time that certain operations take. – Specify qt_debug,CALL_SUMMARY to print information on cursors and timing. – Specify qt_debug,VERBOSE to print all of the information gathered by the other qt_debug options. For more detailed descriptions of these options, see Table 4–3.
qt_lookahead qt_no_lookahead	Specifies whether the DataServer uses lookahead or standard cursors. The default with statements that use NO-LOCK is qt_lookahead. Specify qt_no_lookahead for behavior that is consistent with Progress.

Table 4–2: Connection Query-tuning Options *(2 of 2)*

Option	Description
qt_separate_connection qt_no_separate_connection	Specifies whether each cursor should use a separate connection to the SQL Server™ database. The default is qt_no_separate_connection, which provides behavior that is consistent with Progress. Specify qt_separate_connection to use a separate connection. Executing cursors in separate connections can improve performance because the DataServer does not have to restart the cursors.
qt_cache_size,integer	Specifies the size in bytes of the cache used by lookahead cursors. A larger cache size can improve performance for queries that return a large number of records because the DataServer might need fewer SQL statements to get the results. Minimum: The DataServer always caches at least one record. Maximum: None Default: 30000

The following example shows how to use the query-tuning options to enhance performance. The DataServer opens a separate connection to MSS (Microsoft® SQL Server™) for each cursor and writes an extended report on the SQL statements it executes:

```
CONNECT holder -db infdb -dt MSS -ld demo -U user -P password -Dsrv
qt_separate_connection,qt_debug,EXTENDED.
```

Progress provides a startup parameter called Server Join (-nojoinbysqldb) that controls the default JOIN–BY–SQLDB behavior. You specify this parameter in the startup command for your Progress session. It overrides the JOIN–BY–SQLDB default so that the client evaluates and performs joins. Using this parameter might slow performance, but it provides results that are consistent with Progress behavior. See Chapter 2, "Programming Considerations," for more information.

4.4.5 Analyzing Performance

The qt_debug option of the DataServer (-Dsrv) startup parameter (and the QUERY–TUNING DEBUG phrase) instructs the DataServer to print information on the queries it generates to the `dataserv.lg` log file. The qt_debug option provides extended diagnostic capabilities that you can use to determine which parts of your application might be causing additional network traffic or extra processing by the SQL Server™ database. Table 4–3 lists the diagnostic capabilities of qt_debug.

Table 4–3: Diagnostic Options *(1 of 2)*

Option	Description
qt_no_debug	Supplies no debug information. This is the default.
qt_debug,SQL	Prints the SQL statements that the DataServer executes.
qt_debug,EXTENDED	Prints the SQL statements that the DataServer executes **plus** additional information such as cursor statistics.
qt_debug,CURSOR	Prints information about the cursors that the DataServer uses for internal calls and for opening queries. It tracks when cursors open and close and when the DataServer reuses them. It also summarizes each cursor's activity. These diagnostics are especially helpful when determining Progress maximum cursor settings or cursor "leaks" that your application might have.
qt_debug,PERFORMANCE	Prints information on the amount of time that certain operations take. These statistics are available only for some platforms. Note that any time differences between what the DataServer and data source report might be due to network performance issues rather than to DataServer or data source behavior.

Table 4–3: Diagnostic Options *(2 of 2)*

Option	Description
qt_debug,CALL_SUMMARY	Prints information on cursors and timing. This information is supplied in summary form as a Progress data (.d) file. Call Progress Software Corporation Consulting for assistance with this file. **NOTE:** This option is not available as a QUERY–TUNING phrase option.
qt_debug,VERBOSE	Prints all of the information gathered by the other qt_debug options. **NOTE:** This option generates a very large log file. Be sure to clear your log file before using this option to test a procedure.

NOTE: Turning on debugging options decreases DataServer performance. Be sure to turn off debugging options when you run DataServer applications in production mode.

This connection statement causes the DataServer to report on the time that ODBC operations take to run:

```
CONNECT msscdb -U password
  -Dsrv qt_cache_size,32000,qt_debug,PERFORMANCE.
```

4.4.6 Local Schema Caching

By using a local file to store schema definitions, you can access them more quickly. Running DataServer applications with a local schema cache can result in better performance in networked environments.

The Progress 4GL SAVE CACHE COMPLETE statement creates a binary file that contains the entire schema for a Progress database. Use this statement to create a cache file for a connected schema holder:

```
SAVE CACHE COMPLETE schema-holder-name TO filename.
```

For example, the following statement creates a cache file named `sqlcache` for the sqlhold schema holder:

```
SAVE CACHE COMPLETE sqlhold TO sqlcache.
```

To use the cache file specify the Schema Cache File (-cache) startup parameter and the cache filename when you connect to the schema holder. For example, the following CONNECT statement connects a SQL Server™ database whose data source name is sqlserv1 with the schema sqlhold and tells Progress to use the cache file:

```
CONNECT sqlhold -RO -db sqlbdb -dt MSS -ld sqldemo -U bob -P bobpass
   -Dsrv qt_debug,EXTENDED, -cache sqlcache.
```

If you make any changes to a schema holder, you must create a new cache file for it. For more information, see the *Progress Programming Handbook* and the SAVE CACHE Statement reference entry in the *Progress Language Reference*.

4.5 Connection Failures and Progress Responses

Table 4–4 lists circumstances under which a connection might fail and describes how Progress responds.

Table 4–4: Failure Responses *(1 of 2)*

Failure Circumstance	Result
During startup	The system displays an error message and returns to the operating system prompt.
During a CONNECT statement	The system aborts the remainder of the CONNECT statement as though the remainder never existed, and a run-time error condition occurs. Any connections that you made prior to the failed connection remain in effect. You can use the NO–ERROR option with the CONNECT statement to trap run-time errors. If you use the NO–ERROR option and it fails, you see the same failure behavior as you do with an unsuccessful CONNECT statement. However, run-time error conditions do not occur.

Table 4–4: Failure Responses *(2 of 2)*

Failure Circumstance	Result
During an attempted auto-connect	The system aborts the remainder of the connect as though the remainder never existed, and a run-time error condition occurs. Any connections that you made prior to the failed connection remain in effect. You cannot trap auto-connect run-time error conditions.
During an attempt to connect using the Progress Data Dictionary	The Data Dictionary displays an error message and returns to the main window.
During an attempt to connect a connected SQL Server™ database with a different logical name	The system responds with a run-time error condition but you can continue. You can use the NO–ERROR option to suppress the error.
During an attempt to connect a connected SQL Server™ database with the same logical name	The system responds with a warning but you can continue. You can use the NO–ERROR option to suppress the warning.
During an attempt to connect an unconnected SQL Server™ database whose logical name is already in use by a connected SQL Server™ database	The system responds with a run-time error condition and you cannot connect to the second database.

4.5.1 Connection Troubleshooting

Here are some reasons why a connection attempt to a SQL Server™ database might fail:

- The schema holder is not connected.
- The Progress or MSS (Microsoft® SQL Server™) required environment variables are not set correctly when using the Enterprise DataServer and a broker. For environment-variable information, see Chapter 3, "Configuring the DataServer."
- The data source is not registered properly for ODBC client connectivity.
- You have an outdated version of an ODBC DLL; for example, `ODBC16.DLL`, which runs on 16-bit machines only. This prevents Progress from accessing the data source, though you might still be able to access the data source through your ODBC driver using another product, such as Query Analyzer.

- The data source has not been started or is not running correctly. Use the data source utilities to check the status of the data source and the ability to connect to it.
- The login name and password combination that you provided during connection is invalid for the data source.
- You specified an incorrect ODBC data source name when you created the schema holder.

For more information, see Chapter 6, "Troubleshooting."

4.5.2 Managing Connections to a SQL Server™ Database

Typically, the DataServer maintains one connection to an MSS data source. In some instances, such as for joins and catalog queries, the DataServer automatically starts separate connections. However, there are circumstances in which a DataServer application might require more than one connection. For example, the DataServer cannot send a query to a data source while a stored procedure is still open unless you specify that the DataServer uses separate connections for request. The following cases require a connection to accommodate additional cursors:

- Running a stored procedure, including using the send–sql–statement option
- Performing a join on the server
- Creating or updating the schema image for the data source

In the first case, additional connections are necessary only if your application executes additional database requests while a cursor on a stored procedure is still open.

You can use the -Dsrv qt_separate_connection parameter or the corresponding QUERY–TUNING option (SEPARATE–CONNECTION) to specify that the DataServer uses a separate connection for each statement that requires a cursor. However, if you want to use the main connection when performing joins on the server, use the -Dsrv qt_no_separate_connection parameter when you connect. Note that using a separate connection allows only read-only access to the database. You must issue transactions that require update access to your database from your main connection.

For example, the following statement specifies that the DataServer use a separate connection for the FOR EACH customer query:

```
FOR EACH customer QUERY-TUNING (SEPARATE-CONNECTION)
    DISPLAY customer.
```

4.5.3 Accessing the DataServer Log

Progress supports a log file named `dataserv.lg` that is dedicated to tracking information related to DataServers. This single file contains information about the processes for all DataServers operating on your host machine. It provides a useful record of connection and disconnection processes and error messages that you can use to diagnose problems or failures. Each new connection to the same log file appends to the end of the existing file. If you output a detailed log over multiple connections that do a lot of database I/O, the log can quickly take up disk space. For each process, the log provides the following information:

- Physical database name
- Database type
- User ID
- ODBC function calls
- SQL statements

Specifying the -Dsrv qt_debug option causes the DataServer to write to the `dataserv.lg` file information about the SQL ODBC calls that it generates as well.

To obtain access to the Enterprise DataServer log file, follow these steps on the host machine:

1 ♦ Before starting up the broker process, set the DSLOGDIR environment variable to the name of the directory where you want to place the log file.

If you set the environment variable, Progress writes the information to the `dataserv.lg` file. If Progress cannot open this file or $DSLOGDIR is unset, it writes the information to the `dataserv.lg` file in the process' current directory, and appends to it with each subsequent process that uses it.

2 ♦ Open the `dataserv.lg` file to read the DataServer log. For information on debug options that affect DataServer log output, see the "Analyzing Performance" section.

5

The DataServer Tutorial

This chapter presents step-by-step instructions for tasks associated with the DataServer. Some of these exercises relate to maintaining the schema holder. After providing an overview of the MSS demonstration databases, the tutorial describes how to perform the following tasks:

- Prepare to use the DataServer utilities
- Update a schema holder
- Verify a schema holder
- Change logical name and connection information in a schema holder
- Modify a schema holder to use Progress features
- Change the code page in a schema holder
- Delete a schema holder
- Add extended 4GL support to a SQL Server™ database
- Migrate a Progress database to a SQL Server™ database

5.1 Demonstration Databases for MSS DataServers

The demonstration databases for the DataServer for MSS allow you to run the tutorials from this chapter, code examples, or your own procedures.

The demonstration databases are not part of the Progress installation media. To create and initialize a demonstration data source, you run the Progress-to-MSS utility, which migrates the Progress Sports database to your data source and then creates a schema holder. (See the "Preparing to Use the DataServer Utilities" section for detailed instructions.) After you do this, you are ready to run the tutorial exercises or your own Progress applications against the data source.

Before you create a demonstration database, be sure that your system meets the following prerequisites:

- Sufficient privileges to create a database, add users, and create tables
- Sufficient disk space for your database

5.2 Preparing to Use the DataServer Utilities

Subsequent sections contain tutorials that illustrate using the DataServer utilities to maintain your schema holder. Before you use these tutorials, you must perform some preparatory steps, including creating the demonstration database and schema holder and accessing the DataServer's SQL Server Utilities menu.

To prepare to use the DataServer utilities, follow these steps:

1 ♦ Make sure that you have done the following:

   a) Installed and started your data source

   b) Installed and configured your drivers

   c) Installed and configured the client software

2 ♦ Create an empty database for your data source.

3 ♦ Start the ODBC administration tool for your data source.

4 ♦ Select the appropriate ODBC driver.

5 ♦ Select the target ODBC data source.

6 ♦ Make any additional selections required by your data source.

7 ♦ Specify **mysport** as data source name. The name that you use to register a data source is the name that Progress recognizes.

8 ♦ Exit the ODBC administration tool.

9 ♦ Make sure that the data source has started.

10 ♦ Start a local Progress session using the following command on NT or Windows:

```
install-path\dlc\bin\prowin32
```

Starting a local Progress session also automatically starts the local DataServer.

11 ♦ Access the Progress Data Administration tool. Create a copy of the sports database and connect to it.

12 ♦ Select DataServer→ MS SQL Server Utilities→ Schema Migration Tools→ PROGRESS DB to MS SQL Server to start the Progress-to-MSS Utility.

13 ♦ Specify Progress-to-MS SQL Server parameter values as noted in Table 5–1.

Table 5–1: Progress-to-MSS Utility *(1 of 2)*

Interface Element	Description
Name of the Original Progress Database	Accept the default value of **sports**, which is displayed.
Other connect parameters for Progress Database	Accept the default value, which specifies the current working database.
Enter name of Schema holder Database	Type **myholder.**
Enter the Logical Database name	Type name that Progress will use to identify the SQL Server™ database.
Enter name for Database in the ODBC data source	Type **mysport**, which is the name you used when registering the data source.
Enter the Username	Type the user ID for SQL Server™ database.
Enter user's Password	Type the password of the user.

Table 5–1: Progress-to-MSS Utility *(2 of 2)*

Interface Element	Description
Enter other connect parameters	See Chapter 4, "Connecting the DataServer," for connection parameters.
Enter codepage for Schema Image	Accept the default code page.
Maximum length of VARCHAR field	Enter the maximum size of the VARCHAR field. Any size greater will be text. Max size is 8000.
Create RECID field	Check this toggle box to obtain the cursor scrolling capability.
Load SQL	Leave this box checked to have the generated SQL script loaded into the data source.
Move Data	Check this toggle box to dump and load data from Progress to SQL Server™.
Create Shadow Columns	SQL Server™ is case insensitive by default. Leave unchecked unless the default has been changed and you want Progress behavior.
Create Desc Index	SQL Server™ does not support Descending Indexes. So any indexes that have descending components are skipped. If this box is checked, all indexes are created as ascending.
Include Defaults	Check if you want the Progress initial value used as the default in the SQL script.
Use Field's SQL Width Column	If your Progress database contains data that is larger than the display format, you can check this toggle box. Change the SQL width field in the Data Dictionary to the proper size. The utility will use the SQL width field instead of the display format.

NOTE: For a complete description for running while in batch mode, see Table 5–5 in the "Running the Progress-to-MSS Utility in Batch Mode" section.

Running the utility creates and connects a schema holder and the SQL Server™ database. It operates as follows:

a) SQL script is generated.

b) SQL that creates the schema is sent to the SQL Server™ data manager.

c) The schema is pulled back to the schema holder.

d) The schema holder and the Progress database are compared and all information needed by Progress is applied to the schema holder.

e) Data is loaded if selected.

f) The schema holder is disconnected.

g) A message is displayed that tells the user which startup procedure to use to connect. Additionally, the entered connect information is also stored in the db record which the DataServer can use to automatically connect.

14 ♦ Choose DataServer→ MS SQL Server Utilities to see the available DataServer utilities, described in Table 5–2.

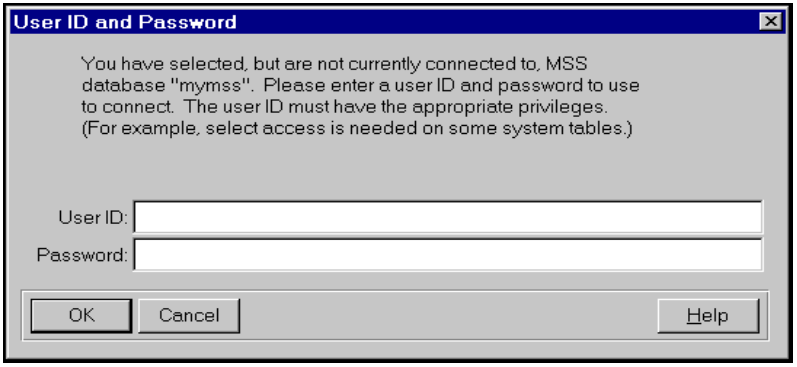
Table 5–2: DataServer Utilities *(1 of 2)*

Utility	Description
Create DataServer Schema...	Creates a schema image in the schema holder for the SQL Server™ database.
Update/Add Table Definitions...	Updates the schema holder to reflect any changes that you make to data definitions.
Verify Table Definition...	Makes sure that the data definitions in the schema holder match your data source data definitions.
Edit Connection Information...	Changes connection information or the logical database name for a SQL Server™ database.
Change DataServer Schema Code Page...	Changes the code page in the schema holder associated with the SQL Server™ database.

Table 5–2: DataServer Utilities *(2 of 2)*

Utility	Description
Delete DataServer Schema...	Deletes a schema image from the schema holder.
Schema Migration Tools	Accesses a utility for migrating a Progress database to a SQL Server™ database.

15 ♦ When you access a DataServer utility (as you will do in the tutorials that follow this section), the following dialog box might appear before the utility opens:



In the User ID and Password dialog box, choose OK if you are satisfied with the user ID and password combination that you supplied. If you want to change them, or they were never specified, enter a user ID and password with the privileges required for creating and updating a schema holder. See the "Establishing Permissions" section in Chapter 3, "Configuring the DataServer," for information on privileges.

5.3 Updating a Schema Holder

The Update/Add Table Definitions utility allows you to update a schema holder to reflect changes that you make to the data definitions in your data source.

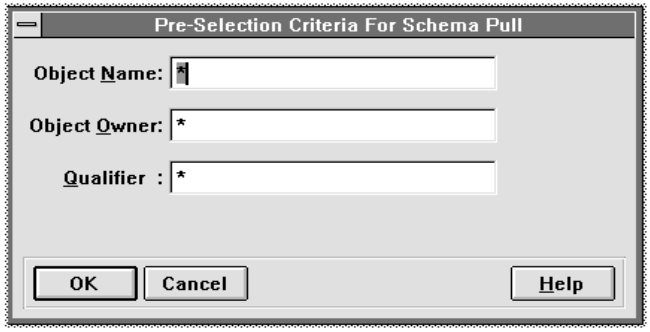
NOTE: Using this utility to update the schema holder is the **only** way to make sure that your applications can access modifications or additions to a data source.

The Update/Add utility allows you to:

- Add object definitions from the data source to a schema holder. Use this option if you add a new table, stored procedure, or view to the data source data definitions and want the schema holder to reflect the change.
- Update existing object definitions in a schema holder to reflect a change in the supporting data-source object definitions.

To update a schema holder, follow these steps:

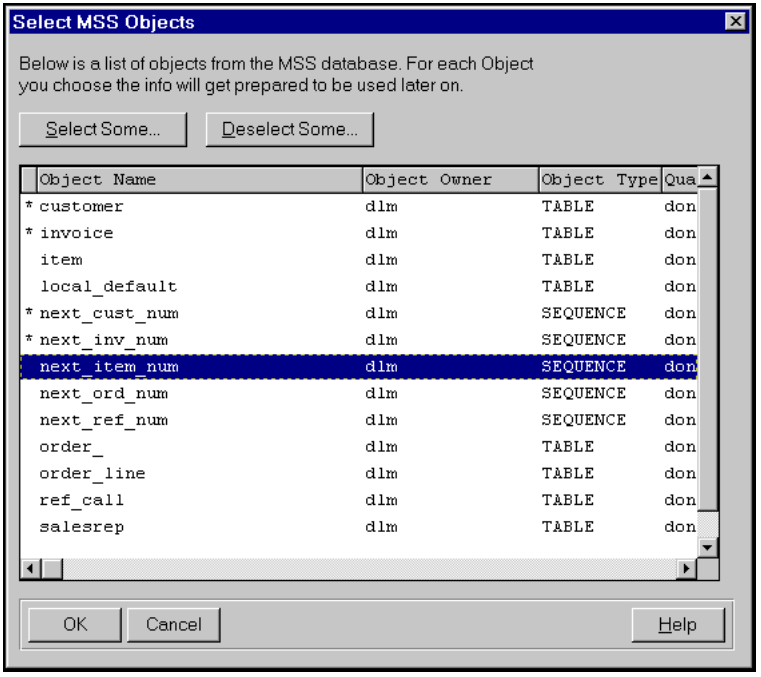
1 ♦ Access the Progress Data Administration tool, if you are not already there, and select DataServer→ MS SQL Server Utilities→ Update/Add Table Definitions. The following dialog box appears:



2 ♦ Type preselection criteria values into the fields as required. These values preselect the data source objects that the utility uses to update the schema holder. By default, the wildcard symbol (*) appears; it specifies that the utility uses all of the objects in the data source.

NOTE: For each of the three entry fields in the dialog box, if you enter a value that consists only of wild cards, you might degrade the performance of the database when you perform a schema pull. (It will include system catalog files that are not typically included in user databases.)

3 ♦ Choose OK. A dialog box lists the objects and table information that you have preselected, for example:



4 ♦ Select the objects that you want to update, then choose OK. When the update completes, Progress returns to the Data Administration main window.

When the update completes, Progress reminds you to check the `ds_upd.e` file. This file contains information about the tables that did not support record IDs as well as other warnings.

When you update a definition, Progress overwrites the old definition with the new one based on the current data source object. It also preserves the Progress-specific table information. As a result, if you want to add a new column to a table in your data source and then update the definition, you do not have to re-enter all of the Progress-specific information for the previously existing columns (fields) in the definition.

NOTE: When you update a table in the schema holder with the Update/Add Table Definitions utility, the information for the user-defined ROWID is lost. You must reselect an index to support the ROWID.

5.4 Verifying a Schema Holder

The Verify Table Definition utility allows you to verify that the schema holder for your data source matches the data definitions in the data source. For example, if you delete the Customer table from the data source but not from the schema holder, this utility reports that the schema holder contains an orphaned object. You can verify the schema information in a single table or from multiple tables, and then choose to update the tables so that the schema information matches the data source definitions.

The Verify utility reads the definitions in the data source and compares them to the information in the schema holder. It reports the differences that it finds and their degree of severity. These are the categories of differences and how they impact your database applications:

- **Minor** — These differences have no impact on the usability of your application.
- **Retained** — These differences cannot be corrected by the Update/Add utility, hence the term "retained." You must determine how severely they impact your application and change the data definitions either in the schema holder using the Data Dictionary or in the data source.
- **Severe** — These differences might cause your application to malfunction. When the Verify utility detects severe differences, it automatically updates the schema holder to solve the discrepancies by adjusting the schema-image information in the schema holder to match the data source definitions. Severe differences in definitions that the DataServer uses internally also cause the schema holder to be updated.

Table 5–3 lists the differences that this utility detects.

Table 5–3: Verify Utility Report *(1 of 2)*

Database Object	Difference	Category
Table	Description	Retained
Table	Foreign type	Severe
Table	Name in Progress	Retained
Table	Package name	Severe
Table	ROWID index	Retained
Index	Active	Minor

Table 5–3: Verify Utility Report *(2 of 2)*

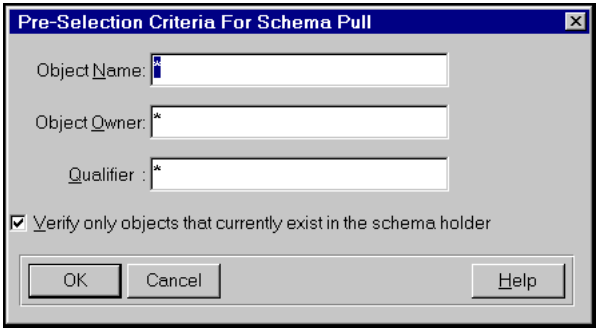
Database Object	Difference	Category
Index	Description	Retained
Index	Name in Progress	Retained
Index	Unique	Retained[1]
Index field	Abbreviated	Minor
Index field	Ascending	Severe
Index field	Order	Severe
Field	Case sensitivity	Retained
Field	Decimals	Retained[2]
Field	Description	Retained
Field	Extent	Severe
Field	Initial value	Retained[2]
Field	Mandatory	Retained
Field	Name in Progress	Retained
Field	Order	Retained
Field	Progress data type	Retained[2]
Field	Progress format	Retained[2]
Field	Shadow-column Name	Severe

[1] When you update an index, it is flagged as unique if it was defined as unique in the data source or the schema holder.

[2] If the corresponding information in the data source is incompatible with the information in the schema holder, the affected fields are not updated. For example, if the data source data type is decimal and the Progress data type is CHARACTER, the data-type information is not updated.

To verify a table, follow these steps:

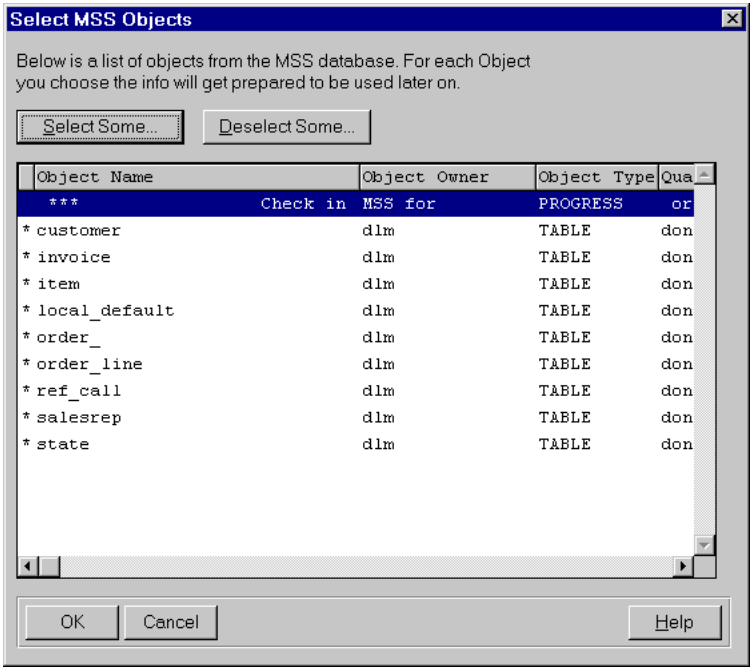
1 ♦ Access the Progress Data Administration tool, if you are not already there, and select DataServer→ MS SQL Server Utilities→ Verify Table Definition. The following dialog box appears:



2 ♦ Type preselection criteria values into the fields if desired. These values preselect the data source objects that the utility uses to update the schema holder. By default, the wildcard symbol (*) appears; it specifies that the utility uses all of the objects in the data source.

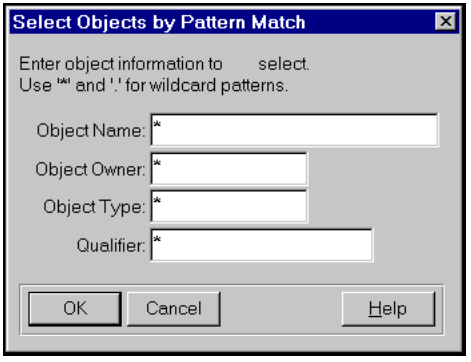
3 ♦ By default, the utility verifies objects in the schema holder that match objects in the data source. To check whether there are objects in the data source that are not represented in the schema holder, deselect the toggle box labeled “Verify only objects that currently exist in the schema holder.”

4 ♦ Choose OK. A dialog box lists the objects and table information that you preselected, for example:



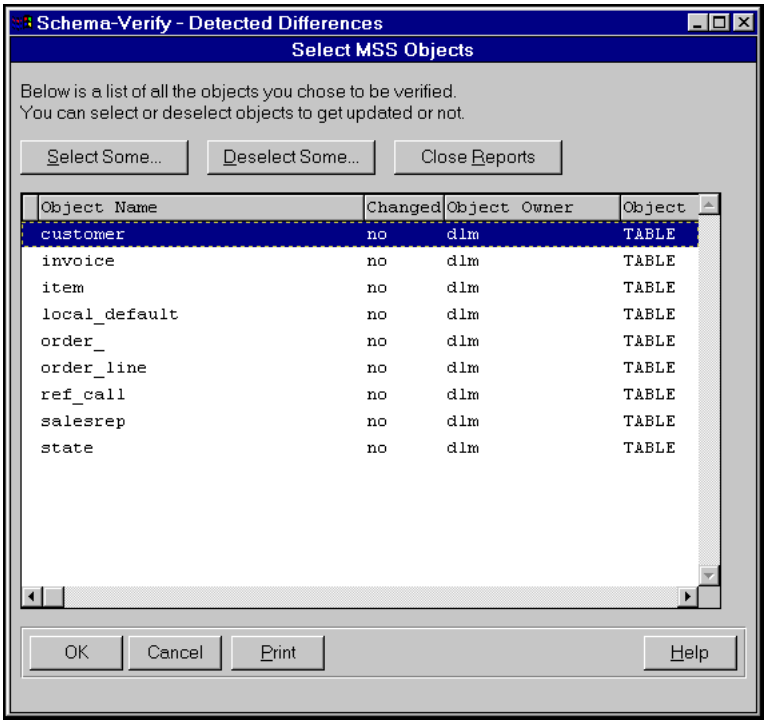
5 ♦ Select the objects that you want to update, then choose OK.

6 ♦ To select tables by matching a pattern, choose the Select Some button. The following dialog box appears:



7 ♦ Type the pattern that you want to match, then choose OK to start the verification. A dialog box that lists the objects and the verification results appears.

8 ♦ If on UNIX, choose the View Reports button to view a description of the differences found. On Windows, the Schema-Verify-Detected differences automatically displays, as shown below:



9 ♦ Choose Close to return to the Schema Verify dialog box.

10 ♦ The utility automatically selects objects with severe differences for updating. You can select or deselect all other objects as you wish. Note that you **must** resolve retained differences manually. Retained differences appear in subsequent reports until you resolve them.

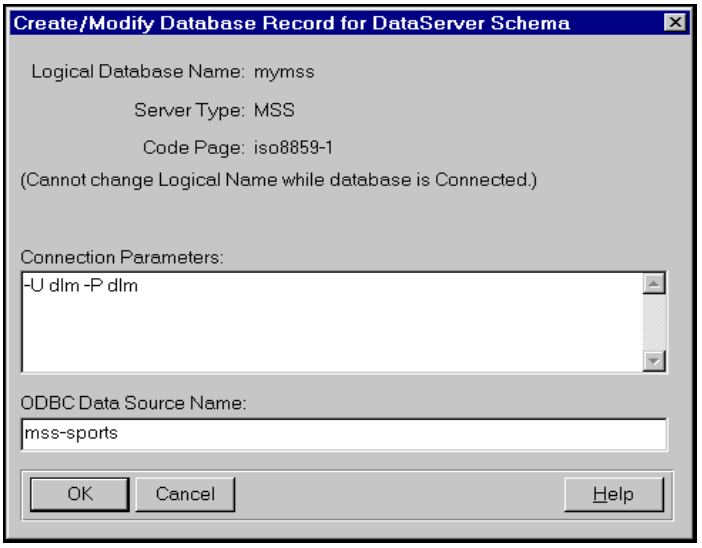
11 ♦ Choose OK to start the update or Cancel to quit the utility without updating the schema holder.

5.5 Changing Connection Information in a Schema Holder

The Edit Connection Information utility allows you to change a SQL Server™ database's connection information or logical name in the associated schema.

To change connection information for a schema holder, follow these steps:

1 ♦ Access the Progress Data Administration tool, if you are not already there, and select DataServer→ MSS SQL Server Utilities→ Edit Connection Information. The following dialog box appears:



2 ♦ Make changes to the Connection Parameters fields as required. When you are done, choose OK to return to the Data Administration main window.

The changes do not take effect until you disconnect and reconnect the schema holder. When you reconnect, Progress uses the new connection parameters.

For details on connection parameters, see Chapter 4, "Connecting the DataServer," and the *Progress Startup Command and Parameter Reference*.

NOTE: If you change the ODBC Data Source Name (DSN), do not select a DSN that uses a different ODBC driver than the original DSN. Configuration switches residing in the schema holder are dependent on the driver name. You will receive only a warning if you use a different driver, but the schema holder configuration may no longer match the characteristics of the driver and might cause unpredictable run-time results.

5.6 Modifying a Schema Holder

You can begin using the DataServer as soon as you load your data source data definitions into the schema holder. However, you might want to use Progress features such as labels, validation expressions, or validation messages. You also might want to change the default data type provided for fields in the schema-image tables.

You can define Progress information at the table and field levels in the schema holder. The following sections describe how to enter Progress information at both levels. An additional section describes how to select an index to support the ROWID function.

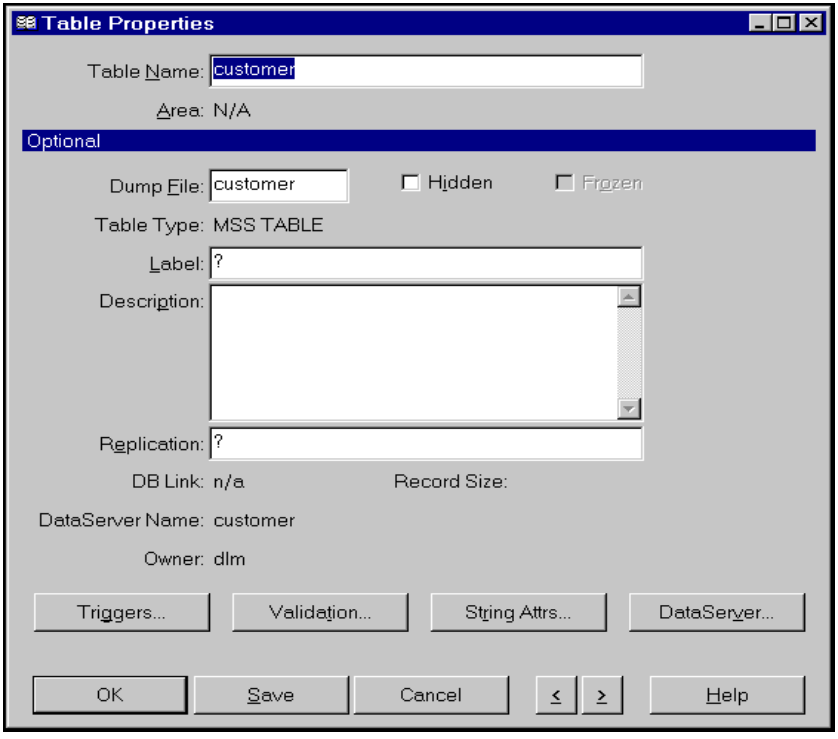
5.6.1 Modifying Table-level Information

To modify information in the schema holder at the table level, follow these steps:

1 ♦ Access the Data Dictionary.

2 ♦ Select a table from the Tables list; for example, the customer table.

3 ♦ Choose the Table Properties button. The following dialog box appears:



4 ♦ Choose the Validation button. The Table Validation dialog box appears. You can change either the validation expression or the message by typing new text in the fields.

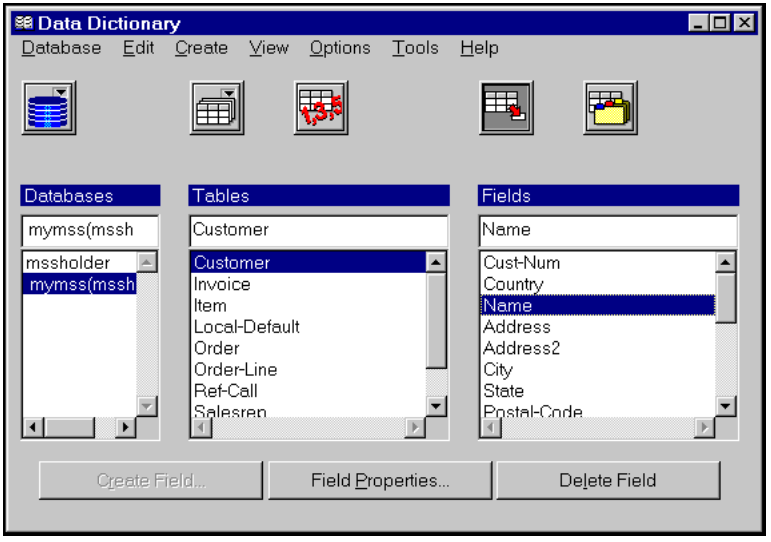
5 ♦ Choose OK to return to the Table Properties dialog box.

6 ♦ Choose OK to return to the Data Dictionary main window.

5.6.2 Modifying Field-level Information

To modify information in the schema holder at the field level, follow these steps:

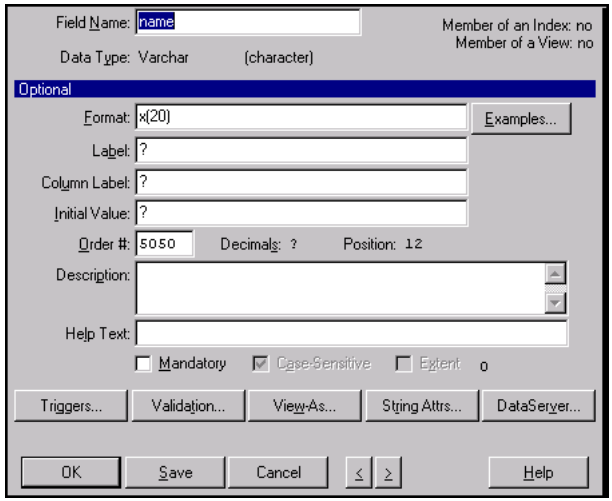
1 ♦ Access the Data Dictionary, if not already showing, choose the Fields button. The following Fields list appears.



2 ♦ Select a table from the Tables list.

3 ♦ Select a field from the Fields list.

4 ♦ Choose the Field Properties button. The following dialog box appears:



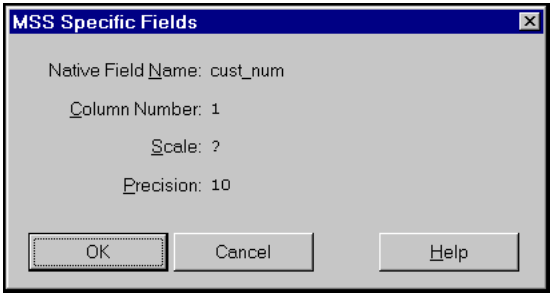
You can enter Progress information at the field level, such as a validation expression or a validation message.

The Data Dictionary displays the standard SQL Server™ names for data types and not the native ODBC data source names. Using the Data Dictionary, you can make the following changes:

- Change the data type or the format in which Progress displays the data. For example, choose the SQL Server™ smallint data type, which in turn maps to the Progress INTEGER data type. However, you can change the SMALLINT mapping to the Progress DECIMAL or LOGICAL data type instead. (See the "Data Types" section in Chapter 2, "Programming Considerations," for information on optional settings for data types.)
- For CHARACTER fields that are not indexed, you can change the case sensitivity.

NOTE: You cannot create fields or add mandatory or extent properties to them.

5 ♦ Choose the DataServer button to view the field name and position as stored in the data source. A dialog box similar to the following appears:



NOTE: You cannot change data source information using the Data Dictionary. For example, the Cust–num field is named Cust_num in the demonstration database.

6 ♦ Choose OK.

7 ♦ When you are done making changes, choose OK to return to the Data Dictionary main window.

NOTE: You can override field-level validation expressions in your application by including the appropriate Progress 4GL statement.

5.6.3 Defining the ROWID

When you create or update a schema holder, the DataServer uses the following guidelines to select a column in an data source table to support the ROWID function:

- If the data source table has a PROGRESS_RECID column, the DataServer selects that column. A column of this type provides optimal support for the ROWID function; you cannot select an alternative to it.
- If the data source table does not have a PROGRESS_RECID column, the DataServer evaluates the available indexes and selects one according to the following preferences:

  a. Unique, single-component, mandatory, integer

  b. Unique, single-component, integer

  c. Unique, single-component, mandatory, any data type except float

  d. Unique, multi-component, any data type

If more than one index in the data source table meets the second level—unique, single-component, integer—the DataServer selects the first such index that it encounters to support the ROWID function. Note that the indexes in this class are **not** mandatory; hence it is essential that you enforce the column supporting ROWID as mandatory at least through code if not through definitions. If your application handles an index in such a way as to make it a better support for the ROWID function, you can designate it in the Progress Data Dictionary.

NOTE: An index that you select as a ROWID must be defined as a unique index. It must also be mandatory, if not by definition, then by means of the application code.

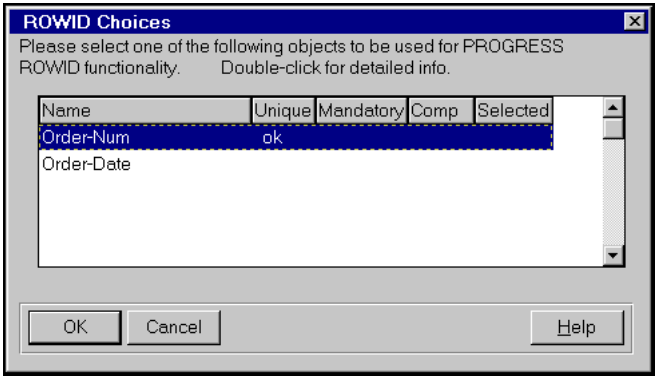
To select an index to support the ROWID function, follow these steps in the Data Dictionary with the schema holder connected (you do not have to connect to the SQL Server™ database):

1 ♦ Choose the Tables mode button.

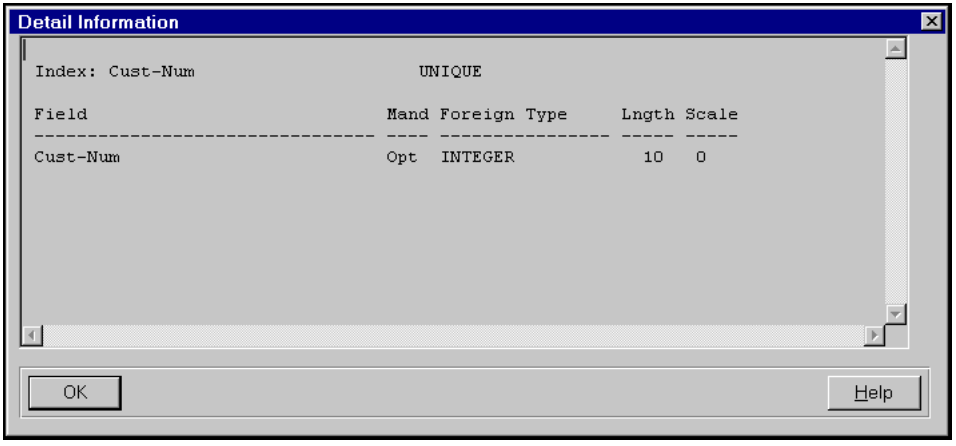
2 ♦ Select the table whose ROWID you want to modify.

3 ♦ Choose the Table Properties button.

4 ♦ Choose the DataServer button. The following dialog box appears:



5 ♦ Double-click an index to see detailed information on its attributes. The following dialog box appears:



6 ♦ Choose OK to return to the ROWID Choices dialog box.

7 ♦ Select the index you want to use to support ROWID.

8 ♦ Choose OK to return to the Table Properties dialog box.

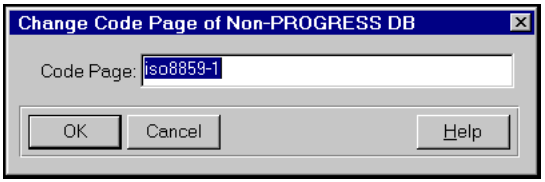
5.7 Changing a Code Page in a Schema Holder

Using the Change DataServer Schema Code Page utility, you can change the code page in a schema holder at any time. As a result, you can create a schema holder even if you do not have the correct code-page information; you can add or correct the code-page information at a later date. However, if you have been writing 8-bit character data to your data source with the DataServer and then you change the code page, the data is unaffected by the code-page change. Note that your database might be corrupted if you start writing data with the DataServer and a schema holder that uses a new code page.

To change a code page, follow these steps:

1 ♦ Access the Progress Data Administration tool, if you are not already there, and select DataServer→ MSS SQL Server Utilities→ Change DataServer Schema Code Page. The utility displays a message about the possibility of corrupting your database by using the wrong code page.

2 ♦ Choose OK to continue. The following dialog box appears:



3 ♦ Either accept the current value or type the data source code page that the data source supports.

See the *Progress Internationalization Guide* for a complete list of code pages that Progress supports. If you are using an unsupported code page, Progress allows you to create your own conversion tables.

4 ♦ Choose OK to change the code page and return to the Data Administration main window.

If you were connected to the schema holder and the data source when you chose to change the code page, Progress disconnects you to make the change. The Connect Database dialog box appears to allow you to reconnect.

5.8 Deleting a Schema Holder

The Delete DataServer Schema utility allows you to delete the schema holder for a data source.

To delete a schema holder, follow these steps:

1 ♦ Access the Progress Data Administration tool and select DataServer→ MS SQL Server Utilities→ Delete DataServer Schema. A dialog box appears, prompting you to verify the deletion.

2 ♦ Choose Yes to verify your selection. After Progress deletes the schema holder, it displays a confirmation message.

3 ♦ Choose OK to return to the Data Administration main window.

5.9 Adding Extended 4GL Support

The DataServer provides support for arrays and the 4GL ROWID function when certain objects exist in a data source. You can modify your data source to support these extended features. You can also add support for these features when you create new tables.

The DataServer maintains the columns that you add to your data source tables to support these extended features. If non-Progress applications are updating those data source tables, these applications must be aware of the additional columns and update them appropriately. You can use triggers, if your data source supports them, to make sure that the columns are updated correctly.

The following sections describe how to modify an existing data source so that it supports arrays, and Progress record identifiers. See the "ROWID Function" section in Chapter 2, "Programming Considerations," for alternate ways to support this function.

5.9.1 Modifying Tables to Support Arrays

The DataServer allows you to extend Progress's ability to support database arrays to your data source. To have access to this functionality, make the following changes to your data source table:

1 ♦ Name the columns of a data-source table that you want the DataServer to roll into an array *column*##1, *column*##2, etc. The columns must be adjacent and in sequence.

2 ♦ Make sure that these columns are of the same data type and size. For example, if you want the schema holder to include an array named MONTH with 12 elements, the data source table must have 12 adjacent columns of the same data type named month##1, month##2, month##3, and so forth. Progress names the corresponding field in the schema holder month. In your applications, refer to each element of the array as month[1], month[2], month[3], and so forth.

3 ♦ If you have already created your schema holder, update it to reflect your changes to the data source table.

5.9.2 Modifying Tables to Support the ROWID Function

If you want to use the ROWID function with a data source, you must make certain changes to your data source table. The following example procedure illustrates how to do this.

To use the ROWID function with a SQL Server™ database, make the following changes to your data source table:

1 ♦ Add a column of the integer data type named PROGRESS_RECID. The new column must be able to contain null:

```
alter table table
    add PROGRESS_RECID integer null
```

2 ♦ Add a column with identity characteristics named PROGRESS_RECID_IDENT_. The new column must have the numeric data type:

```
alter table table
    add PROGRESS_RECID_IDENT numeric(10,0) identity
```

3 ♦ Create a trigger to maintain the PROGRESS_RECID column:

```
create trigger _TI_table on table for insert as
begin
    if (select max(inserted.PROGRESS_RECID) from inserted) is NULL
    begin
        update table set PROGRESS_RECID = @@identify
            where PROGRESS_RECID is null
        select convert (int, @@identity)
    end
end
```

4 ♦ Change the nonunique indexes so that they include a PROGRESS_RECID column as the last component:

```
create index table##index on table (column, PROGRESS_RECID)
```

5 ♦ If you have already created your schema holder, delete and recreate it.

5.10 Migrating a Progress Database to SQL Server™

The Progress DataServer for SQL Server™ supports the Progress-to-MSS utility that allows you to migrate a Progress database. While the DataServer typically makes a data source conform to a Progress database, this utility provides compatibility in the opposite direction. It copies an existing Progress database schema into a target data source.

The Progress-to-MSS utility performs the following tasks:

- Creates objects in the target data source
- Creates the schema holder and schema image
- Optionally populates the data source by dumping and loading the data from the Progress database

The SQL Server™ data source that you create with this utility is a basis for an application database. Before deploying your new data source, you might want to make manual adjustments to take advantage of additional SQL Server™ features that are not supported by the migration utility.

The Progress-to-MSS utility requires a local Progress database.

5.10.1 Preparing a Data Source for the Utility

The Progress-to-MSS migration utility translates definitions for Progress objects into an SQL DDL script to load into your target data source. It automatically makes adjustments in the target data source in order to provide the functionality of the Progress and target systems.

When you use the Progress-to-MSS utility to make a SQL Server™ database compatible, it converts arrays by using one column in the SQL Server™ database for each extent of the Progress array. The utility generates names for the columns that it adds to a table in a target data source using the format *field–name##extent–number*. For example, a Progress field called monthly–amount with an extent of 12 will have 12 columns in the data source with names such as MONTHLY_AMOUNT##1 through MONTHLY_AMOUNT##12.

5.10.2 Running the Progress-to-MSS Utility

The Progress-to-MSS utility runs on Windows with a DataServer accessing SQL Server™ through an ODBC driver.

You can run the utility interactively from Windows or NT, Unix, or in batch mode.

Running the Progress-to-MSS Utility Interactively

To run the Progress-to-MSS utility interactively, follow these steps:

1 ♦ Create a target SQL Server™. You must use an empty target data source when you run the Progress-to-MSS utility.

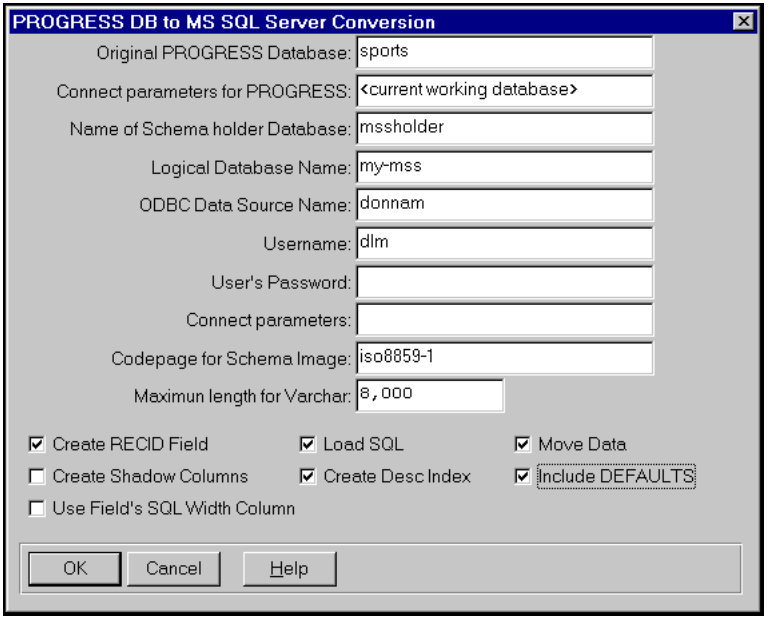
2 ♦ Configure your ODBC driver to connect to your new target data source.

3 ♦ Start the Progress client and connect to the Progress database that you want to migrate to the target data source.

NOTE: For a DBE (double-byte enabled) DataServer application, you must specify the Internal Code Page (-cpinternal) and Stream Code Page (-cpstream) parameters when you start the Progress client. The values that you specify for these parameters must match the code page that the target data source uses.

4 ♦ From the Data Administration tool, choose DataServer→ MS SQL Server Utilities→ Schema Migration Tools→ PROGRESS DB to MS SQL Server.

5 ♦ The following screen appears:



It prompts you for the information described in Table 5–4.

Table 5–4: Progress-to-MSS Utility *(1 of 2)*

Interface Element	Description
Name of the Original Progress Database	Accept the default value of **sports**, which is displayed.
Other connect parameters for Progress Database	Accept the default value, which specifies the current working database.
Enter name of Schema holder Database	Type **myholder.**
Enter the Logical Database name	Type the name that Progress will use to identify the SQL Server™ database.
Enter name for Database in the ODBC data source	Type **mysport**, which is the name you used when registering the data source.
Enter the Username	Type the user ID for SQL Server™.
Enter user's Password	Type the password of the user.
Enter other connect parameters	See Chapter 4, "Connecting the DataServer," for connection parameters.
Enter codepage for Schema Image	Accept the default code page.
Maximum length of VARCHAR field	Enter the maximum size of the VARCHAR field. Any size greater will be text. Max size is 8000.
Create RECID field	If this parameter is available for your data source type, check this toggle box to obtain the Extended 4GL capability.
Load SQL	Leave this box checked to have the generated SQL script loaded into the data source.
Move Data	Check this toggle box to dump and load data from Progress to SQL Server™.

Table 5–4: Progress-to-MSS Utility *(2 of 2)*

Interface Element	Description
Create Shadow Columns	SQL Server is case insensitive by default. Leave unchecked unless the default has been changed and you want Progress behavior.
Create Desc Index	SQL Server™ does not support Descending Indexes. So any indexes that have descending components are skipped. If this box is checked, all indexes are created as ascending.
Include Defaults	Check if you want the Progress initial value used as the default in the SQL script.
Use Fields SQL Width Column	If your Progress database contains data that is larger than the display format, you can check this toggle box. Change the SQL width field in the Data Dictionary to the proper size. The utility will use the SQL width field instead of the display format.

If you want a complete migration of your Progress database to a target data source, you must enter information in all appropriate fields and check all appropriate toggle boxes.

The utility creates a schema holder, updates the empty target data source that you created to contain the objects stored in your Progress database, and creates a startup procedure that you can use to connect your schema holder. The startup procedure derives its name from the logical name for your target database. For example, if you specified "sports" as the logical data source name, the utility creates the `csports.p` startup procedure.

Running the Progress-to-MSS Utility in Batch Mode

To run the Progress-to-MSS utility in batch mode, follow these steps:

1 ♦ Create a target SQL Server™ database. You must use an empty target data source when you run the Progress-to-MSS utility.

2 ♦ Configure your ODBC driver to connect to your new target data source.

3 ♦ On your client machine, pass parameters to the utility by setting the environment variables listed in Table 5–5.

Table 5–5: Progress-to-MSS Utility Batch Parameters *(1 of 2)*

Environment Variable	Description
PRODBNAME	Specifies the source Progress database name.
PROCONPARMS	Specifies parameters for the connection to the source Progress database.
SHDBNAME	Specifies the new schema-holder name.
MSSDBNAME	Specify ODBC data source name.
MSSPDBNAME	Specify logical database name.
MSSUSERNAME	Specifies the user name for the target data source.
MSSPASSWORD	Specifies the password of the user for the target data source.
MSSCONPARMS	Specifies additional connection parameters for the schema holder.
MSSCODEPAGE	Specifies the Progress name for the code page that the ODBC data source uses. By default, the code page for a schema holder is ibm850. You can leave this field blank and use the Change Code page utility to add the code page information for the schema holder later.
VARLENGTH	Total number of characters a VARCHAR can have. Any field whose format is greater that will be defined as a text field. Maximum is 8000.
SQLWIDTH	If present, use the Progress file _WIDTH field size instead of calculating length by using the display format.
DESCIDX	If present, create an index that contains descending components as ascending since SQL Server™ does not support descending indexes.

Table 5–5: Progress-to-MSS Utility Batch Parameters *(2 of 2)*

Environment Variable	Description
LOADSQL	Allows you to specify whether you want the utility to create the schema in your empty SQL Server™ database. Specify YES to enable this behavior.
MOVEDATA	Allows you to specify whether to populate the database. Specify YES to dump and load data or NO to not populate the database. For example, you might specify NO if your database is large, and you want to dump and load data at a more convenient time. The default is NO.

4 ♦ Enter the following commands to set and export environment variables at the system prompt:

NOTE: You might need to append the Progress libraries to your PROPATH environment variable in order for the executable to find `.p` or `.r` files.

```
PRODBNAME=db-name; export PRODBNAME
PROCONPARMS="-1 -i"
SHDBNAME=schema-holder-name; export SHDBNAME
.
.
.
pro -b -p prodict/mss/protomss.p
```

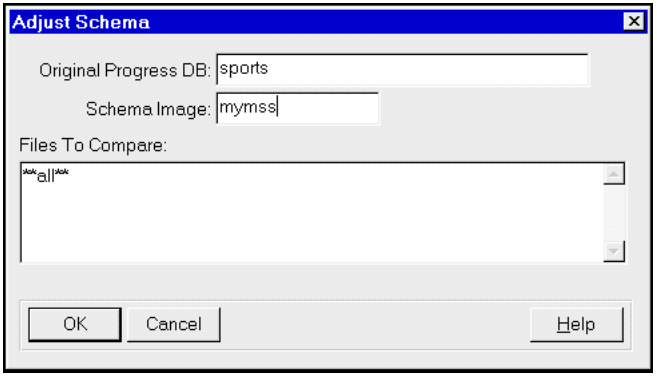
5 ♦ Run `protomss.p`.

5.11 Adjust Schema Utility

The Adjust Schema utility allows you to compare your SQL Server™ Schema holder information with the original Progress database, updating any Progress Attributes in the schema holder. These attributes include field names, labels, help, validation expressions and messages. You must be connected to the Progress database and the schema holder that you want to compare to before running this utility.

Follow these steps to adjust your schema:

1 ♦ Select the utility from the Data Admin Tool→ DataServer→ MS SQL Server Utilities→ Schema Migration Tool→ Adjust Schema. The following screen appears:



If you leave the **all** in the Files to Compare section, Progress compares all objects and makes changes. If you only want to compare certain objects, you must enter the information in the following format:

```
<comma-separated list of tables>|<comma-separated list of
sequences>|<comma-separated list of views>
```

For example:

```
customer,order|next-order-num,next-cust-num|custview,ordview
```

2 ♦ Select OK.

All of the objects that Progress compares will be displayed on screen as they are processed.

6

Troubleshooting

This chapter describes common problems and how to work around them. Specifically, it explains troubleshooting techniques for:

- Tuning your environment with the -Dsrv startup parameter, including information on the following:
  - Transactions and auto-commit
  - Connection problems
  - Key-buffer size
  - Locking error messages
  - Large rows
  - Schema import
  - SQL statements cache
- ODBC driver problems

For information on troubleshooting DataServer connections, see the "Connection Failures and Progress Responses" and "Accessing the DataServer Log" sections in Chapter 4, "Connecting the DataServer."

6.1 Tuning Your Environment with the –Dsrv Startup Parameter

The DataServer (–Dsrv) startup parameter allows you to use special DataServer options to tune your environment. You pass these options as arguments to –Dsrv when connecting to a SQL Server™ database.

There are two versions of the syntax, as follows:

SYNTAX

```
CONNECT data-source-name -ld logical-name -dt MSS
-Dsrv arg1,val1 -Dsrv arg2,val2 -Dsrv arg3,val3...
```

SYNTAX

```
CONNECT data-source-name -ld logical-name -dt MSS
-Dsrv arg1,val1,arg2,val2,arg3,val3...
```

In this syntax:

- The schema holder is already connected.
- The *data-source-name* argument is the name of the data source and the *logical-name* argument is its logical name, which is defined when you create your schema image.
- You pass the options as the *argn*,*valn* pairs.

Here is an example of how to use the CONNECT statement with the –Dsrv parameter:

```
CONNECT mssql_db -ld mydb -Ux -Py
   -Dsrv qt_debug,EXTENDED
   -Dsrv PRGRS_IDBUF,100
   -Dsrv PRGRS_MINBUF,10000
   -Dsrv MAX_R,10000.
```

Note that MAX_R is an abbreviation of MAX_ROWS. You can abbreviate option names as long as they identify parameters uniquely.

Both the syntax statements and the example show the use of the –Dsrv startup parameter in CONNECT statements. You can also specify –Dsrv options in a parameter file, on a program item command line, or in the Connection Parameters field in the Database Connect dialog box.

6.1.1 ODBC Options

The DataServer allows access to selected options defined in the ODBC interface by providing corresponding options that you can use with the -Dsrv startup parameter. Each Progress-supplied ODBC option has a name of the form *option–name* and corresponds with a startup option in the ODBC interface having the name SQL_*option–name*. Table 6–1 lists the Progress-supplied ODBC startup options and the corresponding startup options defined in the ODBC interface.

Table 6–1: ODBC Options

Progress-supplied Option	ODBC-defined Option
ACCESS_MODE	SQL_ACCESS_MODE
ASYNC_ENABLE	SQL_ASYNC_ENABLE
AUTOCOMMIT	SQL_AUTOCOMMIT
LOGIN_TIMEOUT	SQL_LOGIN_TIMEOUT
MAX_LENGTH	SQL_MAX_LENGTH
MAX_ROWS	SQL_MAX_ROWS
NOSCAN	SQL_NOSCAN
OPT_TRACE	SQL_OPT_TRACE
QUERY_TIMEOUT	SQL_QUERY_TIMEOUT
TXN_ISOLATION	SQL_TXN_ISOLATION

Refer to an ODBC application developer's guide for information on the ODBC-defined options.

When you specify a Progress-supplied ODBC option with the -Dsrv startup parameter, the DataServer sends the option to the ODBC driver for processing by the ODBC interface.

The following example of the -Dsrv startup parameter tells the ODBC driver to return no more than 1,000 rows to the Progress application:

```
-Dsrv MAX_ROWS,1000
```

NOTE: The DataServer generally sets the correct value automatically. Therefore, you should reserve use of the ODBC options for troubleshooting and fine-tuning purposes only. See the *ODBC Programmer's Reference* for more information on these options.

6.1.2 DataServer Options

In addition to the ODBC-defined options, the DataServer provides a number of -Dsrv options that are defined on the DataServer side of a connection. Each DataServer option has a name of the form PRGRS_*option–name*, to reflect its origin. Table 6–2 lists and describes these options.

Table 6–2: DataServer Options *(1 of 2)*

Option	Description
PRGRS_CONNECT	Passes a connection string to the ODBC driver.
PRGRS_IDBUF	Specifies the number of keys in the scrolling buffer. The value must be greater than 0. The default value is 25 keys. This parameter applies to all nonlookahead cursors.
PRGRS_LOCK_ERRORS	Specifies that when an application gets the stated error message, standard Progress error-handling behavior occurs (that is, Progress waits and retries rather than halting). Valid values are native ODBC data source error message numbers.
PRGRS_MINBUF	Specifies the minimum size of the buffer used to get data from the DataServer. Use this keyword only with drivers that fail to fetch large records. The default value is 500 bytes. If you specify a value smaller than the default, Progress uses the default value.

Table 6–2: DataServer Options *(2 of 2)*

Option	Description
PRGRS_NO_INDEX	Suppresses index creation at schema import time. If you specify a value of 1 for -Dsrv PRGRS_NO_INDEX, you must create the indexes manually in the schema holder.
PRGRS_PREPCACHE	Specifies the maximum number of prepared statements to be kept in cache. The default value is 20 statements.

The following example of the -Dsrv startup parameter sets the number of keys in the nonlookahead scrolling buffer to 100:

```
-Dsrv PRGRS_IDBUF,100
```

See the "Using SQL Server™ and DataServer Options" section for information on when and how to use these options.

6.1.3 Using SQL Server™ and DataServer Options

This section provides information on using various ODBC and DataServer options.

To avoid using the SQL Server™ transaction manager, specify -Dsrv AUTOCOMMIT,1. This option exercises the Progress client's local before-image mechanism to manage transactions. This creates a transaction from each SQL statement that the DataServer issues. Progress emulates the logical transaction on behalf of the application using the local before-image mechanism. This connection technique is useful in multi-database applications.

NOTE: Progress does not guarantee crash recovery when it uses the local before-image mechanism to emulate transactions.

Connection Problems: The PRGRS_CONNECT Option

The PRGRS_CONNECT option allows you to pass ODBC-specific information to the ODBC driver. This option has the following syntax:

SYNTAX

```
-Dsrv PRGRS_CONNECT,connection-string;
```

The connection string is separated from the option by a comma (,) and ends with a semicolon (;).

Use the PRGRS_CONNECT option in the following cases:

- To connect to a data source whose name is not allowed by Progress; for example, a name that includes blank spaces, ampersands (&), commas (,), and/or carets (^). In the connection string, pass the following characters rather than the unallowed characters. The driver resolves the passed characters to the unallowed character:
  - Pass the ampersand (&) character as two ampersand (&&) characters.
  - Pass the caret (^) character as two caret (^^) characters.
  - Pass the blank space character as an ampersand and a caret (&^).
  - Pass the comma (,) character as a caret and ampersand (^&).
- To establish complex connections that require more than the Physical Database Name (-db), User ID (-U), and Password (-P) parameters. In all cases, the values must not be space delimited and must be passed in a single connection string. For example, the following connection string sets the user ID and password for the server and user ID and password for the data source:

```
DSN=sports;UID=server;PWD=serverpass;UIDDBMS=bob;PWDDBMS=bobpass
```

For more information and syntax examples, see the "Special Connection Issues" section in Chapter 4, "Connecting the DataServer."

Key-Buffer Size: The PRGRS_IDBUF Option

The PRGRS_IDBUF option sets the size of the key buffers. Generally, a default of 25 is sufficient. Key buffers are used with all nonlookahead cursors.

Locking Error Messages: The PRGRS_LOCK_ERROR Option

DataServer for SQL Server™ identifies and handles conditions and errors. However, the PRGRS_LOCK_ERROR option lets you control how your application reacts if it encounters an error that is actually a lock problem when accessing a data source. Use this option to pass the native error number to the DataServer so that it handles this error as it would a Progress lock problem; that is, the DataServer waits and retries, rather than halting the application:

```
CONNECT data source name -ld logical-name -dt mss
     -Dsrv PRGRS_LOCK_ERROR,error-number1,error-number2.
```

Large Rows: The PRGRS_MINBUF Option

Some data rows can be very large; for example, in a SQL Server™ database, rows often have large fields such as IMAGE and MEMO. The ODBC protocol specifies a dynamic buffer allocation process for handling large rows that do not initially fit into clients' buffers; however, some drivers do not yet follow the correct ODBC protocol and do not handle these large rows correctly. Use the -Dsrv PRGRS_MINBUF,*size* option to force a minimum buffer size. For example, -Dsrv PRGRS_MINBUF,15000 enables the DataServer to handle 15K rows even with drivers that fail to follow the ODBC protocol.

NOTE: It is often difficult to determine when there is a buffer size problem and how to choose the correct value for PRGRS_MINBUF. Be careful when using this option.

SQL Statements Cache: The PRGRS_PREPCACHE Option

The DataServer keeps a cache of prepared SQL statements that it reuses with new parameters. This mechanism improves the DataServer performance. You can use the PRGRS_PREPCACHE option to manage this cache in two ways:

- The SQL Server™ drivers are capable of re-using prepared statements and should do so whenever possible. However, using -Dsrv PRGRS_PREPCACHE,0 instructs the DataServer to re-prepare each SQL statement.
- Use the PRGRS_PREPCACHE option to control the size of the cache. The default cache size is 20 statements. You can increase the size for large applications that reuse many queries. The maximum size depends on the amount of resources you have available.

6.2 ODBC Driver Problems on Windows Platforms

The ODBC drivers use the Microsoft Windows DLL technology. A driver problem can leave the driver's DLL in an unstable state. In this case, the next attempt to connect to the driver might fail with a GPF or with an unexpected error message. If this occurs, you might need to unload the driver's DLL by using Windows NT utilities or by restarting Windows NT.

In addition, drivers utilized by ODBC can get overlaid with incompatible DLL versions as other products are installed on your system. You might need to reinstall ODBC drivers and products if such incompatibilities occur.

A

Migrating from ODBC DataServer

This appendix explains how to migrate from the DataServer for ODBC to the DataServer for SQL Server™. Specifically, this appendix documents the following:

- Upgrading to SQL Server™ Version 7 or later
- Creating a schema holder
- Modifying your application

A.1 Upgrading to SQL Server 7™ or Later

Before you upgrade your application to work with the DataServer for SQL Server™, Progress recommends that you follow the instructions from Microsoft to upgrade your Version 6.5 database and then test your application using the DataServer for ODBC. This will allow you to find differences that might exist between the different versions of SQL Server™.

After you have tested your application with the DataServer for ODBC, you can migrate your application to use with the DataServer for SQL Server™.

A.2 Creating a Schema Holder

The MS SQL Server Utilities are located under the DataServer menu option in the Data Administration Tool. Progress recommends that you use the Create DataServer Schema option to pull in all the objects that your application needs. This will create the schema information as a database type of MSS (Microsoft SQL Server™) instead of ODBC. After completing the pull, follow these steps:

1 ♦ Connect to the original Progress database.

2 ♦ Select the DataServer→ MS SQL Server Utilities→ Schema Migration Tools→ Adjust Schema option to fix the Progress attributes needed for your application to run. You can specify the entire schema holder or specific objects.

A.3 Modifying Your Application

The DataServer for SQL Server™ supports the same functionality as the DataServer for ODBC. Therefore, you do not have to change your application. However, if you used any reference to DBTYPE as ODBC, you will have to change these to MSS (Microsoft SQL Server™).

B

Stored Procedure Reference

This appendix contains reference entries for the following Progress 4GL extensions, which support sending SQL statements from within Progress procedures:

- CLOSE STORED–PROCEDURE statement
- PROC–HANDLE function
- PROC–STATUS function
- RUN STORED–PROCEDURE statement

CLOSE STORED–PROCEDURE Statement

For a non-Progress stored procedure, PROC-STATUS indicates that the procedure has completed execution and retrieves any return status. For a send–sql–statement stored procedure, CLOSE STORED–PROCEDURE closes the SQL cursor used by the procedure.

SYNTAX

```
CLOSE STORED-PROCEDURE procedure
  [ integer-field = PROC-STATUS ]
  [ WHERE PROC-HANDLE = integer-field ]
```

procedure

The name of the stored procedure that you want to close or the built-in procedure name, send–sql–statement.

integer-field = PROC-STATUS

Assigns the return value from a stored procedure to the specified integer field or variable (*integer-field*).

WHERE PROC-HANDLE = *integer-field*

An integer field or variable whose value uniquely identifies the stored procedure that produces the results returned from the data source or the SQL cursor of a send–sql–statement stored procedure.

NOTES

- If you specified a PROC–HANDLE when you ran a stored procedure, you must specify the PROC–HANDLE when you close the stored procedure.
- If you do not specify a PROC–HANDLE, the CLOSE STORED–PROCEDURE statement closes the procedure if there is only one stored procedure running. If there is more than one stored procedure running, an error is returned.
- You cannot close a send–sql–statement procedure until you have retrieved all row results.
- You can close all stored procedures at once with the following statement:

```
RUN STORED-PROC closeallprocs.
```

SEE ALSO

PROC–HANDLE Function, PROC–STATUS Function, RUN STORED–PROCEDURE Statement

PROC–HANDLE Function

SYNTAX

```
PROC-HANDLE
```

EXAMPLE

This procedure runs the stored procedure pcust and writes the procedure handle to the variable hdl. It writes the results of the stored procedure identified by this procedure handle into the Progress-supplied buffer, proc–text–buffer, and displays it as:

```
DEFINE VAR hdl AS INTEGER.

RUN STORED-PROCEDURE pcust hdl = PROC-HANDLE
(10, OUTPUT 0, OUTPUT 0).
FOR EACH proc-text-buffer WHERE PROC-HANDLE = hdl:
  DISPLAY proc-text.
END.
CLOSE STORED-PROCEDURE pcust WHERE PROC-HANDLE = hdl.
```

NOTES

- Progress Software Corporation recommends that you specify a procedure handle for each stored procedure that you run.
- You do not have to specify a handle if there is only one active stored procedure and you do not include SQL statements in the Progress application.

SEE ALSO

CLOSE STORED–PROCEDURE Statement, PROC–STATUS Function, RUN STORED–PROCEDURE Statement

PROC–STATUS Function

SYNTAX

```
PROC-STATUS
```

EXAMPLE

This procedure runs the stored procedure pcust and writes the results of the stored procedure into the Progress-supplied buffer, proc–text–buffer. The CLOSE STORED–PROCEDURE statement then retrieves the output parameters. The return status is written to the variable stat and is displayed as:

```
DEFINE VAR stat AS INTEGER.

RUN STORED-PROCEDURE pcust (10, OUTPUT 0, OUTPUT 0).
FOR EACH proc-text-buffer:
END.
CLOSE STORED-PROCEDURE pcust stat = PROC-STATUS.
  DISPLAY stat.
```

NOTE: For descriptions of the possible values for the return status of a non-Progress stored procedure, see your non-Progress documentation.

SEE ALSO

CLOSE STORED–PROCEDURE Statement, PROC–HANDLE Function, RUN STORED–PROCEDURE Statement

RUN STORED–PROCEDURE Statement

Runs a non-Progress stored procedure or allows you to send SQL to an SQL Server™ based data source using a Progress DataServer.

SYNTAX

```
RUN STORED-PROCEDURE procedure
  [ integer-field = PROC-HANDLE ]
  [ NO-ERROR ]
  [ ( parameter [ , parameter ] ... ) ]
```

procedure

The name of the stored procedure that you want to run or the Progress built-in procedure name, send–sql–statement, to send SQL to an SQL Server™ based data source.

integer-field = PROC-HANDLE

Assigns a value to the specified integer field or variable (*integer-field*) that uniquely identifies the stored procedure returning results from the non-Progress database or that uniquely identifies the SQL cursor used to retrieve results from a SQL Server™ data source.

NO-ERROR

Specifies that any ERROR conditions that the RUN STORED–PROCEDURE statement produces are suppressed. Before you close a stored procedure, check the ERROR–STATUS handle for information on any errors that occurred. You receive an error when you attempt to close a stored procedure that did not start.

NOTE: This option must appear before any run-time parameter list.

parameter

A run-time parameter to be passed to the stored procedure. A *parameter* has the following syntax:

SYNTAX

```
[ INPUT | OUTPUT | INPUT-OUTPUT ]
  [ PARAM parameter-name = ] expression
```

An *expression* is a constant, field name, variable name, or expression. INPUT is the default. OUTPUT and INPUT–OUTPUT parameters must be record fields or program variables.

If you run send–sql–statement for an SQL-based data source, you must pass a single character expression *parameter* containing the SQL statement you want the data source to execute.

If you do not specify *parameter–name* (the name of a keyword parameter defined by the stored procedure), you must supply all of the parameters in correct order. If you do specify *parameter–name*, you must precede your assignment statement with the keyword PARAM. If you do not supply a required parameter, and no default is specified in the stored procedure, you receive a run-time error.

EXAMPLES

This procedure runs the stored procedure pcust and writes the results of the stored procedure into the Progress-supplied buffer, proc–text–buffer:

```
DEFINE VAR hdl AS INTEGER.

RUN STORED-PROCEDURE pcust hdl = PROC-HANDLE NO-ERROR
(10, OUTPUT 0, OUTPUT 0).
IF ERROR-STATUS:ERROR
THEN DO:
  MESSAGE "Stored Procedure failed to run".
END.
FOR EACH proc-text-buffer WHERE PROC-HANDLE = hdl:
  DISPLAY proc-text-buffer.
END.
CLOSE STORED-PROCEDURE pcust WHERE PROC-HANDLE = hdl.
```

This procedure uses the send–sql–statement option of the RUN STORED–PROCEDURE statement to send SQL to SQL Server™. It writes the results of the stored procedure into the Progress-supplied buffer, proc–text–buffer:

```
DEFINE VAR handle1 AS INTEGER.

RUN STORED-PROC send-sql-statement handle1 = PROC-HANDLE
     ("SELECT name, cust_num FROM customer").
FOR EACH proc-text-buffer WHERE PROC-HANDLE = handle1:
   display proc-text.
END.
CLOSE STORED-PROC send-sql-statement WHERE PROC-HANDLE = handle1.
END.
```

This code example shows how to trap errors from the SQL Server™ RDBMS within a procedure:

```
DEFINE VAR h1 AS INTEGER.
DEFINE VAR j AS INTEGER.
RUN STORED-PROC send-sql-statement h1 = PROC-HANDLE NO-ERROR
  ("select count (*) from pcust.customer where name between 'A' and 'Z' ").

IF ERROR-STATUS:ERROR THEN DO:
  DO j = 1 TO ERROR-STATUS:NUM-MESSAGES:
    MESSAGE "error" ERROR-STATUS:GET-NUMBER(j)
        ERROR-STATUS:GET-MESSAGE(j).
  END.
END.

CLOSE STORED-PROC send-sql-statement WHERE PROC-HANDLE = h1.
```

NOTE: The RUN STORED–PROCEDURE statement starts a transaction with the same scope as transactions started with the UPDATE statement.

SEE ALSO

CLOSE STORED–PROCEDURE Statement, PROC–HANDLE Function, PROC–STATUS Function

C

Building DataServer Executables

This appendix describes how to build DataServer executables for all possible configurations on UNIX. The instructions apply to building executables on NT and Windows also. Specifically, this appendix includes information on:

- Overview of the building process
- PROBUILD utility
- Building executables on the UNIX client
- Setting environment variables
- Linking DataServer executables

C.1 Overview

Building executables for the DataServer for SQL Server™ is optional. The following sections provide detailed instructions on using the Progress PROBUILD utility to build executables. These instructions apply if you are building custom executables by choice.

Table C–1 lists the possible configurations that include UNIX and describes what you must do to set up the client and server modules.

Table C–1: Building DataServer Processes

Client	Server	Building and Configuring
Windows or NT	Remote NT	On Windows, use the default Progress client executable (`prowin32.exe`). On the NT host machine, build two executables: - The remote DataServer broker - The remote SQL Server™ DataServer This configuration is available only with the Enterprise DataServer for SQL Server™.
UNIX	Remote NT	On the UNIX client machine, build a client executable that includes the remote SQL Server™ DataServer as a configurable element using PROBUILD. Building the client is optional—by default, the UNIX client executable supports access to the remote DataServer. On the NT host machine, configure the default broker executable using ProControl on NT. This configuration is available only with the Enterprise DataServer for SQL Server™.

C.2 PROBUILD and the DataServer

Building a customized executable with PROBUILD involves these steps:

1. Run the PROBUILD utility to build a link script that includes the DataServer objects in your Progress modules. You must build executables on the platform where you will run them.

2. Set the environment variables required during the linking process.

3. Link the objects together to create an executable Progress module.

The PROBUILD utility allows you to customize Progress by letting you include various software modules or products. For each product you select, the PROBUILD utility prompts you to enter a link script name and an executable name. PROBUILD then displays a list of configurable elements for that product. You select the elements you want to include.

C.2.1 PROBUILD Requirements

The PROBUILD utility has software and environment requirements. Follow these steps to set up your environment to run PROBUILD:

1 ♦ Run PROBUILD to create executables on the machine where you plan to run them. For example, if you are building a remote DataServer configuration, to build the broker and the DataServer, run PROBUILD on the host. To build the client, run PROBUILD on the client machine. If you are building executables for deployment, be sure to build the executables on the same platform where your users will run them.

2 ♦ Make sure that you have a linker installed on your system.

3 ♦ Verify settings for Progress environment variables. Set $DLC to the directory where you have installed Progress.

4 ♦ Verify that the `buildenv` script (located in the `$DLC/probuild/eucapp` directory) includes the appropriate environment-variable settings for your system.

The link script that PROBUILD creates calls the `buildenv` script. The `buildenv` script sets default values for environment variables that point to objects, libraries, and options involved in linking executables. You can edit this script to customize environments for different sites.

PROBUILD also generates a log file, `buildenv.log`, which includes information about how your DataServer environment is configured. This information is useful to you and Technical Support when troubleshooting connection problems.

For more information on the PROBUILD utility, see the *Progress Client Deployment Guide*. For more information on environment variables, see the *Progress Installation and Configuration Guide Version 9 for UNIX* or the *Progress Installation and Configuration Guide Version 9 for Windows*.

After setting up the PROBUILD environment, you can run PROBUILD to generate link scripts that create the executables required by the various DataServer configurations. The following sections describe how to build, set up, and run client and host DataServer modules.

C.3 Building Executables on the UNIX Client

You can build a client executable for accessing a remote DataServer on NT. The default UNIX client executable supports access to the remote DataServer.

This section contains instructions for building a client executable. To run the PROBUILD utility and generate the link script for the client executable, follow these steps:

1 ♦ Set up the PROBUILD environment as described in the "PROBUILD and the DataServer" section.

2 ♦ Enter **$DLC/probuild/eucapp/probuild** at the system prompt. The Progress End User Configuration display appears.

3 ♦ Enter the directory where you want to put the link script (and ultimately, the executable for the client). The default is your current directory. If you leave the Install Link Script Field empty, the utility places the link script in your current directory.

4 ♦ Choose Continue. A display that includes a product list appears.

5 ♦ Select PROGRESS CLIENT, then choose Continue. A display containing several prompts appears.

6 ♦ Type names for the link script and the executable.

The default link script name for the UNIX client is `ldpro`. The default executable name is `_progress`. Progress Software Corporation recommends that you do not use the default names for custom executables.

7 ♦ Choose Continue. A list of configurable elements appears.

8 ♦ Select REMOTE SQL Server DS.

9 ♦ Select TCP.

10 ♦ Select DataServer Library.

You can include other elements from this list to further customize your Progress client module.

11 ♦ Choose Continue to return to the product list.

12 ♦ Choose Return, then choose Exit to exit PROBUILD.

PROBUILD builds a link script that you then run to create a customized Progress executable that references the SQL Server™ database objects for the DataServer.

Set client DataServer environment variables and run the link script to create the client executable, then set up the host modules for your configuration.

C.4 Setting Environment Variables

After you run the PROBUILD utility to create your link scripts or command procedures, you must set some of the DataServer environment variables or logicals on the host machine before you run the scripts to create the executables. The others you can set after linking the executables. Be sure to set and export the environment variables in the same environment from which you plan to link and run the executables. Some environment variables (for example the library path variables) are required, while others are optional.

Once you set these variables, you can run the link scripts you created with PROBUILD to generate executables. See the "Linking DataServer Executables" section for instructions.

C.4.1 Linking DataServer Executables

After using PROBUILD to generate the link scripts for the DataServer executables and setting environment variables as required, you must run the link scripts to create the executables.

Run the link scripts that you generated with PROBUILD by entering the link script name at the system prompt.

You can now use the executables to start the DataServer and build a schema holder. For instructions, see Chapter 3, "Configuring the DataServer," and Chapter 4, "Connecting the DataServer."

CAUTION: Do not attempt to run your PROBUILD executable until you finish configuring and connecting it. The attempt will be unsuccessful.

D

Server Related Command Line Utilities and Startup Parameters

This appendix describes the following utilities and parameters that you use to configure, manage, start, and stop the DataServer host and client:

- Progress Explorer command line utilities allow you to start, stop, and configure the Progress Explorer tool broker components. See the *Progress Installation and Configuration Guide Version 9 for Windows* for additional information about the utilities and their role and relationship to other system administration facilities.
- Startup commands and parameters for UNIX and Windows allow you to start and manage DataServer clients. See the *Progress Startup Command and Parameter Reference* for additional information about syntax and usage.

D.1 Progress Explorer Command Line Utilities for the DataServer

This section describes the utilities you use to configure, manage, start, and stop a DataServer. It discusses the purpose, syntax, and primary parameters for each operating system.The utilities are presented in alphabetical order.

MSSCONFIG Utility

Use the MSSCONFIG utility to help you debug existing DataServer for SQL Server™ configurations defined in a properties file, such as the `ubroker.properties` file. This utility displays the property settings associated with a DataServer for SQL Server™ configuration, and checks that the syntax and values are valid.

The MSSCONFIG utility runs locally, on the machine on which the AdminServer is running. The utility does not run across the network:

`-propfile` *`path-to-properties-file`*

Specifies a filename or pathname to a file that contains the property settings to be validated, for example `test.properties`. If a filename or pathname is not specified, it defaults to the installation version of the `ubroker.properties` file, such as `%DLC%\properties\ubroker.properties` on Windows NT.

SYNTAX

Operating System	Syntax

Windows	`mssconfig`
	`[`
	`[`
	`[ -name` *`DataServer-name`* `]`
	`[ -propfile` *`path-to-properties-file`* `]`
	`[ -validate ]`
	`]`
	`| -help`
	`]`

PARAMETERS

`-name` *DataServer-name*

Specifies which existing configuration to examine. The name must match the name of an existing DataServer for SQL Server™ configuration defined in the specified properties file. If you do not specify a DataServer by name, the MSSCONFIG utility analyzes all DataServer for SQL Server™ configurations defined in the properties file specified by the `-propfile` parameter.

`-validate`

Checks the syntax and values of property settings defined in the specified properties file.

`-help`

Displays command-line help.

NOTES

- The `ubroker.properties` file stores all the configuration definitions for each instance of the NameServer, AppServer, DataServer and WebSpeed Transaction Server products. Each configuration definition contains environment variables, registry entries if Windows NT, and property settings for each product instance. Progress Explorer and certain command-line utilities such as MSSCONFIG, use this file to store, validate and manage the configurations for the products.
- The `ubroker.properties` file is installed in the `properties` subdirectory of the Progress installation directory. For example, `%DLC%\properties\ubroker.properties` on Windows NT.
- The file consists of a hierarchical structure of configuration entities, where parent entities provide configuration information that you can override or extend in each child entity. Each configuration entity has a name that begins the entity definition, and the definition contains configuration settings for one or more product instances.

For example, the DataServer for SQL Server™ configurations in `ubroker.properties` may include:

Configuration Entity Name	Configuration Entity Function
`[UBroker]`	Defines default property settings for all NameServer, AppServer, DataServer, and WebSpeed Transaction Server brokers.
[UBroker.MS]	Defines default property settings for all instances of DataServers for SQL Server™.
[UBroker.MS.*product-instance-name*]	Defines property settings for this instance of a DataServer for SQL Server™. The `ubroker.properties` file may contain several of these entities each with a unique *product-instance-name*.

- Parent entities provide default values for all of their child entities. For example, the parent `[UBroker]` contains a set of definitions that can be inherited by its child `[UBroker.MS]`, and then again by its child `[UBroker.MS.`*product-instance-name*`]`. However, at any child level, a redefinition of any value supersedes the default value of its parent. All children from the redefinition level down inherit this new value.
- Optionally, you may edit the `ubroker.properties` file using a text editor such as vi or Notepad. If you want to manually edit this file to create or modify a product configuration, begin by making a backup copy from the installed `ubroker.properties` file (and naming it for example, `test.properties`). Once you edit the properties file, use the relevant validation utility such as MSSCONFIG to validate the changes and make sure there are no syntax errors or conflicts.

MSSMAN Utility

Use the MSSMAN utility to control the operation of a configured DataServer for SQL Server™. The utility allows you to start a broker, query its status, and start and stop additional DataServer servers, trim by a certain number of servers, and shut down the broker.

SYNTAX

Operating System	Syntax

Windows	mssman
	{
	{ -name *DataServer-name*
	{
	-kill
	| -start
	| -stop
	| -query
	}
	[
	-host *host-name* -user *user-name*
	| -user *user-name*
	]
	[-port *port-number*]
	}
	| -help
	}

PARAMETERS

-name *DataServer-name*

This parameter is required. It specifies the name of a broker.

-kill

Stops and removes the DataServer from memory, no matter what it is doing.

-start

Starts the DataServer.

`-stop`

Tells the DataServer to stop itself.

`-query`

Queries the DataServer for its status.

`-host` *`host-name`*

Specifies the name of the machine where the AdminServer is running. If a host name is not specified, it defaults to the local host name.

`-user` *`user-name`*

Specifies a user name and prompts for a password. A user name and password are required only when you use the `-host` parameter and specify a remote host name. If you specify a remote host name with the `-host` parameter, but do not specify a user name with the `-user` parameter, you receive a prompt for a username and password.

`-port` *`port-number`*

Specifies the port number of the machine on which the AdminServer is running. If a port number is not specified, it defaults to 20931.

`-help`

Displays command-line help.

NOTES

- When you specify a user name with the `-user` parameter, Windows NT supports three different formats:
  - A user name as a simple text string, such as "`mary`," implies a local user whose user account is defined on the local NT server machine, which is the same machine that runs the AdminServer.
  - A user name as an explicit local user name, in which the user account is defined on the same machine that runs the AdminServer except the user name explicitly references the local machine domain, for example "`.\mary`".
  - A user name as a user account on a specific NT domain. The general format is *`Domain`*`\`*`User`*, in which the *`User`* is a valid user account defined within the domain and the *`Domain`* is any valid NT Server, including the one where the AdminServer is running.

NSCONFIG Utility

Use the NSCONFIG utility to help you debug existing NameServer configurations defined in a properties file, such as the `ubroker.properties` file. This utility displays the property settings associated with a NameServer configuration, and checks that the syntax and values are valid.

The NSCONFIG utility runs locally, on the machine on which the AdminServer is running. The utility does not run across the network.

SYNTAX

Operating System	Syntax
Windows	`nsconfig [[[-name name-server] [-propfile path-to-properties-file] [-validate]]	-help]`

PARAMETERS

`-name` *name-server*

Specifies which existing NameServer configuration to examine. The name must match the name of an existing NameServer configuration in the specified properties file. If you do not specify a NameServer, the NSCONFIG utility analyzes all NameServer configurations defined in the properties file specified by the `-propfile` parameter.

`-propfile` *path-to-properties-file*

Specifies a filename or pathname to a file that contains the property settings to be validated, for example `test.properties`. If a filename or pathname is not specified, it defaults to the installation version of the `ubroker.properties` file, such as `%DLC%\properties\ubroker.properties` on Windows NT.

`-validate`

Checks the syntax and values of property settings defined in the specified properties file.

`-help`

Displays command-line help.

NOTES

- A single NameServer can simultaneously support all of the AppServer, WebSpeed and DataServer products using Progress Explorer.
- The `ubroker.properties` file stores all the configuration definitions for each instance of the NameServer, AppServer, DataServer and WebSpeed Transaction Server products. Each configuration definition contains environment variables, registry entries if Windows NT, and property settings for each product instance. Progress Explorer and certain command-line utilities, such as NSCONFIG, use this file to store, validate and manage the configurations for the products.

  The `ubroker.properties` file is installed in the `properties` subdirectory of the Progress installation directory. For example, `%DLC%\properties\ubroker.properties` on Windows NT.

  The file consists of a hierarchical structure of configuration entities, where parent entities provide configuration information that you can override or extend in each child entity. Each configuration entity has a name that begins the entity definition, and the definition contains configuration settings for one or more product instances.

For example, the NameServer configurations in `ubroker.properties` may include:

Configuration Entity Name	Configuration Entity Function
[UBroker]	Defines default property settings for all NameServer, AppServer, DataServer, and WebSpeed Transaction Server brokers.
[NameServer]	Defines default property settings for all instances of a NameServer.
[NameServer.*product-instance-name*]	Defines property settings for this instance of an NameServer. The `ubroker.properties` file may contain several of these entities each with a unique *product-instance-name*.

Parent entities provide default values for all of their child entities. For example, the parent `[UBroker]` contains a set of definitions that can be inherited by its child `[NameServer]`, and then again by its child `[NameServer.`*product-instance-name*`]`. However, at any child level, a redefinition of any value supersedes the default value of its parent. All children from the redefinition level down inherit this new value.

Optionally, you may edit the `ubroker.properties` file using a text editor such as vi or Notepad. If you want to manually edit this file to create or modify a product configuration, begin by making a backup copy from the installed `ubroker.properties` file (and naming it for example, `test.properties`). Once you edit the properties file, use the relevant validation utility such as MSSCONFIG to validate the changes and make sure there are no syntax errors or conflicts.

NSMAN Utility

Use the NSMAN utility to control the operation of a configured NameServer. The utility allows you to start a NameServer, query its status, and shut down a NameServer.

SYNTAX

Operating System	Syntax

Windows	nsman
	{
	{ -name *name-server*
	{
	-kill
	| -start
	| -stop
	| -query
	}
	[
	-host *host-name* -user *user-name*
	| -user *user-name*
	]
	[-port *port-number*]
	}
	| -help
	}

PARAMETERS

-name *name-server*

This parameter is required. It specifies the name of the NameServer.

-kill

Stops and removes the NameServer from memory, no matter what it is doing.

-start

Starts the NameServer.

`-stop`

Tells the NameServer to stop itself.

`-query`

Queries the NameServer for its status.

`-host` *`host-name`*

Specifies the name of the machine where the AdminServer is running. If a host name is not specified, it defaults to the local host name.

`-user` *`user-name`*

Specifies a user name and prompts for a password. A user name and password are required only when you use the `-host` parameter and specify a remote host name. If you specify a remote host name with the `-host` parameter, but do not specify a user name with the `-user` parameter, you receive a prompt for a user–name and password.

`-port` *`port-number`*

Specifies the port number of the machine on which the AdminServer is running. If a port number is not specified, it defaults to 20931.

`-help`

Displays command-line help.

NOTES

- A single NameServer can simultaneously support all of the AppServer, WebSpeed and DataServer products.
- When you specify a user name with the `-user` parameter, Windows NT supports three different formats:
  - A user name as a simple text string, such as "`mary`," implies a local user whose user account is defined on the local NT server machine, which is the same machine that runs the AdminServer.
  - A user name as an explicit local user name, in which the user account is defined on the same machine that runs the AdminServer except the user name explicitly references the local machine domain, for example "`.\mary`".

– A user name as a user account on a specific NT domain. The general format is *Domain\User*, in which the *User* is a valid user account defined within the domain and the *Domain* is any valid NT Server, including the one where the AdminServer is running.

D.2 Non-Progress Explorer Command-line Utilities for the DataServer

This section describes the command line utilities you use to start and stop a DataServer. It discusses the purpose, syntax, and primary parameters for each operating system. The utilities are presented in alphabetical order.

PROBRKR Command

Starts the DataServer broker. To use the DataServer from a remote client, you must first start the broker. Once you start the broker, it can receive the client requests and spawn the DataServer.

SYNTAX

Operating System	Syntax
Windows	probrkr *dbname* -S *service-name* [-H *host-name*] [-N *network-type*]

PARAMETERS

dbname

Specifies the name of the database where you are connecting to.

service-name

Specifies the name of the broker process on the host machine.

host-name

Specifies the name of the machine where the DataServer broker is installed. The default value is the current host.

network-type

Specifies the network protocol for which the broker will run.

Progress supports the TCP protocol.

NOTES

- See the *Progress Startup Command and Parameter Reference* for more details on the Server Name (-S), Host Name (-H), and Network Type (-N) startup parameters.
- You can use any of the startup parameters with the PROBRKR command. See the *Progress Startup Command and Parameter Reference* for details.
- You must start the remote broker in the same machine where your ODBC data source names (DSNs) are defined because the servers spawned by the broker inherit the setup of the environment from the broker. For example, set the environment variable MSSSRV to the name of the executable (including the path) of the DataServer for SQL Server™. Be sure to set this variable on the host machine. Also, in the same environment, make sure you have set all MSS (Microsoft SQL Server™) environment variables required to connect to the data source. See Chapter 3, "Configuring the DataServer," for examples of required variables.
- Start the broker on a node that is locally connected to the disk containing the data source.

PROSHUT Command

Shuts down the Progress database server and individual Progress processes. Before you shut down the server, have all application users quit their Progress sessions. If necessary, you can disconnect users by using the PROSHUT command's Disconnect a User or Unconditional Shutdown parameters.

SYNTAX

Operating System	Syntax
NT	proshut *db-name* [-b | -by | -bn | -H *host-name* | -S *service-name* | -N *network-type* | -F | -Gw | -C list | -C disconnect *usernum*]

PARAMETERS

db–name

Specifies the database the server is running against.

-b

Directs Progress to perform a batch shutdown. When no client is connected, the database automatically shuts down. When one or more clients are connected, PROSHUT prompts the user to enter "yes" to perform an unconditional batch shutdown and to disconnect all active users; or "no" to perform a batch shutdown only if there are no active users. The -b parameter combines the functionality of the -by or -bn parameters.

-by

Directs Progress to perform an unconditional batch shutdown and to disconnect all active users.

Using -by in conjunction with -F causes an emergency shutdown.

-bn

Directs Progress to perform a batch shutdown only if there are no active users.

-H *host-name*

Specifies the machine where the database server runs. You must specify the host name if you issue the shutdown command from a machine other than the host.

-N *network-type*

Specifies the networking protocol used by the requesting application to connect to the Progress DataServer. This optional parameter is always TCP.

-S *service-name*

Specifies the database server or broker process. You must specify the service name if you issue the shutdown command from a machine other than the host.

-F

Starts an emergency shutdown. To use this parameter, you must run PROSHUT on the machine where the server resides. This parameter is not applicable for remote shutdowns or DataServer shutdowns.

-Gw

For DataServers, specifies that a DataServer broker is to be shut down.

When you enter the PROSHUT command without the -by, -bn, or -F parameters, the following menu appears:

```
1 Disconnect
2 Unconditional Shutdown
3 Emergency Shutdown (Kill All)
x Exit
```

The following table lists the menu options and their actions:

Option	Action
1	Prompts you for the number of the user you want to disconnect.
2	Disconnects all users and shuts down the database.
3	Prompts you to confirm your choice. If you cancel the choice, you cancel the shutdown. If you confirm the choice, Progress displays the following message: Emergency shutdown initiated... PROSHUT marks the database for abnormal shutdown, kills all remaining processes connected to the database, and deletes shared-memory segments and semaphores. The database is in a crashed state. Progress performs normal crash recovery when you restart the database and backs out any active transactions.
x	Cancels the shutdown without taking any action.

`-C list`

Lists all of the users connected to the database. The list is printed out to the screen without any page breaks.

`-C disconnect` *`usernum`*

Allows you to initiate a disconnect for the specified user. This is similar to option 1 of the PROSHUT menu.

NOTES

- The PROSHUT command runs the following executable:

```
_mprshut
```

- You can access PROSHUT from the PROMON utility by using the Shut Down Database qualifier.

- The user who shuts down the server must have been the user who started it.
- When you initiate PROSHUT over a network, the amount of time that it takes to actually shut down all of the Progress processes and to free any ports varies depending on the number of clients, brokers, and servers that must be shut down. The PROSHUT utility might return control to the terminal before all of the processes are stopped and resources are freed.

D.3 DataServer Startup Parameters

Table D–1 lists the parameters that you use with the command line utilities to start a DataServer for SQL Server™.

Table D–1: DataServer Parameters for UNIX and Windows

Parameter	Syntax
DataServer	-Dsrv *keyword, value*
Database Type	-dt *db-type*
Host Name	-H *host-name*
Service name	-S *service-name*
Network protocol	-N *protocol-type*
Data Source User Name	-U *user-name*
Data Source User Name Password	-P *password*
Server Join	-nojoinbysqldb

Index

A

Aggregates
  in views 2–12

Analyzing performance 4–20

Arrays 2–16
  in joins 2–54
  manually creating support 5–22
  supporting in data source 5–22

Audience xi

Auto-commit 6–5

Auto-connect
  failure response 4–23

B

-b startup parameter
  PROSHUT command D–14

BEGINS function 2–34

-bn shutdown parameter D–15

Bold typeface
  as typographical convention xiii

Brokers
  DataServer D–13
  shutting down
    permissions D–17
  starting from MS-DOS 3–9, 4–6

Buffers
  allocating dynamically 6–7
  defining for stored procedures 2–42

BY option
  FOR EACH statement 2–9
  joins 2–54

-by shutdown parameter
  PROSHUT command D–15

C

-C disconnect usernum qualifier
  PROSHUT command D–16

-C list qualifier
  PROSHUT command D–16

Cache files
  changes to schema holders 4–22
  creating 4–12
  local schema cache 4–21

Caching
  schema 1–7

Case-insensitive indexes
supporting in data source 5–22

Case-sensitive indexes 2–10

Character sets 2–6
changing in a schema holder 5–20

CLOSE STORED-PROCEDURE statement 2–39, B–2
retrieving parameter values 2–41

Code pages 2–6
changing in a schema holder 5–20
collation table 2–8
conversion table 2–8
setting in schema holder 3–20

COMPILE statement 2–32

Configuration examples
local DataServer 4–13
remote DataServer 4–14

CONNECT statement
failure response 4–22
NO-ERROR option 4–22

Connecting schema holders 4–9
at startup 4–10
connection order 4–10

Connection to data source
blank space in data source name 4–16
connection order 4–10
DataServer opens multiple connections 4–24
failure response due to existing connection 4–23
failure response due to logical name 4–23
failures with SQL Server 4–22
guidelines 4–8
interactive connection 4–17
non-standard connections 4–16
security on NT 4–8
troubleshooting 4–23, 6–5

CONTAINS operator 2–34

COUNT-OF function 2–34

CREATE statement 2–18, 2–34

Creating Progress databases 3–18

Cursors 2–27
multiple 4–24

D

Data Dictionary
connection failure response 4–23
modifying schema holders 5–15

Data sources 1–4
adding objects to schema holder 3–22
code page 3–20
data types 2–13
database objects 2–3
keys 2–9
locking behavior 2–19
logical name 3–19
login name 2–35, 3–17
naming conventions 2–4
null value 2–16
password 2–35, 3–17
permissions 3–16
physical name 3–20
registering 3–4
security levels 1–9
stored procedures 2–35
triggers 2–12
views 2–11
zero-length character strings 2–16

Data types 2–13
changing 5–18
display format 5–18
stored procedures 2–36
supported by DataServer 2–13
user-defined 2–15

Database conversion
Progress-to-MSS utility 5–24

Database Type (-dt) startup parameter 4–11

Databases
Progress objects 2–3
SQL Server data-source objects 2–3
triggers 2–12

DataServer
architecture 1–5
architecture components 1–4
broker
Auto Start during host boot cycle 3–9
maximum name length 3–8
on host machine 3–5
building executables C–3
compiling applications 2–32
components 3–2
configurations 1–10
configuring for distributed applications
using probrkr 1–14
using Progress Explorer 1–15
configuring with Progress Explorer on NT 3–6
connection failure 4–22
creating schema holders 3–19
diagnostic options 4–20
enhancing performance 2–47
error handling 2–24
internal logic 1–7
local configuration 1–11
log file 3–10, 4–25
on host machine 3–5
options 6–4
record creation 2–17
remote configuration 1–12
scrolling 2–27
security considerations 1–8
SQL Server utilities 1–4
starting
in Progress Explorer 4–3
MS Windows client 4–7
NT broker processes 4–6
UNIX client 4–7
utilities 1–9

DataServer (-Dsrv) startup parameter 4–12
alternate syntax 4–17
DataServer options 6–5
ODBC options 6–4
PRGRS_CONNECT option 4–16
qt_debug option 4–18
qt_lookahead option 4–18
qt_no debug option 4–18
qt_no_lookahead option 4–18
qt_no_separate_connection option 4–19
qt_separate_connection option 4–19
query-tuning 4–17
stored procedures 2–36
tuning your environment 6–2

DataServers
starting a broker D–12

Datetime data type
on Microsoft SQL Server 2–15

DBE DataServer
startup parameters 5–25

DBRESTRICTIONS function 2–34

Deadlock 2–23
stored procedures 2–36

DEFINE BROWSE statement 2–30

DEFINE BUFFER statement 2–42

Deleting schema holders 5–21

Demonstration databases 1–10, 5–2
creating 5–2
requirements for use 5–2

Diagnostic options
qt_debug 4–20

Disconnect usernum qualifier
PROSHUT command D–16

Display formats 5–18

Distributed DataServer
configured using probrkr 1–14
configured using Progress Explorer 1–15

Documentation
suggested reading list 1–19

DSLOGDIR environment variable 3–10, 4–25

Dummy indexes
for non-indexed data sources 2–9

E

Enterprise DataServer for SQL Server 1–2, 1–4, 3–2

Environment variables
DSLOGDIR 3–10
MSSSRV 3–10
PROBRKR 3–10
UNIX C–3

Error handling 2–24
stored procedures 2–47

Error messages
connection failures 4–22
displaying descriptions xvii
locking problems 6–6

Exclusive locks 2–21

EXCLUSIVE-LOCK option 2–35
in joins 2–54
record locking 2–19

Executables
UNIX C–6

Extended 4GL
adding support to data source 5–22
supported features 1–17

F

Field extents 2–16

Field formats 5–18

Field lists 2–30

Field properties
modifying 5–17

Field size
retrieval limits 2–15

FIELDS option 2–30

FIND statement 2–9, 2–31, 2–34
LAST option 2–31
PREV option 2–31

FOR statement 2–33
EACH option 2–9, 2–34

Foreign keys 2–9

Format phrase
field size limits 2–15

G

-Gw shutdown parameter
PROSHUT command D–15

H

-H shutdown parameter
PROSHUT command D–15

Help, Progress messages xvii

Holding locks 2–21

Host Name (-H) shutdown parameter
PROSHUT command D–15

Host Name (-H) startup parameter 4–11

I

Indexes 2–9
case-sensitive 2–10
dummy 2–9
for stored procedure results 2–11
for views 2–11
problems importing 6–7
selecting for views 2–10

simulated 2–9
unique 2–10
FIND statements 2–10
for browsing 2–10

INPUT option
stored procedures B–7

INPUT-OUTPUT option
retrieving parameter values 2–41
returning parameter values 2–39
stored procedures B–7

Internal Code Page (-cpinternal) startup parameter 2–8
PROGRESS-to-MSS utility 5–25

Isolation levels 2–20

Italic typeface
as typographical convention xiii

J

JOIN-BY-SQLDB option 2–53

Joins 2–11
controlling by SQLDB behavior 2–55

K

Key buffer
troubleshooting 6–6

Keys
foreign 2–9
primary 2–9

Keystrokes xiii

L

List qualifier
PROSHUT command D–16

Local DataServer 1–11
starting 4–2

LocalCache (-cache) startup parameter 4–12

Locks
deadlock 2–23
page-level 2–20, 2–21
record locking 2–20
stored procedures 2–36
table-level 2–20, 2–21
upgrading 2–20

Log files 3–10
accessing 4–25

Logical database name 3–19
changing 5–14

Logical Database Name (-ld) startup parameter 4–11

Login names 3–17, 4–11, 4–24
SETUSERID function 2–35

M

Maintaining schema holders 5–2
recompiling code 3–23

Manual
syntax notation xiv

Manual, organization of xi

MATCHES function 2–34

Messages
displaying descriptions xvii

Microsoft SQL Server
data types 2–13

Modifying schema holders 5–15

Monospaced typeface
as typographical convention xiii

MSSCONFIG Utility D–2

MSSMAN Utility D–5
  DataServer processes on UNIX 4–5

MSSSRV environment variable 3–10

N

-N shutdown parameter
  PROSHUT command D–15

Naming conventions 2–4
  keywords in object names 2–5
  non-unique table names 2–5
  unsupported characters 2–5

Network Type (-N) shutdown parameter
  PROSHUT command D–15

Network Type (-N) startup parameter 4–11

No kill users (-bn) shutdown parameter
  PROSHUT command D–15

NO-JOIN-BY-SQLDB option 2–53

NO-LOCK option 2–35
  record locking 2–19

Non-unique table names 2–5

NO-WAIT option 2–23

NSCONFIG D–7

NSMAN D–10

NULL search condition 2–35

Null value
  in query 2–16
  ROWID 2–29

O

ODBC
  Application Programming Interface (API) 1–7
  compliance 1–1
  data source 1–4
  driver 1–4
    interactive connections 4–17
    NT 3–4
    troubleshooting 6–7
    unstable DLLs 6–7
    Windows 3–4
  Driver Manager 1–4
  options 6–3

OF option
  FIND statement 2–9
  FOR EACH statement 2–9

OPEN QUERY statement 2–34

OUTPUT option
  retrieving parameter values 2–41
  returning parameter values 2–39
  stored procedures B–7

P

PARAM option
  retrieving parameter values 2–41
  stored procedures B–7

Parameter (.pf) files
  connection requirements for DataServer 4–13

Parameters
  abbreviating names 6–2

Passing ODBC-specific information
  PRGRS_CONNECT option 4–16

Password (-P) startup parameter 4–12

Passwords 3–17, 4–24
  SETUSERID function 2–35

Performance
  analyzing 4–20
  caching records 2–52
  enhancing on DataServer 2–47
  joins 2–53
  modifying run-time schema check 2–57
  Progress query tips 2–48

query tuning 2–48
writing queries 2–56

Permissions
data source security 1–9
establishing 3–16

Personal DataServer for SQL Server 1–2, 1–4, 3–2

Physical database name 3–20

Physical Database Name (-db) startup parameter 4–11

PRGRS_CONNECT option
DataServer (-Dsrv) startup parameter 6–5

PRGRS_IDBUF option
DataServer (-Dsrv) startup parameter 6–6

PRGRS_LOCK_ERROR option
DataServer (-Dsrv) startup parameter 6–6

PRGRS_MNBUF option
DataServer (-Dsrv) startup option 6–7

PRGRS_PREPCACHE option
DataServer (-Dsrv) option 6–7

Primary keys 2–9

pro executable
UNIX example 4–8

PROBRKR command D–12

PROBRKR environment variable 3–10

PROBUILD utility C–3

Procedures
compiling 2–32

PROC-HANDLE function 2–45, B–4

ProControl utility
managing DataServer host environment 3–6, 3–8
starting brokers on NT 4–6

PROC-STATUS function 2–40, B–5

proc-text-buffer 2–39
database results 2–41
returns from stored procedures 2–37

Progress
data types 2–13
database objects 2–3
database security levels 1–8
locking behavior 2–19
naming conventions 2–4
triggers 2–12
unknown value 2–16
zero-length character strings 2–16

Progress Explorer
configuring the DataServer on NT 3–6
starting DataServer on NT 4–3
stopping DataServer on UNIX 4–5

PROGRESS-to-MSS utility 5–24, 5–27, 5–29
migrating Progress database to SQL Server data source 5–23
parameters 5–3, 5–5, 5–9, 5–26
running in batch mode 5–25
running interactively 5–24

PROLOAD environment variable C–3

PROSHUT command D–14

Q

qt_debug
CALL_SUMMARY option
DataServer (-Dsrv) startup parameter 4–18
CURSOR option
DataServer (-Dsrv) startup parameter 4–18
EXTENDED option
DataServer (-Dsrv) startup parameter 4–18
PERFORMANCE option
DataServer (-Dsrv) startup parameter 4–18

SQL option
DataServer (-Dsrv) startup parameter 4–18
VERBOSE option
DataServer (-Dsrv) startup parameter 4–18

qt_debug option
CALL SUMMARY option 4–21
CURSOR option 4–20
EXTENDED option 4–20
PERFORMANCE option 4–20
SQL option 4–20
VERBOSE option 4–21

qt_lookahead option
DataServer (-Dsrv) startup parameter 4–18

qt_no_debug option 4–20
DataServer (-Dsrv) startup parameter 4–18

qt_no_lookahead option
DataServer (-Dsrv) startup parameter 4–18

qt_no_separate_connection option
DataServer (-Dsrv) startup parameter 4–19
managing data sources 4–24

qt_separate_connection option
DataServer (-Dsrv) startup parameter 4–19
managing data sources 4–24

Query tuning
precedence of methods 4–17
startup parameters 4–17, 4–19

Query-tuning options 2–50
ARRAY-MESSAGE 2–50
CACHE-SIZE 2–50
DEBUG SQL 2–50
DEBUG-EXTENDED 2–50
JOIN-BY-SQLDB 2–51
LOOKAHEAD 2–51
NO-ARRAY-MESSAGE 2–50
NO-DEBUG 2–50
NO-JOIN-BY-SQLDB 2–51
NO-LOOKAHEAD 2–51
NO-SEPARATE-CONNECTION 2–51
SEPARATE-CONNECTION 2–51

QUERY-TUNING phrase 2–49

R

R-code 2–33
size 2–33

Read-only (-RO) startup parameter 4–10, 4–12

Read-only schema holders 4–12

RECID function 2–29
replacing with ROWID 2–29

Recompiling code
schema changes 3–23

Record caching
lookahead cursors 2–53
standard cursors 2–52

Record creation 2–17, 2–35

Record locking
data sources 2–19
exclusive locks 2–21
EXCLUSIVE-LOCK option 2–19
NO-LOCK option 2–19
shared locks 2–20
SHARE-LOCK option 2–19

Recovery
on data sources 2–24

Registering
data sources 3–4

RELEASE statement 2–18

Remote databases
starting a DataServer broker D–12

Remote DataServer 1–12
starting 4–3

Requirements
software 1–16

Roll back
on data sources 2–24

ROWID function 2–28
defining supporting index 5–18
listing objects not supported 3–22
manually creating support 5–22
supporting in data source 5–22

Rows
troubleshooting large rows 6–7

RUN STORED-PROCEDURE statement 2–37, B–6
retrieving parameter values 2–41
send-sql-statement option 2–46
syntax 2–39

S

-S shutdown parameter
PROSHUT command D–15

SAVE CACHE COMPLETE statement 4–12, 4–21

Schema
loading process 1–7

Schema caching 1–7

Schema holders 1–4, 1–7
changing logical name 5–14
connecting 4–9
connection failure 4–22
connection permissions 3–17
creating 3–16
deleting 5–21
maintaining 3–23, 5–2
modifying 5–15
setting code page 3–20
system administration permissions 5–6
updating 3–23, 5–6
upgrading version 7 to 8 A–2
upgrading version 8 to 9 A–2
verifying 5–9
where best to build 3–16

Schema images
creating and updating 3–16
security 3–16

Scrolling
backward 2–27
forward 2–27

Security
guideline considerations 1–8

send-sql-statement option
nchar and nvarchar data types 2–15
RUN-STORED-PROCEDURE statement 2–46

Sequence generator 2–12

Server Join (-nojoinbysqldb) startup parameter 2–51, 2–53, 4–19

Service Name (-S) shutdown parameter
PROSHUT D–15

Service Name (-S) startup parameter 4–11

SETUSERID function 2–35

Shared locks 2–20

SHARE-LOCK option 2–35
record locking 2–19

Shutdown commands D–14

Simulated indexes
for nonindexed data sources 2–9

Single-user Mode (-1) startup parameter 4–12

Skip schema verification 2–57

smalldatetime data type
on Microsoft SQL Server 2–13, 2–14, 2–15

Software requirements 1–16

SQL ORDER BY option
Progress USE-INDEX option 2–9

SQL Server
Administrator 1–4
creating schema holders 3–19

SQL statement cache
troubleshooting 6–7

SQL statements
sending to data source 2–46

Starting a DataServer broker D–12

Startup parameters
Database Type (-dt) 4–11
DataServer (-Dsrv) 4–12, 4–17
Host Name (-H) 4–11
Internal Code Page (-cpinternal) 2–6
LocalCache (-cache) 4–12
Logical Database Name (-ld) 4–11
Network Type (-N) 4–11
Password (-P) 4–12
Physical Database Name (-db) 4–11
Read-only (-RO) 4–10, 4–12
Server Join (-nojoinbysqldb) 4–12, 4–19
Service Name (-S) 4–11
Single-user Mode (-1) 4–12
unsupported 4–15
User ID (-U) 4–11

Stored procedures
browsing results 2–10
closing all 2–40
data types 2–36
defining 2–35
defining buffers 2–42
error handling 2–47
INPUT option B–7
INPUT-OUTPUT option B–7
joining database results 2–45
OUTPUT option B–7
PARAM option B–7
retrieving database results 2–41
retrieving parameters 2–39, 2–40
return status 2–40
returned values 2–37
returning status values B–2
running 2–35, 2–38
unique indexes for results 2–10

Stream Code Page (-cpstream) startup parameter 5–25

Syntax notation xiv

T

Table properties
modifying 5–15
ROWID 5–19

Table validation 5–16

Terminology
differences between data sources 2–3

Transaction scoping
Progress 2–24

Transactions 2–24

Transact-SQL 2–46

Triggers 2–12

Troubleshooting 6–1

Typographical conventions xiii

U

ubroker.properties file 3–11
hierarchy D–4, D–9

Unique indexes 2–10
stored procedures results 2–10

Unknown value
in query 2–16
ROWID 2–28

Unsupported 4GL statements 2–34

Updating schema holders 5–6

Upgrading
schema holders A–2

USE-INDEX option
FOR EACH statement 2–9

User ID 3–17
SETUSERID function 2–35
startup parameter (-U) 4–11

User-defined data types 2–15

USING phrase
joins 2–54

V

VALIDATE statement 2–18

Validation
expressions 5–16, 5–18
messages 5–16

Verifying schema holders 5–9

Views 2–11
aggregates 2–12
browsing 2–11

W

Word index
CONTAINS operator 2–34

Z

Zero-length character strings 2–16