Progress®



Use Microsoft SQL Server DataServer

OpenEdge®

Table of Contents

Copyright....11

Preface....13

Purpose....13
Audience....14
Organization....14
Use ABL documentation....15
References to ABL compiler and run-time features....15
References to ABL data types....15
Before you begin....16
Typographical conventions....16
Examples of syntax descriptions....17
Long syntax descriptions split across lines....19
Complex syntax descriptions with both required and optional elements....19
OpenEdge messages....19
Obtain more information about OpenEdge messages....20

Introduction....21

MS SQL Server DataServer overview....22
DataServer components....22
DataServer for MS SQL Server logic....24
Schema holder....25
Security....25
DataServer utilities....26
DataServer demonstration databases....27
DataServer configurations....27
The local, self-service, DataServer configuration....28
The remote, client/server, DataServer configuration....29
Configuring distributed DataServer applications using ProBroker....30
Configuring DataServer applications with a self-service OpenEdge AppServer or Progress AppServer....31
Configuring distributed DataServer applications using the Unified Broker Framework....32
TLS enabled DataServer for MS SQL....36
Distributed DataServer configuration with TLS for MSS....36
Distributed DataServer using Unified Broker....37
Distributed DataServer using ProBroker....37
Software requirements....37
Guidelines for using the DataServer....38
Some functionality highlights....38

Using the DataServer for MS SQL Server for the first time....38
Migrating an OpenEdge database to an MS SQL Server data source....39
Documentation resources....39

Initial Programming Considerations....41

Database design issues....42
OpenEdge database objects and MS SQL Server database objects....42
Naming conventions....43
MS SQL Server DataServer data source limits....44
Code pages....44
Support for Unicode....47
Indexes and sorting....49
Case-sensitivity....51
Interaction of code page, collation, and case sensitivity....52
Initial value....52
MS SQL Server data source views....53
Triggers....54
OpenEdge sequences....54
Data types....56
Working with unsupported data types....60
Working with non-updatable data types....60
Processing considerations for 32-bit and 64-bit data types....61
DataServer for MS SQL Server support for datetime data types....61
Using datetime data types in a WHERE clause....71
Support for OpenEdge ABL CLOB data type....71
Support for OpenEdge ABL BLOB data type....72
User-defined data types....73
Arrays....74
Unknown value (?)....74
Zero-length character strings....74
Record creation....75
Data source record locking....78
Share locks....79
Exclusive locks....79
Handling lock timeouts....79
Additional record locking details....80
NO-LOCK....81
Locking impact on queries....81
Locking examples....81
The NO-WAIT option....83
Transactions....83
Handling rollback of sub-transactions....83
Transaction scoping and buffer management....84
Error handling....85

Overflow checking....86
Cursors....87
Forward and backward scrolling....87
Block cursors....88
ABL issues....91
ROWID function....91
RECID function....92
DEFINE BROWSE statement....93
Field lists....93
FIND statements....95
Compiling OpenEdge procedures....96
Relaxing CRC Validation Rules for pre-compiled r-code....97
FOR statements....97
Features and language differences....97
RDBMS stored procedures....99

RDBMS Stored Procedure Details....101

Overview....101
Defining native stored procedures to ABL....102
RDBMS stored procedure basics....102
Run Stored-Procedure details....105
Complete syntax reference for Run Stored-Procedure....106
Run Stored-Procedure statement....106
Close Stored-Procedure statement....109
Run Stored-Proc statement execution without the LOAD-RESULT-INTO phrase....109
Run Stored-Proc statement execution with the LOAD-RESULT-INTO phrase....110
Run Stored-Proc statement execution using the send-sql-statement option....111
Data output and retrieval options....112
Return codes....112
Values of output parameters defined when creating a procedure....112
Results retrieved from a database....112
Loading results into a temp-table....113
Interfacing with RDBMS stored procedures....114
Retrieving return codes....114
Retrieving output parameter values....115
Retrieving result sets using proc-text-buffer....115
Defining a view to use as a buffer....117
Loading result sets into temp-tables....120
Handling errors....127
Trapping errors when using Load-Result-Into....130
Error messages related to using the Load-Result-Into phrase....130
ROWID Support....131
Understanding the ROWID implementation....131
ROWID characteristics....131

Using ROWID with RUN STORED-PROCEDURE and LOAD-RESULT-INTO....132
Additional ProDataSet support....135

Additional Features to Enhance DataServer Performance....137

Enhancements overview....138
Connection pooling....139
Main components....139
ODBC Connection Pooling....140
DataServer connection management....140
Firehose, Firehose block, and Fast Forward-Only Cursors....141
Monitoring cursor and connection use....143
OpenEdge query types....145
Query tuning....145
Caching records....150
Join by SQL DB....151
Statement level lock upgrade for JOIN query....153
Improving join performance for queries with FIRST or LAST criteria....156
Performance Optimization....157
Writing queries for performance....157
Modifying the run-time schema check....158
Skip schema verification....159
Replacing FIND FIRST for performance....159
Replacing FOR FIRST for performance....160

Configuring the DataServer....161

DataServer components....162
DataServer configuration prerequisites....162
Configuring an ODBC driver and registering the data source....163
Configuring the ODBC driver....163
Registering your data source....163
Configuring a local DataServer....163
Configuring a remote AppServer....164
Configuring a remote DataServer....164
Configuring with OpenEdge Management or OpenEdge Explorer....164
Configuring from the command line....167
Configuring a self-service DataServer 174
Configuring the OpenEdge AppServer agent as a self-service DataServer session....174
Configuring the Progress Application Server for OpenEdge agent as a self-service DataServer session....176
Creating a schema holder....177
Before you begin....177
Authorization and authentication....177
Preparing to create the schema holder....180

Creating a new empty OpenEdge database....180
Creating a schema holder....181
Maintaining a schema holder....184
Updating a deployed schema holder....185
Typical configuration for a remote client to connect to a remote DataServer....185
Configuring an ODBC data source....186
Starting a broker....186
Starting the schema holder to accept remote client connections....186
Connecting remote clients to the MS SQL Server DataServer broker....187

Connecting the DataServer....189

Starting a local DataServer....189
Starting a remote DataServer....190
Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client....190
Starting and stopping a broker process using MSSMAN....192
Starting and stopping a broker process from the command line....193
Starting the Windows client process....193
Starting the UNIX client process....194
Connection guidelines....194
Connections and security in Windows....194
Starting ProBroker distributed DataServer broker with TLS-enabled Progress Networking....195
Non-default TLS certificates....195
Connecting a schema holder....196
Connecting a schema holder at startup....197
Unsupported connection parameters....202
Special connection issues....202
Binding with connection and startup parameters....204
Query tuning with connection and startup parameters....204
Analyzing application execution with Enhanced Logger....206
Local schema caching....221
Connection failures and OpenEdge responses....222
Connection troubleshooting....223
Managing connections to an MS SQL Server database....223
Accessing the DataServer log....224

The DataServer Tutorial....227

Demonstration databases for DataServer for MS SQL Server....228
Preparing to create demonstration databases....228
DataServer utilities....233
Prime to ROWID identification....234
ROWID sub-section of MS SQL Server Conversion....235
Recommendation for choosing best clustered index....238

Creating a schema holder 238
Updating a schema holder....244
Verifying a schema holder....246
Verifying a table....248
Changing connection information in a schema holder....251
Changing a code page in a schema holder....251
Deleting a schema holder....252
Managing server attributes....252
Overview of foreign database constraint definitions....255
Constraint definition names....255
Defining constraint definitions in Data Administration....255
Defining a primary key constraint definition....256
Defining a unique key constraint definition....258
Defining a foreign key constraint definition....259
Defining a check constraint definition....261
Defining a default constraint definition....263
Modifying an existing constraint definition....264
Activating or deactivating a class of existing constraint definitions....265
Deleting contstraint definitions....265
Generating constraint definitions from ROWID....266
Migrating an OpenEdge database to MS SQL Server....267
Preparing a data source for the utility....268
Migrating defined constraint definitions to your foreign data source....268
Running the OpenEdge DB to MS SQL Server utility....268
Column width....294
Adjust schema utility....294
Delta df to MS SQL Server Incremental Schema Migration utility....295
Running the Incremental Schema Migration utility interactively....296
Running the Incremental Schema Migration utility silently (batch mode)....300
Adjusting field widths during migration....309
Updating the MS SQL Server database....309
Modifying a schema holder....309
Modifying table-level information....309
Modifying field-level information....310
Defining the ROWID....312
Independent OpenEdge MS SQL Server Schema Pull....315
Running the Independent schema pull utility in batch mode....322
Adding extended ABL support....329
Modifying tables to support arrays....329
Using triggers to enable PROGRESS_RECID support of the ROWID function....330
Using computed columns to enable PROGRESS_RECID support of the ROWID function....331
Migrating RECID-trigger mechanism to RECID computed column mechanism....331

Troubleshooting....335

Tuning your environment with the -Dsrv startup parameter....335
ODBC options....336
DataServer options....337
Using MS SQL Server and DataServer options....340
Using the block cursor switches....346
ODBC driver problems....348

Appendix A: Migration Issues....349
Creating a schema holder....349
Modifying your application....350
Modifying your insert trigger....350

Appendix B: Server Related Command Line Utilities and Startup Parameters....353
OpenEdge Management or OpenEdge Explorer command line utilities for the DataServer....353
MSSCONFIG utility....354
MSSMAN utility....355
NSCONFIG utility....357
NSMAN utility....359
Other command line utilities for the DataServer....361
PROBRKR command....361
PROSHUT command....362
DataServer startup parameters....363

Appendix C: Data Type Details....365
Data type conversion details....365

Appendix D: Using qt_debug to Analyze Performance....369

Index....371

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

July 2021

Product version: Progress OpenEdge 12.4

Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Use ABL documentation
- Before you begin
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages

Purpose

This manual explains how to use the OpenEdge® 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 OpenEdge and MS SQL Server database management systems, and provides tuning and troubleshooting information.

Note: In this manual, any reference to the term MSS or data source in the text, screen captures, or code examples presented refers to the MS SQL Server database.

Audience

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

Organization

Introduction on page 21 introduces you to the DataServer for MS SQL Server and discuss how OpenEdge and WebSpeed applications work with it to access data sources through the ODBC standard.

Subsequent chapters provide additional information about using the DataServer. If you are using the DataServer with WebSpeed and with applications written in Progress® SpeedScript®, all information regarding ABL applies to your application. For the latest documentation updates see the OpenEdge Product Documentation Overview page on PSDN: http://communities.progress.com/pcom/docs/DOC-16074.

Chapter names and details follow:

Introduction on page 21

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

Initial Programming Considerations on page 41

Discusses the differences between the OpenEdge database and MS SQL Server and the DataServer strategies for resolving these differences. Includes database design issues and application issues.

RDBMS Stored Procedure Details on page 101

Defines relational database management system (RDBMS) stored procedures and describes how to create and implement them in an OpenEdge environment. It discusses various techniques to execute RDBMS stored procedures and Send SQL statements on the MS SQL server and load results sets directly into temp-tables. ProDataSet functionality, available through the use of temp-tables, is also briefly discussed.

Additional Features to Enhance DataServer Performance on page 137

Presents various DataServer performance enhancement techniques you can use, including connection pooling.

Configuring the DataServer on page 161

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

Connecting the DataServer on page 189

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 OpenEdge responses.

The DataServer Tutorial on page 227

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 OpenEdge DB to MS SQL Server migration utility.

Troubleshooting on page 335

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

Migration Issues on page 349

Provides information on how to migrate from the DataServer for ODBC to the DataServer for MS SQL Server, and information on issues migrating from the Version 9.1 DataServer for MS SQL Server.

Server Related Command Line Utilities and Startup Parameters on page 353

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

Data Type Details on page 365

Presents a detailed, comprehensive list of the MS SQL Server data types and their associated OpenEdge-data type equivalencies.

Using qt_debug to Analyze Performance on page 369

The `qt_debug` option of the DataServer (`-Dsrv`) startup parameter can be used as a supplement to the OpenEdge Enhanced Logger to provide you with the means to perform diagnostics.

Use ABL documentation

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is *ABL (Advanced Business Language)*. With few exceptions, all keywords of the language appear in all UPPERCASE, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates go to https://docs.progress.com.

References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the *ABL Virtual Machine (AVM)*. When the documentation refers to ABL source code compilation, it specifies *ABL* or *the compiler* as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies *the AVM* as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time: "ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error." The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: "The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all UPPERCASE, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever *integer* appears, this is a reference to the `INTEGER` or `INT64` data type.
- Wherever *character* appears, this is a reference to the `CHARACTER`, `LONGCHAR`, or `CLOB` data type.
- Wherever *decimal* appears, this is a reference to the `DECIMAL` data type.

- Wherever *numeric* appears, this is a reference to the `INTEGER`, `INT64`, or `DECIMAL` data type.

References to built-in class data types appear in mixed case with initial caps, for example, `Progress.Lang.Object`. References to user-defined class data types appear in mixed case, as specified for a given application example.

Before you begin

Before attempting to use the MS SQL Server DataServer, first read the programming guidelines in Initial Programming Considerations on page 41. Also, be sure to follow the step-by-step instructions in Configuring the DataServer on page 161 and Connecting the DataServer on page 189 for installing, configuring, and connecting to a data source.

See Documentation resources on page 39 for additional information.

Typographical conventions

This documentation uses the following typographical and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax, code examples, system output, and file names.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax.
`Fixed-width bold`	Fixed-width bold italic indicates variables in syntax with special emphasis.
`UPPERCASE fixed width`	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.

Convention	Description
Period (.) or colon (:)	All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

Examples of syntax descriptions

In this example, ACCUM is a keyword, and *aggregate* and *expression* are variables:

Syntax

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
  DISPLAY Customer.Name.
END.
```

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

Syntax

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

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

Syntax

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

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

Syntax

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

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

Syntax

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

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

Syntax

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

In this example, you must specify MESSAGE and at least one *expression* or SKIP [(*n*)], and any number of additional *expression* or SKIP [(*n*)] 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" ] ... }
```

Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, `WITH` is followed by six optional items:

Syntax

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

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, `ASSIGN` requires either one or more *`field`* entries or one *`record`*. Options available with *`field`* or *`record`* are grouped with braces and brackets:

Syntax

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

OpenEdge messages

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

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.
- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before 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 OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge 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 after execution messages.

- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

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

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

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtain more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help** > **Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help** > **Messages** and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the OpenEdge `pro` command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

   ```
   OpenEdge-install-dir/bin/pro
   ```

2. Press **F3** to access the menu bar, then choose **Help** > **Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File** > **Exit**.

1

Introduction

The OpenEdge® DataServer for Microsoft SQL Server allows you to develop and deploy ABL (Advanced Business Language) or WebSpeed® applications that access a Microsoft SQL Server database. These applications use ABL specifically designed to translate knowledge about business operations into software.

The DataServer for MS SQL Server supports Microsoft SQL Server Versions 7 and later. 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.

This chapter introduces you to the DataServer for MS SQL Server and discusses how OpenEdge and WebSpeed applications work with it to access data sources through the ODBC standard.

For details, see the following topics:

- MS SQL Server DataServer overview
- DataServer components
- DataServer utilities
- DataServer demonstration databases
- DataServer configurations
- TLS enabled DataServer for MS SQL
- Software requirements
- Guidelines for using the DataServer
- Documentation resources

MS SQL Server DataServer overview

The OpenEdge DataServer for MS SQL Server allows you to access your MS SQL Server data source with the ABL and develop applications within the OpenEdge® Studio. The OpenEdge Studio is a set of tools that helps you maintain data sources and develop applications with graphical user interfaces. When you develop applications with the OpenEdge Studio, you can design a separation of your user interface, business logic, and data sources. OpenEdge Studio provides user interface independence, whether you need a character UI, a Graphical Windows UI, or a Web interface. When you incorporate a DataServer with OpenEdge Studio, you are enabling the use of your ABL business logic to access a foreign data source.

OpenEdge provides you to develop, deploy, manage, and integrate your application. The DataServer allows you to implement the OpenEdge features and ABL expansions in applications that run with the supported data sources.

Some of these tools and features are:

- **OpenEdge® Application Server** — Use to partition your application across multiple servers.
- **Progress Dynamics®** — Use to design and develop a repository-based application.
- **Data Dictionary** — Use to modify database schema; create indexes; and define database triggers, validation expressions, and help messages.
- **Data Administration tool** — Use to manage database data and perform DataServer tasks.
- **Database triggers** — Use to fire a block of ABL 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 a MS SQL Server data source does not provide you with access to all ABL, WebSpeed, and database features. For details, see Initial Programming Considerations on page 41.

The OpenEdge® DataServer for MS 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 can be configured on remote servers.
- The ODBC driver for the data source is bundled with the product.

Note: To install and use an ODBC driver successfully, your system must meet the driver system requirements for connecting to data sources. For more information, see the appropriate DataDirect product documentation at http://web.datadirect.com/support/product-info/documentation/index.html.

DataServer components

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

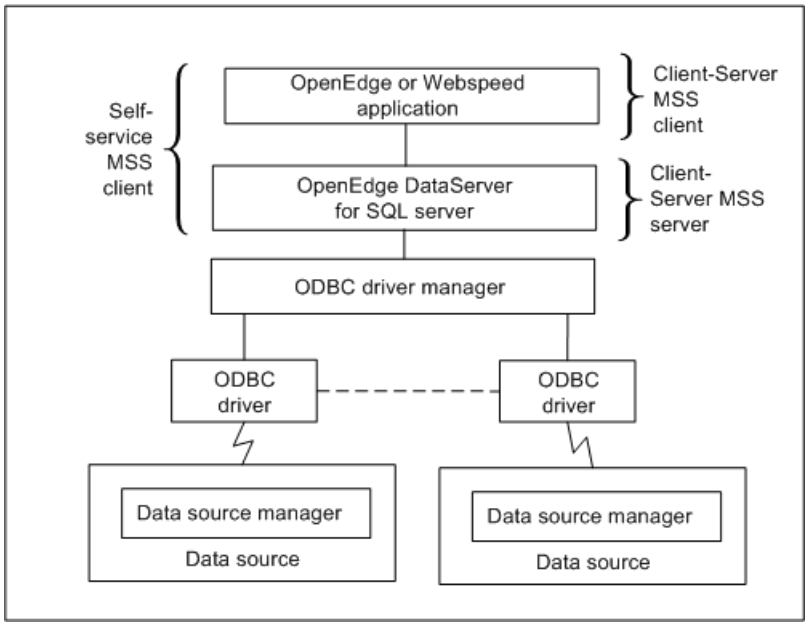
The DataServer consists of the components described in the following table.

Table 1: DataServer architecture components

Component	Description
DataServer for MS SQL Server (Windows platforms only)	An OpenEdge software module that allows you to use ABL or WebSpeed Speedscript.
ODBC Driver Manager (Windows platforms only)	A software module that manages the connection between an application and ODBC drivers. It is implemented in Windows by default as a Microsoft-supplied dynamic link library (DLL) module.
Schema holder	A repository for foreign data definitions.
ODBC driver (Windows platforms only)	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 (Windows platforms only)	A name that identifies a specific set of data and how to obtain it. You must register a MS 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 utilities for MS SQL Server	A set of utilities that allows you to perform certain tasks related to the MS SQL Server DataServer. You access them from either the Data Administration or the Data Dictionary.

The following figure illustrates how the DataServer components are organized.

Figure 1: Architecture for DataServer for MS SQL Server



As shown in the figure above, the components for the self-service DataServer for MS SQL Server are organized similarly to the client-server configuration except that the DataServer is embedded in the client rather than executing separately. With a client-server configuration, the components can be deployed so that the DataServer resides on the same machine as the client (loop back), on the server for efficiency and close proximity to the database, or distributed to a separate third tier.

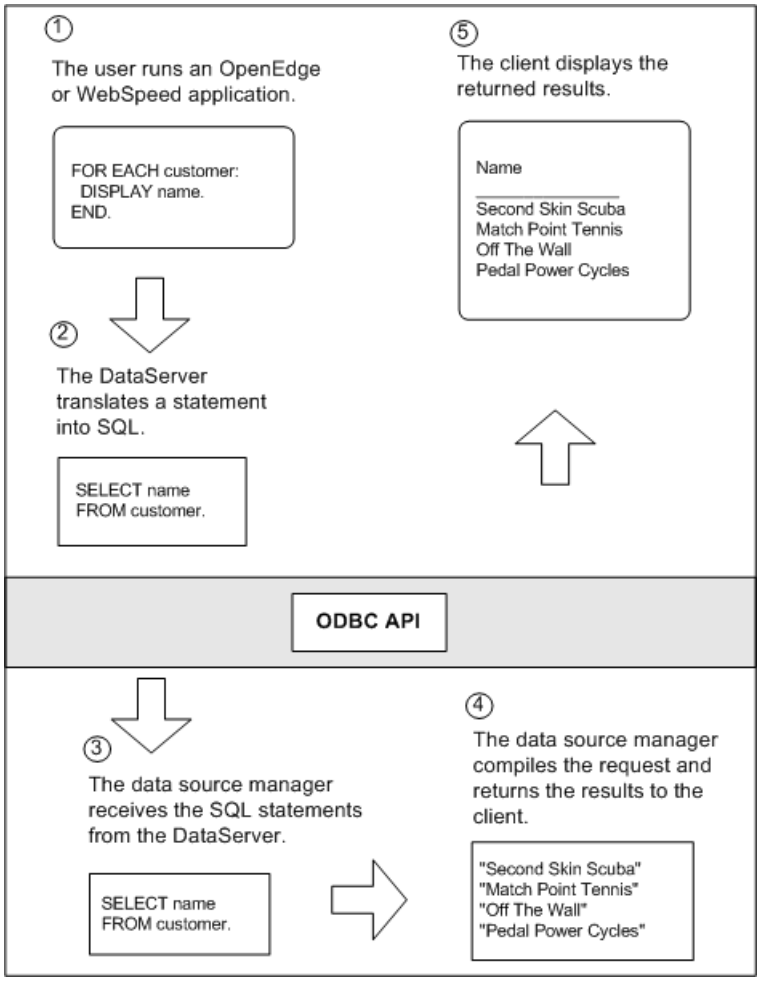
The ODBC driver manager must reside on the same machine as the DataServer "server." To ODBC, the "server" half of a client-server configuration, or the self-service DataServer appears as an ODBC client. In either case, an OpenEdge or WebSpeed application uses the DataServer to work with the ODBC Driver Manager, which accesses the desired ODBC driver. The driver then accesses the MS SQL Server database.

Some of the DataServer components are linked with elements of the standard OpenEdge architecture, as described in DataServer configurations on page 27.

DataServer for MS SQL Server logic

The flow chart in the following figure illustrates the DataServer's internal logic.

Figure 2: DataServer logic



As shown in the figure above, when you execute an application that accesses a MS SQL Server database, the compiler translates ABL 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 OpenEdge through the ODBC API.

Note: The SQL presented in the figure has been simplified for the purpose of focusing on the data flow.

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

Schema holder

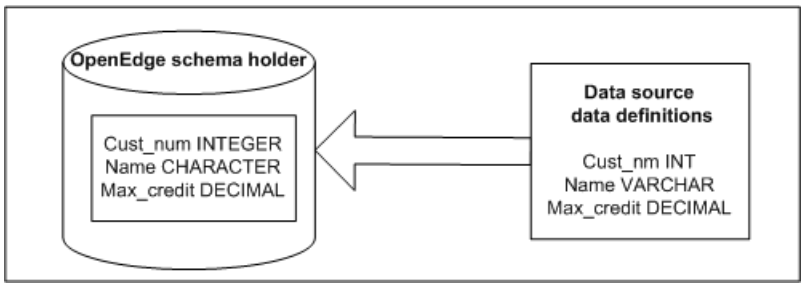
As noted in the previous section, a *schema holder* contains information about the MS SQL Server data sources. This information, called the *schema* of the data source, is a description of its structure, its tables, the fields within its tables, and its indexes.

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

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

The following illustrates the schema-loading process.

Figure 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.

However, the RDBMS stored procedures also supports s and ProDataSet functionality which does support returning data to the foreign data source. For information on using `RUN STORED-PROCEDURE`, see RDBMS Stored Procedure Details on page 101.

Security

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

OpenEdge security

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

- Database-connection security
- Schema security
- Compile-time security
- Runtime security

For more information about compile-time and run-time security, see *OpenEdge Deployment: Managing ABL Applications*. For general information about OpenEdge security, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.

MS SQL Server database security

As noted previously, you should follow the security guidelines that your MS SQL Server data source has established for your applications. The MS 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** — Grants or revokes permissions to other users to create or own a wide type of objects; for example, databases
- **Database owner** — Grants other users permission to access or modify a database or its objects
- **Database object owner** — Defines a user who can be the owner of objects in a database owned by another user
- **Public owner** — Allows access to public database objects by any users without restriction

Additional security considerations

Note the following additional security points:

- The DataServer for MS SQL Server will allow you to use either the Windows or MS SQL Server authentication model for your connection to the MS SQL Server data source. For more information on database security, see Microsoft's SQL Server documentation.
- There are specific security requirements for accessing data with the DataServer that relate to creating a schema holder. For details, see Creating a schema holder on page 177.

DataServer utilities

Progress Software Corporation 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 OpenEdge database to a MS SQL Server database, generating delta SQL files to update your MS SQL Server data source, and adjusting a pulled schema to match an existing OpenEdge database
- Adjusting the pulled schema to match an existing OpenEdge database

In addition, you can use the 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 Data Administration to manipulate data definition files for the schema holder.

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 OpenEdge Installation media. Instead, you create them by running the OpenEdge DB to MS SQL Server utility, which migrates the Sports database—objects and data—to your MS SQL Server data source. For instructions, see Demonstration databases for DataServer for MS SQL Server on page 228.

DataServer configurations

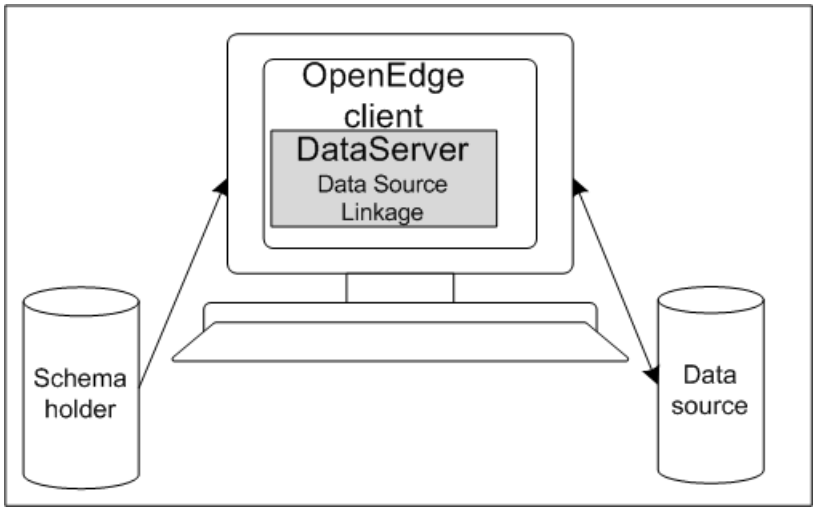
The DataServer for MS SQL Server can run in a variety of configurations. Depending on its configuration, it is either a single OpenEdge executable or a set of OpenEdge executables that you can distribute across operating systems. There are two general types of configurations—local and remote—with variations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

- **Local DataServer** — All of the DataServer software components, the schema holder, the ODBC software, and MS SQL Server client software run on one machine.
- **Remote DataServer** — The DataServer software components run on different machines. Typically, the OpenEdge client component runs on one machine, while the OpenEdge server component runs on another machine called the host machine. Note that the ODBC software and any data source 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 TCP/IP networking. A remote configuration exists within the framework of a distributed application, whereby DataServers are spawned on the host machine using the ProBroker executable or broker in the Unified Broker framework.

The local, self-service, DataServer configuration

The following figure shows the local DataServer configuration.

Figure 4: The local, self-service, DataServer for MS SQL Server



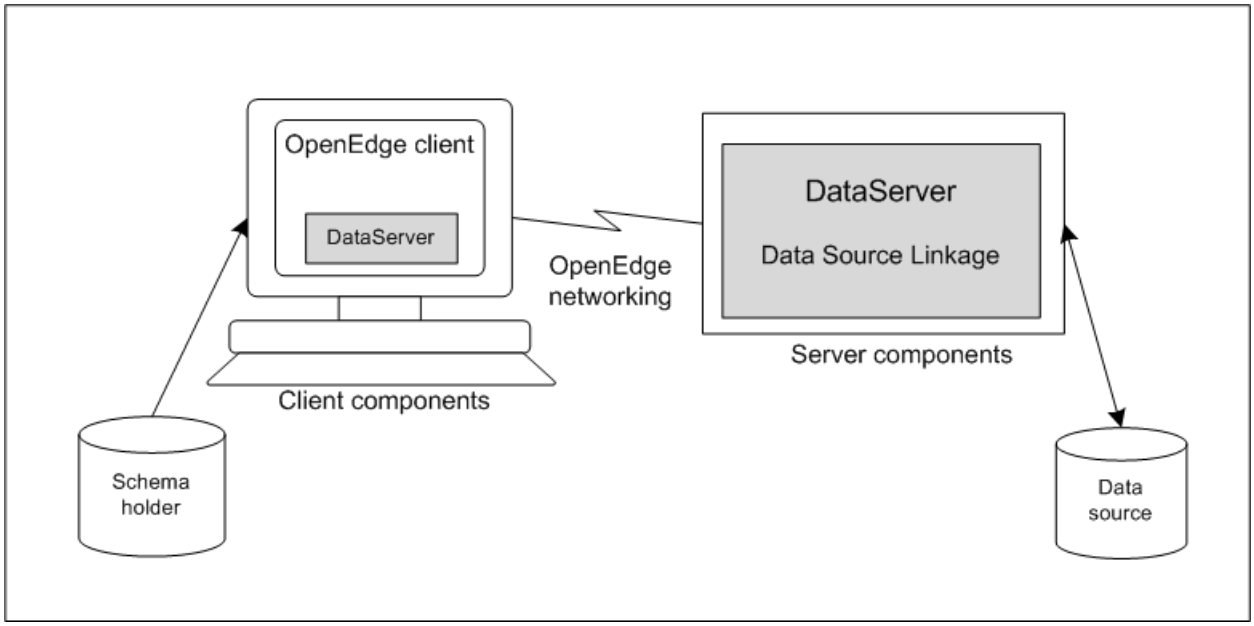
In the local DataServer configuration, the DataServer software components, the data source client software run on the same machine. Depending on the ODBC and data source client software implementation, the actual target database can be local or remote to the machine where the local OpenEdge DataServer for MS SQL Server executes.

With the MS SQL Server client software-supplied modules that manage networked database communications, a remote database configuration is possible. The DataServer cares only about the local MS SQL Server data source definition. Remote database access is transparent to the DataServer. Similarly, the schema holder can be accessed locally or remotely depending upon your OpenEdge Database configuration.

The remote, client/server, DataServer configuration

The following figure shows the remote DataServer configuration.

Figure 5: The remote, client/server, DataServer for MS SQL Server



In the remote configuration, the OpenEdge client runs on a client machine and the OpenEdge server components and any MS SQL Server client software run on a remote server machine. The actual target database can be local or remote to the machine where the OpenEdge DataServer for MS SQL Server executes. With the MS 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 addition, the schema holder can be accessed locally or remotely depending upon your OpenEdge database configuration.

In remote DataServer configurations, OpenEdge 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.

The following table lists the supported configurations for setting up a remote DataServer. It contains possible client-server combinations and networking options.

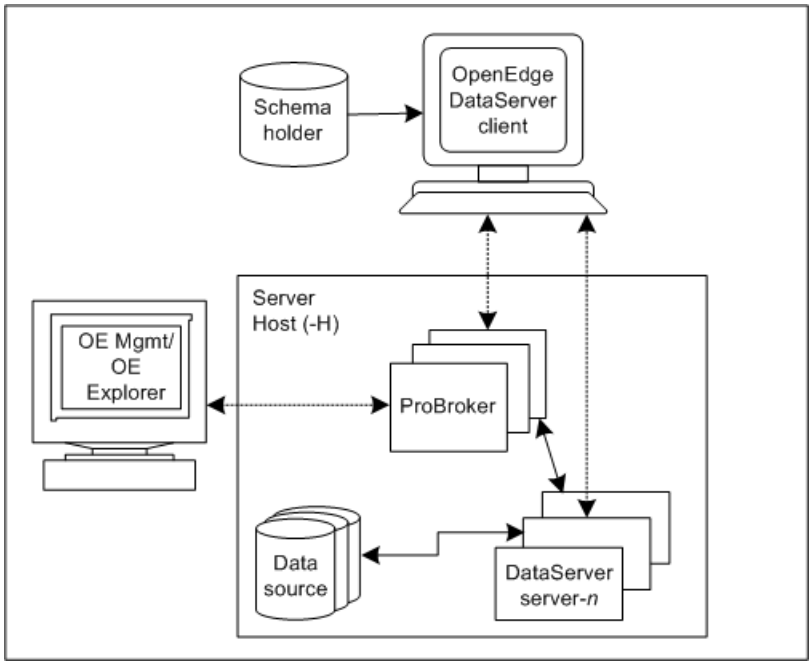
Table 2: Supported configurations

Client	Networking	Server
OpenEdge client in Windows	None	The server, client, ODBC client, and data source are on the same machine.
OpenEdge client in Windows or on UNIX	OpenEdge	The client can reside in Windows or on UNIX. The server, ODBC driver, and data source are on a Windows server machine.

Configuring distributed DataServer applications using ProBroker

The figure below shows an example of a distributed DataServer configuration that uses ProBroker. The figure illustrates how to combine clients and servers that run on different platforms.

Figure 6: The distributed DataServer for MS SQL Server using ProBroker



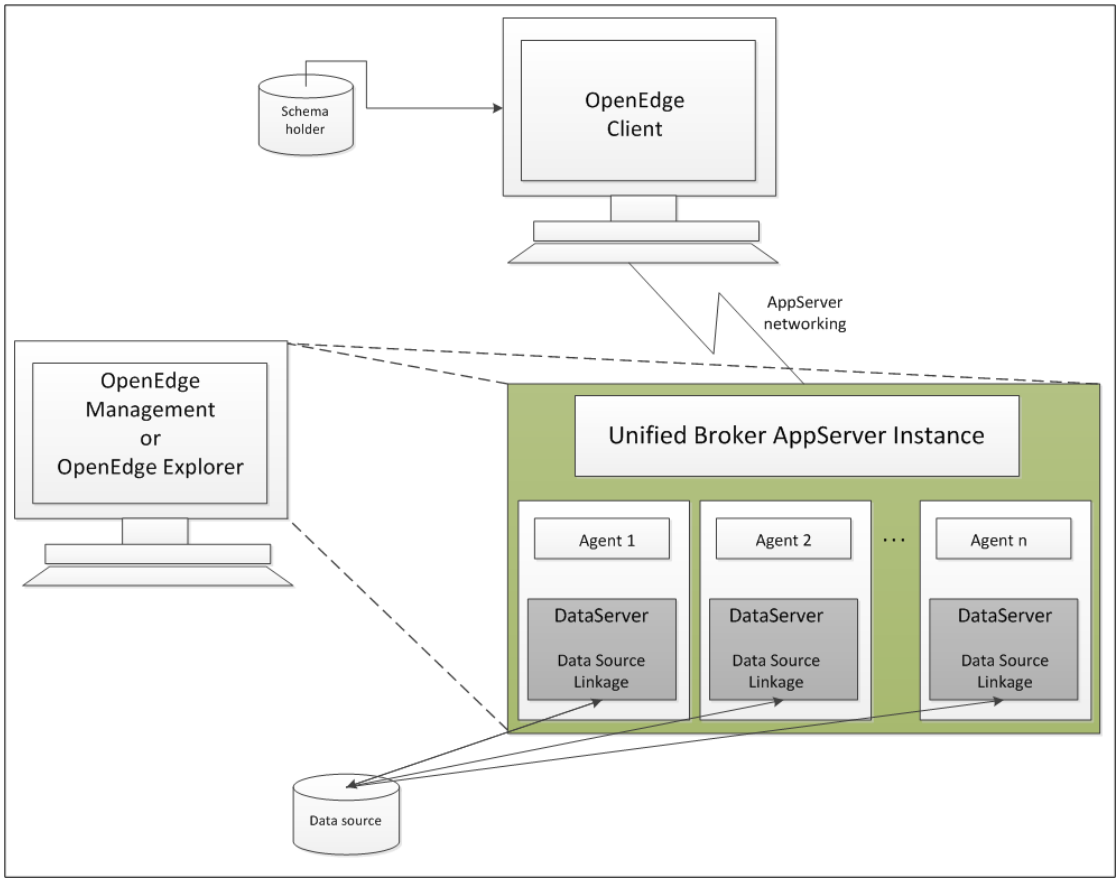
In the figure above, the OpenEdge client runs on a client machine (either in Windows or on UNIX) and can access a remote Windows server. Multiple brokers can be running simultaneously on one server machine. The client connects initially to the ProBroker executable by locating the host and port of the executing broker. It is then automatically reconnected to a DataServer established for it by the broker. Each executing broker can 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 connections, there is always a one to one relationship between a specific DataServer client and its DataServer server process. OpenEdge 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.

Configuring DataServer applications with a self-service OpenEdge AppServer or Progress AppServer

The figure below shows an example of a DataServer application that is configured with an AppServer. The figure illustrates how to combine clients and servers that run on different platforms.

Figure 7: The DataServer configuration with OpenEdge or Progress AppServer



In the figure above, the OpenEdge client runs on a client machine (either in Windows or in UNIX) and can access a remote machine that has OpenEdge Management or OpenEdge Explorer to manage the AppServer instance. The connection between the OpenEdge client and the AppServer residing in the remote machine is established using the AppServer networking.

With the OpenEdge 11.6 release, the DatServer for MS SQL Server is a candidate module Progress multi-threaded AppServer.

The remote machine has a Unified Broker AppServer instance with multiple AppServer agents in it or it has a Progress AppServer instance with multiple AppServer agent threads. For every self-service AppServer Agent process or thread, the connection between the DataServer component of the agent process and the Data source is established on the same computer as the AppServer using the respective DataServer source linkage. Data Source Linkage specifies how the DataServer component of the agent processes use the ODBC driver to locate and connect to the data source.

The DataServer for MS SQL Server can be accessed from agent processes of the OpenEdge AppServer or from agent threads of the Progress Application Server for OpenEdge. The Progress Application Server for OpenEdge can only manage DataServer sessions fully within the Progress Application Server for OpenEdge if those sessions are self-service. The Progress Application Server for OpenEdge does not support distributed DataServer components although a DataServer client component in the Progress Application Server for OpenEdge can access a server component configured in a DataServer instance of the Unified Broker or a server process spawned from ProBroker.

For more information on configuring and managing the AppServer, see *OpenEdge Application Server: Administration.*

Configuring distributed DataServer applications using the Unified Broker Framework

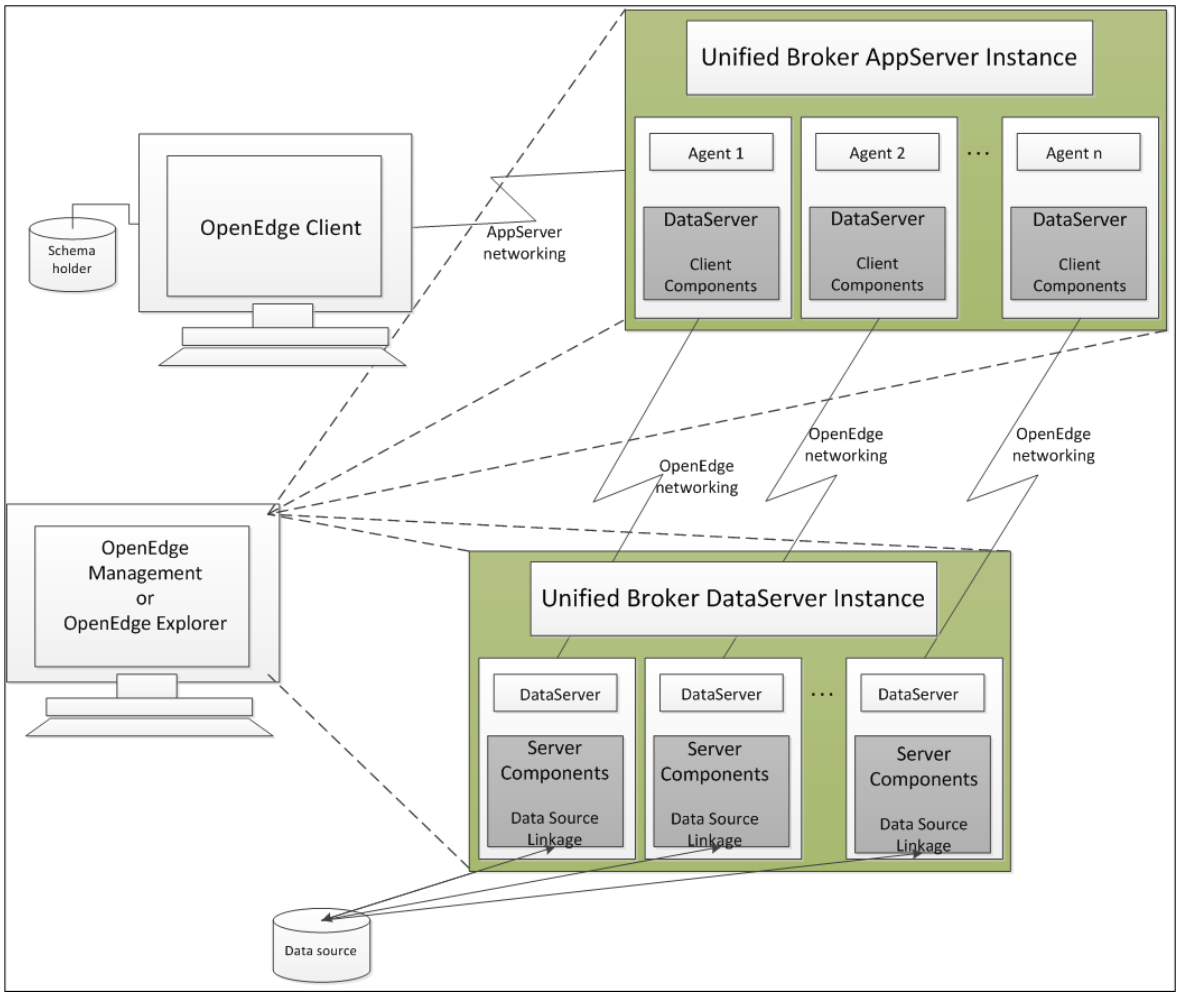
The Unified Broker Framework is a system administration framework that provides a consistent interface in which specific OpenEdge products such as the DataServer for MS SQL Server can be managed. It supports elements that allow you to perform such common administrative tasks as

- Starting and stopping processes
- Managing, configuring, and validating property files

These elements include the OpenEdge Management/OpenEdge Explorer configuration tool, mergeprop utility, and command-line utilities.

The following figure shows an example of a client/server DataServer application that is configured with Unified Broker AppServer Instances. The figure illustrates how to combine clients and servers that run on different platforms.

Figure 8: Distributed DataServer applications using a Unified Broker Framework



In the figure above, the OpenEdge client runs on a client machine (either in Windows or in UNIX) and can access a remote machine that has the OpenEdge Management or OpenEdge Explorer to manage the Unified Broker instances. The connection between the OpenEdge client and the AppServer residing in the remote machine is established using AppServer networking, and the connection between the several DataServer broker instances is established using OpenEdge networking.

For every AppServer Agent, the connections between the DataServers residing in other broker instances is established using the OpenEdge networking, that is, the DataServer client components of the Agent establish connection with the server components of the DataServers in corresponding DataServer broker instances.

Note: Server component of the DataServer cannot be distributed using the Progress Application Server for OpenEdge. However, a Progress Application Server for OpenEdge

---

Each DataServer can spawn a multitude of DataServer processes, and a Data Source Linkage specifies how the DataServer processes use the ODBC driver to locate and connect to the data source. instance can connect to server components in a Unified Broker DataServer Instance of the traditional OpenEdge AppServer or server component processes spawned from the traditional ProBroker product.

For more information on configuring and managing the AppServer, see *OpenEdge Application Server: Administration*.

instance can connect to serverThe framework also facilities activities that are fundamental to the MS SQL Server DataServer's broker-based technology. For example, the AdminService, the framework's central element, enables supported products like the DataServer for MS SQL Server in managing an application's resources.

For details about the Unified Broker Framework, its elements, and the OpenEdge products that employ its features, see *OpenEdge Getting Started: Installation and Configuration*.

A closer look at the OpenEdge Management or OpenEdge Explorer interface

OpenEdge Management or OpenEdge Explorer is a graphical user interface that provides an easy way for you to manage OpenEdge servers. This interface runs as a Windows client and works with another administration component, the AdminService, to provide a client/server framework for managing the following OpenEdge servers.

Using OpenEdge Management or OpenEdge Explorer, you can:

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

For more information about working with OpenEdge Management or OpenEdge Explorer, see OpenEdge Management or OpenEdge Explorer online help.

Using the mergeprop utility

The `mergeprop` utility is a command-line utility that supports functionality similar to that supported by the OpenEdge Management or OpenEdge Explorer configuration tool. It is an alternative approach to configuring and managing the content of the OpenEdge property files. Property files, such as the ubroker.properties file used by the DataServer, store configuration information that specifies and controls the behavior of various components.

Typical configuration scenarios

In a distributed client-server configuration, the OpenEdge client runs on a client machine (either in Windows or on UNIX) and accesses a remote server on a Windows server where the broker for the DataServer for MS SQL Server is running.

Multiple NameServers and/or Unified Broker instances can be running simultaneously on one server machine. The DataServer client connects to a broker instance from OpenEdge Management or OpenEdge Explorer either directly or through a controlling NameServer. (See the important caution that follows.) It is then automatically reconnected to a DataServer established for it by the broker instance. Each executing broker instance can 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. The schema holder can run on the client or you can 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, OpenEdge handles the communication between the client and the server. OpenEdge Management or OpenEdge Explorer supports only the TCP/IP network configuration. Distributing the client and server components of the DataServer is only possible using DataServer broker instances of the Unified Broker. The Progress Application Server for OpenEdge (PAS for OpenEdge) does not support distributed DataServer components although the client component of the DataServer run from a PAS for OpenEdge can connect to a distributed server component in a DataServer broker instance of the Unified Broker.

Caution: 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:

- Configuring with OpenEdge Management or OpenEdge Explorer on page 164
- Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190

For more information about the Unified Broker Framework, its elements, and how Unified Broker products such as the DataServer for MS SQL Server work within the framework, see *OpenEdge Getting Started: Installation and Configuration*.

TLS-enabled DataServer

DataServer components have the option of configuring DataServer client and server instance to require Transport Layer Security (TLS) client connections in remote DataServer configurations. Users can maintain both TLS-enabled and non-TLS Transaction Server instances. However a given instance supports only one type of connection, either secure or non-secure.

DataServer Security is based on the client authenticating the server's identity using a Public Key Infrastructure (PKI) and a symmetric data encryption system. To configure a Transaction Server instance for TLS operation, you must:

- Install a server private key and a public key certificate. OpenEdge provides built-in keys and certificates that are suitable for use on development or demonstration servers; for production machines, you should obtain server certificates from an internal or public Certificate Authority (CA).
- Specify an alias and password for access to the private key/digital certificate.
- Disable session caching, or enable it with a specified timeout.

To connect to a TLS-enabled OE DataServer component, Client and Servers must have access to a digital (public key) certificate that can authenticate with the digital certificate used by the server, and the client must be configured to send TLS requests. All OpenEdge-managed TLS servers rely on a common OpenEdge key store to manage the private keys and server digital certificates required to support TLS connections from clients. Similarly, most OpenEdge-managed TLS clients and servers rely on a common OpenEdge certificate store to manage the root CA digital certificates that enable them to establish connections to appropriate TLS servers. With OpenEdge installation a third party Public/Private key pair is provided for testing. The root CA certificate is located in $DLC/keys/default_server.pem and the public key is located in $DLC/certs/pscca.cer.

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

- Configuring with OpenEdge Management or OpenEdge Explorer on page 164
- Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190

For more information about the Unified Broker Framework, its elements, and how Unified Broker products such as the DataServer for MS SQL Server work within the framework, see *OpenEdge Getting Started: Installation and Configuration*.

TLS enabled DataServer for MS SQL

Configuring the client and server components of a remote DataServer connection with Transport Layer Security (TLS) communications is optional. Users can maintain both TLS-enabled and non-TLS instances of a DataServer broker environment. However a given instance supports only one type of connection, either secure or non-secure.

DataServer Security is based on the client authenticating the server's identity using a Public Key Infrastructure (PKI) and a symmetric data encryption system. To configure a Broker instance for TLS operation, you must:

- Install a server private key and a public key certificate. OpenEdge provides built-in keys and certificates that are suitable for use on development or demonstration servers; for production machines, you should obtain server certificates from an internal or public Certificate Authority (CA).
- Specify the `keyAlias` and `keyAliasPasswd` parameters in the `UBroker.MS.mssbroker1` section of the ubroker.properties file for access to the private key/digital certificate
- Disable session caching of the orabroker using the `noSessionCache` parameter, or enable it with a specified timeout using the sessionTimeout parameter.

For more information see Establishing the TLS protocol in a DataServer broker instance of the Unified Broker on page 169 and *Unified Broker section and the OpenEdge Getting Started : Core Business Services – Security and Auditing* guide.

To connect to a TLS-enabled OE DataServer component, Client and Servers must have access to a digital (public key) certificate that can authenticate with the digital certificate used by the server, and the client must be configured to send TLS requests. All OpenEdge-managed TLS servers rely on a common OpenEdge key store to manage the private keys and server digital certificates required to support TLS connections from clients. Similarly, most OpenEdge-managed TLS clients and servers rely on a common OpenEdge certificate store to manage the root CA digital certificates that enable them to establish connections to appropriate TLS servers. With the OpenEdge installation, a third party Public/Private key pair is provided for testing. The root CA certificate is located in $DLC/keys/default_server.pem and the public key is located in $DLC/certs/pscca.cer.

Distributed DataServer configuration with TLS for MSS

The DataServer broker (_probrkr.exe) or an MSS DataServer instance of the Unified Broker) on a host machine determines the protocol requirements of the clients that connect to server components (_orasrv.exe) started by the broker on behalf of a connecting client. A distributed ProBroker or Unified Broker environment can accepts a `-ssl`startup or command line argument on the host machine to indicate that the types of client requests that the spawned DataServer (_msssrv.exe) can handle should be using TLS protocol over the network. OpenEdge DataServer client should only send TLS enabled requests over the network. A `-ssl` start-up parameter is also accepted by the client to instantiate an TLS connection with the server component. If an OpenEdge DataServer client is started without a `-ssl` startup parameter and it sends a connection request to a running broker in TLS mode, the broker fails to authenticate the client connection request. However, a message is then returned internally to the client requesting it to send a TLS enabled handshake. If the client is capable of sending a TLS request, the TLS connection is automatically established without specifying the client startup parameter.

See the *OpenEdge Getting Started: Installation and Configuration* for an overview of the Unified Broker framework.

Distributed DataServer using Unified Broker

The Unified Broker framework is a system administration framework that provides a consistent interface in which specific OpenEdge products, such as the MSS DataServer, can be managed. It supports elements that allow you to perform such common administrative tasks as

- Starting and stopping processes
- Managing, configuring and validating property files

These elements include the OpenEdge Management/OpenEdge Explorer configuration tool, mergeprop utility, and command-line utilities.

The framework also facilities activities that are fundamental to the MS SQL DataServer's broker-based technology. For example, the AdminService, the framework's central element, enables supported products like the MS SQL DataServer in managing an application's resources.

See the *OpenEdge Getting Started: Installation and Configuration* for an overview of the Unified Broker framework.

Distributed DataServer using ProBroker

ProBroker is the original broker framework for spawning server component of the DataServer products. ProBroker is started from a command line or from an operating system script. It uses a traditional “fork/exec” broker model as it receives client connection requests. It is a standalone process rather than part of a system framework. As such, it has none of the management, configuration or validation capabilities of the Unified Broker Framework. There is one additional command, “proshut”, that allows the user to terminate a started ProBroker process.

For a tutorial on configuring ProBroker for DataServer connectivity, see Starting the remote DataServer on page 174

Software requirements

The following list describes the software requirements for using the DataServer.

- OpenEdge-supplied software

  Your installation must have:

  - DataServer for MS SQL Server
  - Progress Application Server for OpenEdge

- Data-source server software

  Your installation must have:

  - Microsoft SQL Server Version 7.0 or higher

- Data source client software

  Your installation must have:

  - 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 http://www.microsoft.com.

- ODBC software

  Your installation might have:

  - An ODBC Driver for the version of Microsoft SQL Server you are using. (optional)

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

  - Progress Software Corporation supplies the DataDirect ODBC driver, which is automatically installed when you install the DataServer for MS SQL Server.

    For additional information, see the DataDirect Web site at: http://web.datadirect.com/products/odbc/.

Guidelines for using the DataServer

OpenEdge database management system 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 OpenEdge databases and MS SQL Server databases.

Some functionality highlights

The DataServer allows you to use ABL features as extensions to your data source. Some of the OpenEdge programming and database design techniques that you can implement on your MS SQL Server 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 MS SQL Server data-source or application is organized. For a discussion of these issues and instructions for modifying your data source, see Initial Programming Considerations on page 41

If you create a MS SQL Server data source from an existing OpenEdge database with the OpenEdge DB to MS SQL Server migration utility and select the **Create RECID Field** option, you can use the `FIND PREV/LAST` statements in addition to taking advantage of OpenEdge-like cursor behavior.

How you use the DataServer depends on whether you plan to access information in a data source through an OpenEdge application or migrate an OpenEdge database to a MS SQL Server data source. The following sections summarize.

Using the DataServer for MS SQL Server for the first time

Before using the DataServer, you must:

1. Install the DataServer modules on the machines your configuration requires.
2. If you chose not to use the bundled ODBC driver, purchase and install the ODBC driver software on the machine that will execute the server component.

3. Verify connectivity between ODBC and the MS SQL Server database.
4. Create a local schema holder on the client or server machine, as appropriate. Schema holders cannot be transferred between different host machines.

RDBMS Stored Procedure Details on page 101 provides information about where to install DataServer modules and how to create a schema holder.

Migrating an OpenEdge database to an MS SQL Server data source

You can migrate an OpenEdge database to an MS SQL Server data source.

Use the following steps to set up and use the DataServer:

1. Install the DataServer components on the machines that your configuration requires.
2. If you chose not to use the bundled ODBC driver, purchase and install the ODBC driver software on the machine that will execute the server component.
3. Verify connectivity between ODBC and the MS SQL Server database.
4. Run the OpenEdge DB to MS SQL Server migration utility.

See Migrating an OpenEdge database to MS SQL Server on page 267 for specific instructions.

Documentation resources

The following table shows reference sites in this manual that explain different approaches to using the DataServer for MS SQL Server.

Table 3: How to use this manual

If you are . . .	Refer to this information . . .
New to the DataServer for MS SQL Server	Initial Programming Considerations on page 41 Configuring the DataServer on page 161 Connecting the DataServer on page 189 The DataServer Tutorial on page 227
Migrating an OpenEdge database to a MS SQL Server data source	Running the OpenEdge DB to MS SQL Server utility on page 268 Initial Programming Considerations on page 41 Configuring the DataServer on page 161 Connecting the DataServer on page 189
Upgrading from an earlier version of MS SQL Server	Migration Issues on page 349 Configuring the DataServer on page 161

The following table lists manuals from the OpenEdge documentation set that contain useful information on different aspects of DataServer usage.

Table 4: DataServer-related topics

Topic	OpenEdge manuals
Starting and setting up OpenEdge products	*OpenEdge Getting Started: Installation and Configuration*
Using the Data Dictionary	*OpenEdge Getting Started: ABL Essentials*
Defining security for an OpenEdge database	*OpenEdge Data Management: Database Administration*
Writing applications in the ABL	*OpenEdge Development: ADM and SmartObjects* *OpenEdge Getting Started: ABL Essentials*
Connecting your data source with WebSpeed	*OpenEdge Getting Started: Installation and Configuration*
Writing WebSpeed applications	*OpenEdge Application Server: Developing WebSpeed Applications*

2

Initial Programming Considerations

An application developed in the OpenEdge Studio can use the DataServer for MS SQL Server to access data from multiple sources transparently. With the DataServer, your applications can access information in both OpenEdge databases and MS SQL Server databases.

This chapter discusses fundamental considerations you need to know to plan your applications and design your databases. For example, you need to know the differences between OpenEdge databases and MS SQL Server databases, as well as the DataServer strategies for resolving these differences, which your application might need to accommodate.

Follow these initial guidelines carefully when you develop your application to ensure that it can access OpenEdge databases and MS SQL Server databases transparently. Once you are comfortable with this information, refer to the stored procedure details presented in RDBMS Stored Procedure Details on page 101, and the advanced guidelines in Additional Features to Enhance DataServer Performance on page 137.

Note: The material in this chapter is also of interest to users who plan to migrate an OpenEdge database to a MS SQL Server data source. However, such a migration raises additional issues that you must consider when designing your application. For details, see Running the OpenEdge DB to MS SQL Server utility on page 268.

For details, see the following topics:

- Database design issues
- Data types
- Record creation
- Data source record locking
- Transactions

- Error handling
- Cursors
- ABL issues
- RDBMS stored procedures

Database design issues

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

OpenEdge database objects and MS SQL Server database objects

The OpenEdge database and the MS 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 OpenEdge database or MS SQL Server databases's logical structure. The following table lists OpenEdge database objects and the MS SQL Server database equivalents.

Table 5: OpenEdge Database objects and MS SQL Server database objects

OpenEdge	MS 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
View	View

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

OpenEdge	MS SQL Server
No equivalent to stored procedures	Stored procedure
Schema Trigger	Trigger

Naming conventions

When planning for maximum transparency across OpenEdge databases and MS SQL Server data sources, be sure to consider the restrictions that each places on naming data source objects when you perform a OpenEdge DB to MS SQL Server conversion. The following table describes these restrictions, and the notes that follow discuss how the DataServer resolves differences between OpenEdge and MS SQL Server database naming conventions.

Table 6: Database naming restrictions

Category	OpenEdge	MS SQL Server
Alphanumeric characters	A-Z or a-z 0-9	All alphanumeric characters from the character set that you defined for your MS SQL Server
Special characters	Allows $, &, #, %, -, and _[2]	Allows @, $, #, _, ¥, and £[3]
Initial character	Alphabetic only	Alphabetic only
Maximum length of object name	32 characters	24 characters[4]
Keywords	Not allowed[5]	Not allowed[5]
Unique table names	Required[6]	Required[7]

MS SQL Server DataServer data source limits

A MS SQL Server 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 OpenEdge applications run. See your Microsoft SQL Server documentation for full details.

Code pages

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

[2] Although OpenEdge allows you to use the ampersand (&) in database-object names, the OpenEdge development tools use this character internally to denote preprocessor directives.

[3] When the DataServer encounters characters not supported by OpenEdge in a sequence, table, index, or field name, it converts them to underscores (_) in the data source. 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.

[4] Object names are limited to 32 characters. When you use the OpenEdge DB to MS SQL Server 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, array fields, sequences, triggers, and so forth. If you intend to use the OpenEdge DB to MS SQL Server utility, be sure to limit object names to 24 characters to accommodate this name resolution during conversion.

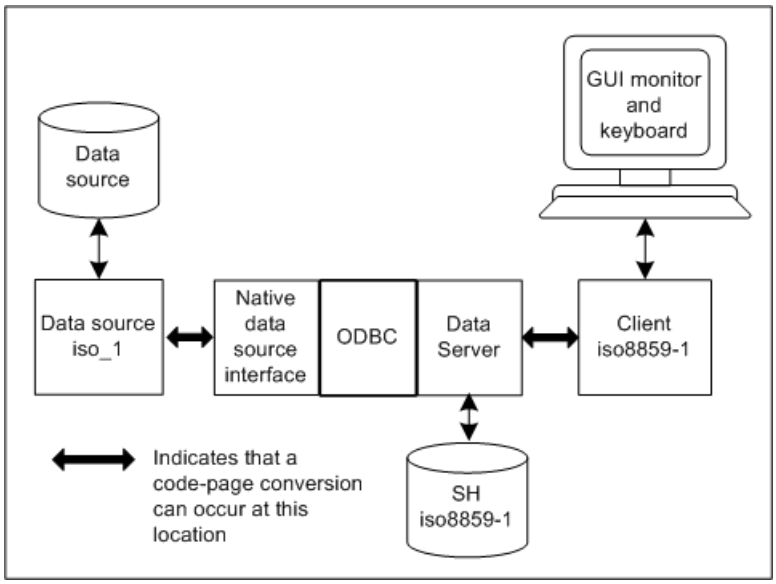
[5] OpenEdge and MS SQL Server databases contain restrictions against using keywords as database object names. If a data source object name consists of an OpenEdge keyword, the DataServer appends an underscore character (_) to the name. For example, an object named "each" becomes "each_"

[6] MS SQL Server qualifies nonunique table names to make them unique (see your data source documentation for details); however, OpenEdge still regards these qualified names as nonunique. The DataServer resolves nonunique table names for you as follows: when OpenEdge 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 OpenEdge 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.Keep in mind that 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 an OpenEdge keyword. Therefore, you should limit your object names to 29 characters so that you do not exceed the OpenEdge 32-character limit for object names in the schema holder.

[7] MS SQL Server allows you to qualify an object name by adding the database name, the owner's name, or the table name (for columns).

The following figure shows a possible configuration of code pages for the DataServer components and processes.

Figure 9: DataServer processes and code pages



In the configuration shown in the figure above, all components use the same code page. The OpenEdge iso8859-1 code page is a subset of the iso_1 code page used by the default installation of MS SQL Server. For information on setting character sets for your data source, see the administration guide supplied by the vendor. On the OpenEdge 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 a MS SQL Server data source accurately, you must specify the correct code page and collation in the schema holder. For OpenEdge 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 MS SQL Server data source supports. To minimize the number of translations, specify the default code page that the data source uses. If OpenEdge 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 OpenEdge 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 MS SQL Server 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 MS SQL Server 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 OpenEdge client uses when it manipulates data in memory. If the OpenEdge 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.

OpenEdge 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 a MS SQL Server data source. The data source collation tables, not the OpenEdge collation tables, have first priority when you perform comparisons and sorts. After performing comparisons and sorts, the OpenEdge 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 MS 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 ~)
```

MS 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 MS SQL Server collation.

Normally, the default OpenEdge collation sorts a tilde character above all alphanumeric characters. Therefore, in order for the above example to exhibit OpenEdge-like behavior and return alphanumeric records as well as records beginning with the tilde, the MS 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)
```

MS SQL Server returns records with columns that begin with a tilde character followed by all that begin with alphanumeric characters. However, the default OpenEdge 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 MS SQL Server from the OpenEdge client would require modifying the collation table associated with the OpenEdge code page and weighting it to match the MS SQL Server sort order.

For a complete discussion of how OpenEdge handles code-page issues, see *OpenEdge Development: Internationalizing Applications.*

Code page, collation, and case conflicts

How you specify code page, case sensitivity, and shadow column usage can lead to conflict between the attributes. It is best to establish the settings on these attributes prior to schema creation, and allow the schema pull to manage the integration into the schema image. The following table describes the behavior of the combinations.

Table 7: Case sensitivity interactions

Code Page[8]	Shadow Columns	Case Sensitive Field Attribute	Result
CS	Yes	Off	Shadow column provides case insensitivity
CS	No	Off	`UPPER( )` function provides case insensitivity[9]
CS	Yes	On	Code page provides case sensitivity
CS	No	On	Code page provides case sensitivity
CI	Yes	Off[10] (Is case-insensitive)	Shadow column provides case insensitivity[11]
CI	No	Off (Is case-insensitive)	Code page provides case insensitivity
CI	Yes	On	Code page provides case insensitivity[12]
CI	No	On	Code page provides case insensitivity[13]

Support for Unicode

Enabling Unicode support in the DataServer for MS SQL Server provides the following functionality:

- Completes the connection between UTF-8 support in the OpenEdge Advanced Business Language (ABL) client and UCS-2 support in MS SQL Server
- Allows Unicode data to be passed to and from MS SQL Server data sources
- Provides a migration path for Unicode character data type definitions in ABL to MS SQL Server, and vice versa

The result is that OpenEdge DataServer for MS SQL Server applications can be developed and deployed without significant language or country restrictions.

For general Unicode information, see *OpenEdge Development: Internationalizing Applications*.

Unicode support in the DataServer for MS SQL Server is enabled while migrating an OpenEdge database to MS SQL Server. For details seeThe DataServer Tutorial on page 227.

[8] The MS SQL Server code page is either Case Sensitive (CS) or Case Insensitive (CI).
[9] Be aware that relying on the UPPER() function can have a negative performance impact.
[10] This combination is not recommended. Don't use shadow columns if the MS SQL Server code page can handle case-insensitivity instead.
[11] This combination is not recommended. Use a code page to support case insensitivity by identifying the code page as case insensitive on a schema pull. Fields will be marked case insensitive on the pull.
[12] Shadow columns are present, but play no role.
[13] Case sensitivity cannot be supported by OpenEdge. The client will throw out any results not returned in case insensitive order.

Configuration requirements

In order to use Unicode in the OpenEdge DataServer for MS SQL Server, you must use Microsoft SQL Server 2005 or later. All MS SQL server drivers, including the DataDirect Connect ODBC driver for MS SQL Server, Version 5.1 or later, support Unicode. Additionally, you must perform the following prerequisite tasks:

- **Prepare your Unicode Schema Holder** — If you create a new OpenEdge application to be Unicode-enabled for migration to an MS SQL DataServer, ensure that you create the database from a Unicode source. A copy of the empty OpenEdge `UTF-8` database can be obtained from `$DLC/prolang/utf/empty`. If you use the Create DataServer Schema MS SQL DataServer utility to create a DataServer schema and you plan to set the schema holder's code page to `UTF-8`, the schema holder must also be derived from a copy of the empty `UTF-8` database located at `$DLC/prolang/utf/empty`.

  Note: The migration process derives a schema holder this way automatically if you set the code page of your migration to `UTF-8`.

  The following command creates an empty OpenEdge database named "empty" that can be used to describe a Unicode-enabled foreign data source:

  ```
  Prodb OpenEdge_utf-8_schema_holder $DLC/prolang/utf/empty
  ```

  Note: The best way to ensure that the database path is correct for utilities that create a Unicode-enabled database is to set the `$DLCDB` environment variable to `$DLC/prolang/utf`. This ensures that databases will be constructed with Unicode enablement. If you do not set `$DLCDB`, you will need to reference the `$DLC/prolang/utf` directory explicitly whenever you want to create a Unicode-enabled database.

- **Use appropriate startup parameters** — To process Unicode data in your client application, set your `cpinternal` client code page to `utf-8`. If you are executing file I/O against your Unicode data, set your client's `cpstream` code page to `utf-8`. You also may consider several other available client code page settings for Unicode. For more information, see *OpenEdge Development: Internationalizing Applications*.
- **Prepare OpenEdge databases for Unicode** — Your OpenEdge database does not need to be Unicode-enabled in order to be migrated to an MS SQL DataServer. The migration process will convert data from any ANSI code page to Unicode as part of the migration if you chose to move data during a migration and have specified a Unicode destination. If you have existing data in an OpenEdge database that you wish to convert to Unicode prior to migration, you can convert your non-Unicode data using one of two conversion utilities:
  - Data Administration dump and load utility
  - `PROUTIL CONVCHAR` character conversion utility

  For more information on these utilities, see *OpenEdge Development: Internationalizing Applications.*

Data type support

Unicode data and data types can be migrated to a MS SQL Server data source and pulled into a MS SQL Server DataServer schema holder. The following table defines the supported MS SQL Server data types and their maximum sizes.

Table 8: Supported Unicode data types in MS SQL Server

Data type	Maximum size
NCHAR	4000 characters
NVARCHAR	4000 characters
NVARCHAR(MAX)	2^{30-1} characters
NTEXT	2^{30-1} characters

MS SQL Server limits the size of CHAR and VARCHAR data types to 8000 bytes. When server data is represented by a single-byte ANSI code page, CHAR and VARCHAR data types can store 8000 characters. When SQL Server stores UCS-2 data in double-byte format, it restricts CHAR and VARCHAR data types to only 4000 characters. Therefore, when a non-Unicode OpenEdge database is migrated to MS SQL Server 2005 (or greater), the **Maximum Varchar Length** threshold is set to 8000 by default. Character data sized smaller than this threshold (which the user can configure lower) is converted to the VARCHAR native type. Character columns with widths greater than 8000 characters continue to be converted to the TEXT data type, MS SQL Server's legacy CLOB equivalent data type. However, when a Unicode OpenEdge database is migrated to MS SQL Server 2005 (or greater), the **Maximum Varchar Length** threshold is set to 4000 by default and cannot be set any higher. Character columns with widths greater than 4000 characters are converted to the NVARCHAR(max) data type (MS SQL Server's Unicode CLOB-equivalent data type).

Note:

Progress recommends using the DBTOOL utility and the **Adjust Field Width** option when migrating an OpenEdge database with Unicode data to MS SQL Server. Prior to performing the migration, use the database administration utility, DBTOOL, to size your character columns to store your Unicode data. The DBTOOL utility has the ability to count the number of Unicode characters and make the required width adjustment for sizing columns on the foreign data source.

For more information on DBTOOL utility, see *OpenEdge Data Management: Database Administration.* For more information on the **For Field widths use** migration option, see Table 57: OpenEdge DB to MS SQL Server Conversion Advanced Options on page 275 and review Column width on page 294.

Indexes and sorting

You create and maintain all indexes from within the MS SQL Server data source using native tools, rather than with the 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.

Give careful consideration to benefit and cost of creating indexes. Having indexes for frequently executed queries can greatly improve record retrieval performance. An abundance of unused or infrequently used indexes can have a negative impact on performance due to the overhead cost of maintaining the indexes.

Using index definitions in the MS SQL Server data source, the DataServer builds index information in the schema holder. OpenEdge index definitions for the data source schema serve two purposes:

- They allow you to use the OF option in ABL 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 custnum field is common to both the order and customer tables, you could write the following statement:

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

- They translate USE-INDEX to SQL ORDER BY for DataServer operations. A MS SQL Server data source uses the ORDER BY clause to assist in selecting the optimal index for the query. For example, if you define city-dept as a MS SQL Server data source primary key on the city and department fields, it is a unique index in the schema holder. In this case, the following OpenEdge 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. If your application requires a predictable order, use include a USE-INDEX or BY clause.

USE-INDEX and BY clause considerations

The BY clause offers greater programming flexibility than the USE-INDEX clause, and should be used in most cases. If indexes are added or deleted, applications coded with the BY clause only require a recompile to adjust to the index changes. Applications coded with the USE-INDEX clause require code modifications to be current with the index changes. If the index specified by a USE-INDEX clause is dropped, the code will fail to compile until the code is modified. If there is a more efficient index than the one specified in the USE-INDEX clause, it will not be selected by ABL and it is unlikely to be used in the data source's query plan.

Dummy indexes for sort order

You can add dummy index definitions to your schema holder independent of the actual index definitions in your MS SQL Server data source to provide a sort order. These dummy indexes can improve the readability of your ABL code. An index in the schema holder need not match an index in the MS SQL Server 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 MS SQL Server 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 a MS SQL Server 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 runtime errors. See The DataServer Tutorial on page 227 for instructions on using the Data Dictionary to designate unique indexes.

MS SQL Server data source views and result sets from stored procedures do not have unique indexes. Just as for tables, you can use the 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. For more information about views, see MS SQL Server data source views on page 53.

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

Large key entry support

Updates in the OpenEdge database enable existing index key limits, currently set at approximately 200 bytes, to be widened. For example, this enhancement will enable databases with a block size of 4K or 8K to support a maximum index key of approximately 2000 bytes. Also, the index key width expansion extends the maximum key size supported in the OpenEdge client that can be used with databases exported to foreign data sources.

When you are migrating OpenEdge keys to a foreign data source, the key sizes cannot be larger than those supported by the foreign data source. Because the maximum OpenEdge key size is data-source dependent, you should consider your target data source's capacity with respect to the maximum OpenEdge key-size capacity before you perform a migration.

For example, migrating data to a MS SQL Server 2000 database will need to be limited to a wide key capacity of 900 bytes due to the 900-byte key restriction imposed by the foreign data source.

Case-sensitivity

By default, an OpenEdge database is case-insensitive. An MS SQL Server database is also case-insensitive by default, which is why the **Insensitive** field in the OpenEdge DB to MS SQL Server Conversion tool is set to "yes". Using case-insensitivity in both OpenEdge and MS SQL Server enables seamless compatibility between the two, and provides the best performance and least maintenance. Whether or not your data source code page is case-sensitive, you can set the attributes of any field to be either case-sensitive or case-insensitive.

If you intend to use case sensitivity, consider the following:

- Pattern-matching literals in data source access statements retrieve case-sensitive data.
- The OpenEdge database considers the user ID and password submitted at connection time to be case-sensitive.
- If you define a case-sensitive code page in your foreign data source, you must set the **Insensitive** field in the OpenEdge DB to MS SQL Server Conversion dialog to "no". Case-insensitive columns, in your case-sensitive database migration, are given shadow columns to support case-insensitivity in the foreign data source if they participate as a component in an index.

After the migration process, if you perform a schema-pull operation, the columns migrated as case insensitive that were used as index components get appropriately pulled as case-insensitive columns into the schema holder by virtue of the shadow columns indicating case-insensitivity. But, the other columns that did not participate in indexes do not get any indication from MS SQL Server of their case-insensitivity. When the OpenEdge database migration to MS SQL Server sets the **Insensitive** indicator in the schema holder to "no", the foreign code page is assumed to be case-sensitive, so, when the **Update/Add Table Definitions** option is selected during the schema-pull operation to pull table definitions directly from a MS SQL Server, the unindexed table columns are assumed to be case-sensitive, even if they were migrated as case-insensitive and you had not made any changes to the table definitions you pulled. One side effect is that if your r-code was compiled before the schema pull operation, it would now have a schema mismatch with the pulled server definitions. This is because the r-code takes into account case-sensitivity during the schema check when files are opened at run-time.

To avoid schema mismatch during r-code execution, run the **Adjust Schema utility** against the original OpenEdge database after the **Update/Add Table Definitions** operation on tables that were originally migrated with unindexed case-insensitive columns. This causes the schema holder to be adjusted with the case-insensitive attribute of the column originating from the migrated OpenEdge database.

If an indexed field is case-insensitive, an OpenEdge database 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 code page, the DataServer can make this feature compatible across OpenEdge and MS SQL Server data sources. To support case insensitivity with a case-sensitive code page, an extra column, known as a shadow column, must be added to the data source immediately before the indexed column. This column is named `_S#_column`. See Migrating an OpenEdge database to MS SQL Server on page 267 for instructions on adding this column automatically with the OpenEdge DB to MS SQL Server utility.

Note: By default, sort order in MS SQL Server is not case sensitive.

Interaction of code page, collation, and case sensitivity

Properly setting code page, collation, and case sensitivity values such that they compliment each other, will yield the best scenarios for data access. To avoid conflict between code page, collation, and case sensitivity, set these characteristics at schema creation, and allow a schema pull to manage the DataServer integration. If any of these settings are changed, the schema holder should be regenerated. Table describes the interaction between code page, collation, and case sensitivity.

Initial value

By default, the OpenEdge database generates an initial value of blank for character fields and zero for integer and decimal fields when they are created in an OpenEdge database table. Since the **Include Defaults** option, by default, is unchecked in the database migration dialog, the initial value does not get generated as a default value on the server. Therefore, you must select the **Include Defaults** checkbox to generate initial value.

After migration, when the field is created in an OpenEdge application, the field receives its initial value from the schema holder as it is adjusted against the original OpenEdge database, which has the initial value set to 0 for the integer fields and blank for the character fields. Note that the default value from the server is not used because in OpenEdge, the initial value in the schema holder takes precedence over the default value from the Server.

If a field with an initial value is also marked with the **mandatory** attribute, the field receives the NOT NULL constraint on the server, and if the **Include Defaults** option is not selected during migration, the server does not receive any information about the "initial values" in the OpenEdge schema holder. In a situation where you mark the **mandatory** attribute for a field but you do not select the **Include Defaults** option, that is, you have a mandatory field without a default value on the server; you might encounter problems such as OpenEdge run-time error 110.

Note: In such a situation, a field would be pulled back with the UNKNOWN value as its initial value; and because the field is mandatory, the NULL translation of the UNKNOWN value would now be invalid as an initial value for the mandatory column during record creation if you did not assign any value during record creation.

To avoid errors under such conditions, run the **Adjust Schema utility** against the original OpenEdge database on the tables where fields were originally migrated with the mandatory attribute but without the **Include Defaults** option selected. The utility appropriately adjusts the original OpenEdge initial value of the associated **mandatory** field in the schema holder.

For more information on migrating an OpenEdge database to MS SQL Server and the Adjust Schema Utility, see Migrating an OpenEdge database to MS SQL Server on page 267, and Adjust schema utility on page 294.

MS SQL Server data source views

MS SQL Server 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` OpenEdge 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 ABL 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 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 Modifying field-level information on page 310 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 MS SQL Server 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.

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 OpenEdge databases (including the schema holder) and MS SQL Server data sources. In an application that executes both types, the OpenEdge trigger (`CREATE`, `FIND`, `UPDATE`, `DELETE`) executes first. If processing an OpenEdge trigger results in a data source request, the DataServer passes the request to the MS SQL Server data source and the operation (`INSERT`, `UPDATE`, `DELETE`) executes.

Triggers for OpenEdge databases and MS SQL Server data sources are independent of each other. A data source trigger that rolls back does not affect OpenEdge triggers. Defining a trigger in OpenEdge does not create a data source trigger definition. An OpenEdge trigger that rolls back does so independently of the data source's transaction scope. Note, however, that although triggers for OpenEdge databases and MS SQL Server data sources are independent, they might affect each other based on the kind of transaction your application is executing. For example, ensure that triggers running in both OpenEdge and at the data source **do not** dead lock on the same record access.

OpenEdge 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 an OpenEdge DB to MS SQL Server migration, the DataServer has traditionally simulated support for OpenEdge sequences using native MS SQL Server stored procedures and a sequence tables. Starting with MS SQL Server 2012 and OpenEdge 11.4, the DataServer for MS SQL Server began providing the ability to support OpenEdge sequences using sequences native to MS SQL Server. You cannot create sequences for MS SQL Server tables directly through the Data Dictionary.

The DataServer for MS SQL Server now offers a legecy, a revised legacy and a native version of the sequence generator. Each legacy sequence has one stored procedure and one table object associated with it whose names are prefixed with _SEQP_ and _SEQT_ respectively. The same objects in the new sequence generator are prefixed with SEQP_REV_ and SEQT_REV respectively and include a single common table object named SEQT_REV_SEQTMGR. When a sequence is migrated to MS SQL Server, the new sequence generator is used if the **Use Revised Sequence generator** option is checked in the migration dialog. If you remigrate with the same option turned off, the sequences will revert to the legacy model in the foreign database and the schema holder. If the "Try Native Sequences" option is checked, then, if the version of MS SQL Server supports native sequences, OpenEdge sequences will be instead supported by native sequences in MS SQL Server. If the version of MS SQL Server does not support native sequences, then the selections on the options described above will determine which traditional sequence generator will instead be used to support OpenEdge sequences during migration.Only one sequence generator is implemented for the DataServer at a given time. If you elect to use the generator and the legacy version already exists, the legacy version is replaced, and visa versa. It is also possible to change the sequence version using the delta.sql utility by setting or unsetting the option.

The sequence procedures all run in separate connections from your ABL transactions. In that connection, a different transaction is established to handle modifications to the sequence table rows when the legacy or revised sequence generator is used. With all the sequence requests made against the legacy sequence generator, if a sequence is being actively modified when another user attempts to access the sequence record, the second user must wait for a lock. Under heavy use, it is possible for a lock timeout to occur on a legacy or revised sequence generator. To avoid any probability of sequence timeouts, sequence retry logic in the DataServer attempts to handle concurrency automatically. In the remote instance of a sequence lock timeout that is not resolved internally, the user is given the option to continue to wait and retry or to cancel the retries and abandon the sequence request. Setting a non-zero value for the -Dsrv switch PRGRS_NATIVE_LOCKWAIT may further reduce any possibility of an unmanaged sequence timeout. For more information on setting the PRGRS_NATIVE_LOCKWAIT value, refer to Handling lock timeouts on page 79 or DataServer options on page 337.

The revised version of the sequence generator eliminates lock contention when using the commonly utilized NEXT-VALUE sequence function. Lock contention is also greatly minimized for the GET-VALUE function. Lock contention for SET-VALUE function is not reduced and may not be a better choice over the legacy sequence generator for this function. If your particular application is atypical in the sense that it uses the SET-VALUE function frequently as compared to your use of NEXT-VALUE and GET-VALUE, then the previous version of the sequence generator may provide an advantage. The above lock contention that can occur for a legacy and revised sequence generator does not exist if OpenEdge sequences are migrated using MS SQL Server native sequences instead.

Note: When using the generator, your first next-value operation produces a value that is equal to the initial value specified for the sequence generator. By comparison, the first value produced for the legacy sequence generator is equal to the initial value plus the increment value.

The bit size of the legacy database sequences can be either 32-bit INTEGER or 64-bit INT64. The generator was introduced after the DataServer had already transitioned to 64-bit values. Therefore all generators are 64-bit. The following section highlights changes regarding 64-bit sequences.

Native Sequence generator

OpenEdge native sequences can now be mapped directly to native sequences in MS SQL Server. This mapping alternative eliminates contention-related issues during ABL sequence operations. When using native sequences in your OpenEdge sequencemapping, sequence numbers are assigned independent of the application's transaction and this ensures that there is no contention. The performance is improved as there is no locking and sequence caching is possible with the use of Native Sequence generator. It simplifies the support infrastructure required for sequence generation and the sequence equation.

The following are improved with the introduction of the Native Sequence generator:

- Sequence value acquisition performance
- record creation performance

Note: While using DataServer for MS SQL server, if a user sets the current value of a native sequence, the start value is modified with the value provided in the SET CURRENT VALUE statement. If the NEXT-VALUE is called in a repeated loop till it reaches the maximum value of the sequence with the Cycle option active, the NEXT-VALUE restarts from the native sequence start value that is set by the last SETCURRENT-VALUE operation. It does not restart from the start value when the sequence was created. If the SET CURRENT-VALUEis not called, the NEXT-VALUE restarts with the start value that is set when the sequence is created.

Support for 64-bit sequences

The MS SQL Server DataServer supports the following changes for 64-bit sequences:

- An existing database sequence will remain defined as a 32-bit `INTEGER` and migrate as 32-bit, unless you explicitly choose to define a sequence as 64-bit `INT64`.
- The MS SQL Server DataServer can invoke a sequence function to process `INT64` values through a standard implementation. That is, use `GET`, `NEXT`, and `SET` sequence functions to get or set an `INT64` value.
- Starting with OpenEdge Release 10.1B, an OpenEdge database, by default, defines a Sequence field in the schema as an `INT64` data type. Therefore, sequences migrated from a new OpenEdge database will automatically expand to support `INT64` data types whereas previously they returned `INTEGER` values. You can override the `INT64` return value using the `INTEGER` function. However, note that there is a potential for an overflow condition to occur if the target functionality expects an `INTEGER` value.
- When the MS SQL Server is the target database, and the sequence's upper limit is defined as the Unknown value (?) in the source database, the migration utility will specify the sequence's upper limit value as 922337203685775807. However, if support for large sequences is not enabled for an OpenEdge database, the sequence's upper limit value is specified as 2147483647.
- An overflow condition can occur and subsequently an error message can appear on a client machine when a sequence generator is defined as a 64-bit value, but the value loaded into a client is OpenEdge Release 10.1A or earlier.
- Native sequences in MS SQL Server can be pulled into the logical database of a schema holder and used as an OpenEdge sequence in an OpenEdge application. There are a number of numeric data types in SQL Server to which native sequences are allowed to be mapped. The Dictionary will allow native sequences of any valid type to be pulled into your schema holder logical database. However, if you want full compatibility with your OpenEdge sequences, use the default 64-bit data type with your native sequences.

Data types

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

The DataServer translates MS SQL Server data types into OpenEdge 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 MS SQL Server `datetime` and `smalldatetime` data types as `timestamp`, which is the equivalent ODBC SQL data type and **not** the MS SQL Server `timestamp` data type. There can also be some restrictions in data source compatibility with OpenEdge.

You can also modify these definitions using the Data Dictionary. For example, the DataServer maps the MS SQL Server `tinyint` data type to the OpenEdge 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 Modifying a schema holder on page 309 for an explanation of how to use the Data Dictionary to change OpenEdge data types in the schema holder.

The table that follows lists these data types and their default OpenEdge equivalents. The data types in parentheses are alternative data types that you can specify in the schema holder for your MS SQL Server data source. See Data Type Details on page 365 for more details about each MS SQL Server data type, its ODBC SQL equivalent, and OpenEdge data type equivalent.

Table 9: MS SQL Server data type equivalents in OpenEdge

MS SQL Server data type	OpenEdge data type equivalent[14]

`integer`	`INTEGER`
	`(DECIMAL)`
	`(LOGICAL)`
	`(INT64)`

`bigint`	`INT64`
	`(DECIMAL)`
	`(INTEGER)`

`smallint`	`INTEGER`
	`(DECIMAL)`
	`(LOGICAL)`
	`(INT64)`

`tinyint`	`INTEGER`
	`(DECIMAL)`
	`(LOGICAL)`
	`(INT64)`

`decimal`	`DECIMAL`
	`(INTEGER)`
	`(INT64)`

`numeric`	`DECIMAL`
	`(INTEGER)`
	`(INT64)`

MS SQL Server data type	OpenEdge data type equivalent[14]
float	DECIMAL (INTEGER) (INT64)
double precision	DECIMAL (INTEGER) (INT64)
real	DECIMAL (INTEGER) (INT64)
char	CHARACTER[15]
nchar	CHARACTER[15]
varchar	CHARACTER[15]
nvarchar	CHARACTER[15]
text [16]	CHARACTER[15] CLOB[17]
varchar(max)	CHARACTER[15, 18] CLOB[17]
ntext[16]	CHARACTER[15] CLOB[17]
nvarchar(max)	CHARACTER[15,18] CLOB[17]
money	DECIMAL (INTEGER)
smallmoney	DECIMAL (INTEGER)
date[18]	DATE (DATETIME) (DATETIME-TZ) (CHARACTER)[15]

MS SQL Server data type	OpenEdge data type equivalent[14]
time[18]	CHARACTER[15] (DATETIME)
datetime	DATE (CHARACTER)[15]
datetime2[18]	DATETIME (DATE) (DATETIME-TZ) (CHARACTER)[15]
smalldatetime	DATE (CHARACTER)[15]
datetimeoffset[18]	DATETIME-TZ (DATETIME) (DATE) (CHARACTER)[15]
binary	CHARACTER[15]
varbinary	CHARACTER[15]
varbinary(max) [19]	CHARACTER[15, 18] BLOB[20]
image[16]	CHARACTER[15] BLOB[20]
bit	LOGICAL
timestamp	CHARACTER[21, 15]
identity [22]	Depends on underlying type[21]
uniqueidentifier	CHARACTER[15, 21]
computed columns	Depends on underlying type[21]
cursor	Not supported[23]
sql_variant	Not supported[23]
geometry	Not supported[23]
geography	Not supported[23]

MS SQL Server data type	OpenEdge data type equivalent[14]
hierarchyid	Not supported[23]
raw	Not supported[23]

Working with unsupported data types

You can read data from tables containing unsupported data types by using two possible workarounds - first, by creating a view or, secondly, by using a stored procedure.

To use the first workaround, create a view in the SQL Server database that excludes the unsupported column so that only supported data types are available. For more information on views, see MS SQL Server data source views on page 53.

To use the second workaround, write your own stored procedure with the RUN STORED PROCEDURE statement or write SQL into a SEND-SQL-STATEMENT stored procedure call that expresses Data Manipulation Language (DML) adequate to handle the table object without the unsupported types. You can omit the unsupported data type in your DML statements such that they are not returned in the result set or are CAST to data types that are supported. For more information on stored procedures, see RDBMS Stored Procedure Details on page 101.

Working with non-updatable data types

Because non-updatable columns can be read, they can also be dumped to a .d data file. But, these .d data files cannot be loaded with the load .d utility because of their non-updatable column status. The non-updatable values are applied by the server and therefore cannot be retained from the client. Use foreign database tools to dump and load tables with non-updatable columns that are controlled by the server. Alternatively, write a program to read data from your .d data file with non-updatable types into a temp table omitting the non-updatable columns using the EXCEPT phrase when you write them.

14 The initial entry identifies the default data type. Data types in parentheses identify supported options.

15 The value of this datatype changes, during migration, based on the settings of "_USE_OE_INIT_VALUE" or "_BLANKDEFAULT" environment variables. For more information, refer to Table 59: OpenEdge-to-MS SQL Server Conversion Hidden Environment Variables on page 284

16 Starting in MS SQL Server 2005, legacy "image" and "text" LOB types on the server can be expressed as BLOB and CLOB types respectively. While still supported, Progress recommends legacy LOB types be converted to varbinary(max) and varchar(max) types for better forward compatibility.

17 The default CHARACTER mapping for binary LOB server types can be changed to OpenEdge CLOB by selecting the **CLOBs** checkbox on the **Default to OpenEdge LOB for** option during object selection on schema pull. Changing the mapping from OpenEdge character to OpenEdge LOB may require code changes. However, Progress recommends mapping MSS character LOB types to OpenEdge CLOBs for better scalability.

18 Starting in MS SQL Server 2008 using native drivers, these data types can be pulled into an OpenEdge schema holder for MS SQL Server.

19 The FILESTREAM variant of varbinary(max) data type is also supported.

20 The default CHARACTER mapping for binary LOB server types can be changed to OpenEdge BLOB by selecting the **BLOBs** checkbox on the **Default to OpenEdge LOB for** option during object selection on schema pull. Changing the mapping from OpenEdge character to OpenEdge LOB may require code changes. However, Progress recommends mapping MSS binary LOB types to OpenEdge BLOBs for better scalability.

21 These non-updatable columns can be mapped into an MSS DataServer schema holder but cannot be written to on the server.

22 You can add an identity data type to a table if you have not already set the PROGRESS_RECID column to the table's identity column. You can only use the PROGRESS_RECID identity column if you want to set an identity column for ROWID designation in your OpenEdge file schema. identity columns created by the user cannot be designated for OpenEdge ROWID or used as part of an index composite that designates ROWID.

23 For more information on unsupported data types, see Working with unsupported data types on page 60.

Processing considerations for 32-bit and 64-bit data types

Starting with OpenEdge Release 10.1B, DataServer for MS SQL Server supports an existing or new 32-bit INTEGER data type defined in ABL. However, the internal bit size for an INTEGER data type and ROWID will automatically expand from a 32-bit field size to a 64-bit field size to accommodate larger values. In most situations, this expansion also supports the option to convert a database column from a 32-bit INTEGER to a 64-bit INT64 without performing a dump and load. For more details about ROWIDs, see ROWID function on page 91. Also, refer to the following sources for additional language details about INT64 and ROWID:

- *Release 10 4GL Data Types* Web paper available at the Web paper category of the PSDN Web site located at http://communities.progress.com/pcom/community/psdn.
- *OpenEdge Development: ABL Reference*

Processing activities that require additional action

The following processing activities require the defined action to successfully convert existing 32-bit INTEGER columns to 64-bit INT64 columns:

- When existing r-code references a column's original 32-bit INTEGER definition, you must recompile the code and check the table's Cycle Redundancy Check (CRC).
- Contrast the actions required to perform the following data definition changes:
  - If you change data definitions in your original OpenEdge database from INTEGER to INT64 and you had previously migrated your database to an MS SQL Server, you must repeat the migration step to change the field's physical storage on the foreign data source.
  - If you change data definitions in your original OpenEdge database from INT64 to INTEGER and you had previously migrated your database to an MS SQL Server, you must repeat the migration step to change the field's physical storage on the foreign data source, and then dump and reload the field's data.
- If you changed your native definition to include a bigint data type, it is recommended that you repeat the step to pull the table into your schema holder. Otherwise, a conversion from the native bigint data type might occur at run-time if your schema holder defines a different data type. Also, if the schema holder definition of the native bigint is INTEGER, an overflow condition might occur when accessing data in the foreign data source.

Determining your application needs

Determine your application needs and goals before changing INTEGER data types to INT64 data types in applications. Not all INTEGER data types will necessarily require the additional bit size, nor will wholesale changes in data types necessarily work efficiently with other interfaces to ABL. For more information about determining when to use INT64 versus INTEGER, refer to the *Release 10 ABL Data Types* Web paper available at the Web paper category of the PSDN Web site located at http://communities.progress.com/pcom/community/psdn.

DataServer for MS SQL Server support for datetime data types

OpenEdge DataServer for MS SQL Server supports DATETIME data types in MS SQL Server up to SQL Server 2008 with full support for the ABL DATETIME-TZ data type.

Working with MS SQL Server and ABL datetime data types

The following table defines the MS SQL Server 2000, 2005 and 2008 data types and identifies compatible ABL data types.

Table 10: MS SQL Server 2000 and 2005 datetime data types

MS SQL Server data type	Description	Compatible OpenEdge data types
DATETIME	Provides date and time data to an accuracy of one three-hundredth of a second (equivalent to 3.33 milliseconds or 0.00333 seconds). Values are rounded to increments of .000, .003, or .007.	DATE[24] CHAR DATETIME[25, 26] DATETIME-TZ[26]
SMALLDATETIME	Provides date and time data with accuracy to the minute.	DATE[24] CHAR DATETIME[25, 27] DATETIME-TZ[27]

Beginning in OpenEdge Release 10.2B, the DataServer for MS SQL Server supports mappings of MS SQL Server datetime data types DATETIME and SMALLDATETIME to OpenEdge data type DATETIME-TZ along with the compatible OpenEdge types listed in Table 9: MS SQL Server data type equivalents in OpenEdge on page 57.

MS SQL Server 2008 offers the DATETIMEOFFSET data type to support time zones in a datetime column as well as three other datetime data types. The following table defines the MS SQL 2008 data types and identifies compatible ABL data types.

Table 11: MS SQL Server 2008 datetime data types

MS SQL Server data type	Description	Compatible OpenEdge data types
DATE	Provides day of the year based on the Gregorian calendar.	DATE[28] CHAR DATETIME[28, 29] DATETIME-TZ[29]

[24] Default, compatible data type to which the native type can be converted.

[25] Check the **Default to OpenEdge DATETIME** option from **Pre-Selection Criteria For Schema Pull** on the **Update/Add Table Definitions** dialog to get this data type instead of the default.

[26] When converting OpenEdge DateTime/TZ data types to the legacy MS SQL Server DateTime data type, the same 3.33 millisecond accuracy that SQL Server uses is applied to OpenEdge values with millisecond precision. This means that .989, .990 and .991 round to .990, .992, .993 and .994 round to .993, .995, .996, .997 and .998 round to .997 and .999 rounds up to 1.000.

[27] When converting OpenEdge DateTime/TZ data types to the legacy MS SQL Server SmallDateTime data type, the same minute accuracy that SQL Server uses is applied to OpenEdge values to with second or greater precision. This means that seconds values up to 29.998 seconds will round down to zero and values from 29.999 seconds and up will round up to the next minute.

MS SQL Server data type	Description	Compatible OpenEdge data types
TIME	Provides time of day based on 24-hour clock.	CHAR[28] DATETIME[29]
DATETIME2	Provides variable precision of up to 100 nanoseconds.	DATE CHAR DATETIME[28, 29, 30] DATETIME-TZ[29, 30]
DATETIMEOFFSET	Provides date and time data with time zone offset and variable precision of up to 100 nanoseconds.	DATE CHAR DATETIME[29, 30] DATETIME-TZ[28, 29, 30]

Note: Date time related data types of MS SQL Server 2008 will work only with the SQL Native Client10 (SNAC 10.0) driver or above, and not with earlier native and non-native drivers, as earlier version drivers will not support 2008 date time data types.

The ABL DATETIME and DATETIME-TZ data types are compatible with datetime data types supported by MS SQL Server. The following table provides details on compatibility:

Table 12: ABL datetime data types

ABL Data Type	Definition	MS SQL Server compatible data type
DATE	ABL date contains only a calendar date component and there is no time zone element to it.	DATETIME [31] SMALLDATETIME DATE[32] DATETIME2 DATETIMEOFFSET

[28] Default data type mapping for this MS SQL Server data type.

[29] New MSS data types with a time component support fractional accuracy up to 100 nanoseconds. Hence, when we receive any time data from an OpenEdge datetime data type with a time component, the fractional portion of the time value greater than millisecond accuracy is dropped in the conversion. For example, a time value '12:00:00.1235678' in MS SQL Server would yield a resultant value of '12:00:00.123' in an OpenEdge DATETIME column. NOTE: The time component is truncated rather than rounded in order to be consistent with MS SQL Server which also truncates the value when converting between two datetime data types defined with inconsistent fractional accuracy.

[30] ABL data type limits MSS data type accuracy to millisecond precision.

ABL Data Type	Definition	MS SQL Server compatible data type
DATETIME [33]	The DATETIME data type consists of two parts, one an ABL date and one an ABL time. The unit of time is milliseconds from midnight.	DATE[34] TIME SMALLDATETIME[34] DATETIME2[32, 34] DATETIME[31, 34] DATETIMEOFFSET[34]
DATETIME-TZ [35]	Variation of DATETIME data type with time zone offset.	DATE[34] SMALLDATETIME[34] DATETIMEOFFSET[32, 34] DATETIME2[34] DATETIME[31, 34]

Legacy mapping based on MAP to MSS DateTime:

The following table explains the mapping:

Map to MSS Date Time (Checked/ Unchecked)	Data type	Corresponding Legacy/MSSQL Server 2008 compatible data type
Checked (mapping provides backwards compatibility to earlier versions of MS SQL Server)		
	Date	DateTime
	DateTime	DateTime datatype
Unchecked (Use Date, DateTime2 and DateTimeOffset datatypes introduced in MSSQL Server 2008)		
	Date	Date
	DateTime	DateTime2
	DateTime-TZ	DateTimeOffset

[31] Default and backward-compatible, data type mapping used for an OpenEdge migration to the MS SQL Server DataServer when the Map to MSS Datetime Type check box is checked.

[32] Default data type mapping on an OpenEdge migration to the MS SQL Server DataServer when the Map to MSS Datetime Type check box is unchecked. NOTE: You should only uncheck this box when your target database is MS SQL Server 2008 or later.

[33] Compatible with MS SQL Servers 2000 and later.

[34] Since the valid date range for OpenEdge date and datetime data types with a date component is wider than the range available to MS SQL Server data types with a date component, it is recommended that a validation expressions be applied to fields in the schema holder mapped to MS SQL Server datetime types with a narrower date range. This would ensure that the client first blocked out-of-range dates before they were invalidated by server operations.

[35] Compatible with MS SQL Server 2008 and later.

Performing data type conversions

Making changes between ABL and MS SQL Server data types will affect how the data is stored. The following tables describe the effects of changing one data type to another.

The following table provides details on converting ABL DATETIME data types and MS SQL Server data types:

Table 13: Converting ABL datetime types to MS SQL Server data types

Source (OpenEdge)	Target (MSS)	Default Conversion
DATE	DATE	Straight copy of date portion.
DATE	DATETIME DATETIME2	Straight copy of date portion. Time set to midnight up to millisecond precision (00:00:00.000).
DATE	SMALLDATETIME	Straight copy of date portion. Time set to midnight up to 1 minute of accuracy.
DATE	DATETIMEOFFSET	Straight copy of date portion. Time set to midnight up to millisecond precision (00:00:00.000). Time zone set to the OpenEdge client session time zone context.
DATETIME	DATE	Straight copy of date portion of DATETIME value. Time and time zone portion are dropped.
DATETIME	DATETIME DATETIME2	Straight copy of the datetime portion up to millisecond precision.
DATETIME	SMALLDATETIME	Straight copy of the datetime portion up to a minute of precision.
DATETIME	TIME	Straight copy of time portion up to millisecond precision. Date portion is dropped.
DATETIME	DATETIMEOFFSET	Straight copy of the datetime portion. Time zone set to the OpenEdge client time zone.
DATETIME-TZ	DATETIME DATETIME2	Straight copy of the datetime portion of DATETIME-TZ value (in client time-zone context) up to millisecond precision. Time zone portion is dropped.
DATETIME-TZ	SMALLDATETIME	Straight copy of the datetime portion of DATETIME-TZ value (in client time-zone context) up to 1 minute of precision. Second and millisecond portion is dropped. Time zone portion is dropped.

Source (OpenEdge)	Target (MSS)	Default Conversion
DATETIME-TZ	DATETIMEOFFSET	Straight copy with millisecond precision.
DATETIME-TZ	DATE	Straight copy of date portion of DATETIME-TZ value (in client time-zone context). Time and time zone portion are dropped.

The following table provides details on converting ABL DATETIME data types and MS SQL Server data types:

Table 14: Converting ABL datetime types to MS SQL Server data types

Source (MSS)	Target (OpenEdge)	Default Conversion
DATE	DATE	Straight copy of date portion.
SMALLDATETIME DATETIME DATETIME2	DATE	Straight copy of date portion. Time portion is dropped.
DATETIMEOFFSET	DATE	Receives MSS DATETIMEOFFSET (with time zone context from the database). DataServer converts to OE client time zone context. Date portion of converted timestamp is copied. Time and time zone portions are dropped.
DATE	DATETIME	Straight copy of the date portion into the DATETIME. Time portion is set to 00:00:00
DATETIME DATETIME2	DATETIME	Straight copy of datetime portion up to millisecond level accuracy.
SMALLDATETIME	DATETIME	Straight copy of datetime portion up 1 minute of accuracy. Second and millisecond portion set to 0.
DATETIME	DATETIME	Straight copy with millisecond accuracy.
DATETIMEOFFSET	DATETIME	Receives MSS DATETIMEOFFSET with stored time zone context. DataServer converts to OE client time zone context. Timestamp portion (in client time) is copied with millisecond accuracy. Time zone portion is dropped.

Source (MSS)	Target (OpenEdge)	Default Conversion
TIME	DATETIME	Straight copy of TIME portion up to millisecond accuracy. Date portion is set to TODAY based on client time zone context.[36]
DATE	DATETIME-TZ	Straight copy of the date portion into the DATETIME-TZ. Time portion is set to 00:00:00. Time zone will be set to client time zone context.
DATETIME DATETIME2	DATETIME-TZ	Straight copy of the date and time portions into the DATETIME-TZ up to millisecond of accuracy. Time zone will be set to client time zone context.
SMALLDATETIME	DATETIME-TZ	Straight copy of the date and time portions of the MSS SMALLDATETIME with 1 minute accuracy. Time zone will be set to client time zone context.
DATETIMEOFFSET	DATETIME-TZ	Straight copy with millisecond accuracy.

Enabling Datetime data types using the Data Administration tool

The OpenEdge Data Administration tool provides utilities for migrating an OpenEdge database to a MS SQL Server data source and for pulling definitions from an MS SQL Server data source into an OpenEdge database, referred to as a schema holder. In doing so, OpenEdge applications use the DataServer to access and update data stored in MS SQL Server data sources.

The Data Administration tool ensures that utilities can recognize and handle date time data types supported in starting in MS SQL Server 2008 and later. Those utilities include:

- Update/Add Table Definitions
- Verify Table Definitions
- Schema Migration > OpenEdge DB to MS SQL Server
- Adjust Schema
- Generate Delta.sql OpenEdge to MS SQL Server
- Dump as Create Table Statement

Update/Add Table definitions utility

The Update/Add Table Definitions utility allows you to pull definitions from a table in a foreign data source. The table may be new or already exist in the schema holder, in which case you can add definitions or update definitions that are outdated. By default, MS SQL Server DATETIME and SMALLDATETIME data types are mapped to the OpenEdge DATE data type. The utility allows you to change the mapping to OpenEdge DATETIME.

[36] When a TIME data type in MS SQL Server is mapped to an OpenEdge Datetime data type, the date portion defaults to TODAY's date unless a valid date is specified for the -Dsrv PRGRS_DATEPART_OVERRIDE switch at connection time. This switch allows the user to set a date for this conversion other than TODAY's date.

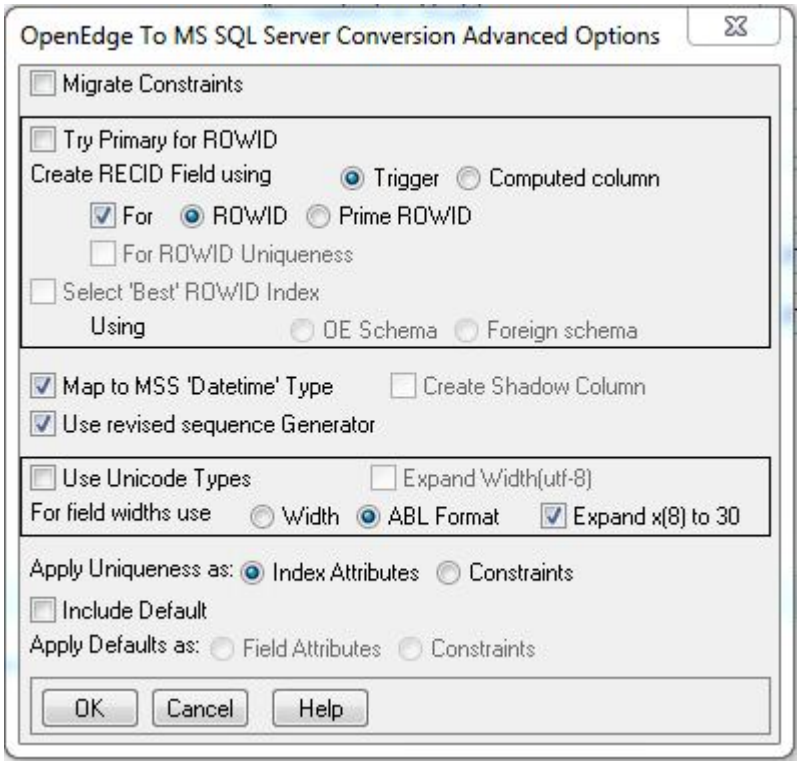
Note: Since time zone support is new in 10.2B, any DATETIMEOFFSET column pulled from an MSS SQL Server 2008 or later database is automatically mapped to DATETIME-TZ by default.

Schema Migration > OpenEdge DB to MS SQL Server

By default, ABL DATE and DATETIME data types were mapped to MS SQL Server DATETIME data types. However, the migration tool provides you with the option of selecting the following datetime data type mapping:

OpenEdge Data Type	MSS Data Type
DATE	DATE
DATETIME	DATETIME2
DATETIME-TZ	DATETIMEOFFSET

To override the default mapping and choose a mapping to the data types supported by MS SQL Server 2008, uncheck the **Map to MSS 'Datetime' Type** option in the **OpenEdge to MS SQL Server Conversion Advanced Options** dialog (You can open this dialog by clicking the **Advanced** button on the **OpenEdge DB to MS SQL Server Conversion** dialog.)



Note:

Do not uncheck the **Map to MSS 'Datetime' Type** option if your migration target is not an MS SQL Server 2008 database or above.

Date time related data types introduced with MS SQL Server 2008 will work only with the SQL Native Client 10 (SNAC 10) driver or above, and not with earlier native and non-native drivers.

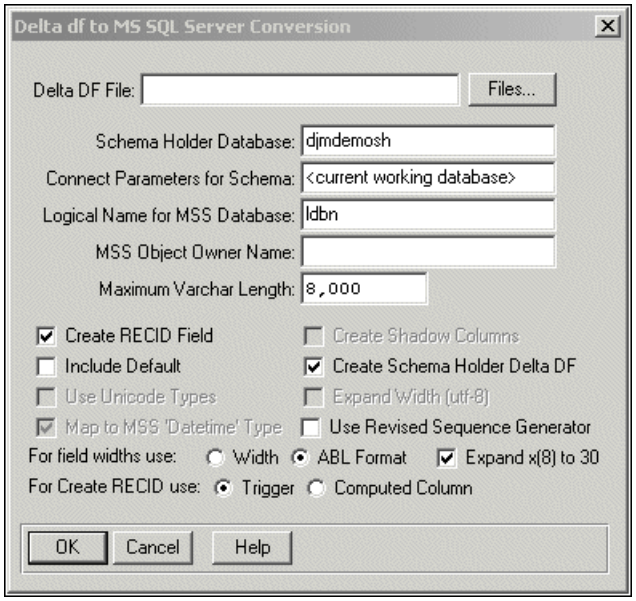
Verify Table Definitions and Adjust Schema utilities

The **Verify Table Definitions** utility enables you to verify that the definitions in the schema holder match the definition of objects in the foreign database. The **Adjust Schema** utility compares the connected OpenEdge database to the schema image in the connected schema holder and updates the OpenEdge attributes necessary for your ABL code to run against both data sources. Both utilities are updated to work with datetime data types.

Generate delta.sql OpenEdge to MS SQL Server utility

The **Generate delta.sql** utility allows you to migrate schema changes from an OpenEdge database to an MS SQL Server database. The utility reads a delta.df file that has been created using the incremental dump procedure and creates a new file containing SQL DDL for updating the .df file which can then be used to complete the migration process.

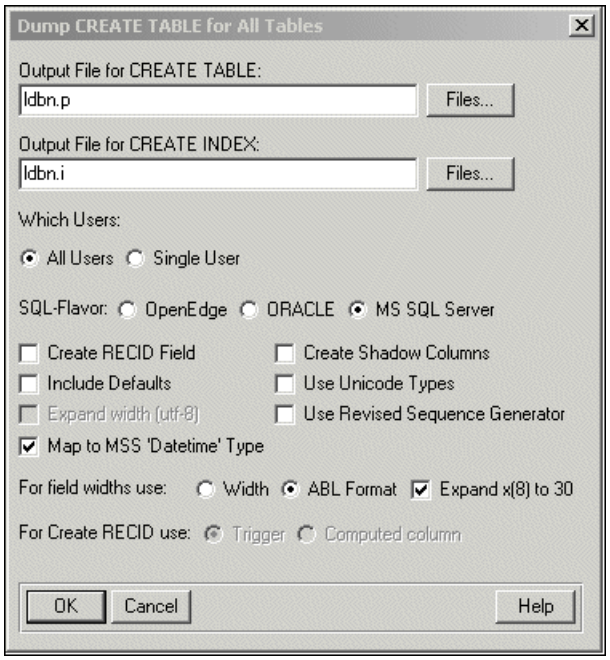
By default, the **Map to MSS Datetime Type** option is checked. Uncheck the box to map MS SQL Server 2008 data types to ABL `DATE`, `DATETIME`, and `DATETIME-TZ` data types:



Note: An extent field can not be modified without being dropped. If you are modifying and extent field in an OpenEdge Database while generating a Delta.df, the Delta SQL utility generates a Delta.df file contains corresponding Drop and ADD statements for the field. Data in the old extent field will be lost as the Delta SQL utility will replace the existing extent field.

Dump as Create Table Statement

The **Dump as Create Table Statement** utility allows you to select one or all tables of the working database and generate scripts for MS SQL Server that contain statements equivalent to those originally used to define selected tables in the foreign data source. By default, the **Map to MSS Datetime Type** option is selected. Uncheck the option to generate columns with datetime data types available in MS SQL Server 2008.



Default and Special Datetime Default Values

When a DATETIME or a DATETIME-TZ value is defined to a table in the DataServer schema without a default value, its initial value is automatically set to "?" (the Unknown value). An option for setting the initial value is to use the ABL NOW function. It initializes both date and time parts of the current date and time and the time zone portion for DATETIME-TZ columns. Using NOW for initialization sets the date, time, and time zone based on SESSION:TIME-SOURCE.

Using Datetime Data Types with Stored Procedures

Beginning in OpenEdge Release 10.2B, DATETIME-TZ is included as a parameter type that can be passed to and received from stored procedures.

The DATETIME-TZ data types are definable in both static and dynamic temp tables. Therefore, along with parameterization, SENDSQL and stored procedure result sets can convert DATETIME-TZ data types when using LOAD-RESULT-INTO.

If query results derived from a stored procedure are defined by foreign data types that require special mapping in order to retain their value or a portion of their value, they should be CAST to a type that can handle them. For instance, a TIME data type in a stored procedure result set should be CAST to a datetime data type that can receive the time component since OpenEdge does not support a TIME data type directly. The CAST should be done as part of the result set before it is received into a SQL View or LOAD-RESULT-INTO temp tables. Similarly, an MS SQL Server datetime data type with time precision greater than OpenEdge should be CAST to a native datetime data type with the same or less precision than OpenEdge before it is returned in the result set so it can be properly received. Or, you could CAST a more precise value to a VARCHAR column in advance and preserve the precision in the alphanumeric form of an OpenEdge CHARACTER field.

Datetime index components

Indexing of DATETIME and DATETIME-TZ columns is supported in schema holders. Using DATETIME-TZ fields in an index is particularly useful since the date and time value is stored in UTC and therefore will display results in absolute time.

Using datetime data types in a WHERE clause

Mixing DATE, DATETIME, DATETIME-TZ in a WHERE clause or in other comparisons will cause compile time errors and are not valid comparisons.

Support for OpenEdge ABL CLOB data type

The DataServer provides support for the OpenEdge ABL CLOB (Character Large Object) data type by mapping it to the equivalent data types in the MS SQL Server. This support is limited to MS SQL Server 2005 and later versions. The existing default and legacy mappings of server CLOB to the OpenEdge CHARACTER data type can be optionally replaced with OpenEdge CLOB mapping. OpenEdge CLOB mapping provides additional support for CLOB server types that are unavailable with CHARACTER mapping. You must keep in mind that CHAR-mapped columns that are replaced with CLOB mappings requires that the table in which the CLOB resides has a ROWID index designation.

The ability to map to an OpenEdge CLOB data type enables you to:

- Migrate an OpenEdge database with CLOB data type to an MS SQL Server database as VARCHAR(MAX) or NVARCHAR(MAX) and to pull data back as a CLOB date type into the schema holder. For more information on enabling the CLOB data type during a schema migration, see Migrating an OpenEdge database to MS SQL Server on page 267.
- Pull MS SQL Server VARCHAR(MAX), NVARCHAR(MAX), TEXT, or NTEXT data types as an OpenEdge CLOB data type into the schema holder by using the **Default to OpenEdge LOB** option.
- Read and write LOB data of any encoding, including UNICODE, from or to an MS SQL Server database that is CLOB-enabled by using OpenEdge ABL ASSIGN or COPY-LOB operations.
- Pass a CLOB as an INPUT or an OUTPUT parameter to the stored procedure. You can use the OpenEdge MEMPTR and LONGCHAR data types as parameters to RUN STORED-PROCEDURE operation to map BLOB and CLOB server types, respectively. For more information on RDBMS Stored Procedure Details, see RDBMS Stored Procedure Details on page 101.

  Note: RUN STORED-PROCEDURE is not referred as an RDBMS transaction. It is a part of a transaction but is not in itself inherently a transaction. The ability to pass LOB parameters does not give you the ability to run stored procedures.

Note: OpenEdge provides partial read/write capability to BLOB and CLOB data types using STARTING AT, FOR, and OVERLAY AT n [TRIM] operations. This capability does not extend to the MS SQL Server legacy native data types, TEXT and NTEXT.

When the **Default to OpenEdge LOB for: CLOBs** is selected in **Pre-Selection Criteria** of the schema pull (see Independent OpenEdge MS SQL Server Schema Pull on page 315), it breaks the backward compatibility of the MS SQL Server DataServer which previously mapped server LOB types to the OpenEdge CHARACTER data type. Since the Adjust Schema operation requires CHARACTER type mapping for server LOB types, an Adjust Schema operation will fail to "adjust" back to CHARACTER . Since this schema adjustment will not occur, server CLOBs will now be mapped to OpenEdge CLOBs and will be useable as LOB objects in your OpenEdge DataServer application. LOB objects are manipulated differently from their original CHARACTER data types and the requirements for your CLOB-mapped application may be different from your original database if such server LOBs were originally migrated from CHARACTER types.

For more information, on using the ASSIGN and COPY-LOB statements, see *OpenEdge Development: ABL Reference*.

ABL-to-MS SQL Server data type mapping

The **Map to MSS Datetime Type for** option is disabled by default for BLOBs in order to provide backward compatibility, which means that the default behavior is to use the behavior from the previous release, mapping to the DATETIME data type, and not the new data types.

Table 15: CLOB data type in schema pull

MS SQL Server Data Type		OpenEdge Data Type	
MSS 2005	**MSS 2008**	**Prior to 11.0**	**11.0 and later**
VARBINARY(MAX)	VARBINARY(MAX)	CHAR	CHAR, CLOB
IMAGE	IMAGE	CHAR	CHAR, CLOB

For more information on ABL-to-MS SQL Server data type mapping, see Data types on page 56.

Support for OpenEdge ABL BLOB data type

The DataServer provides support for the OpenEdge ABL BLOB (Binary Large Object) data type by mapping to equivalent data types in the MS SQL Server. This support is limited to MS SQL Server 2005 and later versions. The ability to map to an OpenEdge BLOB data type enables you to:

- Migrate to an OpenEdge database with BLOB data types to an MS SQL Server database as VARBINARY(MAX) and to pull the server types back as a BLOB date types into the schema holder. For more information on enabling the BLOB data type during a schema migration, see Migrating an OpenEdge database to MS SQL Server on page 267.
- Pull MS SQL Server VARBINARY(MAX), IMAGE, or VARBINARY(MAX) FILESTREAM as OpenEdge BLOB data type into the schema holder by using the **Default to OpenEdge LOB** option.
- Read and write data to and from an MS SQL Server database that is BLOB-enabled by using ABL ASSIGN and COPY-LOB operations.
- Pass a BLOB as an INPUT or an OUTPUT parameter to the stored procedure. You can use the OpenEdge MEMPTR and LONGCHAR data types as parameters to RUN STORED-PROCEDURE to map BLOB and CLOB server types, respectively. For more information on RDBMS Stored Procedure Details, see Chapter 1, RDBMS Stored Procedure Details on page 1.

  Note: Do not refer to RUN STORED-PROCEDURE as an RDBMS transaction. It is a part of a transaction but is not in itself inherently a transaction. The ability to pass LOB parameters does not give you the ability to run stored procedures.

Note: MS SQL provides partial read/write capability to BLOB and CLOB data types using STARTING AT, FOR, and OVERLAY AT n [TRIM] operations. This capability does not extend to the SQL server legacy native data type, IMAGE.

When the **Default to OpenEdge LOB for: BLOBs** is selected in **Pre-Selection Criteria** of the schema pull (see Independent OpenEdge MS SQL Server Schema Pull on page 315), it breaks the backward compatibility of the MS SQL Server DataServer which previously mapped server LOB types to the OpenEdge CHARACTER data type. Since the Adjust Schema operation also expects CHARACTER type mapping for server LOB types, an Adjust Schema operation fails to "adjust", those server LOBs that are now mapped to BLOBs, back to RAW mappings. Since this schema adjustment will not occur, server BLOBs will now be mapped to OpenEdge BLOBs and will be useable as LOB objects in your OpenEdge DataServer application. LOB objects are manipulated differently from the original RAW data types and the requirements for your BLOB-mapped application may be different from your original database if such server LOBs were originally migrated from RAW data types.

For more information, on using the `ASSIGN` and `COPY-LOB` statements, see *OpenEdge Development: ABL Reference.*

ABL-to-MS SQL Server data type mapping

The **Map to MSS Datetime Type for** option is disabled by default for BLOBs in order to provide backward compatibility, which means that the default behavior is to use the behavior from the previous release, mapping to the `DATETIME` data type, and not the new data types.

The following table depicts mapping between the ABL `BLOB` data type and its MS SQL Server equivalent during a schema pull.

Table 16: BLOB data type in schema pull

MS SQL Server Data Type		OpenEdge Data Type	
MSS 2005	**MSS 2008**	**Prior to 11.0**	**11.0 and later**
`VARBINARY(MAX)`	`VARBINARY(MAX)`	`CHAR`	`CHAR, BLOB`
`IMAGE`	`IMAGE`	`CHAR`	`CHAR, BLOB`
	`VARBINARY(MAX) FILESTREAM`	Non-applicable	`BLOB`

For more information on ABL-to-MS SQL Server data type mapping, see Data types on page 56.

User-defined data types

MS SQL Server allows you to define your own data types that map to native MS SQL Server data types. When the DataServer reads the schema information for a user-defined data type, it reads the MS SQL Server base data type and maps it to the equivalent OpenEdge 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.

Arrays

The OpenEdge database allows you to define fields as arrays, also called *field extents*. The DataServer interprets specially named data source columns of the same data type as an OpenEdge field with the same number of array elements. You name the data source columns *column_name*##1, *column_name*##2, and so forth, to correspond to an OpenEdge array named *colunm_name*. The DataServer creates a single field definition in the schema holder for the field extents. See Migrating an OpenEdge database to MS SQL Server on page 267 for instructions on adding these columns automatically with the OpenEdge DB to MS SQL Server utility.

Unknown value (?)

The DataServer supports null values. Procedures that use a null value behave exactly as they do when accessing an Unknown value (?) in an OpenEdge 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 `custnum` field is not null-capable, the following statement fails at run time:

```
FIND customer WHERE customer.custnum NE ?
```

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.
customer.address2 = ?.
```

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 `NULL`s 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 customer.address2 EQ " ".
FIND FIRST customer WHERE customer.address2 EQ "".
```

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 customer.name BEGINS "":
```

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

```
FOR EACH customer WHERE customer.name BEGINS " ":
```

_BLANKDEFAULT

When records with a NULL string in equality matches but not records with the NULL value are queried on a server, MS SQL Server distinguishes between a NULL value and a NULL string unlike Oracle which excludes records with both a NULL or a NULL string in equality matches and treats both as NULL.

To maintain the distinction between NULL and NULL string in Oracle, a NULL string ("") migrated from OpenEdge to Oracle is converted to a single-spaced string (" "). This enables Oracle to achieve equality matches on null strings separate from NULL values.

If you are migrating from the Oracle DataServer to the MS SQL Server DataServer and have developed queries against Oracle that depend on the default value for migrated fields being a single-space string (" ") rather than as a null-string (""), a special, hidden environment variable "_BLANKDEFAULT" is made available to allow an OpenEdge Initial Value of the null-string ("") to be converted to a single-space blank string (" ") so that the initial values are consistent with the expectations for server data in Oracle.

Note: The "_BLANKDEFAULT" environment variable does not affect the conversion of the OpenEdge Unknown value (?) to NULL on the server.

Record creation

Record creation is handled differently for OpenEdge databases and MS SQL Server data sources accessed through the DataServer. The difference occurs because ABL code run against an OpenEdge database follows different record-scoping rules than ABL code run against a MS SQL Server data source.

Records are scoped to the outermost block in which they are used, or the smallest enclosing block that encompasses all references to a record. In the absence of enclosing blocks, a record is scoped to the procedure block. Outside of record scope, a buffer associated with a record is cleared by ABL.

The OpenEdge database writes a record to the database after all its indexed columns are assigned. The MS SQL Server DataServer writes records or record changes out to the database at the end of record scope or at the end of a transaction boundary within that record scope. For this reason, the duration of record locks can vary between OpenEdge and a MS SQL Server data source since the record lock will be held from the time of the write until the end of a transaction.

While both OpenEdge and MS SQL Server data sources both commit changes at a transaction boundary, OpenEdge is capable of downgrading the lock on a record within scope after a commit boundary has been traversed. A MS SQL Server data source drops all record locks at the end of a transaction. See Transaction scoping and buffer management on page 84 for more information.

The following ABL statements will force a physical database write to a MS SQL Server data source before the end of a transaction block or the end of record scope:

- RELEASE
- VALIDATE
- RECID/ROWID

While these statements can change lock status and force a write, they have no affect on transaction scope. Locks will be held from the time of the write until the time of a commit or rollback. For the RELEASE statement, the record is released from its buffer but is inaccessible and still locked awaiting a transaction boundary. After a RELEASE, a record would need to be reread in order to refill buffer content. The OpenEdge database waits for all the key values to be assigned or for the end of record scope to write a record regardless of the outcome of a VALIDATE statement. The MS SQL Server DataServer writes a record at the time of the VALIDATE since a write is required in order to resolve the integrity of a record within the MS SQL Server data source. OpenEdge can assign a RECID or ROWID without writing a record. For a MS SQL Server data source, the generation of RECID and ROWID are contingent on a record write in order to produce its value.

The availability of database changes is dependent on when the record is written. The user writing the record can read back the in-memory copy of the record as soon as it has been written out to the database. Another user trying to access the same record may or may not see the changes written to the database, depending on the transaction isolation level of the foreign data source. While OpenEdge will show other users records modified but not yet committed, a MS SQL Server data source will not report on the state of record content until a transaction boundary has been traversed. Therefore, while OpenEdge might indicate that a record already exists or has certain content before another user has committed such changes, a MS SQL Server data source will report that the record is in use and block access until the transaction is complete. The code fragments in this section illustrate these differences.

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

```
DO TRANSACTION:
    CREATE customer.
  ASSIGN
    customer.name    = "SMITH"
    customer.custnum = 10
    customer.address = "1 Main St".
END.
```

When you run this procedure:

- The OpenEdge database 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, the OpenEdge database writes the record after executing the statement custnum = 10.
- The DataServer writes the record later, 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 the OpenEdge database and DataServer record creation:

```
DEFINE BUFFER xcust FOR customer.
CREATE customer.
customer.custnum = 111.

FIND xcust WHERE xcust.custnum EQ 111.
DISPLAY xcust.
```

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

- The OpenEdge database 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.
customer.custnum = 111.
VALIDATE customer. /* or RELEASE customer. */

FIND xcust WHERE xcust.custnum EQ 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. Using the VALIDATE or RELEASE statements forces a write to the data source even if the transaction has not yet completed. This makes the record available in the local database cache and it will be found if the record must be re-read before the end of the transaction.

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 a MS SQL Server 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 customer.custnum EQ 12.
DO TRANSACTION.
  ASSIGN customer.address = "14 Oak Park".

  FIND customer WHERE customer.address EQ "14 Oak Park".
END.
```

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

```
FIND customer WHERE customer.custnum EQ 12.
DO TRANSACTION:
    ASSIGN customer.address = "14 Oak Park".
    VALIDATE customer.
    FIND customer WHERE customer.address EQ "14 Oak Park".
END.
```

For more information about record scoping and transaction behavior, see *OpenEdge Getting Started: ABL Essentials*.

Data source record locking

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

The following table provides data source specific comparisons.

Table 17: OpenEdge database and data source locking

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

The DataDirect drivers provide four transaction isolation levels in the following order from least to most restrictive: read uncommitted, read committed, repeatable read, and serializable. In a multi-user configuration, you can isolate users from each other in your data source by setting the isolation level. In your OpenEdge 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 Microsoft documentation and the MS SQL Server documentation for more information.

Note: MS 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 MS SQL Server documentation for details.

The following table shows the possible -Dsrv TXN_ISOLATION,n values with the respective meaning.

Table 18: TXN_ISOLATION values in the -Dsrv parameter

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

Share locks

The default isolation level for the MS SQL Server DataServer is read uncommitted. At this level, a SHARE-LOCK and NO-LOCK are identical from the perspective of the DataServer and the MS SQL Server data source. The higher isolation levels will determine what kind of share locks will take effect. In MS SQL Server, a repeatable read and serializable isolation level are synonymous.

The MS SQL Server DataServer ignores ABL SHARE-LOCK option when used in ABL 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 an ABL statement, regardless of whether you include the SHARE-LOCK option, the MS SQL Server data source typically performs 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 an ABL 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 documentation that addresses ODBC programmingor data source reference manuals for more information.

Exclusive locks

When you update, delete, or create a record, MS 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 OpenEdge 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.

Handling lock timeouts

The default behavior for handling a lock timeout condition in the OpenEdge DataServers is to return control immediately to the OpenEdge client. Therefore, the lock wait timeout at the data source is set to zero at the beginning of a client session when using the OpenEdge DataServer for MS SQL Server. This is desirable behavior for clients that want immediate control over lock handling. The client application can choose to handle the lock timeout directly using the NO-WAIT and keywords. Then, when a record cannot be accessed because it is locked by another user, the application can test for the server timeout condition by testing for TRUE returned from the LOCKED function. The application consumes the timeout condition in this case and is free to perform whatever action is deemed necessary.

If the client application does not specify NO-WAIT, then the application automatically loops back to the server in an internal wait mode and retries record access. It continues to do so until the Lock Timeout period set on the client (-lkwtmo parameter specified in seconds) is exceeded. If this wait period is exceeded without being able to successfully access the server resource, the process times out, the wait is canceled and the client raises a stop condition.

When NO-WAIT is unspecified, the client consistently returns to the server to retry access to the locked resource. If is also unspecified, then during the Lock Timeout period (-lkwtmo) a resource wait dialog box continues to be displayed to the user. It allows the user to select **cancel** from the dialog to end the wait period. If the user does not cancel and the -lkwtmo period has not been exceeded, the client performs constant retries and multiple round trips to the server. This constant re-cycling, especially during a period of high resource contention, can be normalized by setting a small timeout period on the server in which to handle lock conditions before returning timeouts to the client application. The server wait period is set through the PRGRS_NATIVE_LOCKWAIT -Dsrv connection parameter. The disadvantage to setting this parameter to a non-zero value is that the client application is blocked for the timeout period set on the server. This may produce some amount of server-bound latency that should be considered when setting the number of milliseconds for this option. However, if the server is able to complete the resource request in the server timeout period, the resource is returned to the client immediately and the application unblocks. Therefore, the advantage of setting a non-zero server timeout is that the server is given the opportunity to resolve record access without further round trips from the client repeatedly request the same resource. A nonzero value may be especially useful during periods of high contention and may increase the overall efficiency of the DataServer application. Progress recommends a nominal but non-zero setting for the number of milliseconds in most cases. Evaluate your average contention for resources in setting this value for your own application.

The PRGRS_NATIVE_LOCKWAIT -Dsrv option permits an application to set a maximum time threshold that the server will block the application waiting to fulfill a resource request that is locked. When the server waits for the resource longer than the PRGRS_NATIVE_LOCKWAIT number of milliseconds, control is returned to the client application which then handles the lock condition as described earlier. As the PRGRS_NATIVE_LOCKWAIT time is increased, the number of retries from the client within the -lkwtmo period is decreased (assuming NO-WAIT is unspecified).

The PRGRS_NATIVE_LOCKWAIT setting will affect all transactions for all connections to the foreign data source for a given application session. This includes read-only connections, stored-procedure connections, and transactions on the sequences connection.

Note: Setting the PRGRS_NATIVE_LOCKWAIT is equivalent to setting the "LOCK_TIMEOUT" value natively in MS SQL Server.

Additional record locking details

In MS SQL Server, if transactions are scoped within other transactions, nested or embedded, all commit/rollback operations on the nested transactions are ignored by SQL server. Not until the transaction nesting levels are reduced to zero are all the nested levels actually committed or rolled back. This is a restriction of the data source with ramifications illustrated by the following code:

```
DO TRANSACTION:
  RUN STORED-PROC SP.
  CLOSE STORED-PROC SP.

  UPDATE record 2.
  ASSIGN fld2 = X.
END.
```

If the stored procedure SP has a commit or rollback statement within its TSQL, the commit or rollback isn't actually executed until the END statement in ABL, when the transaction level returns to zero. If `record 2` is also updated by SP, then `record 2` will be locked out to an ABL transaction that's also trying to update it. SP will continue to hold `record 2` and lock out an ABL update even if SP explicitly commits the update inside the stored procedure. To avoid this problem, it is necessary to recode the example as follows:

```
DO TRANSACTION:
  RUN STORED-PROC SP.
  CLOSE STORED-PROC SP.
END.

DO TRANSACTION:
  UPDATE Record 2.
  ASSIGN fld2 = X.
END.
```

NO-LOCK

The DataServer can emulate the OpenEdge NO-LOCK behavior if the isolation level is set to read uncommitted at connection time and you specify the NO-LOCK syntax in your ABL statement.

Locking impact on queries

The DataServer processes query statements marked with SHARE-LOCK and EXCLUSIVE-LOCK queries very differently from NO-LOCK queries.

Note: There is an exception to the previous statement; queries marked SHARE-LOCK when the record isolation level is read uncommitted are treated like a NO-LOCK.

The records of a NO-LOCK query are pre-fetched, which means that changes to the result set after the query has been executed are unknown. Records modified in the database after the query is established will not be reflected in the result set. SHARE-LOCK and EXCLUSIVE-LOCK queries do not pre-fetch records, but do preselect record keys. The query behavior is that of a "keyset-driven" cursor. Since the record keys are stored at the time an ABL query is first executed, new records added to the database since the key list was established are not included in the query result set. However, records that have been deleted or modified in the database will be reflected in the query results as records are accessed by the application.

Locking examples

The following example illustrates how the end of a transaction affects OpenEdge and MS SQL Server data source locks differently:

```
DO TRANSACTION:
   FIND customer WHERE customer.custnum EQ 10.
   UPDATE customer.
END.
```

Note the following points:

- When you access an OpenEdge database with this procedure, the `customer` record is share-locked when the first transaction ends.
- When you access a MS SQL Server data source with the DataServer, the `customer` record is released when the first transaction ends.

This example illustrates how OpenEdge and MS SQL Server data source share locks differ in scope and duration:

```
FIND customer WHERE customer.custnum EQ 10 SHARE-LOCK NO-WAIT .
IF AVAILABLE customer THEN DO:
  DISPLAY customer.
  PROMPT-FOR customer.

  tx:
  DO TRANSACTION ON ERROR UNDO tx, RETRY tx:
    FIND customer WHERE customer.custnum EQ 10
      EXCLUSIVE-LOCK NO-WAIT .
    IF LOCKED customer THEN DO:
      MESSAGE "customer locked - retrying".
      UNDO tx, RETRY tx.
    END.
    ELSE
      ASSIGN customer.
  END.
END.
```

In this example, the first record is only share-locked within the MS SQL Server data source if the isolation level setting requires it. (Recall that a `SHARE-LOCK` specified in an ABL statement is ignored by the DataServer.) As a result, if the isolation level is read uncommitted or read committed, the record might be updated by another client before the second `FIND` statement executes. This could cause the record that the second `FIND` statement fetches to be different from the record fetched by the first `FIND` statement. This procedure might cause updated information to be lost because the procedure applies updates based on the first find of the record, and these updates will overwrite the values from the second find of the record.

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

```
FIND customer WHERE customer.custnum EQ 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 under 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, OpenEdge 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 MS SQL Server 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 either has an exclusive lock on it, or 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 a MS SQL Server 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 ABL displays a message that the transaction was killed.
- The system responds as if you had chosen Cancel.

For details on how OpenEdge database locks work, see *OpenEdge Getting Started: ABL Essentials*. See ODBC and MS SQL Server documentation for more information about locks in MS SQL Server.

The NO-WAIT option

The `NO-WAIT` option works for DataServer applications in the same way that it works for OpenEdge applications: the DataServer uses a time-out mechanism. If MS SQL Server does not return a record, the DataServer considers the record to be locked. It then cancels the request to the MS 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.

Transactions

With DataServer operations, a MS SQL Server data source handles its own transaction roll back and recovery operations. However, the OpenEdge transaction scoping rules apply: a transaction ends when the code exits the outermost block that performs an update. With the DataServer, ABL performs as follows:

- Sends a `COMMIT` to the data source, when a transaction that updates a MS SQL Server data source ends successfully.c
- Sends a `ROLLBACK` to the data source if you interrupt the transaction.

See *OpenEdge Getting Started: ABL Essentials* for details on how ABL handles transactions and error conditions.

Handling rollback of sub-transactions

DataServer for MS SQL Server supports savepoints for backing out sub-transactions and to maintain sub-transaction boundaries efficiently. From OpenEdge Service Pack Release 11.7.2, ABL sub-transactions are translated to SQL Server savepoints and `UNDO` of a sub-transaction is handled by SQL Server.

The following sections describe how rollback of sub-transactions is handled:

UNDO of sub-transactions with LOBs

In previous releases of OpenEdge, backing out a sub-transaction with LOBs aborted the entire transaction and resulted in the following error:

```
"Undo of sub-transaction modifying a large object.  Backing out transaction. (11599)"
```

Using savepoints, DataServer for MS SQL Server allows you to rollback only that specific sub-transaction with LOBs without backing out the entire transaction.

UNDO of DELETE

In previous releases of OpenEdge, submitting a CREATE after performing an UNDO of a DELETE sub-transaction generated a gap in PROGRESS_RECID value. For example, consider that the PROGRESS_RECID value is 1, and you performed an UNDO of a DELETE sub-transaction. If you create a new record after the UNDO, the PROGRESS_RECID value changes to 3 instead of 2—a gap in the value. This gap in PROGRESS_RECID value is not generated using savepoints.

Transaction scoping and buffer management

The DataServer and an OpenEdge database hold locks differently at transaction boundaries. The OpenEdge database will hold a lock past the transaction boundary where the DataServer will not. This causes different behavior in the use of buffers. Consider the following example:

```
DEFINE BUFFER st_buf1 FOR state.
DEFINE BUFFER st_buf2 FOR state.

DO TRANSACTION:   /* Transaction 1 */
  FIND st_buf1 WHERE st_buf1.state EQ "NH" EXCLUSIVE-LOCK.
  DISPLAY st_buf1.state-name.  /* state-name = "New Hampshire" */
END.

/* During this pause, state-name is changed from "New Hampshire" to "Granite    State"
 by an external user */
PAUSE MESSAGE " state.state-name ".

DO TRANSACTION:   /* Transaction 2 */
  FIND st_buf2 WHERE st_buf2.state EQ "NH" EXCLUSIVE-LOCK.
  DISPLAY st_buf1.state-name st_buf2.state-name.
END.
```

Executing the above code against an OpenEdge database will downgrade the EXCLUSIVE-LOCK in Transaction 1 to a SHARE-LOCK. This occurs at the end of Transaction 1 and remains in effect at the time of the PAUSE statement and prior to the start of Transaction 2. This SHARE-LOCK prevents another user from modifying the state-name value between the transactions. As an optimization, when Transaction 2 is executed, the client does not refresh the st_buf1 buffer since the SHARE-LOCK prevented its contents from becoming stale between transactions. Also, since st_buf2 will have the same data integrity as st_buf1, they share the same buffer content.

Executing the above code against the DataServer will have different results because the DataServer does not retain any lock conditions beyond the scope of the transaction boundary. The DataServer will release the EXCLUSIVE-LOCK on the record at the END statement of Transaction 1. This leaves the record exposed to modification by another client during the PAUSE statement. If another client modifies the state-name value to "Granite State" during the PAUSE, Transaction 2 will read the updated value during the FIND statement. However, because of OpenEdge buffering rules and record scoping, neither buffer is refreshed with the updated value and the DISPLAY statement displays "New Hampshire New Hampshire."

To avoid this type of problem, the following workarounds are available:

- Change the lock status in Transaction 1 to NO-LOCK if an EXCLUSIVE-LOCK is not required. The lock upgrade in Transaction 2 will force the buffer to be updated in this transaction.
- If the lock can not be changed in Transaction 1, release the first buffer before the start of Transaction 2 with the statement "RELEASE st_buf1." The RELEASE statement will force a refresh of the data when the FIND in Transaction 2 executes.

Error handling

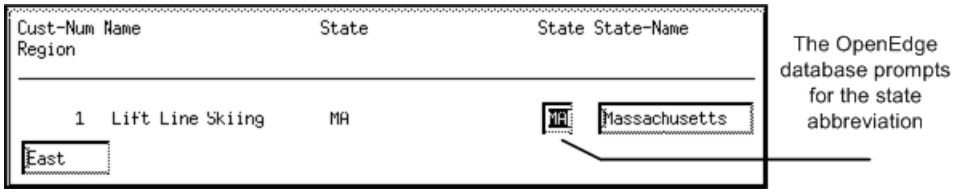
One common data entry error is attempting to add a duplicate record; that is, trying to create a record using a unique key that already exists in the database. Suppose, for example, that a user tries to add a customer with custnum = 1 (where custnum is a unique key), but a customer record with this custnum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

When this type of error occurs, ABL run time engine tries to resolve it by working back through the procedure, looking at each block header until the closest block with the error-handling property is found, and then undoing and retrying the block. (See *OpenEdge Getting Started: ABL Essentials* for more information about error handling.) However, because the DataServer is accessing a non-OpenEdge data source, duplicate-key errors can not be detected until the end of a transaction block. Therefore, if an error occurs in a subtransaction, it is not detected until the end of the entire transaction block and default error handling must be performed for the entire transaction block.

The following example illustrates OpenEdge database and DataServer error handling:

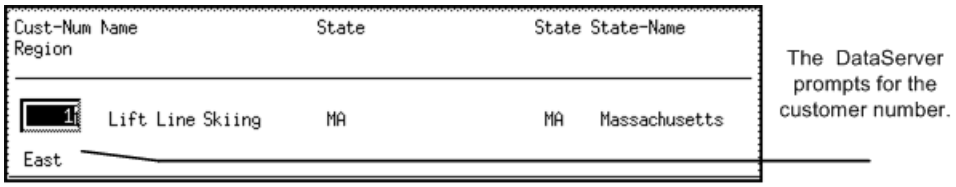
```
rep-blk:
REPEAT:
  PROMPT-FOR customer.custnum. /* User input */
  FIND customer USING custnum .
  IF AVAILABLE customer THEN
    UPDATE customer.custnum customer.name customer.state. /* User input */

  do-blk:
  DO ON ERROR UNDO do-blk, RETRY do-blk:
    FIND state WHERE st.state EQ 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 custnum field and then the state field:



Suppose that the user enters an existing state (for example, NH) while ABL is processing the DO block. When this duplicate-key entry occurs for an OpenEdge database, ABL 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, ABL 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:



If you use NO-ERROR to do your own error handling, you must account for the fact that a MS SQL Server data source creates or updates a record later than an OpenEdge database 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 .
ASSIGN customer.custnum = 45 .
ASSIGN customer.name = "Smith" .
```

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

```
DEFINE VAR jx AS INTEGER.

DO TRANSACTION:
  CREATE customer .
  ASSIGN customer.custnum = 45 .
  VALIDATE customer.
  IF ERROR-STATUS:ERROR THEN DO:
    MESSAGE "error: number of messages = " ERROR-STATUS:NUM-MESSAGES.
    DO jx = 1 TO ERROR-STATUS:NUM-MESSAGES:
      MESSAGE "error" ERROR-STATUS:GET-NUMBER(j)
        ERROR-STATUS:GET-MESSAGE (jx).
    END.
    UNDO, LEAVE.
  END.

  ASSIGN customer.name = "Smith" .
  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 OpenEdge keys. An OpenEdge database 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.

Overflow checking

Overflow checking is a process by which the DataServer evaluates whether the value defined for a data type exceeds the data type's capacity. If the value is greater than the bit size defined for a data type, an error message occurs.

Beginning with OpenEdge Release 10.1B, all in-memory operations on INTEGER and INT64 data types are now 64-bit, promoting potential conflicts when data is exchanged between OpenEdge Release 10.1A and earlier OpenEdge clients, and 10.1B MS SQL Server DataServers. For example, if a networked 10.1B DataServer produces an INT64 data type that is sent to a pre-10.1B client, and the INT64 data type is mapped to a 32-bit data type that exceeds 32-bits of data storage, the MS SQL Server will detect an overflow condition and an error message appears on the client machine.

Cursors

A *cursor* is like a pointer that points to consecutive records in a table. The ABL 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 custnum index, and your current record is customer number 50. This means that ABL has a cursor positioned at custnum 50. Note that ABL maintains cursor positioning across queries.

The DataServer allows applications that access MS SQL Server data sources to imitate OpenEdge cursor behavior for FIND cursors. FOR EACH and OPEN QUERY statements do not retain cursor position across other queries or against a FIND statement.

This section discusses the following cursor-related activities:

- Forward and backward scrolling on page 87
- Block cursors on page 88

For details about firehose and fast forward-only cursors, see Firehose, Firehose block, and Fast Forward-Only Cursors on page 141.

Forward and backward scrolling

A query is scrolling if you specify SCROLLING in the DEFINE QUERY statement or if you define a browse for the query. You can use the REPOSITION statement to change your current position with the result set. For a non-scrolling query, the DataServer allows you to only move sequentially forward through the rows by using the FIRST and options of the GET statement. Scrolling basically means that the result set is cached on the client for backward and forward scrolling.

With the DataServer, forward and backward scrolling works with a MS SQL Server just as it does with an OpenEdge database **except** when the data source fails to find a record. In ABL, the cursor position can change after a failed search; however, with the DataServer, a failed search does not affect the cursor. For example, if a cursor is positioned at custnum 50 and a request for the next customer from an OpenEdge table fails, the cursor moves to the next sequential record beyond custnum 50. The same failed request from a data source leaves the cursor at custnum 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 an OpenEdge database or a MS SQL Server data source.

Impact of MAX-ROWS

The MAX-ROWS qualifier on a query determines the number of rows returned. The MAX-ROWS calculation is applied **before** any sorting is applied to the query, causing the results to be random. Consider the following example:

```
DEFINE QUERY x FOR customer SCROLLING.

OPEN QUERY x FOR EACH customer MAX-R0WS 15.
GET LAST x.
DISPLAY customer.custnum.

REPEAT:
  GET PREV x.
  DISPLAY customer.custnum.
END.
```

When this example is run with the OpenEdge sports database, the first record return has `custnum 54`, and the repeat loop displays other `custnums` in descending order from there. When this example is run with the sports database migrated to MS SQL Server, the first record returned has `custnum 15`, and the repeat loop displays values in descending order from there. This difference in results is a caused by the MAX-ROWS being applied prior to the sorting.

Block cursors

Block cursors are available as a performance enhancement. They allow blocks of records to be bound and fetched together from a single database request, thus reducing the network overhead and minimizing record copies for ABL queries and finds, and minimizing record copies.

Considering when to use block cursors and firehose block cursors

A firehose block cursor is the default cursor processing mechanism to process the NO-LOCK queries for the DataServer for the MS SQL Server. A NO-LOCK query is one that executes with the NO-LOCK lock condition attached to its ABL statement. Firehose block cursors provide optimum performance for NO-LOCK queries and generally, surpass the performance benefits of the server-side block cursors because firehose block cursor does bulk processing at the client-side. For more information on firehose, firehose block, and Fast Forward-only cursors, see Firehose, Firehose block, and Fast Forward-Only Cursors on page 141.

While both the firehose block and the block cursors are the default cursor options for NO-LOCK queries at the client-side and at the server-side respectively, you can use the -Dsrv connect options available from your DataServer session to turn off either or both of these options. In addition, firehose cursors can be turned off for individual queries (at the statement-level) using the QUERY-TUNING options. For information on the query tuning options, see Query tuning on page 145.

When firehose cursors are turned off, a server-side cursor, managed by SQL Server, is used to process queries. When block cursors are turned off, a look-ahead cache is used instead to batch records retreived from the server for the NO-LOCK queries, and if the look-ahead cache is also turned off through a connection switch or a QUERY-TUNING option, then the records are fetched one at a time from the server as requested by the client.

Additional details about block cursor behavior

DataServer connections that run at the read uncommitted isolation level will also execute queries that specify the SHARE-LOCK condition as NO-LOCK queries and will also utilize block cursors for their result sets. You can shut off block cursor behavior and revert back to look-ahead cursors for NO-LOCK queries by setting the -Dsrv PRGRS_BLOCK_CURS,0. If you leave block cursors on but wish to turn off the block and/or look-ahead cursor optimizations for a specific query, you can set the QUERY-TUNING option QUERY-TUNING(NO-LOOKAHEAD) on your ABL statement.

While block cursors increase efficiency, they might also require more memory than look-ahead cursors. Block cursors will try to reuse memory from previous result sets whenever possible. You can adjust the memory available to block cursors according to your specifications. Adjust the cache size of an individual block with the same connection-level parameter you used to set the look-ahead cache size. The -Dsrv QT_CACHE_SIZE option allows you to specify at connect time how large the block cache should be for individual query results. When block cursors are active, the default QT_CACHE_SIZE is set to 10,000 bytes. When look-ahead cursors are active, this size defaults to 30,000 bytes.

Use the parameters to set the number of Dataserver connections for the read-only queries when run on the server. By setting the suitable connections for your application can increase the performance of the application. Than following -Dsrv is to set the managed connections -Dsrv RGRS_CACHE_CONN,n. Here 'n' is the number of connections. Let Dataserver to run the ABL queries like JOINs or stored procedures in separately connections by setting the switch -Dsrv QT_SEPARATE_CONNECTION improve the application performance.

You can override the connection-level QT_CACHE_SIZE at the statement level by setting the QUERY-TUNING option called CACHE-SIZE on the query itself. Determining the proper block size should be based on the maximum length of your returned records multiplied by the expected number of resultant rows and then compared to your available memory. Whenever an existing block is not reused, the block of memory allocated for the query will be adjusted downward to an exact multiple of the number of rows that can fit into the allocated area.

To prevent large applications from potentially overutilizing memory for block cursors, two other settings can be adjusted to limit the amount of memory available to block cursors. The first is the maximum block memory usage per table. If you have multiple queries open against the same table simultaneously, each query uses its own query block. The -Dsrv PRGRS_TABLE_BLOCKSIZE option puts an upper limit on the total memory available to query blocks of an individual table. This number should normally be set larger than the QT_CACHE_SIZE value. It can be set as high as two or three times the QT_CACHE_SIZE. If the maximum block memory available to the table will be exceeded by allocating space for the current NO-LOCK query in your ABL, the query is instead executed with a look-ahead cursor. The default maximum block memory area per table is set at 65,000 bytes.

The second adjustment switch available is the -Dsrv PRGRS_MAX_BLOCKSIZE option. This value sets an upper limit to the overall memory provided to block cursors, irrespective of per table allocations. The default maximum is 1048576 bytes (or 1MB). Your accumulated total memory allocated for block cursors will not exceed this value. If allocating block cursor memory for a NO-LOCK query in your ABL will cause this limit to be exceeded, the query would instead be executed with a look-ahead cursor. The PRGRS_MAX_BLOCKSIZE value should be set with respect to the amount of memory available on your machine. If you are running in client-server mode, the number should be set with respect to both the available memory on the server machine as well as the number of clients that will be connecting.

The following example uses the state table in the demo database, which has a maximum record size of 84 bytes:

```
FOR EACH state NO-LOCK QUERY-TUNING(CACHE-SIZE 850):
  DISPLAY state.
END.
```

In this example, the QUERY-TUNING CACHE-SIZE value overrides the -Dsrv QT_CACHE_SIZE query block size default of 10,000 bytes. Ten records at 84 bytes per record totals 840 bytes in the query block. If a previously established block is located for that table that is greater than or equal to the requested size, it will be reused. However, if a new block is established instead, its memory allocation will be adjusted downward to 840 bytes in order to fit an exact multiple of rows in the block. Then, if the result set contains 30 records, the query block will get refilled three times before the end of the result set if read sequentially.

Block cursors on versus off

Block cursors are the default over lookahead cursors when PRGRS_BLOCK_CURS is enabled and a server-side cursor is needed. The switch -Dsrv PRGRS_BLOCK_CURS,0 will turn off block cursors in MS SQL Server if you want to turn off the feature for this connection. You can always disable block cursors for a specific query by disabling lookahead with a query tuning option: QUERY-TUNING(NO-LOOKAHEAD). Block cursors is **not** a one size fits all feature. There are application environments where the default values for block allocation sizes can result in either under-utilization or over-utilization of memory. See Using the block cursor switches on page 346 for information on tuning the amount of memory that block cursors utilizes.

For any given query, the use of a block cursor or a lookahead cursor is mutually exclusive. In certain circumstances, block cursors will downgrade to lookahead. These are the criteria:

- When -Dsrv BINDING,0 is set. When binding is off, block cursors are disabled.
- Predictable single record result sets—FOR FIRST/FOR LAST or dynamic find operations.
- The PRGRS_TABLE_BLOCKSIZE has been exceeded and all existing block table space is in-use by open cursors.
- The accumulation of memory allocated to all block cursors has reached PRGRS_MAX_BLOCKSIZE.
- The QT_CACHE_SIZE query-tuning (CACHE-SIZE) value is not larger than 2 times the maximum row size. At least 2 rows need to fit in the result block to use a block cursor.

Block versus lookahead cursors

Block cursors and lookahead cursors are both associated with NO-LOCK queries. These are queries where the lock status has been explicitly set to NO-LOCK or where the lock status has been explicitly set to SHARE-LOCK and the transaction isolation level is set to read-uncommitted. Block cursors and lookahead cursors are mutually exclusive.

Block cursors are more efficient than lookahead cursors for the following reasons:

- The results of a query are bound directly to the area from which they are copied into the record buffer on the client. Lookahead cursors copy out of a common bind area into the lookahead cache and then are copied into client record buffer resulting in multiple copies of the data.
- The fetch process used by block cursors fetches multiple rows at a time, reducing the number of driver calls and potentially reducing the total number of network round trips to the server.
- Block cursor memory is preallocated prior to fetching a result set, whereas lookahead cursors post allocate memory as records are fetched from the result set.
- Block cursors are directly linked to the result set binding feature whereas lookahead cursors have no particular dependency on whether binding or late-binding is active.

ABL issues

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

ROWID function

The ROWID data type provides a unique record identifier that is compatible across OpenEdge and MS SQL Server databases. Applications that use the ROWID function behave in the same way whether they access records in an OpenEdge databases or in a MS SQL Server data source. The ROWID function is more consistent than RECID across data sources, since RECID cannot guarantee uniqueness. Progress Software Corporation recommends that you replace the RECID function with ROWID in existing applications.

The DataServer supports the ROWID function for MS SQL Server data source tables that have a unique index. The DataServer utilities use an index that meets this criterion to provide values for the ROWID function. If you build your schema holder using the OpenEdge DB to MS SQL Server utility and you select the **Create RECID field**option, 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 Defining the ROWID on page 312 for instructions.

Note: Starting with OpenEdge Release 10.1B, the default external representation of ROWID enables the MS SQL Server Dataserver to use a 64-bit integer value for ROWID. Using an integer value to define ROWID is not required, but rather an option to efficiently represent ROWID as a numeric binary value. The MS SQL Server DataServer supports the INT64 data type to emulate ROWID expansion support in this use.

The ROWID value in a MS SQL Server data source differs from the ROWID value in an OpenEdge database if you use the unique index method in the following ways:

- In an OpenEdge database, 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 OpenEdge ROWID function fails if you search for an Unknown value (?), because an OpenEdge 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) EQ ?.
  ```

- 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 OpenEdge databases and/or MS SQL Server 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.

Note: Reposition functions such as REPOSITION-BACKWARDS and REPOSITION-TO-ROW typically use ROWID to identify records. Functions of this type require integer expressions, which can be either INTEGER or INT64.

For a complete description of the ROWID function, see its reference entry in *OpenEdge Development: ABL Reference*.

RECID function

For backward compatibility, the DataServer supports the RECID function for MS SQL Server data source tables that have a unique 4-byte integer column defined as the key for the ROWID/RECID index of a given table in the schema holder. Whenever the ROWID index selection for a schema holder table in the Dictionary has multi-component key composite or is a single-component key but not a single unique integer component, the RECID function is not supported and the compiler will disallow the use of the RECID function in a WHERE clause.

Note: The ROWID function does not have this same restriction and is the recommended alternative for this limitation.

When the **Create RECID Field** option is selected, the OpenEdge DB to MS SQL Server migration utility creates an indexed column with unique values for each row called PROGRESS_RECID. Starting with OpenEdge Release 10.1B, the field is defined as bigint and in Release 10.1A or earlier, the field is defined as integer. You can also add this column to tables manually if you are using an existing MS SQL Server database or if you ported an OpenEdge database without the **Create RECID Field** option selected.

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 *OpenEdge Development: ABL Reference*.

RECID values can be stored in an INTEGER data type. The internal bit size for the INTEGER data type is not expanded from 32-bit to a 64-bit field size. If the RECID value stored in the INTEGER data type does exceed 32 bits however, an overflow condition occurs. See Overflow checking on page 86.

You can make an existing application that includes RECID behave more consistently across data sources by replacing RECID with ROWID. See ROWID function on page 91 for more information.

DEFINE BROWSE statement

The DEFINE BROWSE statement relies on a unique record identifier for forward and backward scrolling. If your MS SQL Server data source table does not support the OpenEdge ROWID function (through either a PROGRESS_RECID column or an indexed column with unique values, defined as bigint in OpenEdge Release 10.1B and later or as integer in Release 10.A or earlier), you can write code that explicitly requests the default OpenEdge browse scrolling behavior, similar to this:

```
DEFINE VARIABLE iRow AS INTEGER NO-UNDO.
DEFINE QUERY q FOR customer FIELDS (custnum name address) SCROLLING NO-LOCK.
DEFINE BROWSE b QUERY q DISPLAY custnum 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:
  iRow = CURRENT-RESULT-ROW("q").
  GET PREV q.
  GET NEXT q EXCLUSIVE-LOCK.
  IF CURRENT-RESULT-ROW("q") EQ iRow 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 customer.address WITH BROWSE b.
END.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

Field lists

The DataServer fully supports the use of field lists in queries (DEFINE QUERY, FOR EACH, PRESELECT, and SQL SELECT statements). Using FIELD and EXCEPT clauses can greatly improve performance for NO-LOCK queries, particularly if the DataServer is a remote configuration where the query results must be passed over a network. When a field list is used, unneeded data is not returned. The field list does not guarantee that it will restrict the data to the specified fields. It can be necessary to return additional fields such as those required to accommodate the values of a selected index.

For example, the following statement returns the same results for an OpenEdge database and a MS SQL Server data source:

```
DEFINE QUERY myquery FOR customer FIELDS (custnum name) SCROLLING.

OPEN QUERY myquery FOR EACH customer NO-LOCK WHERE custnum LT 6
  BY customer.name.
```

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. If a lock upgrade is required, the field list is ignored and all fields are retrieved.

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 the data source. For example, the DataServer fetches the name and the postalcode field to process the following query:

```
FOR EACH customer FIELDS (name) WHERE customer.postalcode EQ 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. Lookahead and block cursors gain performance by prebinding the fields of your result set. For maximum efficiency, any text or image fields should be explicitly excluded from your field list if possible because MS SQL Server does not allow those fields to be pre-bound.

Programmers are responsible for coding their applications to restrict the use of their query buffers to the fields specified by the field list. References to fields outside the field list are not caught by a compile time error. Sometimes such a reference will return a run time error, but that is not guaranteed. The following code **will** return a run time error reporting that the st field is missing from the customer buffer:

```
FOR EACH customer FIELDS (custnum name)
  WHERE customer.name EQ "Off the Wall" NO-LOCK:
  FIND FIRST order WHERE order.st EQ customer.st NO-LOCK.
END.
```

The following code **will not** return a run time error because the CAN-FIND expression resolves to FALSE, masking the fact that there was in fact no customer.st value to compare:

```
FOR EACH customer FIELDS (custnum name)
  WHERE customer.name EQ "Off the Wall" NO-LOCK:
  CAN-FIND(FIRST order WHERE order.st = customer.st NO-LOCK).
END.
```

See the "Record Phrase" entry in *OpenEdge Development: ABL Reference* for more information on the FIELDS option.

FIND statements

The DataServer fully supports the FIND FIRST, CURRENT, LAST, PREV, and NEXT statements for MS SQL Server 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 NEXT statement

The FIND NEXT statement can generate different results with the DataServer when compared to the same statement executed against an OpenEdge database. A FIND NEXT statement refers to a previous FIND statement only if the WHERE clauses of each statement are identical. If the WHERE clauses are different, or if one of the statements does not have a WHERE clause, the FIND NEXT will behave like a FIND FIRST statement.

FIND PREV and FIND LAST statements

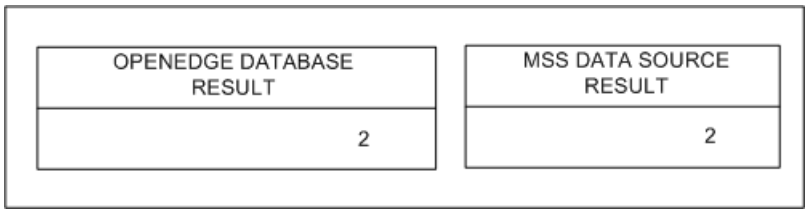
Applications that use the FIND PREV or FIND LAST statements work on tables in a manner consistent with the OpenEdge database. The only exception occurs when the FIND PREV or FIND LAST statement fails. In OpenEdge, the cursor is located after the last record that was scanned. In the MS SQL Server data source, the cursor behaves as if the failed FIND had never occurred. To support these statements, a table must include support for the OpenEdge ROWID function (through either a PROGRESS_RECID column or an indexed NUMBER column with unique values). See ROWID function on page 91 for more information.

For example, the procedure find.p accesses OpenEdge and MS SQL Server databases using the same FIND and FIND PREV statements in each case, as shown:

find.p

```
/* This code accesses an OpenEdge database. */
FIND demo.customer WHERE demo.customer.custnum EQ 3.
FIND PREV demo.customer.
DISPLAY demo.customer.custnum
  LABEL "PROGRESS DATABASE RESULT" WITH FRAME new-frame.

/* This code accesses a MS SQL Server data source.*/
FIND mssdemo.customer WHERE mssdemo.customer.custnum EQ 3.
FIND PREV mssdemo.customer.
DISPLAY mssdemo.customer.custnum
  LABEL "MSS DATA SOURCE RESULT" WITH COL 29.
```

When you run find.p with an OpenEdge table and a MS SQL Server table, you get the following 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.custnum 3 in the OpenEdge table, which was the last record scanned. In the data source table, the cursor is positioned at custnum 2. Failed finds do not affect cursor position in data source tables.

Compiling OpenEdge procedures

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

To compile procedures that access a MS SQL Server data source, start up an OpenEdge session 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 *OpenEdge Development: ABL Reference*.

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

---

r-code

r-code is generated when an ABL procedure is compiled. The compiled r-code is portable across user interfaces. For example, r-code that you compile on Sun Solaris can run on any other UNIX system (character user interface).

r-code is **not** portable across Window systems; that is, r-code compiled for a character application will not run on Windows and r-code compiled for Windows will not run on a character user interface.

r-code is also **not** portable among database management systems. ABL 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 MS SQL Server will not run on an OpenEdge database.

Prior to OpenEdge Version 11, r-code stored the physical position of a database column. r-code was not compatible with the server schema if the position of a column changed on the server. In Version 11, r-code stores a logical identifier for the physical position of a database column. Therefore, in OpenEdge 11, column positions in the server table can be rearranged without the need to recompile. However, introducing a new column or deleting an existing column from an existing record structure always requires the r-code to be recompiled against the new schema layout.

The r-code is dependent only on a logical identification of the column, and not the physical position of the column. The logical identification, in the OpenEdge database, takes place before migrating to a foreign data source, and the logical identification does not change once the column is defined in the OpenEdge table. When the columns are rearranged on the server, the code need not be recompiled, but the foreign schema must be pulled back into the schema image. This is to ensure that the new physical position of a column is matched in the schema to the logical identifier that is also known to the r-code.

You can only run r-code or load data (.d file) while connected to a schema image that is derived from the same database that was migrated and connected to the server when the code was first compiled and/or the data was first dumped. This ensures that the same logical column identifier is preserved after migration and/or after any subsequent schema pulls.

Note: When the server table is re-pulled into the same schema image (after the columns are rearranged on the server), the logical column identifier is re-matched to the physical column position so that the r-code can find again the column in a run-time application.

See *OpenEdge Deployment: Managing ABL Applications* for more details on r-code portability.

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

Relaxing CRC Validation Rules for pre-compiled r-code

When a DataServer application is presented to a non-matching schema holder the application partner may be able to avoid the recompilation of r-code despite schema inconsistencies should the application partner or ISV desire to ship r-code instead of source code. If adding a new field or modifying certain field attributes of an existing field in the database does not change the logical record position, the OpenEdge data type or the number of extents of a field being referenced by the r-code, r-code can still be run against the non-compliant schema without recompilation.

Internally, the DataServer schema holder stores the table CRC value for each user table loaded into the schema holder. This CRC value is the currency/signature for that particular table and is compared against the CRC value stored in the r-code if that particular table (the signature at the time of compile) is referenced by the r-code. Any mismatch between the two values is reported as CRC mismatch error. Normally, this problem is corrected by recompiling r-code for the application against the schema holder that had been modified since the last r-code compilation. This is the best way to both avoid CRC failure and ensure that there will be no inconsistencies in the CRC values or application behavior due to inconsistencies. However, it is now possible to run r-code against a schema holder logical database that has been updated with the following server changes:

Adding a New Field

New fields, added in the MSS DataServer, will now get a higher **field number than** the existing fields. This avoids position conflicts with any existing fields in the table. Existing fields will retain the **rpos** throughout their life time and will never change once assigned an **rpos**. If a field is deleted, **rpos** of that field will not be reused an it leaves a **hole** in the **rpos** structure. However the DataServer for MS SQL server, **rpos** of a deleted field may be reused by new field, if the new field is added from a .df file that has the same **position** attribute as that of the deleted field **rpos**.

Changing the Precision of a Character Column

Reducing the precision of a CHARACTER data type is possible without requiring a recompile. However this will cause a String Data right truncation ODBC error if the r-code tries to insert a value larger than the precision of the new column. However, reducing the precision does not cause an error for read only operations because the client code does not use precision from the r-code while retrieving records.

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 a MS SQL Server data source than when returning data from an OpenEdge database. The data source might not sort data like the OpenEdge database does, so the order in which records return might differ. In addition, the server-side result set is limited to a single record to improve performance.

Features and language differences

The DataServer supplies you with complete ABL functionality when accessing MS SQL Server data sources. Many ABL language elements (statements, functions, and so forth) and Data Dictionary features work the same whether your application accesses a data source through the DataServer or an OpenEdge database.

Use the DBRESTRICTIONS function to find out which OpenEdge features your MS SQL Server data source does not support. For the DataServer for MS 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 *OpenEdge Development: ABL Reference* for information on syntax.

The following table summarizes ABL differences between OpenEdge databases and MS SQL Server data sources.

Table 19: ABL differences between OpenEdge Databases and MS SQL Server data sources

OpenEdge feature	MS SQL Server 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. Any run-time substitution of the operands to the MATCHES and BEGINS functions causes the client to evaluate such phrases because only the OpenEdge client can handle wild card contents native to OpenEdge. Also, the wild card content is evaluated based on OpenEdge wild cards NOT based on the wild cards native to the foreign data source. **Note:** It is theoretically possible to do this with an OpenEdge database, but using this kind of criteria results in poor performance.
NO-LOCK option[37] SHARE-LOCK option EXCLUSIVE-LOCK option	An OpenEdge EXCLUSIVE-LOCK is emulated in the DataServer. An OpenEdge NO-LOCK can be emulated in the MS SQL Server database when the isolation level is set to read-uncommitted. SHARE-LOCK behavior is data source and isolation- level dependent. See Data source record locking on page 78 for more information.

[37] For more information, see Data source record locking on page 78.

OpenEdge feature	MS SQL Server data source difference
NULL	MS SQL Server NULL = Unknown value (?) OpenEdge empty string ("") = a one space string in MS 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.
SESSION:TIME-SOURCE handle	This system handle returns the MS SQL Server's server time information.
SETUSERID function	You cannot use this function to change the login name and password.

RDBMS stored procedures

Relational database management system (RDBMS) stored procedures can be an efficient, productive approach to obtain data from a foreign data source. In general, using stored procedures can help reduce network traffic, promote better performance, and improve response time because the server-side processing of the stored procedure can be accomplished without communicating with the client; once this processing is completed, the server returns only the data results to the requesting client.

See RDBMS Stored Procedure Details on page 101 for a complete discussion of various RDBMS stored procedure and send-sql- statement techniques and enhancements you can use to run against the MS SQL Server DataServer.

3

RDBMS Stored Procedure Details

This chapter defines relational database management system (RDBMS) stored procedures and describes how to create and implement them in the OpenEdge environment. It discusses various RDBMS stored procedure and `send-sql-statement` techniques you can employ. It also details how to execute an RDBMS stored procedure or send-sql-statement against a MS SQL Server and load its result sets directly into temp-tables.

For details, see the following topics:

- Overview
- RDBMS stored procedure basics
- Run Stored-Procedure details
- Data output and retrieval options
- Interfacing with RDBMS stored procedures
- Handling errors
- ROWID Support

Overview

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.

Note: This chapter uses the phrase RDBMS stored procedure and stored procedure interchangeably.

Enhancements to stored procedure techniques also allow you to extend your code in new ways and effectively employ more ABL features and functionality. For example, you can retrieve data from a foreign data source through a stored procedure and load this result set into a temp-table. Also, this technique is especially useful if you elect to further manage and manipulate your result set using ProDataSet capabilities.

Stored procedures can allow you to improve your dataserver's performance because they utilize the foreign data source's native capabilities. Stored procedures can be an efficient, productive approach to obtain data from a foreign data source or execute native business rules and procedures. In general, using stored procedures can help reduce network traffic, promote better performance, and improve response time because the server-side processing of the stored procedure can be accomplished without sustained communication with the client; once this processing is completed, the server returns the data results to the requesting client.

The first task to effectively employ any stored procedure technique or approach is to determine what information you want to receive or process you want to run. Once you know your data requirements, you can proceed to define the stored procedure details.

Note: A complete discussion of creating and using native stored procedures is beyond the scope of this chapter. For this type of information, see your Microsoft SQL Server documentation.

Defining native stored procedures to ABL

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 the precompiled procedure, assuming it is still cached in the database. This makes access to the database quicker and more efficient than when you access it with new queries each time.

The DataServer allows you to use ABL to run native MS SQL Server stored procedures. All pre-defined stored procedures initiated on behalf of the MS SQL Server are executed from within ABL, using the `RUN STORED-PROCEDURE` statement. You define specific ABL language elements to the `RUN STORED-PROCEDURE` statement to match the profile or signature of your native stored procedure. You can also provide additional ABL statements subsequent to the `RUN STORED-PROCEDURE` statement to handle result sets from stored procedures.

RDBMS stored procedure basics

In the OpenEdge environment, you can think of a stored procedure definition as having two basic, interrelated parts:

- **Execution controls to run and close a stored procedure** — Comprises the information needed to execute a stored procedure request against the MS S data source. At a minimum, all stored procedures discussed in this guide are assessable using the `RUN STORED-PROCEDURE` statement.
- **Language elements that enable access to specific data results** — Qualifies the retrieved data, or *result sets*, that the stored procedure's execution returns. Various keywords, phrases, statements, and syntax elements support different retrieval options for stored procedure output. This part of the stored procedure implementation reflects your analysis of your data needs; based on this analysis, you determine the additional syntax elements you need to define the output elements and data results you want retrieved.

The following table identifies and briefly introduces the elements that comprise a stored procedure definition; each of these elements is also more fully discussed later in this chapter.

Table 20: Stored procedure language elements

ABL language element	Description
RUN STORED-PROCEDURE statement	Executes a stored procedure
PROC-HANDLE phrase	Allows you to specify a handle to identify a stored procedure
PROC-STATUS phrase	Reads the return value
LOAD-RESULT-INTO phrase	Allows data from a result set that is returned for a foreign data source to be put into one or more temp-tables
PARAM phrase	Identifies run-time parameters to be passed to and/or from the stored procedure
CLOSE STORED-PROCEDURE statement	Enables the values to be retrieved from the output parameters that you defined for the stored procedure, finalizes result sets data processing, and tells OpenEdge that the stored procedure has ended
NO-ERROR phrase	Allows native database errors (that occur during stored procedure execution) to be evaluated and handled by the application. **Note:** This can also be achieved by a CATCH end block in the stored procedure. For more information on handling errors using the NO-ERROR option or a CATCH end block, see 'Handling errors' in *Use Microsoft SQL Server DataServer*.

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

For more information about the reference entries presented in the previous table, see 'Run Stored-Procedure details' in *Use Microsoft SQL Server DataServer*.

As previously noted in the table above, you can pass data types in the RUN STORED-PROCEDURE statement using the PARAM phrase. For supported data types with stored procedures, see 'Data types' in *Use Microsoft SQL Server DataServer*. The table below lists issues that occur when you pass certain data types as parameters.

Table 21: Argument data types for stored procedures

OpenEdge	MS SQL Server data source
DECIMAL INTEGER INT64	The DataServer converts each of these data types in the schema image to the equivalent default OpenEdge data type as follows: • DECIMAL=DECIMAL(default) , or Float • INTEGER=INTEGER • INT64=BIGINT However, you can use the 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( ) in the Transact-SQL of your native stored procedure to isolate parts of the date structure for which you might want to test.
DATETIME	You can specify a DATETIME data type in a temp table used to receive results from a stored procedure using the LOAD-RESULT-INTO phrase.
MEMPTR	You can specify a MEMPTR data type in a Param phrase as an INPUT or an OUTPUT parameter to receive results from a corresponding BLOB data type parameter to or from a RUN STORED-PROCEDURE statement.
LONGCHAR	You can specify a LONGCHAR data type in a Param phrase as an INPUT or an OUTPUT parameter to receive results from a corresponding CLOB data type parameter to or from a RUN STORED-PROCEDURE statement.

Note these stored procedure points:

- Input and output parameters are displayed as fields.
- Stored procedures called from within OpenEdge applications cannot return Boolean values to LOGICAL data types.
- If you are running several stored procedures, run them serially and process all the results from one stored procedure and close the procedure before you run a second one. By default, the DataServer allows one active request for running a stored procedure. It is not necessary to specify the PROC-HANDLE phrase when procedures are run serially.

When you run stored procedures concurrently, the DataServer uses one connection to the data source per procedure. If different 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.

Note:

You must define a PROC-HANDLE phrase for each stored procedure phrase that is simultaneously active. This technique provides a CLOSE STORED-PROC statement that can identify the targeted open procedure and close it.

In contrast, since a stored procedure executed with the LOAD-RESULT-INTO phrase implicitly closes the procedure once the execution ends and the data retrieved is placed into temp tables, it essentially runs serially and has no use for a PROC-HANDLE.

- When you create or update your schema image, the stored procedures appear in the list of accessible objects along with tables, view, and sequences. OpenEdge allows you to run the stored procedures that you create in data sources using the procedure definitions in your schema image. See your Microsoft SQL Server documentation for complete information about creating and using stored procedures.
- If the ABL that executes a stored procedure is already within a transaction block, the stored procedure becomes an extension of that transaction and will not commit to the database until the ABL transaction is completed. However, because the stored procedure does not execute as part of ABL client process, it cannot be rolled back by ABL.
- The DataServer cannot roll back sub-transactions in the stored-procedure context since it has no control over what the stored procedure executes.

The following section expands on the use of the RUN STORED-PROC statement.

Run Stored-Procedure details

This section provides:

- A complete, detailed reference for all the syntax expressions and options you can use to define the RUN STORED-PROCEDURE and the associated language elements that comprise a stored procedure definition.
- Syntax to use the RUN STORED-PROCEDURE statementwith and without the LOAD-RESULT-INTO phrase.
- Detailed information about using the RUN STORED-PROCEDURE statement with the send-sql option. Examples to use the RUN STORED-PROCEDURE statement with and without the LOAD-RESULT-INTO phrase are provided.

Complete syntax reference for Run Stored-Procedure

The following syntax shows the RUN STORED-PROC statement and all options that can be used with the statement:

```
RUN STORED-PROCEDURE procedure-name
[[ LOAD-RESULT-INTO handle [ int = PROC-STATUS ]] |
    [ int = PROC-HANDLE]]
    [ NO-ERROR ]
    [ ( [INPUT | OUTPUT | INPUT OUTPUT ]
                [ PARAM parameter-name = ]expression, ...
      [ INPUT | OUTPUT | INPUT OUTPUT ]
                [ PARAM parameter-name = ] expression ) ]
```

However, keep in mind that not all options shown can be used simultaneously.

Note: In the RUN STORED-PROCEDURE syntax, note that the *handle* is either a pure handle or a handle that points to a temp-table. For more information about the use of temp-table handle with the LOAD-RESULT-INTO phrase, see LOAD-RESULT-INTO phrase on page 107.

```
CLOSE STORED-PROCEDURE procedure-name
   [ integer-field = PROC-STATUS ]
   [ WHERE PROC-HANDLE = integer-field ]
```

The CLOSE STORED-PROCEDURE is not used when the LOAD-RESULT-INTO keyword is used with the RUN-STORED-PROCEDURE statement. See Close Stored-Procedure statement on page 109.

For valid combinations and usage of these syntax elements see:

- Run Stored-Proc statement execution without the LOAD-RESULT-INTO phrase on page 109
- Run Stored-Proc statement execution with the LOAD-RESULT-INTO phrase on page 110
- Run Stored-Proc statement execution using the send-sql-statement option on page 111

The following sections describe the syntax elements and options in detail. This section also presents examples that describe typical situations in which you might use the various options that the RUN STORED-PROC statement supports.

Run Stored-Procedure statement

The RUN STORED-PROC statement runs a RDBMS stored procedure or allows you to send Transact-SQL to a MS SQL Server based data source using an OpenEdge DataServer. It contains a procedure-name which is either the:

- Name of the RDBMS stored procedure that you want to run

- OpenEdge built-in procedure name, `send-sql-statement`, to send Transact-SQL to a MS SQL Server based data source

Note: If your ABL procedure is in a transaction block when a `RUN STORED-PROCEDURE` statement is executed, the RDBMS stored procedure runs within the same RDBMS transaction, even if your stored procedure declares its own transaction levels.

LOAD-RESULT-INTO phrase

The `LOAD-RESULT-INTO` function loads the result sets into a temp-table which is represented by a handle variable. Note that handle can also be defined as extent to enable you to pass more than one temp-table handle in those instances where SQL statement(s) are defined to return more than one result set.

When used with the `RUN STORED-PROC` statement or stored procedure to load result sets into temp-tables, this function carries an implicit `CLOSE-STORED PROCEDURE` statement.

Note: The compiler issues an error at runtime if the variable of type handle with the `LOAD-RESULT-INTO` function does not point to a temp-table.

For additional details about using the `LOAD-RESULT-INTO` phrase with the temp-table handle, see Loading results into a temp-table on page 113.

When used with the `LOAD-RESULT-INTO` phrase, the temp-table handle identifies the temp-table to which the result set will be loaded.

You can specify an array of one or more temp-table handle elements to retrieve stored procedure result sets and have the DataServer load the result set data directly into the associated temp-table(s). This approach allows you to have direct ABL access to the fields defined in the temp-table.

The following types of temp-tables can support result sets:

- **Static** — A temp-table whose schema is defined at compile time.
- **Dynamic** — A temp-table whose schema is defined at run time. There are two types of dynamic temp-tables: dynamic-prepared and dynamic-unprepared.

For additional details about using the `LOAD-RESULT-INTO` phrase with the temp-table handle, see Loading results into a temp-table on page 113.

PROC-STATUS phrase

The `PROC-STATUS` phase returns the return status from a MS SQL Server stored procedure. The *return status* is an integer value that typically indicates whether a stored procedure succeeded or failed; if a stored procedure fails, a code indicates why it failed. See your Microsoft SQL Server documentation for descriptions of the possible values for the return status.

PROC-HANDLE phrase

The `PROC-HANDLE` phrase allows you to specify a handle to act as a unique identifier for a MS SQL DataServer stored procedure. For example, the `PROC-HANDLE` assigns a value to the specified integer field or variable (*`integer-field`*) that uniquely identifies the stored procedure that is returning results from MS SQL Server.

Note these additional points about the `PROC-HANDLE`:

- 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 OpenEdge application.

NO-ERROR option

The NO-ERROR option specifies that any ERROR condition that the RUN STORED-PROCEDURE statement produces is 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.

For more information on handling errors using the NO-ERROR option or a CATCH end block, see Handling errors on page 127. For more information on the NO-ERROR option, CATCH statement or CATCH end block, in OpenEdge ABL, see *OpenEdge Development: ABL Reference*.

Note: This option must appear before any runtime parameter list.

PARAM phrase

The Param phrase identifies a run-time parameter to be passed to the stored procedure. A *parameter* has the following syntax:

```
[( [ INPUT | OUTPUT | INPUT-OUTPUT ]
                        [ PARAM parameter-name = ] expression, ...
      [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.

Note: When you run send-sql-statement for a MS SQL-based data source, it passes 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.

Close Stored-Procedure statement

For a RDBMS stored procedure, a PROC-STATUS received value indicates that the procedure has completed execution and makes available the return code value and any output parameters. For a send-sql-statement stored procedure, CLOSE STORED-PROCEDURE marks the stored procedure handle "reusable" and does not close the SQL cursor associated with the proc handle. When we get subsequent RUN STORED-PROC request, DataServer reuses the proc-handle if the SQL statement matches the existing statement in the cache. if not, it allocates a new proc-handle.

```
CLOSE STORED-PROCEDURE procedure-name
   [ integer-field = PROC-STATUS ]
   [ WHERE PROC-HANDLE = integer-field ]
```

procedure-name

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.

Consider these points concerning the CLOSED STORED-PROCEDURE statement:

- 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.

Run Stored-Proc statement execution without the LOAD-RESULT-INTO phrase

To implement most data retrieval options associated with a stored procedure, (excluding those that involve loading retrieved data into temp-tables) you must define the RUN STORED-PROC statement and explicitly end the stored procedure using the CLOSE STORED-PROC statement.

You must define the RUN STORED-PROC and explicitly complete access to the stored procedure OUTPUT with the CLOSE STORED-PROC syntax expressions for a stored procedure implementation that retrieves the following types of data from the foreign data source:

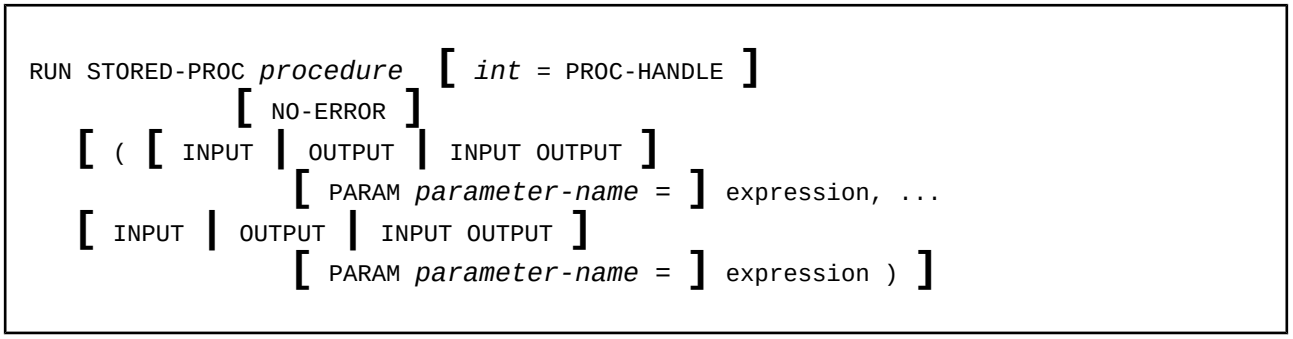
- proc-text-buffer
- user-defined views

For more information about these data retrieval options, see Data output and retrieval options on page 112.

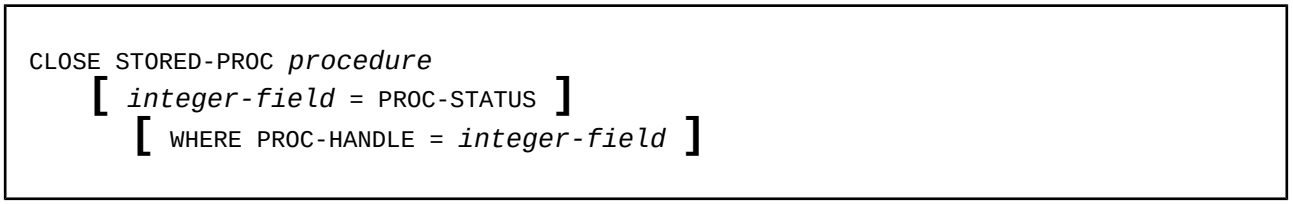
Note: You can also implicitly or explicitly close a stored procedure to retrieve these types of data: return code or output parameters. For more information about each of these data retrieval options, see Data output and retrieval options on page 112.

The syntax details presented in this section describe how to run and close a stored procedure in the MS SQL Server data source.

This is the syntax for the `RUN STORED-PROC` statement to execute an RDBMS stored procedure **without** the `LOAD-RESULT-INTO` phrase:



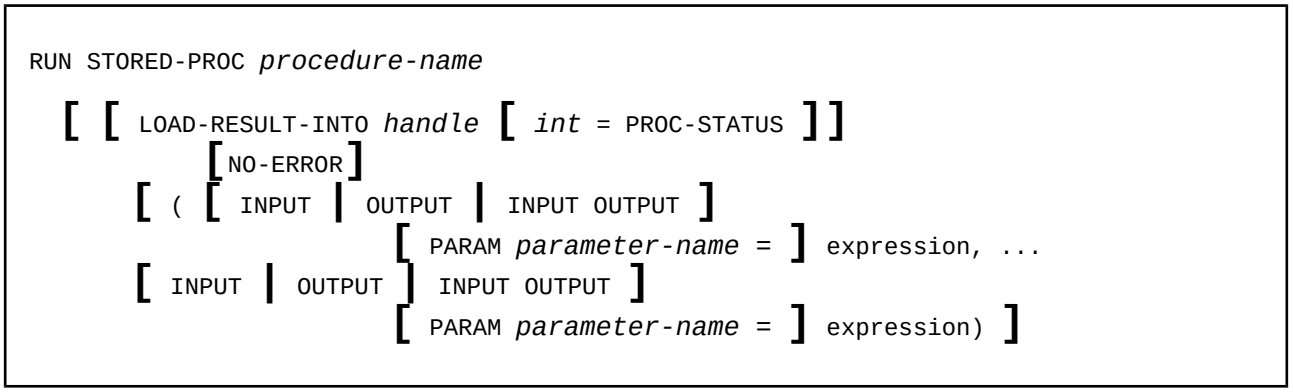
This type of syntax expression requires the explicit `CLOSE STORED-PROC` statement. This is the partial syntax for the `CLOSE STORED-PROC` statement:



Run Stored-Proc statement execution with the LOAD-RESULT-INTO phrase

You can also execute a stored procedure implementation to obtain result sets and load the results into temp-tables. You must define result set(s) target as part of the `RUN STORED-PROC` statement. You do not explicitly close a `STORED-PROC` defined this way as you do with buffer methods of retrieving result sets; the `LOAD-RESULT-INTO` function achieves this goal implicitly.

This is the syntax for the `RUN STORED-PROC` statement implementation when loading result sets to temp-tables:



The RUN STORED-PROC statement sets up the execution of the stored-procedure, retrieves the result set(s), and loads it into the temp-table(s) provided you use the LOAD-RESULT-INTO function. The CLOSE STORED-PROC statement is automatically applied when the result set(s) is loaded into temp-tables because using temp-tables must consume all result sets of the procedure.

Run Stored-Proc statement execution using the send-sql-statement option

ABL also allows you to use stored-procedure syntax to send SQL statements and their native language extensions 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. This option gives you access to Transact-SQL, providing you access to business logic for MS SQL Server. For example, you can issue Data Definition Language (DDL) statements from within OpenEdge procedures. You can send multiple SQL statements by concatenating them and passing the string as a single parameter to the send-sql-statement option.

Like the option to define a RUN STORED-PROC statement either with or without the LOAD-RESULT-INTO options, you can define a send-sql-statement with or without the LOAD-RESULT-INTO option. The following section presents and briefly describes a sample use of each technique.

Without the Load-Result-Into option

You use the RUN STORED-PROC statement with the send-sql-statement option and pass the Transact-SQL statements as a parameter. The syntax of the statement must be valid Transact-SQL syntax. The following example shows how this code passes a SELECT statement as a parameter.

Note: The SQL statement(s) passed to the send-sql-statement stored procedure is not limited to the 32,000 bytes of a OpenEdge CHARACTER field. To pass SQL of greater size, store the SQL text in a LONGCHAR field and pass that as a parameter to the send-sql-statement stored procedure.

Passing a SELECT statement as a parameter

```
DEFINE VARIABLE handle1 AS INTEGER NO-UNDO.

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.
```

The code above 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 an OpenEdge procedure. You can read the results into the proc-text-buffer defined by OpenEdge as shown in the example above. Alternatively, you can define your own buffer from within your data source that can accept other data types as well as the CHARACTER data type.

With the Load-Result-Into option

The following example shows how to use the `send-sql-statement` with the `LOAD-RESULT-INTO` option. It also shows that the `PROC-STATUS` phrase must be defined as part of the `RUN STORED-PROC` statement because of the implicit `CLOSE STORED-PROC` that is associated with the `LOAD-RESULT-INTO` phrase.

Using the send-sql-statement with the LOAD-RESULT-INTO option

```
DEFINE VARIABLE res    AS INTEGER NO-UNDO.
DEFINE VARIABLE tthndl AS HANDLE  NO-UNDO.

CREATE TEMP-TABLE tthndl.

RUN STORED-PROCEDURE send-sql-statement LOAD-RESULT-INTO tthndl
  res = PROC-STATUS ("SELECT * FROM customer").
IF res THEN
  DISPLAY "succeeded".

DISPLAY stat.
```

Also note that the `PROC-STATUS` phrase does not need an associated `PROC-HANDLE` phrase to close the associated procedure because it is retrieved using the `RUN STORED-PROC` statement; although the `PROC-HANDLE` is typically used after the execution of the `RUN STORED-PROC` statement, it is not needed in this context because of the implicit procedure close.

Data output and retrieval options

This section briefly highlights the data retrieval options that you can initiate through the execution of a stored procedure. See Interfacing with RDBMS stored procedures on page 114 for more information and examples of each option.

Return codes

This value might be a success code or a value returned by the stored procedure (as defined by the data source).

Values of output parameters defined when creating a procedure

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; your selected `INPUT` and `OUTPUT` parameter options must match the corresponding signature of the stored procedure at the data source.

Results retrieved from a database

This section identifies and briefly defines three coding techniques that ABL supports to retrieve data source results. In general, you can think of these techniques as handling complex query results for which each technique has its own purpose and benefits.

Using the OpenEdge-supplied proc-text-buffer for row results

This technique:

- Allows you to access result sets available through a pseudo table for which each row is a long character string
- Requires you to parse each long string to access specific fields

The `proc-text-buffer` is part of the schema holder, making this result set option independent of the foreign data source.

Defining a special view on the MS SQL Server data source to use as a buffer for row results

This technique allows you to:

- Define views within the foreign data source
- Use the result set buffer configuration of the view to access the data returned by the stored procedure

Defining of a special view involves some administration on the foreign datasource and therefore is considered a result set option that is dependent on the foreign data source.

Loading results into a temp-table

This technique allows you to:

- Specify one or more temp-table handle elements to retrieve stored procedure result sets and have the DataServer load the result set data directly into the associated temp-tables
- Have direct access to the fields defined in the temp-table

The temp-table approach maintains some separation between the data and the foreign data source which allows you to process the result sets quite independent of its database origin. Also, the result set definition is not contained within the schema holder.

Unless you pass unprepared dynamic temp-table handles into which the result sets are to be loaded, you still need to ensure that the temp-table record definition matches that of the stored procedures result set(s).

Passing empty temp-tables with the `LOAD-RESULT-INTO` phrase generates temp-table definitions dynamically based on a default mapping of data types. This method provides a data retrieval method completely independent of the foreign data source.

The `LOAD-RESULT-INTO` method represents a highly efficient means to transfer stored procedure result sets into temp-tables and provides significant performance gains over other techniques used to accomplish this same goal.

Note: The `proc-text-buffer` and the special native views techniques use the basic `RUN STORED-PROC` statement and are backward compatible to earlier versions of ABL. The `LOAD-RESULT-INTO` phrase provides a new technique to populate temp-tables directly in ABL.

Interfacing with RDBMS stored procedures

As previously mentioned, the RUN STORED-PROCEDURE statement is the initial statement required for the execution of all stored procedures you define and initiate through OpenEdge. It can run an RDBMS stored procedure bound by the data source, or allow you to send SQL to a MS SQL Server data source using an OpenEdge DataServer. The specific, additional keywords and elements you must include in a stored procedure or define for a send-sql-statement depend on the stored procedure characteristics including its signature, data results, and methods chosen to retrieve output to an ABL procedure.

This section identifies and describes the syntax elements, keywords, and other requirements associated with obtaining stored procedure output from a stored procedure data request.

The following example shows a stored procedure with multiple result sets, output parameters, and a return value.

A stored procedure with multiple result sets, output parameters, and a single return value

```
/* pcust stored procedure*/
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
```

This Transact-SQL code creates the stored procedure pcust and defines three parameters: num, which is an input parameter (the default parameter type), 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 associated with a record buffer defined by the procedure definition in the OpenEdge schema image; 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, it is stored as 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.

Retrieving return codes

You can create a stored procedure that provides return code information. For example, it might indicate whether the stored procedure was successful or whether it encountered an error condition.

The following example shows how ABL 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 example

```
/* Return status */

DEFINE VARIABLE handle1 AS INTEGER NO-UNDO.
DEFINE VARIABLE stat    AS INTEGER NO-UNDO.

RUN STORED-PROCEDURE pcust handle1 = PROC-HANDLE (20, output 0, output 0).
CLOSE STORED-PROC pcust stat = PROC-STATUS WHERE PROC-HANDLE = handle1.
IF stat = 0 THEN
  DISPLAY "procedure succeeded".
```

Retrieving output parameter values

When you call a stored procedure, you can specify an 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 OpenEdge data types.

Passing parameters by name using the PARAM option

The following example shows an ABL procedure that 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).
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.

Retrieving result sets using proc-text-buffer

The `proc-text-buffer` technique offers one approach to access results returned from a data source. The following 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 reference entry that discusses the DEFINE BUFFER statement in *OpenEdge Development: ABL Reference*.

Technique to use proc-text-buffer

The following example shows the results of the first ABL stored procedure, `pcust`, which is first presented as an example in A stored procedure with multiple result sets, output parameters, and a single return value.

pcust stored procedure

```
/* pcust stored procedure */
DEFINE BUFFER cust-state-join-buffer FOR proc-text-buffer.

RUN STORED-PROC pcust (20, output 0, output 0).
FOR EACH cust-state-join-buffer:
  DISPLAY cust-state-join-buffer.
END.

CLOSE STORED-PROC pcust.
DISPLAY pcust.orders pcust.states.
```

The OpenEdge-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 next example illustrates returning database results into the `proc-text-buffer` and converting the results to the `INTEGER` data type.

Returning database results into the proc-text-buffer and results conversion

```
DEFINE VARIABLE handle1 AS INTEGER NO-UNDO.
DEFINE VARIABLE iMax    as INTEGER NO-UNDO.

RUN STORED-PROC send-sql-statement handle1 = PROC-HANDLE
 ("SELECT max (cust_num) FROM customer").
FOR EACH proc-text-buffer:
    iMax = INTEGER(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 MS SQL data source. The ABL does not process it, so errors occur only at run time and not when you compile a procedure.

Note: The ABL `QUOTER` function can be useful for quoting values or handling embedded quotes when building SQL statements that will be passed into stored procedures with the `send-sql-statement` option.

Assessing result sets obtained from the proc-text-buffer technique

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, and converts it to the character equivalent. Another benefit of the `proc-text-buffer` is that it can be used to hold 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 results set record form one result set at a time.

However, a disadvantage in using the `proc-text-buffer` technique is that it is much more difficult to manipulate the data after you receive it as it requires parsing the data. 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.

Defining a view to use as a buffer

As an alternative to using the OpenEdge buffer `proc-text-buffer` definition, you can define a view in the data source that can serve as a buffer allowing you to retrieve database results in their original data types. However, keep in mind that using views creates a database dependency beyond the stored procedure itself.

Technique to define a view to use as a buffer

While a stored procedure can include multiple SQL statements, a buffer that you define contains the format of only a single results set. You need to define multiple views to accommodate multiple results sets.

To define a buffer:

1. Define a view in the MS SQL Server data source.

   The data source must have the following characteristics:

   - The naming convention `_BUFFER_buffername`
   - The same number of columns and data types that the stored procedure returns in the results set
   - 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 a 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 helps to indicate that the purpose of the view is its buffer content and not its SQL capabilities. 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 OpenEdge can use. (See Updating a schema holder on page 244 for instructions on using this utility.)

Assessing result sets obtained by defining a view as buffer technique

The buffer in the previous procedure 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 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 a MS SQL Server, or view procedures in the system tables.

The examples in this section do not use the supplied `proc-text-buffer` buffer. Instead, they show how to define formatted buffers by creating views in the data source, using the following syntax:

Syntax

```
CREATE VIEW _BUFFER_buffer-name
```

Views created in your MS SQL Server data source

The following code examples show two views created in your MS SQL Server 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
```

Result set of pcust - typed buffers

The following code example shows ABL procedure results of the previous stored procedure `pcust` as it is written into the new buffers `pcust_orders` and `pcust_states`.

```
/* Results of the first two example code examples - 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 OpenEdge-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 ABL 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 an OpenEdge database; for example, with Frame phrases and `FORM` statements.

Procedure handles

The following 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 VARIABLE handle1 AS INTEGER NO-UNDO.
DEFINE VARIABLE handle2 AS INTEGER NO-UNDO.

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.
```

The results look the same as in the first two examples. However, because you are running a stored procedure twice, ABL uses the procedure handles to identify the different instances. If you have more than one stored procedure open simultaneously in your application, you must explicitly define procedure handles for each.

Joining stored procedure results with other database tables

The following example shows how to use standard OpenEdge syntax to join the results from a stored procedure with other tables in the database.

```
/* Join with procedure results */
RUN STORED-PROC pcust (20, output 0, output 0).
FOR EACH pcust_orders, EACH orderline
  WHERE pcust_orders.ordernum EQ orderline.ordernum:
  DISPLAY orderline.ordernum orderline.itemnum.
END.
FOR EACH pcust_states:
  DISPLAY pcust_states.
END.
CLOSE STORED-PROC pcust.
```

The example code joins the order information returned from the stored procedure with the `orderline` information in the same database.

Loading result sets into temp-tables

Enhancements implemented through changes to the `RUN STORED-PROC` statement allow you to retrieve result sets from a foreign data source and load each result set, for which a temp-table handle is defined, into its own temp-table. The `LOAD_RESULT-INTO` function enables data retrieved to be loaded into temp-tables where the data can then be manipulated, employing all characteristics inherent to temp-tables. The capability to load result sets into temp-tables is not limited by the parsing requirements associated with the `proc-text-buffer` nor the database dependencies associated with views.

Temp-tables can provide data management capabilities associated with ABL directly to the result sets of a stored procedure, but completely independent of the foreign data source from which it was populated and/or derived. Temporary tables are effectively database tables in which OpenEdge stores data temporarily. Because temp-tables have the same support features that actual OpenEdge databases use, you can take advantage of almost all the OpenEdge database features that do not require data persistence and multi-user access. For example, you can define indexes for fields in the temp-table. For more information about temp-tables, see *OpenEdge Getting Started: ABL Essentials*.

RUN STORED-PROC statement with the LOAD-RESULT-INTO phrase

The following example shows how to use the `RUN STORED-PROC` statement with the `LOAD-RESULT-INTO` phrase with a single dynamic temp-table. It highlights the coding techniques discussed in Run Stored-Proc statement execution using the send-sql-statement option on page 111 the and it introduces the dynamic temp-table topic further discussed in later sections.

```
DEFINE VARIABLE bhCustomer AS HANDLE NO-UNDO.
DEFINE VARIABLE hBrowse    AS HANDLE NO-UNDO.
DEFINE VARIABLE hQuery     AS HANDLE NO-UNDO.
DEFINE VARIABLE ttCustomer AS HANDLE NO-UNDO.

DEFINE FRAME BrowseFrame WITH SIZE 80 BY 10.

CREATE TEMP-TABLE ttCustomer.

RUN STORED-PROC send-sql-statement LOAD-RESULT-INTO ttCustomer
  ("SELECT * from CUSTOMER").

bhCustomer = ttCustomer:DEFAULT-BUFFER-HANDLE.
CREATE QUERY hQuery.
hQuery:SET-BUFFERS (bhCustomer).
DISPLAY ttCustomer:Name.
hQuery:QUERY-PREPARE("FOR EACH" + ttCustomer:Name).
hQuery:QUERY-OPEN.

CREATE BROWSE hBrowse ASSIGN
  ROW         = 1
  COL         = 1
  WIDTH       = 79
  DOWN        = 10
  FRAME       = FRAME BrowseFrame:HANDLE
  QUERY       = hQuery
  SENSITIVE   = TRUE
  SEPARATORS  = TRUE
  ROW-MARKERS = FALSE
  VISIBLE     = TRUE.

hBrowse:ADD-COLUMNS-FROM(bhCustomer).
ENABLE ALL WITH FRAME BrowseFrame.
WAIT-FOR CLOSE OF CURRENT-WINDOW.
```

Getting started

If you are using static temp-tables, you must define the temp-table layout in your program to accommodate a specific stored procedure result set **before** attempting to populate these tables. Once this prerequisite task is done, however, temp-tables can also be automatically populated, offering a potential performance gain in most instances.

Unlike the proc-text-buffer technique, you do not have to parse the strings from the proc-text-buffer pseudo table buffer where each row is a character string. Similarly, you do not need to perform any administration to maintain views in the foreign data source or their definitions in the schema holder. For more details about planning your temp-table layout, see Creating a temp-table layout plan on page 122.

Employing additional enhancements

The temp-table technique offers even greater programming benefits as it extends both `send-sql-statement` and stored procedures through the result processing techniques previously described in this chapter. For example, by mapping the `PROGRESS_RECID` to the `ROWID` field in temp- tables, you can easily support `KEY` definitions required by the DataSet (commonly referred to as a ProDataSets) to ensure that your data mappings between the foreign data source and the temp-table are accurate. Accurate data mappings are essential for sending data back from the temp-table to the data source. Non-`ROWID` key definitions can also be described with a unique key.

ProDataSets functionality is based on one or more temp-tables that share and extend basic temp-table functionality. For more information about ROWID field, see ROWID Support on page 131. For in depth discussion of temp-tables and more information about ProDataSets, see *OpenEdge Development: ProDataSets*.

The following table highlights additional language elements you can use with the stored procedure and the `send-sql-statement` language to use `ROWID`.

Table 22: Returning result sets and loading the data into temp-tables

ABL	Description
`RUN STORED-PROCEDURE` statement	Executes the stored procedure or send-sql-statement and tells OpenEdge that the stored procedure has ended.

ABL	Description
LOAD-RESULT-INTO phrase	Allows data from a result set that is returned for a foreign data source either through a stored procedure or a send-sql- statement to be put into one or more temp-tables. Static, unprepared dynamic, and prepared dynamic temp-tables are supported. Pass as many temp-tables as result set instances are returned by SQL statements in your Send-SQL or RUN STORED-PROC definition. **Note:** When using SQL statement(s) through a send-sql-statement or stored procedure to load result sets into temp-tables, RUN STORED-PROC carries an implicit RUN CLOSE-PROC statement. (The stored procedure's output parameters are available after the RUN STORED-PROC executes and closes the procedure.)
PROC-STATUS phrase	Reads the return value (optional).

Note: To optimize enhanced stored procedure calls, it is recommended that you use firehose cursor functionality. Use of firehose cursors provided through the connection pooling functionality supports the fastest way to read and return a result set. For details about connection pooling and firehose cursors, see Connection pooling on page 139.

Creating a temp-table layout plan

When using *static* or *prepared dynamic temp-tables*, you must define the temp-table layout in your application program to accommodate specific result sets **before** you attempt to populate the temp-tables with data. If a SQL statement retrieves more than one results set, you must define multiple temp-tables to be able to retrieve all the data. Therefore, the success of this approach depends to a large extent on your:

- Understanding of the specific data your foreign data source is providing you through a given stored procedure
- Ability to correctly define temp-tables

The following types of temp-tables can support results sets:

- **Static** — A temp-table whose schema is defined at compile time.
- **Dynamic** — A temp-table whose schema is defined at run time. There are two types of dynamic temp-tables: dynamic-prepared and dynamic-unprepared.

Keep in mind that you can pass handles of temp-tables that contain a mixed array. A *mixed array* is one in which some of the temp-table handle elements can be static while others can be dynamic. Also, note that a stored procedure supports the use of an INT64 data type in static and dynamic temp tables when the LOAD-RESULT-INTO phrase processes the procedure's result set on the RUN-STORED-PROC statement.

Using a temp-table handle with an unprepared dynamic temp-table

When a temp-table handle points to an unprepared dynamic temp-table, the MS SQL Server DataServer defines the temp-table schema in the form of the result sets record structure which is passed back to the DataServer from the foreign data source. The data types defined for the temp-table schema are determined based on the default data type mapping that exists between the SQL data type and its equivalent OpenEdge default data type. Once the temp-table schema is dynamically established by the DataServer, the result set begins to populate it.

Recognize that there is the possibility of a small performance price to be paid when you build dynamic temp-tables. However, considering the database independence that this technique affords over building static temp-tables, you might consider the price of dynamically built temp-tables to be a small, reasonable one.

The following table identifies the temp-table options for which you can plan and the requirements you must fulfill for each option.

Table 23: Options to plan the temp-table layout for result sets

To return a result set to this type of temp-table . . .	Then the layout definition is . . .
Static Dynamic-prepared state	Defined by you; you must base the layout on the expected fields to be returned and each of these fields' data types so that the first field defined in the temp-table corresponds to the first column of the result set. This column matching and data type matching must be repeated successfully for each temp-table and its corresponding result set.
Dynamic - unprepared state	Not defined by you; the schema of the temp-table is based on the result-set schema and a mapping of default OpenEdge data types for each SQL type. For more details, see Details about a dynamic temp-table in an unprepared state on page 123.

Note: Once the data is loaded into the temp-table, any updates made to the records in the temp-table are not propagated back to the foreign database. Result sets are available through temp-tables for the purpose of obtaining a snapshot of the data. For example, you can use this technique to populate a browser from a temp-table. You must re-read the record using the proper lock mechanism to actually update the record.

Details about a dynamic temp-table in an unprepared state

A dynamic temp-table is considered to be in an unprepared state after the first definitional method is called until the temp-table is prepared. If a clear dynamic temp-table handle is passed, the DataServer populates the temp-table schema based on the result-set schema and prepares the temp-table. A *clear dynamic temp-table* is a table that is in an unprepared state where definitional methods have not yet been called. The DataServer then executes the temp-table handle: `ADD-NEW-FIELD` (field name, data type) internally for each one of the columns of the result set. If a column in the result set from the foreign schema does not have a name (for example, an array element field or a SQL-derived field), the DataServer assigns a name to it based on the column position on the row.

For example, if you run the following statement, then the temp-table contains these columns:

- Column 1
- Cust_num

- Name
- Column 4

```
/* */
SELECT "mytag," cust_num, name, (salary * 0.10) FROM anytablename
```

The data type associated with each column follows the mapping rules that exist between OpenEdge and the foreign data source's data types. For more information about data types and default mapping, see Initial Programming Considerations on page 41.

Note: Since a stored procedure can return multiple result sets, the DataServer prepares the temp-table as "`result`*n*" where *n* is the result-set sequence number that corresponds to its element position in the temp table handle array, starting with 1. Therefore, if there are 3 result sets and 3 clear dynamic temp-tables are passed, the temp-tables are called `result1`, `result2`, and `result3`.

Note the following error conditions as they specifically apply to a dynamic temp-table:

- If an error occurs during the schema population or during the prepare of a temp-table, the DataServer raises an error condition.
- The dynamic temp-table must be either already prepared or clear with no defined fields in it. If fields have already been added to a dynamic temp-table before the `RUN STORED-PROC` statement is executed and the temp-table is not prepared, the DataSever raises an error condition due to the invalid state of the temp-table.

Details about a dynamic temp-table in a prepared state

The following example shows multiple dynamic temp-tables in a prepared state. The temp tables, `tt1` and `tt2`, are prepared in the code just before the `RUN STORED-PROC` statement is called and the temp-tables are passed. The temp-table definition is considered prepared because of the explicit inclusion of each field name and associated data type.

Multiple temp-tables in a prepared state

```
/* example - Multiple dynamic temp-tables in a prepared state*/

DEFINE VARIABLE tt1      AS HANDLE NO-UNDO.
DEFINE VARIABLE tt2      AS HANDLE NO-UNDO.
DEFINE VARIABLE tt-array AS HANDLE NO-UNDO EXTENT 2.

CREATE TEMP-TABLE tt1.
tt1:ADD-NEW-FIELD("custnum", "integer").
tt1:ADD-NEW-FIELD("name", "character").
/*no more fields will be added */
tt1:TEMP-TABLE-PREPARE ("ordx1").

CREATE TEMP-TABLE tt2.
tt2:ADD-NEW-FIELD("ordernum", "integer").
tt2:ADD-NEW-FIELD("orderdate", "date").
/*no more fields will be added */
tt2:TEMP-TABLE-PREPARE ("ordx2").

ASSIGN
  tt-array[1] = tt1
  tt-array[2] = tt2.

RUN STORED-PROC myproc LOAD-RESULT-INTO tt-array (INPUT 10, INPUT "order").
```

In the dynamic temp-table example above, note that the temp-table definitions must match the result set that the `send-sql-statement` returns. For example, the number of fields and the data type of each field in the result must have a corresponding field defined in the temp-table that will receive the result set. No mapping exists between the result set and the temp-table. Therefore, the first field defined in the temp-table corresponds to the first column of the result set and so forth.

Additional temp-table examples

This section presents more examples that show various techniques to code temp-tables.

Temp-table code technique

The following example shows the initial explanation that the stored procedure `myProc` is comprised of two fields—one is an integer and the other is a character, and a result set.

```
SELECT "myProc," cust_num, cust_name FROM anytablename
```

Basic execution of a stored procedure using LOAD-RESULT-INTO functionality

The following example shows the basics of executing a call to a stored procedure using the `LOAD-RESULT-INTO` functionality. Note that in this example, `PROC-STATUS` function is not specified. Therefore, there is no possibility of a return value.

```
/* Calling a stored procedure, using the LOAD-RESULT-INTO phrase*/

DEFINE VARIABLE res    AS INTEGER NO-UNDO.
DEFINE VARIABLE ttHndl AS HANDLE  NO-UNDO.

DEFINE TEMP-TABLE ttl
  FIELD f1 AS INTEGER
  FIELD f2 AS CHARACTER.

ttHndl = TEMP-TABLE tt:HANDLE.
```

```
RUN STORED-PROC myProc LOAD-RESULT-INTO ttHndl (INPUT 1, OUTPUT res).
```

Using an existing dynamic temp-table without the TEMP-TABLE-PREPARE ( ) method

The following example shows the basics of using an existing dynamic temp-table without the TEMP-TABLE-PREPARE ( ) method. In this instance, send-sql-statement is used rather than a predefined stored proc. In contrast, the third example code that appears later in this section shows the same approach, but explicitly defines the existing dynamic temp-table with the TEMP-TABLE-PREPARE( ) method.

```
/* Calling a stored procedure, using an existing temp-table without temp table prepare
*/

DEFINE VARIABLE ttHndl AS HANDLE NO-UNDO.

CREATE TEMP-TABLE ttHndl.

RUN STORED-PROC send-sql-statement LOAD-RESULT-INTO ttHndl
  ("select * from customer").
```

Using an existing dynamic temp-table with the TEMP-TABLE-PREPARE ( ) method

the following example shows the basics of using an existing dynamic temp-table with the TEMP-TABLE-PREPARE() method.

```
/* Calling a stored procedure, using an existing temp-table with temp table prepare*/

DEFINE VARIABLE ttHndl AS HANDLE NO-UNDO.

CREATE TEMP-TABLE tt1Hndl.
ttHndl:ADD-NEW-FIELD("custNum", "integer").
ttHndl:ADD-NEW-FIELD("name", "character").
ttHndl:TEMP-TABLE-PREPARE("ordX").

RUN STORED-PROC send-sql-statement LOAD-RESULT-INTO ttHndl
  ("select custNum,name from myDB.customer").
```

Note the following points about this example:

- As a prerequisite for creating the code shown, the developer would need to define the schema for the table.
- Once the temp-table schema begins preparation from the clear state, the temp-table must be defined to the exact specifications for the result sets as generated by the RUN STORED-PROC statement or send-sql-statement. Otherwise, the RUN STORED-PROC will end in failure. Also, note the TEMP-TABLE-PREPARE must be called at the completion of the temp-table definition associated with the stored procedure results.

Calling a stored procedure and using the EXTENT phrase

The following example shows passing the handles of empty dynamic temp-tables to enable a MS SQL Server DataServer to create schema based on the result-set schema.

```
/ *Calling a stored procedure, and using the EXTENT phrase to pass handles of empty
temp-tables to enable the MS SQL DataServer to create schema holder definitions based
on each temp-table's result-set schema. */

DEFINE VARIABLE ttCustomer AS HANDLE NO-UNDO.
DEFINE VARIABLE ttHandle   AS HANDLE NO-UNDO EXTENT 2.
DEFINE VARIABLE ttOrder    AS HANDLE NO-UNDO.

CREATE TEMP-TABLE ttCustomer.
CREATE TEMP-TABLE ttOrder.
ttHand1e[1] = ttCustomer.
```

```
ttHand1e[2] = ttOrder.

RUN STORED-PROC send-sql-statement LOAD-RESULT-INTO ttHandle
  ("Select * from customer; select * from order").
```

Calling an enhanced stored procedure with a single temp table and the send-sql-statement

The following example shows how to use the enhanced stored procedure syntax with a single static temp-table and the `send-sql-statement`.

```
/* Calling an enhance stored procedure with a single temp table and the send-sql-statement
 */

DEFINE VARIABLE ttHandle AS HANDLE NO-UNDO.

DEFINE TEMP-TABLE ttHandle
  FIELD f1 AS INTEGER
  FIELD f2 AS CHARACTER.

ttHandle = TEMP-TABLE ttHandle:HANDLE.

RUN STORED-PROC send-sql-statement LOAD-RESULT-INTO ttHandle
  ("Select cust_num,name from customer").
```

Use of a PROC-STATUS phrase

The following example shows the use of a `PROC-STATUS` phrase. The `PROC-STATUS` phrase must be defined as part of the `RUN-STORED-PROC` statement because of the implicit `CLOSE STORED-PROC` that is associated with the `LOAD-RESULT-INTO` phrase.

```
/* Sixth example - Example of the implicit close stored-proc and use of LOAD-RESULT-INTO
 */

DEFINE VARIABLE iStat    AS INTEGER NO-UNDO.
DEFINE VARIABLE ttHandle AS HANDLE  NO-UNDO.

CREATE TEMP-TABLE ttHandle.

RUN STORED-PROCEDURE pcust LOAD-RESULT-INTO ttHandle
  iStat = PROC-STATUS (20, output 0, output 0).

DISPLAY iStat.
```

Note that the `PROC-STATUS` phrase does not need a `PROC-HANDLE` phrase because it is retrieved using the `RUN STORED-PROC` statement and not after this statement's execution as it typically is used.

Handling errors

The `RUN STORED-PROC` statement supports the `NO-ERROR` option.

Trapping errors within a procedure using NO-ERROR option (simple example)

The following shows a simple example how to trap errors within a procedure using the NO-ERROR option.

```
DEFINE VARIABLE ix AS INTEGER NO-UNDO.
RUN STORED-PROC send-sql-statement NO-ERROR
  ("select count (*) from customer where name between a and 'z' ").
/*Select statement syntax is incorrect due to type-mismatch*/
IF ERROR-STATUS:ERROR THEN DO:
  DO ix = 1 TO ERROR-STATUS:NUM-MESSAGES:
    MESSAGE "error" ERROR-STATUS:GET-NUMBER(ix)
    ERROR-STATUS:GET-MESSAGE(ix).
  END.
  RETURN.
END.
CLOSE STORED-PROC send-sql-statement.
```

Trapping errors within a procedure using NO-ERROR option

The following example assumes that you created the sp_img table and defined the sp_img_out stored procedure with two data types, integer and image, in the MS SQL Server as shown in the following code samples:

```
create table sp_img
(  p_int int primary key,
  p_image image
)Go
```

```
create procedure sp_img_out
@ip_int int,
@op_image image out
as
begin
insert into sp_img(p_int,p_image)
values(@ip_int, @op_image)
select p_int,p_image from sp_img where p_int = @ip_int;
end
Go
```

In the ABL code example that follows, the first (input) parameter literal value "3" is mapped to an INTEGER data type and the second (output) parameter "picture" is mapped to an IMAGE data type in MS SQL Server. As the ODBC drivers and the stored procedures in MS SQL Server do not support the legacy data type IMAGE as an OUTPUT parameter, on executing the sp_img_out stored procedure with the NO-ERROR option, a run-time error results, and is handled by the ERROR-STATUS:ERROR check in the IF block.

Note: IMAGE is not an unsupported data type. It is unsupported only as an OUTPUT parameter.

```
DEF VAR intvar AS INTEGER.
DEF VAR ix AS INTEGER.
DEF VAR invalid_param AS LOGICAL INITIAL FALSE.
DEF VAR picture as MEMPTR.
RUN STORED-PROC sp_img_out intvar = PROC-HANDLE NO-ERROR (PARAM ip_int = 3,
                                    OUTPUT PARAM op_image = picture ).
IF ERROR-STATUS:ERROR AND ERROR-STATUS:NUM-MESSAGES > 0 THEN DO:
  DO ix = 1 TO ERROR-STATUS:NUM-MESSAGES:
    IF ERROR-STATUS:GET-NUMBER(ix) = 16684 THEN DO:
      MESSAGE ERROR-STATUS:GET-NUMBER(ix) ERROR-STATUS:GET-MESSAGE(ix).
      invalid_param = TRUE.
    END.
  END.
  IF invalid_param = FALSE THEN DO:
    STOP.
  END.
END.
ELSE DO:
  CLOSE STORED-PROCEDURE sp_img_out WHERE PROC-HANDLE = intvar.
  COPY-LOB FROM op_image TO FILE "C:\myicons\icon.ico".
END.
```

The DataServer internally closes the stored procedure handle when the stored procedure fails to execute. You must ensure that the error handling code block immediately follows the RUN STORED-PROCEDURE call so that errors captured by NO-ERROR are not lost by the execution of another statement. Including the CLOSE STORED-PROCEDURE statement in the error handling block causes a run-time error because closure is implicit. Only if the procedure is run successfully will a procedure handle be established, and only then do you need to execute CLOSE STORED-PROCEDURE on the open handle.

Trapping errors within a procedure using CATCH option

The error demonstrated in the previous example can also be handled using a CATCH end block in the procedure block that performs the RUN STORED-PROC statement. The following code shows how to trap errors within a procedure using a CATCH end block.

```
DEF VAR intvar AS INTEGER.
RUN STORED-PROC sp_img_out intvar = PROC-HANDLE (3, OUTPUT ?).
CLOSE STORED-PROCEDURE sp_img_out WHERE PROC-HANDLE = intvar. /* This statement will
not be executed on error */

CATCH eSysError AS Progress.Lang.ProError:

  MESSAGE eSysError:GetMessageNum(1) SKIP
    eSysError:GetMessage(1) SKIP
    eSysError:GetMessage(2) SKIP
    eSysError:GetMessage(3) VIEW-AS ALERT-BOX BUTTONS OK.
    /* Handler code for SysError condition. */
END.
```

For more information on the NO-ERROR option, CATCH statement or CATCH end block, in OpenEdge ABL, see *OpenEdge Development: ABL Reference*.

Trapping errors when using Load-Result-Into

By properly positioning the NO-ERROR in the RUN statement, error information can also be retrieved from attempted SQL execution at the data source when using LOAD-RESULT-INTO.

Trapping errors using LOAD-RESULT-INTO

The following example shows how errors are trapped after LOAD-RESULT-INTO stored procedure execution.

```
DEFINE VARIABLE hTables AS HANDLE EXTENT 2 NO-UNDO.

DEFINE TEMP-TABLE ttcust
FIELD cust-num AS INTEGER
FIELD NAME AS CHARACTER.

DEFINE TEMP-TABLE ttord
FIELD ord-num AS INTEGER
FIELD ord-date AS DATE.

hTables[1] = TEMP-TABLE ttcust:HANDLE.
hTables[2] = TEMP-TABLE ttord:HANDLE.

RUN STORED-PROC send-sql-statement LOAD-RESULT-INTO hTables NO-ERROR
  ("Select cust_num, name from customer; Select order_num, order_date from    order_;
Select * from order_line")./*Loading three result-sets in two
                                    temp-tables raises an error*/

IF ERROR-STATUS:ERROR THEN
  MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
ELSE DO:
  MESSAGE "SQL call was successful." VIEW-AS ALERT-BOX.
  FOR EACH ttcust:
    DISP ttcust.cust-num ttcust.name.
  END.
  FOR EACH ttord:
    DISP ttord.ord-num ttord.ord-date.
  END.
  /* CLOSE STORED-PROC not required w/LOAD-RESULT-INTO */
END.
```

Error messages related to using the Load-Result-Into phrase

Note the following situations in which error conditions related to the use of LOAD-RESULT-INTO are reported:

- The compiler issues an error if you try to specify PROC-HANDLE and LOAD-RESULT-INTO on the same Run STORED-PROC statement.
- The compiler issues an error if you use PROC-STATUS without specifying the LOAD-RESULT-INTO function.
- The compiler issues error messages when the number of handles is not equal to the number of result-sets specified in the statement. For example:
  - **Error "Only 2 handles supplied but there are at least 3 result-sets (12209)"** — This error message appears when two handles and three result-sets are specified in a statement.
  - **Error "Received an invalid handle (element 3) when processing stored procedure's result-set (12212)"** — This error message appears when four handles and three result-sets are specified in a statement.

- If you try to execute the `CLOSE STORED-PROC` statement after the `Run STORED-PROC` statementfor which you have defined the `LOAD-RESULT-INTO` function, the following error message appears:
  - **Error "No active stored procedures. (2101)"** — This error message appears when no handle is specified in the statement.
  - **Error "No active procedure for specified proc-handle. (2475)"** — This error message appears when a handle is specified in the statement.

  The system generates either of the previous error messages as it assumes a second `CLOSE STORED-PROC` statement has been incorrectly executed in the current implementation.

For more information about the enhanced stored procedure functions and statements, see Table 22: Returning result sets and loading the data into temp-tables on page 121.

ROWID Support

This section presents details about ROWID function.

- Understanding the ROWID implementation on page 131
- Using ROWID with RUN STORED-PROCEDURE and LOAD-RESULT-INTO on page 132
- Additional ProDataSet support on page 135

Understanding the ROWID implementation

The OpenEdge `ROWID` function returns the unique internal identifier of the database record currently associated with the record buffer you name. This internal identifier has the data type `ROWID`, which is supported for OpenEdge and all other DataSever databases.

ROWID characteristics

The `ROWID` value is not stored directly in the MS SQL Server data source, but it is represented by a unique index key in your database. If you migrated your database to a MS SQL Server and elected to use the **Create RECID Field** option during the migration, a unique 8-byte integer column named `PROGRESS_RECID` and a corresponding seed column named `PROGRESS_RECID_IDENT` will have been automatically generated in your database tables along with a corresponding Index containing the `PROGRESS_RECID` value as its key.

Note: While the `PROGRESS_RECID_IDENT` column is not used to resolve the ROWID value, it is important to recognize the presence of this column for purposes of the following discussion. (This point is especially significant because its physical presence is unknown to the schema holder definition that represents this table to the client.)

Databases that are only pulled from the native environment or are migrated without the **Create RECID Field** option must choose a unique index key from each table to represent the `PROGRESS_RECID` to support the `RECID` and `ROWID` functions, and forward and backward scrolling within their ABL sessions. ABL will internally map the selected unique key to the Progress `RECID` and `ROWID` functions.

Note: Reposition functions such as REPOSITION-BACKWARDS and REPOSITION-TO-ROW typically use ROWID to identify records and functions of this type require integer expressions which, beginning with OpenEdge Release 10.1B, can be either INTEGER or INT64.

It is important to note that the unique index key used to derive the Progress ROWID must be a single component index for it to map to the ROWID of a temp-table as described in the following discussion. For more details, see the ROWID function discussion in Initial Programming Considerations on page 41.

Starting in OpenEdge Release 10.1B, the default external representation of ROWID enables the MS SQL Server DataServer to use a 64-bit integer value for ROWID. Also, before discussing ROWID as it relates to the RUN STORED-PROC statement, it is important to understand an important property of the OpenEdge ROWID. For information see ROWID: Standard ABL behavior on page 132.

ROWID: Standard ABL behavior

The ROWID value of a temp-table buffer will be different than the ROWID value of a record BUFFER even if the underlying data is identical. This difference exists because the ROWID function relies on the record buffer name.

Expected ABL behavior—ROWID value of a temp-table buffer

The following example shows the standard, expected ABL behavior.

```
DEFINE VARIABLE rid-1    AS ROWID  NO-UNDO.
DEFINE VARIABLE rid-2    AS ROWID  NO-UNDO.
DEFINE VARIABLE ttHandle AS HANDLE NO-UNDO EXTENT 1.

DEFINE TEMP-TABLE ttCustomer LIKE Sports2000.Customer
  FIELD tRecid       AS INTEGER
  FIELD tRECID_ident AS INTEGER.

ttHandle[1] = TEMP-TABLE ttCustomer:HANDLE.

FIND FIRST Customer WHERE Customer.CustNum = 1 NO-LOCK.
rid-1 = ROWID(Customer).

RUN STORED-PROC send-sql-statement
  LOAD-RESULT-INTO ttHandle ("select * from customer where custnum = 1").

rid-2 = ROWID(ttCustomer).

IF rid-1 NE rid-2 THEN
  MESSAGE "The same record but different ROWID's".
```

The following sections illustrate the differences between the ROWID value of a temp-table buffer and the ROWID value of a record BUFFER so that you can best understand, program for, and leverage the use of the ROWID function with the LOAD-RESULT-INTO clause of the RUN STORED-PROCEDURE command.

Using ROWID with RUN STORED-PROCEDURE and LOAD-RESULT-INTO

The example in the previous section assumes you migrated your database to MS SQL Server using the **Create RECID Field**option, as discussed in ROWID characteristics on page 131. The examples in this section show ROWID as being represented by the 8-byte integer value of the PROGRESS_RECID column as opposed to some other unique single-component index designated in your database to be the PROGRESS_RECID.

Note: If you used a different single-component index to load the ROWID of a temp-table, you would need to map the column accordingly, just as the example maps PROGRESS_RECID.

The RUN STORED-PROC command has no native awareness that the MS SQL Server Database table is being queried for the result set(s) it generates. Therefore, to allow DataServer technology to convert the stored PROGRESS_RECID value into a native OpenEdge ROWID value, the physical name of the target database table needs to be known. To achieve this bond, the temp-table that the stored procedure populates must be associated with an OpenEdge ProDataSet object.

ABL Query filling a ProDataSet temp-table

The following example shows an ABL query filling the temp tables of a ProDataSet. It will be used as the baseline code referenced throughout the remainder of this section.

```
DEFINE VARIABLE phDataSet AS HANDLE NO-UNDO.

DEFINE TEMP-TABLE ttCustomer LIKE Sports2000.Customer
  FIELD tRecid       AS INTEGER
  FIELD tRECID_ident AS INTEGER.

DEFINE DATASET dsCustomer FOR ttCustomer.
DEFINE QUERY qCustomer FOR Customer.

phDataSet=DATASET dsCustomer:HANDLE.
DEFINE DATA-SOURCE srcCustomer FOR QUERY qCustomer.

BUFFER ttCustomer:HANDLE:ATTACH-DATA-SOURCE
  (DATA-SOURCE srcCustomer:HANDLE,?,?,?).
QUERY qCustomer:QUERY-PREPARE ("FOR EACH Customer").
DATASET dsCustomer:FILL().

FOR EACH ttCustomer:
  DISPLAY ttCustomer.name ttCustomer.tRecid.
END.
```

If the table uses computed-column PROGRESS_RECID option, then ttCustomer temp-table definition should be:

```
DEFINE TEMP-TABLE ttCustomer LIKE Sports.Customer
FIELD tRecid AS ROWID /* Must be changed to ROWID type */
FIELD tRECID_ident AS INT64
FIELD tRECID_alt as INT64.
```

Using the LOAD-RESULT-INTO technique to populate the underlying Temp-Table of a ProDataSet

The following example combines code from the examples:

- Expected ABL behavior—ROWID value of a temp-table buffer on page 132
- ABL Query filling a ProDataSet temp-table on page 133

The example applies the results of the `RUN STORED-PROC [LOAD-RESULT-INTO]` technique, rather than an ABL query, to fill the `TEMP-TABLE` associated with a ProDataSet.

```
DEFINE VARIABLE hSendSQL  AS HANDLE NO-UNDO EXTENT 1.
DEFINE VARIABLE phDataSet AS HANDLE NO-UNDO.
DEFINE VARIABLE rid-1     AS ROWID  NO-UNDO.
DEFINE VARIABLE rid-2     AS ROWID  NO-UNDO.

DEFINE TEMP-TABLE ttCustomer LIKE Sports.Customer
  FIELD tRecid       AS ROWID /* Must be changed to ROWID type */
  FIELD tRECID_ident AS INT64.

hSendSQL[1] = TEMP-TABLE ttCust:HANDLE.

DEFINE DATASET dsCustomer FOR ttCustomer.
DEFINE QUERY qCustomer FOR Customer.

phDataSet=DATASET dsCust:HANDLE.

DEFINE DATA-SOURCE srcCustomer FOR QUERY qCust.
BUFFER ttCustomer:HANDLE:ATTACH-DATA-SOURCE
  (DATA-SOURCE srcCustomer:HANDLE,?,?,?).

FIND FIRST Customer WHERE Customer.CustNum = 1 NO_LOCK.
rid-1 = ROWID(Customer).

/* Populate the ttCustomer Temp-Table */
RUN STORED-PROC send-sql-statement
   LOAD-RESULT-INTO hSendSQL ("select * from customer").

FIND FIRST ttCustomer WHERE ttCustomer.CustNum = 1 NO-LOCK.
rid-2 = ttCustomer.tRecid.

If rid-1 NE rid-2 THEN
  MESSAGE "The same record but different ROWID's".
IF rid-1 = rid-2 THEN
  MESSAGE "Congratulations - we have the same ROWID's".

MESSAGE STRING(ttCustomer.tRecid) VIEW-AS ALERT-BOX.
```

Keep the following key points in mind regarding this example:

- The `TEMP-TABLE` field that is mapped to the `PROGRESS_RECID` column should be changed from its standard definition of `INTEGER` to `ROWID`.

  The result column location where `PROGRESS_RECID` is being returned has been named `tRecid` in the temp-table. The `PROGRESS_RECID_IDENT` has been renamed `tRECID_ident`. This renaming occurs because of the following line:

  ```
  FIELD tRecid       AS ROWID /* must be changed to ROWID type */
  FIELD tRECID_ident AS INTEGER.
  ```

If the table uses the computed-column PROGRESS_RECID option, then the ttCustomer temp-table definition should be:

```
DEFINE TEMP-TABLE ttCustomer LIKE Sports.Customer
FIELD tRecid AS ROWID /* Must be changed to ROWID type */
FIELD tRECID_ident AS INT64
FIELD tRECID_alt as INT64.
```

- The TEMP-TABLE must be defined to the ProDataSet. The following line excerpted from the complete example shows this definition:

```
DEFINE DATASET dsCustomer FOR ttCustomer.
```

- The demonstrated technique does not change the default behavior of the ROWID function, but provides a mechanism for the ROWID value to be stored along side its corresponding result rows; therefore, using the ROWID value to access database rows is unconventional with respect to the normal, or more typical, association between ROWID and a database table row. The following code demonstrates this difference.

  Default use of ROWID function on a record buffer, as excerpted:

```
FIND FIRST Customer WHERE Customer.CustNum = 1 NO-LOCK.
rid-1 = ROWID(Customer).
```

  In contrast, the following code excerpt demonstrates an alternative use of the ROWID value with a temp-table:

```
FIND FIRST Customer WHERE Customer.CustNum = 1 NO-LOCK.
rid-2 = ttCustomer.tRecid.
```

Additional ProDataSet support

As previously demonstrated in the example, Using the LOAD-RESULT-INTO technique to populate the underlying Temp-Table of a ProDataSet on page 134, the LOAD-RESULT-INTO functionality provides a very fast and efficient way to populate the temp-table(s) of a ProDataSet Object. The example that follows is a two-part example that builds on the techniques demonstrated in the previous example.

Use the LOAD-RESULT-INTO technique with BEFORE-FILL method to fill the temp-table(s) of a ProDataSet

The example that follows shows the ProDataSet object BEFORE-FILL procedure can be modified to provide a single ProDataSet data access object that can be used against native OpenEdge or against MS SQL Server and other DataServer data sources.

```
DEFINE VARIABLE hSendSQL  AS HANDLE NO-UNDO EXTENT 1.
DEFINE VARIABLE phDataSet AS HANDLE NO-UNDO.
DEFINE VARIABLE rid-1     AS ROWID  NO-UNDO.
DEFINE VARIABLE rdi-2     AS ROWID  NO-UNDO.

DEFINE TEMP-TABLE ttCustomer LIKE Sports.Customer
  FIELD tRecid       AS ROWID
  FIELD tRECID_ident AS INTEGER.

hSendSQL[1] = TEMP-TABLE ttCustomer:HANDLE.

DEFINE DATASET dsCustomer FOR ttCustomer.
DEFINE QUERY qCustomer FOR Customer.

phDataSet = DATASET dsCustomer:HANDLE.
phDataSet:SET-CALLBACK-PROCEDURE
  ("BEFORE-FILL", "preDataSetFill", THIS-PROCEDURE).

DEFINE DATA-SOURCE srcCustomer FOR QUERY qCustomer.
BUFFER ttCustomer:HANDLE:ATTACH-DATA-SOURCE
  (DATA-SOURCE srcCustomer:HANDLE,?,?,?).
DATASET dsCustomer:FILL().

FIND FIRST Customer WHERE Customer.CustNum = 1 NO-LOCK.
rid-1 = ROWID(Customer).

FIND FIRST ttCustomer WHERE ttCustomer.CustNum = 1 NO-LOCK.
rid-2 = ttCustomer.tRecid.

IF rid-1 NE rid-2 THEN
  MESSAGE "The same record but different ROWID's".
IF rid-1 EQ rid-2 THEN
  MESSAGE "Congratulations - we have the same ROWID's".

MESSAGE STRING(ttCustomer.tRecid) VIEW-AS ALERT-BOX.

PROCEDURE preDataSetFill:
  DEFINE INPUT PARAMETER DATASET FOR dsCustomer.
  DEFINE VARIABLE handle1  AS INTEGER NO-UNDO.
  DEFINE VARIABLE hSendSQL AS HANDLE  NO-UNDO EXTENT 1.

  hSendSQL[1] = TEMP-TABLE ttCustomer:HANDLE.

  IF DBTYPE(NUM-DBS) NE "PROGRESS" THEN DO:
    DISPLAY "DataServer FILL using SEND-SQL".
    BUFFER ttCustomer:HANDLE:FILL-MODE = "NO-FILL".
    RUN STORED-PROC send-sql-statement LOAD-RESULT-INTO hSendSQL
      ("select * from customer").
  END.
  ELSE DO: /* normal OpenEddge FILL procedure */
    DISPLAY "Native Progress FILL".
    QUERY qCustomer:QUERY-PREPARE("FOR EACH Customer").
  END.
END PROCEDURE. /* preDataSetFill */
```

4

Additional Features to Enhance DataServer Performance

For configuration purposes, the server component of the DataServer can be a distributed “client/server” or non-distributed “self-service”. Progress recommends running a “self-service” DataServer if there is no specific reason to distribute the components of the DataServer. This avoids Progress networking protocol and the extra network hop between the client and server component systems on different machines. Network bandwidth can be a limiting factor when database access includes large data transfers.

A DataServer for MS SQL Server session can be established from either the OpenEdge AppServer or the Progress AppServer products. DataServers distinguishes between the OpenEdge AppServers and Progress AppServer based on the use of threads versus processes when establishing the agent for the DataServer session. Both AppServer environments require access to a schema holder database from which a logical database can be connected.

The Progress AppServer environment does not provide a broker facility from which the server component of the DataServer can be spawned. It only runs DataServers in “self-service” mode. Self-service mode is the most efficient way to establish a DataServer session from an AppServer and it is the preferred configuration for DataServer access in terms of performance.

The Progress AppServer runs all the DataServer sessions from agents in a single process space and avoids the context switching and communication between processes of the traditional AppServer and the Progress AppServer can provide a significant reduction in overhead by multi-threading the DataServer sessions. Progress recommends use of the Progress AppServer over the OpenEdge AppServer for better DataServer and OpenEdge performance.

Progress also recommends using a shared schema holder database between agent processes of the traditional AppServer or agent threads of the Progress AppServer. By sharing a logical database from a common, read-only, schema holder between AppServer agents, a DataServer session gains access to the same foreign data source for many agents using a single common schema.

Note: This assumes all clients connected through the AppServer want access to the same version of the logical database. Otherwise, multiple logical databases may be required which can also be shared from one or more shared schema holder databases. Progress recommends a schema holder be connected “self-service” in read-only mode using shared memory for better performance since it will not have the overhead of Progress networking.

There are a variety of additional DataServer-related features you can employ to enhance your DataServer's performance.

For details, see the following topics:

- Enhancements overview
- Connection pooling
- OpenEdge query types
- Query tuning
- Caching records
- Join by SQL DB
- Writing queries for performance
- Modifying the run-time schema check
- Replacing FIND FIRST for performance
- Replacing FOR FIRST for performance

Enhancements overview

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 a MS SQL Server database. For example, you might write a query that gives you consistent results across databases or one that takes advantage of MS 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 Query tuning with connection and startup parameters on page 204.

Connection pooling

The DataServer for Microsoft SQL Server is enhanced with the ability to form a connection pool. A *connection pool* is a set of database connections that are available for an application to use and reuse without having to be reestablished. Connection pooling significantly improves the cursor management associated with no-lock queries, particularly multi-table joins. Creating and tearing down connections can be resource intensive. Using a pooled connection to keep existing connections alive results in significant performance gains because the DataServer avoids the overhead of making a connection for each request. ABL applications that open multiple no-lock queries and handle their results simultaneously experience the best cumulative performance gains from connection pooling.

Main components

Connection pooling for the DataServer for Microsoft SQL Server is a combination of ODBC connection pooling and DataServer connection management. These connection components can be used as follows:

- Individually, ODBC connection pooling or DataServer connection management provides the foundation required for firehose cursors, but enabling both provides the best performance. For more information on firehose cursors, see Firehose, Firehose block, and Fast Forward-Only Cursors on page 141.

  Without a connection pool, firehose cursors would block an application until a full result set is retrieved. Because of this, when connection pooling is disabled, firehose cursors are also disabled. By maintaining multiple connections and one cursor per connection, read-only requests only block the connection on which they retrieve results, freeing ABL applications to continue processing data on the other connections.

- ODBC connection pooling and DataServer connection management provide the highest performance improvements when enabled together, but they can also be enabled independent of one another.

  For installations where the number of ODBC connections is limited, you might decide to enable only the DataServer connection management. For deployments in which memory constraints are a concern, enabling only the ODBC connection pool provides on-demand connections, with optimized resource efficiency. If both an ODBC connection pool and managed connections coexist in the same DataServer process, the managed connections will come from the ODBC connection pool. The managed connections are distinguished from the connection pool connections in that they are never released back to the pool for the life of an ABL session.

Considerations when using ODBC connection pooling and DataServer connection management

When both ODBC connection pooling and DataServer connection management are enabled, set the number of managed connections to a value that covers the required number of connections for a typical application. Reuse of the managed connections takes precedence over ODBC connection pool connections. The ODBC connection pool is then an overflow handler for connections required beyond the capacity of the managed connections. This configuration ensures that there is little risk of having a negative impact on performance by downgrading to a Fast Forward-Only server-side cursor in the mainline transaction-oriented connection. For more information on monitoring the performance of your connection pool, see Monitoring cursor and connection use on page 143.

ODBC Connection Pooling

ODBC connection pooling is managed by the ODBC driver manager, external to the DataServer. The size of the pool grows and contracts dynamically based on demand and time-out intervals. The upper limit to the pool size is limited only by memory and other server resource constraints. The DataServer retrieves connections out of the externally managed pool. ODBC connection pooling is maintained with the `CONNECTION_POOLING` option to the `-Dsrv` connection parameter. ODBC connection pooling is enabled by default.

Disable ODBC connection pooling with the following syntax:

Syntax

```
-Dsrv CONNECTION_POOLING,0
```

Note: To take advantage of ODBC connection pooling in the DataServer, you must rebuild your schema holder. However, if your schema holder was built before OpenEdge Version 10.0B, contact Technical Support for information about how to proceed.

DataServer connection management

When an ABL session is established, the DataServer for Microsoft SQL Server opens the number of read-only connections specified in the `PRGRS_CACHE_CONN` value. The DataServer maintains these connections for the duration of an ABL session, reusing them for subsequent queries.

Enabling connection management

DataServer connection management is enabled and tuned with the `PRGRS_CACHE_CONN` option to the `-Dsrv` connection parameter. Connection management is enabled by default, and the default number of managed connections is 5. Change the number of connections using the following syntax:

Syntax

```
-Dsrv PRGRS_CACHE_CONN,n
```

Where *n* is the number of managed connections.

Disabling connection management

DataServer connection management is disabled by setting the number of managed connections to zero, as shown:

```
-Dsrv PRGRS_CACHE_CONN,0
```

Connections exclusively for stored procedures

When DataServer connection management is enabled, connections for stored procedures are allocated from the managed connections. If your stored procedures modify the connection environment or attributes, allocate a set of managed connections exclusively for stored procedures to prevent the modified connections from being reallocated for general use.

Allocate managed connections exclusively for stored procedures with the following syntax:

Syntax

```
-Dsrv PRGRS_SP_CACHE_CONN,n
```

Where *n* is the number of managed connections allocated to stored procedures. Set *n* to a value large enough to handle the largest number of stored procedures you can run simultaneously. If you exceed *n*, the connection will be allocated from the ODBC connection pool, if enabled, making the modified connection available for general reuse.

Note: This switch is off by default, and is only needed for applications that modify the connection environment or attributes within stored procedures.

Impact on commits in stored procedures

Running a stored procedure in a separate connection changes the timing of changes being committed to the data source. When a stored procedure is run in a separate connection, changes not explicitly committed during the execution of the stored procedure are committed at the time the procedure handle is closed and the connection is recycled.

Firehose, Firehose block, and Fast Forward-Only Cursors

Firehose cursors deliver a streamlined, unmanaged, client-side cursor-processing mechanism for handling result sets from a DataServer query. When connection pooling is enabled, firehose cursors are the default mechanism for handling read-only results. If a firehose cursor is denied to an application, the DataServer first attempts to substitute a Fast Forward-Only (FFO) server-side cursor with Auto-Fetch and Auto-Close attributes in its place. If the query cannot be handled by a FFO cursor, the cursor is further downgraded.

A firehose block cursor is the default cursor processing mechanism to process the `NO-LOCK` queries for the DataServer for the MS SQL Server. It provides optimum performance for `NO-LOCK` queries and generally, surpass the performance benefits of the server-side block cursors because a firehose block cursor does bulk processing at the client-side.

Firehose cursors

Firehose cursors are identified in Microsoft SQL Server as the *default result set*. A default result set is generated when the statement attributes of a cursor are left unchanged from their standard MS SQL defaults. The default result set allows rows from a query result to be pulled without locks in forward-only sequence into a client-side cache. The default result set is referred to as a firehose cursor because it can "flood" the client with results. It is unencumbered by the cursor management necessary with server-side cursors.

The following DataServer operations are eligible for the firehose cursor implementation:

- All NO-LOCK queries.
- All SHARE-LOCK queries with transaction isolation level set to read-uncommitted.
- Internal no-lock queries that populate the key cache for transaction-oriented operations.
- All stored procedure result sets.
- All send-sql-statement result sets.
- Queries written with the `QUERY-TUNING(SEPARATE-CONNECTION)` keyword. When connection pooling is enabled, the `QUERY-TUNING(SEPARATE-CONNECTION)` is redundant.

Note: Prepared statements associated with firehose cursors are cached on a statement cache that is associated with the managed connection. Statement reuse may decrease based on the recycling of managed connections. To completely disable the prepared statement cache, use the following connection switch: "`-Dsrv PRGRS_PREPCACHE,0`". For information about monitoring the statement cache reuse, see Monitoring cursor and connection use on page 143.

Firehose block cursors

A firehose block cursor fetches blocks of records from the cached result-set at the client-side. When, at the client side, the DataServer processes blocks of records, as opposed to individual records, it implementes the firehose block cursors for faster cursor-processing. For information on Firehose cursors and block cursors, see Firehose cursors on page 141 and Block cursors on page 88.

Firehose block cursor is the default mechanism to process `NO-LOCK` queries on the cilent-side.

To enable the firehose block cursor, you must enable the firehose cursors and the block cursors. You enable the firehose cursors by enabling the `CONNECTION_POOLING` option and/or by having available connections when either the `PRGRS_CACHE_CONN` option is greater than zero or the `QUERY-TUNING(SEPARATE-CONNECTION)` keyword is added to the query. And, you enable the block cursors by enabling the `PRGRS_BLOCK_CURS` option and ensuring that there are adequate memory resources to allocate record blocks for the result sets.

Note:

`CONNECTION_POOLING` is enabled by default and the `PRGRS_CACHE_CONN` option is non-zero by default. For more information on `CONNECTION_POOLING` and `PRGRS_CACHE_CONN` option, see Connection pooling on page 139 and DataServer connection management on page 140.

The `PRGRS_BLOCK_CURS` option is also enabled by default. Because of the default enabling of a combination of options, the DataServer session is generally configured to take advantage of firehose block cursors whenever possible. For more information on the `PRGRS_BLOCK_CURS` dataserver option, see DataServer options on page 337.

The following DataServer operations are eligible for the firehose block cursor implementation:

- All NO-LOCK queries.
- All SHARE-LOCK queries with transaction isolation level set to read-uncommitted.
- Internal no-lock queries that populate the key cache for transaction-oriented operations.

Note: Prepared statements associated with firehose block cursors are cached on a statement cache that is associated with the managed connection. Statement reuse may decrease based on the recycling of managed connections. To completely disable the prepared statement cache, use the following connection switch: "`-Dsrv PRGRS_PREPCACHE,0`". For information about monitoring the statement cache reuse, see Monitoring cursor and connection use on page 143.

Firehose and Firehose block exclusions

Certain statements and fields automatically exclude a query from consideration for a firehose or firehose block cursor. For example:

- `FOR FIRST` and `FOR LAST` statements. These statements return at most one row, so they are optimized with the Transact-SQL keyword `TOP`, which renders them ineligible for a firehose cursor.

When the DataServer is unable to implement Firehose or Firehose block cursors, the server-side Fast Forward-only (FFO) cursors are implemented. For information about FFO cursors, see Fast Forward-Only cursors on page 143.

Fast Forward-Only cursors

When a firehose cursor cannot be used, the DataServer attempts to use a Fast Forward-Only (FFO) cursor with Auto-Fetch and Auto-Close attributes. FFO cursors are the server-side equivalent of firehose cursors. FFO cursors have special optimization characteristics that distinguish them from other server-side cursors. They require a minimum of server-side resources and are capable of minimizing round trips to the server. FFO cursors are an extension to the ODBC specification and are unique to ODBC drivers that conform to Microsoft SQL Server driver requirements. The Auto-Fetch attribute directs the server to return the initial block of results in the same network message that provided the SQL request to be executed by the server. The Auto-close attribute directs the server to automatically close the cursor on the same round trip in which the last query result is received by the client.

Note: Result sets that include `text` or `image` columns cause an implicit conversion from an FFO to a dynamic cursor type. These are columns that translate through ODBC to SQL `LONGVARCHAR` and `LONGVARBINARY` data types.

Monitoring cursor and connection use

Monitor the use of your managed connections to tune the number of connections you allocate. If you regularly exceed your allocation, consider increasing the number of managed connections. If you never use your total allocation, consider decreasing the number of managed connections.

You can monitor connections either through using OpenEdge logging or by enabling logging using the "`-Dsrv qt_debug,cursor`" switch.

Note: The OpenEdge logging infrastructure offers more extensive reporting capabilities than qt_debug. For details on enhanced logging, see Analyzing application execution with Enhanced Logger on page 206.

Monitoring connections with qt_debug

The DataServer log contains messages indicating the status of connections. At startup, the number of managed connections initialized is written to the log file. If connections are rejected, this is also logged. If a connection from the ODBC connection pool cannot be reused, the ODBC driver issues the message **DEAD Connection** which is written to the log file.

At the end of a session, the DataServer writes summary information about cursor use. The summary contains the following information:

- **Needed connections** — The number of connections actually used during a session
- **Peak connections** — The maximum number of connections used simultaneously during a session

When debug logging is enabled with the "`-Dsrv qt_debug,cursor`" switch, a summary of connection activity is written to the log file. This summary contains:

- Number of connections (defaults to 5)
- Number of peak connections (defaults to 5)
- Highest connection value
- Lowest connection value
- Number of connections pulled out of pool
- Number of connections placed in the pool
- Total sum of active connections

The following excerpt from a `dataserv.lg` file shows the summary of the managed connection activity written when the `"-Dsrv qt_debug,cursor"` switch is used, plus the standard messages:

```
16:32:09 : Standard secondary connection:
16:32:09 : Connection chain 0xbc6e64: num_conns 1
16:32:09 : Connection chain 0xbc6e64: peak_conns 1
16:32:09 : Connection chain 0xbc6e64: high_conns 1
16:32:09 : Connection chain 0xbc6e64: low_conns 0
16:32:09 : Connection chain 0xbc6e64: num_off_conns 1
16:32:09 : Connection chain 0xbc6e64: num_on_conns 1
16:32:09 : Connection chain 0xbc6e64: sum_active 1
16:32:09 : Connection chain 0xbc6e64: Needed Connections: 1, (Needed 4 less than
requested)
16:32:09 : Connection chain 0xbc6e64: Peak Connections: 1, (Peaked at 4 less than
requested)
16:32:09 : Connect: 0x1012398    Stmt: 0x1b95f80   Handle: 0x1011d98   Crc: 1613 Proprty:
      Statement dropped
16:32:09 : Statement cache (-Dsrv PRGRS_PREPCACHE setting) reuse ratio is 0.
```

For more information on using the `qt_debug` option to the `-Dsrv` switch, see Server Related Command Line Utilities and Startup Parameters on page 353

Cursor downgrades

Cursor downgrades are monitored by the DataServer, according to the type of cursors in use:

- For firehose cursors, the DataServer counts the number of times a cursor is downgraded, and writes the count to the log file at the end of the session.

Consider increasing the number of DataServer managed connections if firehose cursors are frequently downgraded.

- For downgrades of Fast Forward-Only cursors, a message is written to the log file when the downgrade occurs.

Statement cache

The reuse of the statement cache is calculated as a ratio. Firehose cursors decrease the probability of a statement being reused. A reuse ratio of 3 indicates that statements are reused an average of 3 times. Reuse ratios of 3 or higher indicate good utilization of the statement cache. Reuse ratios less than 1 indicate poor utilization of the statement cache. If your ratio is less than 1, consider disabling the statement cache with "`-Dsrv PRGRS_PREPCACHE,0`". For more information on the `PRGRS_PREPCACHE` option to the `-Dsrv` switch, see Server Related Command Line Utilities and Startup Parameters on page 353

OpenEdge query types

The DataServer provides several ways to submit an OpenEdge query to a MS SQL Server data source:

- **ABL** — 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.
- **OpenEdge SQL SELECT** — This approach applies to the SQL `SELECT` statement. When you use this statement in an OpenEdge 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 RDBMS 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 MS 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 RDBMS Stored Procedure Details on page 101.

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. ABL 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.

Query tuning

How you structure a query determines how efficiently you access a database. Using your MS SQL Server 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 OpenEdge `QUERY-TUNING` phrase.

You can include the `QUERY-TUNING` phrase in these OpenEdge statements:

- FOR EACH

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

- OPEN QUERY

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

- DO PRESELECT

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

- REPEAT PRESELECT

```
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.

The following table describes the query-tuning options.

Table 24: Query-tuning options

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. This value will override a cache size specified with -Dsrv QT_CACHE_SIZE. Minimum: The DataServer always caches at least one record. Maximum: None. **Default**: 30000.

Option	Description
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. **Default**: NO-DEBUG.
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. For more information on JOIN-BY-SQLDB, see Join by SQL DB on page 151. **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 an OpenEdge session. This parameter does not override the explicit use of JOIN-BY-SQLDB in the QUERY-TUNING phrase. **Note:** JOIN-BY-SQLDB is the default behavior for outer join operations.
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 an OpenEdge database because changes made to the records in the cache might not be immediately visible. Specify NO-LOOKAHEAD for behavior that is consistent with OpenEdge. **Default**: LOOKAHEAD, when statements use NO-LOCK or when statements use SHARE-LOCK with TXN_ISOLATION level set to 1 (read uncommitted.)

Option	Description
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 OpenEdge. **Default**: NO-SEPARATE-CONNECTION except in certain cases. For details, see Managing connections to an MS SQL Server database on page 223.
NO-QUERY-ORDER-ADDED	Specifies that OpenEdge should not choose an index in the absence of a USE-INDEX or BY clause in the query request. OpenEdge may otherwise select an index if it is needed to provide ABL language compatibility. **Note:** If you elect to use this option to omit index selection on the query, you may see better performance using the optimizer's sort selections. However, compatibility with OpenEdge forward/backward scrolling and reposition capability may be lost. Only use this option when compatibility is not required and can be overlooked for the sake of better performance.
NO-QUERY-UNIQUE-ADDED	Specifies that OpenEdge should omit the record identifier from the end of the query's generated ORDER BY clause when trying to obtain record uniqueness from a selected non-unique index. A sort order that is modified to derive uniqueness may produce a query that can't find a useful index to perform sorting thus impacting query performance. **Note:** If you elect to use this option, the query may find an index match to provide better performance. However, turning off uniqueness in a query where scrolling is required may result in behavior that is incompatible with the OpenEdge ABL. Only use this option when compatibility is not required and can be overlooked for the sake of better performance.

Option	Description
FIREHOSE-CURSOR NO-FIREHOSE-CURSOR	Specifies at the query level that the firehose cursor type should be considered to satisfy the query when the NO-LOCK phrase is used. **Note:** This query-level option overrides the connection-level -Dsrv options, QT_FIREHOSE and QT_NO_FIREHOSE that determine if firehose cursors should be considered for the DataServer connection. Specifies at the query level that the firehose cursor type should not be considered to satisfy the query when the NO-LOCK phrase is used. **Note:** This query-level option overrides the connection-level -Dsrv options, QT_FIREHOSE and QT-NO-FIREHOSE that determine if firehose cursors should be considered for the DataServer connection. By default, firehose cursors are available to satisfy NO-LOCK queries during a DataServer session. It is generally recommended this default use be retained and overridden by QT-NO-FIREHOSE on an individual query basis in the event that slow query performance is observed on a very large result set.
SELECT-ON-JOIN JOIN-ON-SELECT	When performing an outer join operation, SELECT-ON-JOIN specifies that any search condition separate from the join predicate be bracketed **after** the join is performed. This option eliminates all the records with NULL values contained in the non-matching results, if the columns with the NULL values appear in the join search criteria. When performing an outer join operation, JOIN-ON-SELECT specifies that any search condition separate from the join predicate be bracketed **before** the join is performed. This option includes all the records with NULL values contained in the non-matching results and causes records that join on NULL to be included in the results of an OUTER-JOIN which is consistent with OpenEdge join handling of the unknown value. **Default**: JOIN-ON-SELECT.

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, JOIN-BY-SQLDB, that the DataServer instructs the MS SQL Server data source to perform by default, as shown:

```
FOR EACH customer, EACH order OF customer WHERE order.ordernum GT 20
  BY customer.custnum QUERY-TUNING (JOIN-BY-SQLDB)
```

The QUERY-TUNING options in this example specifies 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 a MS SQL Server 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.

Caching records

The DataServer caches results sets from a MS SQL Server data source to enhance performance when using block cursors. It caches as much data as fits in its allocated cache size. When using firehose and/or lookahead calls, the allocations are made to fit the size of the result set. 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 MS SQL Server 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 order.ordnum GT 20
  QUERY-TUNING(CACHE-SIZE 20 DEBUG EXTENDED):
```

Join by SQL DB

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

In DataServers, JOIN-BY-SQLDB is the default behavior for outer join operations.

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

- All tables in the join are in the same logical OpenEdge 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 SQLDB will not occur for this query:

  ```
  FOR EACH customer, EACH order OF customer USING order.ordernum:
  ```

- 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.
- There is no nested FOR block in 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.custnum = order.custnum:
```

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

```
FOR EACH customer, EACH order
    WHERE customer.custnum = order.custnum AND customer.custnum GT 100:
```

However, for the following query, the DataServer instructs the client to perform the join because of the OR option:

```
FOR EACH customer, EACH order
    WHERE customer.custnum = order.custnum OR customer.custnum GT 100:
```

Note:

By default, the dynamic joins, the inner joins, and the outer joins are performed by the MS SQL database server, provided the criteria set by the DataServer is fulfilled. For more information on joins, see *OpenEdge Data Management: SQL Development*.

An Unknown value (?) in OpenEdge is mapped to a NULL value in MS SQL Server; When you perform a JOIN-BY-SQLDB, MS SQL does not join the records in the parent table to the records in the child table on NULL join key values. For an outer join performed with JOIN-BY-SQLDB, if the results set contains a parent record with a NULL join key value and a non-matching child record, all field values in the child record are set to NULL. In contrast, when you perform a join by client, records that are joined on the Unknown value (?) (to which NULLs correspond), the results set includes any matching parent and child records in the join results of both inner and outer joins.

By default, the DataServer instructs a MS SQL Server 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.

The following table describes how these controls interact and affect the behavior.

Table 25: Controlling join by SQLDB behavior

QUERY-TUNING	Startup Parameter	Behavior
JOIN-BY-SQLDB	-nojoinbysqldb	The MS SQL Server data source performs the join if possible.
JOIN-BY-SQLDB	None	The MS SQL Server data source performs the join if possible.
NO-JOIN-BY-SQLDB	-nojoinbysqldb	The client performs the join.

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

A join by SQLDB 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 MS SQL Server 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.

When the join keys contain the Unknown value (?) for OpenEdge, those record matches are included in the join results. This is in contrast to the NULL value for SQL databases whereby

Statement level lock upgrade for JOIN query

OpenEdge ABL allows a user to specify a lock type in the OPEN QUERY statement of a JOIN query. An EXCLUSIVE-LOCK can be applied to individual buffers that are participating in a JOIN query. Opening a JOIN query when one or more participating buffers are opened with EXCLUSIVE-LOCK makes DataServer evaluate the query to be joined by the client. Such join queries are not evaluated on the server and their performance might be affected. OpenEdge MSS Dataserver supports statement level lock-upgrade for join query evaluated using server side join. To use this capability, Progress suggests that users should open a query with lock-type as NO-LOCK and specify the desired locking at the statement level. This allows DataServer to evaluate the query as server side join. DataServer caches the results set of ROWID index key values of participating join buffers depending on what lock type is specified when the first GET statement is executed. Users can now evaluate the server side solution to fetch the record with an EXCLUSIVE-LOCK on a join query opened with SHARE-LOCK/NO-LOCK and update the records.

Note: GET statements can be used with different locking types at statement level.

Table 26: Syntax

```
GET { FIRST | NEXT | PREV | LAST | CURRENT } query [ SHARE-LOCK | EXCLUSIVE-LOCK |
NO-LOCK ]
[ NO-WAIT ]
```

For more information on how to run ABL queries and the syntax, see *Progress OpenEdge ABL: ABL Reference* guide.

The GET statement for FIRST, NEXT, and CURRENT are executed on the server side and for LAST and PREV are evaluated on the Client side and not the server side. To use SCROLLING capability and use GET LAST or GET PREV statement, an ABL application should use QUERY-TUNING switch (NO-JOIN-BY-SQLDB) to evaluate the statements on client side.

The MSS DataServer supports the following scenarios and can be evaluated on the server side. The following table provides the possible Lock type in each scenario:

Table 27: Locking Scenarios

QUERY-TUNING	Startup Parameter
NO-LOCK and fetched with	EXCLUSIVE-LOCK
	SHARE-LOCK
	NO-LOCK
	Default Query-level lock
SHARE-LOCK and fetched with	EXCLUSIVE-LOCK
	SHARE-LOCK
	NO-LOCK
	Default Query-level lock
Default lock and fetched with	EXCLUSIVE-LOCK
	SHARE-LOCK
	NO-LOCK
	Default Query-level lock
NO-LOCK, SHARE-LOCK and fetched with	EXCLUSIVE-LOCK
	SHARE-LOCK
	NO-LOCK
	Default Query-level lock
NO-LOCK, default lock (ABL), and fetched with	EXCLUSIVE-LOCK
	SHARE-LOCK
	NO-LOCK
	Default Query-level lock

QUERY-TUNING	Startup Parameter
SHARE-LOCK, default lock (ABL), and fetched with	EXCLUSIVE-LOCK
	SHARE-LOCK
	NO-LOCK
	Default Query-level lock
NO-LOCK, SHARE-LOCK, default lock (ABL), and fetched with	EXCLUSIVE-LOCK
	SHARE-LOCK
	NO-LOCK
	Default Query-level lock

When the first GET statement is executed with EXCLUSIVE-LOCK and the subsequent GET statement is executed with any kind of locking, DataServer prepares the ROWID index key values cache of the tables participating in the JOIN query. DataServer then fetches records from the data source by using its ROWID and locks the record in data source.

When the first GET statement is executed with a lock (NO-LOCK or SHARE-LOCK) other than EXCLUSIVE-LOCK, DataServer caches the whole record in the result set and subsequent fetches retrieves the records from the cache. If the subsequent GET specify exclusive-lock, then the record is locked in the data source.

Note: In all the scenarios mentioned in the above table, if the user specifies QUERY-TUNING(NO-JOIN-BY-SQLDB), join by client is applied and the Join is evaluated on the client side.

Examples:

- In a scenario where the query is opened with NO-LOCK and the first GET statement uses EXCLUSIVE-LOCK, the query syntax is as below:

```
OPEN QUERY cust-query FOR EACH Customer NO-LOCK, EACH order of customer NO-LOCK.
GET NEXT cust-query EXCLUSIVE-LOCK.
GET NEXT cust-query NO-LOCK.
```

- In a scenario where the query is opened with NO-LOCK and the first GET statement uses NO-LOCK, the query syntax looks like this:

```
OPEN QUERY cust-query FOR EACH Customer NO-LOCK, EACH order of customer NO-LOCK.
GET NEXT cust-query NO-LOCK.
GET NEXT cust-query EXCLUSIVE-LOCK.
```

Note: The default lock for the above ABL statements is NO-LOCK as both participating buffer has NO-LOCK as query-level lock types. The default for participating buffer is the respective lock type specified at query-level.

When the join keys contain the Unknown value (?) for OpenEdge, those record matches are included in the join results. This is in contrast to the NULL value for SQL databases whereby

Improving join performance for queries with FIRST or LAST criteria

In versions prior to Progress OpenEdge 11.7.4, processing an ABL join query with multiple tables using the FIRST or LAST keywords was significantly slow as compared to processing a similar ABL query against a single table or against a join query that did not use FIRST or LAST criteria. This happened because the server join was unable to reduce the FIRST or LAST criteria to a single server join record during ABL to SQL translation which led to a widened result set and forced client selection, in some cases, the client was forced to retrieve server records key by key using multiple queries.

The join performance has now been improved by ensuring that all joins with FIRST or LAST criteria translate ABL to a single server join SQL statement. This functionality is known as **Single-shot** functionality.

From Progress OpenEdge 11.7.4 release onwards, the single-shot statement will now be evaluated at the server side.This will help in performance improvement of single-shot joins by:

- Maximizing server optimization
- Minimizing client selection
- Reducing round trips to the foreign data source

An example of single-shot at innermost join level is shown below. Here, single-shot is at innermost join level and remaining buffers are joined using EACH statement. It will be evaluated at server side.

```
FOR EACH customer,
    EACH order OF customer,
    EACH orderline OF order,
    FIRST ITEM OF orderline:
    DISPLAY customer.custnum  order.custnum order.ordernum
    orderline.linenum item.itemnum.
END.
```

The ABL uses an index to evaluate the “FIRST” or “LAST” criteria. By default, the primary index is selected in determining the “MIN” or “MAX” column criteria to be applied in the server sub-query that handles the FIRST or LAST join criteria. When USE-INDEX clause is used, the index specified in the USE-INDEX clause will be used in place of the default primary index.

Index definitions support ABL USE-INDEX modifier. ABL translates USE-INDEX to ORDER BY for DataServer operations. For example, if you define custnum as an index on the custnum field without using the single shot query mechanism, the following ABL statements are equivalent when accessing a SQL database:

```
FOR EACH customer USE-INDEX custnum:
```

```
FOR EACH customer BY custnum:
```

The following examples illustrate the behavior of the index definition if the single-shot query mechanism is used.

If single-shot query mechanism is used, the following query results in a server-side join:

```
FOR EACH customer NO-LOCK USE-INDEX custnum WHERE custnum = 1,
FIRST order OF customer USE-INDEX custorder:
```

If the single-shot query mechanism is used, the following query results in a client-side join as the BY clause has been used:

```
FOR EACH customer NO-LOCK WHERE custnum = 1,
FIRST order OF customer
BY customer.custnum BY order.custnum BY order.ordernum:
```

Note:

If you do not specify USE-INDEX, 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 in your query definition.

Progress recommends that you use a unique index as a USE-INDEX modifier when using it with a FOR EACH query that joins FIRST or LAST on a child table. If the index is not unique, join is still evaluated on client side.

Client startup parameter and QUERY-TUNING Switch

Single-shot evaluation on server will be ON by default. It can be turned OFF and evaluated at client side by using client startup parameter -nojoinbysqldb. It can also be turned OFF at run time with QUERY-TUNING switch NO-JOIN-BY-SQLDB.

Performance Optimization

Users can choose between two query tuning switches – KEYCACHE JOIN and NO-KEYCACHE JOIN. These switches are mutually exclusive and are applicable only when the first GET statement is called using EXCLUSIVE-LOCK. The switches are described in the following sections.

- **KEYCAHE JOIN** – To enable KEYCACHE join when an application has a -Dsrv switch PRGS_JOIN_NOKEYCACHE enabled, users should specify QUERY-TUNING (KEYCHACHE JOIN) in the ABL query. This will result in populating the key-cache for the Join buffer that is participating in a join query.
- **NO-KEYCAHE JOIN** – To prevent or to turn off the populating of key-cache for the participating join buffer, the user should specify QUERY-TUNING (NO-KEYCACHE-JOIN) in the ABL query.

Note: When an ABL application has most of the GET statement called with NO-LOCK and SHARE-LOCK in that case specifying QUERY-TUNING(NO-KEYCACHE-JOIN) or -Dsrv PRGRS_JOIN_NOKEYCACHE may improve the performance of application.

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:

- 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 MS SQL Server 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 MS SQL Server 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 an OpenEdge SQL statement. If you use an OpenEdge 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 MS SQL Server data source for processing. However, avoid nesting CAN-FIND functions.
- Avoid using the RECID function. Instead, use the ROWID function.
- Avoid queries that include server LOB types in their results if the LOB data is not pertinent with the needs of your application. LOB types, and their processing overhead, can be omitted from your query results using field lists.

Modifying the run-time schema check

At runtime, the DataServer for MS 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 OpenEdge 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 MS SQL Server 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:

```
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 can 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`.

Replacing FIND FIRST for performance

When coding ABL applications for the DataServer, `FIND FIRST` statements can often be replaced with other ABL statements to greatly improve performance. `OPEN QUERY`s and dynamic `FIND`s are significantly more efficient. Likewise, using `FOR FIRST` in place of `FIND FIRST` can improve performance when retrieving a single record with `NO-LOCK`. For example, if your application uses the following `FIND FIRST` code:

```
FIND FIRST tablename WHERE where-clause NO-LOCK.
```

The code can be replaced with the following for significant performance gains:

```
FOR FIRST tablename WHERE where-clause NO-LOCK:
END.
```

In the case of the `FOR FIRST`, the record is not reliably available beyond the end of the `FOR FIRST` loop unless a `LEAVE` is specified. Similar performance advantages can be gained when retrieving last record as well.

Replacing FOR FIRST for performance

When coding ABL applications for the DataServer, FOR FIRST statements can be used often be replaced with another ABL statement to greatly improve performance. Using FOR only in place of FOR FIRST can improve performance when retrieving a single record. For example, if your application uses the following FOR FIRST code:

```
FOR FIRST customer WHERE cust-num <=10:
DISP NAME.
END.
```

This code returns a list of all customer names where the customer name is less than or equal to 10. It can be replaced with the following code for retrieving a single record with significant performance gains:

```
FOR customer WHERE cust-num <=10:
DISP NAME.
END.
```

5

Configuring the DataServer

Configuring the DataServer for MS SQL Server involves starting executables for several processes. This chapter provides step-by-step instructions for initially setting up the DataServer.

Before you configure a DataServer, make sure that you have installed all of the required software. For details, see Software requirements on page 37.

For details, see the following topics:

- DataServer components
- Configuring an ODBC driver and registering the data source
- Configuring a local DataServer
- Configuring a remote AppServer
- Configuring a remote DataServer
- Configuring a self-service DataServer
- Creating a schema holder
- Maintaining a schema holder
- Typical configuration for a remote client to connect to a remote DataServer

DataServer components

The DataServer for MS SQL Server can run in a variety of configurations. Some configurations involve a single process running on one machine. Others involve multiple processes running on different machines across multiple platforms. With the OpenEdge 11.6 release, it can involve agent threads running from a Progress AppServer on a machine and other processes running on the same or other machines across multiple platforms.

DataServer configuration prerequisites

Before you configure a DataServer, you must:

- Register your MS SQL Server database as an ODBC data source. For details, see Registering your data source on page 163.
- Determine which components you need on which platforms and then set up the appropriate executables on those platforms.

  The following table 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 28: Installing the DataServer components

Client	DataServer	Installing and configuring
Windows	Local Windows	Use the default OpenEdge client executable (`prowin32.exe`). The server component is embedded in the client executable.
Windows	Remote Windows	On the **client machine**, use the default OpenEdge client executable (`prowin32.exe`). On the **host machine**, to make a connection to the default DataServer executable (`_msssrv.exe`), use the default broker executable (`_probrkr.exe`) or configure a broker using OpenEdge Management or OpenEdge Explorer.
UNIX	Remote Windows	On the **client machine**, use the default OpenEdge client executable (`_progres`). On the **host machine**, to make a connection to the default DataServer executable (`_msssrv.exe`), use the default broker executable (`_probrkr.exe`) or configure a broker using OpenEdge Management or OpenEdge Explorer.

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 Configuring a local DataServer on page 163. If you are building a remote DataServer configuration host, see Configuring a remote DataServer on page 164.

Configuring an ODBC driver and registering the data source

Before you can configure the DataServer you must:

- Install your ODBC drivers and the client connectivity component from MS SQL Server
- Set up your data sources

When you install the DataServer for MS SQL Server, the DataDirect ODBC driver for MS SQL Server is bundled in and installed automatically. If you prefer to use a different driver, you must obtain the drivers from the vendor and install them separately.

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 DataDirect ODBC Driver information, see the DataDirect Web site at: http://web.datadirect.com/products/odbc/.

For any alternate driver, refer to the configuration information provided by your ODBC driver vendor.

Registering your data source

In Windows, before you can use your data source with the DataServer for MS SQL Server, you must register it as an ODBC data source using the ODBC administration tool.

To register a data source:

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-OpenEdge data source vendor. Also, some vendors might provide a similar administration tool.
2. Select the appropriate ODBC driver.
3. Select your MS 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.

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.)

To configure a local DataServer in Windows:

1. Install the ODBC software and the client software on the system where your local DataServer resides.
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 a schema holder for your MS SQL Server database and connect using the client executable. See Creating a schema holder on page 177 for instructions.

Configuring a remote AppServer

The OpenEdge AppServer has at least 3 processes - OpenEdge client, OpenEdge AppServer and AppServer agent. The Progress AppServer runs at least 2 process - OpenEdge client and the Progress AppServer with multi-threaded agents. The remote AppServer can be run as a self-service or client/server. For a client/server, AppServer connections see "Configuring a remote DataServer on page 164" to understand configurations involving a remote host for the server component of the DataServer.

Both the OpenEdge AppServer and the Progress AppServer can run in self-service mode. The Progress AppServer can only run in self-service mode. The following section describes how to set up a client configuration to a self-service AppServer where the AppServer is either the OpenEdge Appserver or the Progress Appserver.

Configuring a 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 unified broker for OpenEdge Management or OpenEdge 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 MS SQL Server database and communicates with the client process.

Note: The OpenEdge MS SQL Server DataServer component is an ODBC **client** with respect to the MS SQL Server database configuration even though from the OpenEdge point-of-view it resides on the **server** machine. As a result, your MS 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 OpenEdge client component requires no special software or configuration; it only requires a standalone OpenEdge Windows or UNIX client. There are no OpenEdge restrictions regarding the location of the actual MS 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. In Windows, you configure the DataServer using OpenEdge Management or OpenEdge Explorer. See Configuring with OpenEdge Management or OpenEdge Explorer on page 164 for details.

Configuring with OpenEdge Management or OpenEdge Explorer

You can use OpenEdge Management or OpenEdge Explorer to configure the DataServer for MS SQL Server.

Note: This section primarily focuses on how to access OpenEdge Management or OpenEdge Explorer to **configure** the DataServer for MS SQL Server. It briefly presents a few examples about how to **connect** the DataServer. However, for more complete connection instructions, see Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190. For information about OpenEdge Management or OpenEdge Explorer, see the OpenEdge Management or OpenEdge Explorer online help.

To access OpenEdge Management or OpenEdge Explorer in Windows:

1. Make sure that the AdminService is running.
2. Start the OpenEdge Management or OpenEdge Explorer. Choose **Start** > **Programs (or All Programs)** > **Progress** > **OpenEdge** > **Management Console or OpenEdge Explorer**.
3. Connect to localhost and log in.
4. From the console's left pane, select the **MSS DataServer** folder and `double-click`. The list of existing DataServer brokers for MS SQL Server appears in the right pane.
5. Select the DataServer instance whose properties you want to create or edit, and `click` **Configuration** so that the DataServer properties appear.
6. `Click` **Edit**.

   Note: The DataServer for MS SQL Server installation provides one predefined DataServer Broker (`mssbroker1`) and one predefined NameServer (`NS1`). (The NameServer is located in the **NameServer** folder.) Each broker is referred to as an instance. See the OpenEdge Management or OpenEdge 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 *OpenEdge Getting Started: Installation and Configuration* for information about the NameServer's role in a configuration.)

7. Configure the DataServer broker by setting the Broker properties, Server properties and Environment Variables. For details on these settings, see the OpenEdge Management or OpenEdge Explorer online help.

   Proceed to 7.a on page 165 to use the DataServer for MS SQL Server broker instance with a controlling NameServer (as provided by a default Data Service). Otherwise, proceed to 7.b on page 165 to use the broker instances or without the NameServer (using the `-DirectConnect` parameter option):

   a) To use the controlling NameServer to manage your broker instances, use the Data Service (This is the default and recommended option. For syntax and examples, see Connecting to a MS SQL Server DataServer broker using the NameServer on page 166.) As a result, the DataServer client's initial connection is to the NameServer.

   b) To connect the DataServer directly to the broker instance, do one of the following:

   - Set the `-DataService` value to `none` in the connection parameters of your schema holder.

     If you plan to always use a `-DataService` value of `none`, remove the controlling NameServer from your broker instance definition.

   - Add the `-DirectConnect` parameter to override your `-DataService` parameter.

See Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190 for more information about connecting the DataServer to the NameServer and the broker.

Connecting to a MS SQL Server DataServer broker using the NameServer

This section presents an example of how to connect to a MS SQL Server DataServer using the NameServer.

To connect to a MS SQL Server DataServer using the NameServer:

1. Create a schema holder.
2. Set your MS SQL Server instance username (`-U`) and password (`-P`) connection parameters for the schema holder.

   Progress Software Corporation recommends that you use the `-U` and `-P` convention to establish the username and password.

3. Set up your connection, using parameters to run the schema holder. This step shows examples from which you can model your connection:

   a) The following example shows how to set up the connection in Windows in a single-user mode:

   ```
   prowin32 f:\wrk\holder -1 -db MSS_DB_DSN -ld MSS_DB_INS -dt MSS -H MSS_DSRV_Host
    -S 5162 -DataService mssbroker1 -Dsrv SVUB,1
   ```

   Note that although the schema holder is defined as a single-user mode, access to the MS SQL Server instance is in multi-user mode.

   b) The following example shows how to set up the connection in Windows in a multi-user mode:

   ```
   prowin32 holder -H MSS_DSRV_Host -S Holder_Service -db MSS_DB_DSN -ld MSS_DB_INS
    -dt MSS -H MSS_DSRV_Host -S 5162 -DataService mssbroker1 -Dsrv SVUB,1
   ```

   c) The following example shows how to set up the connection on UNIX:

   ```
   mpro holder -H MSS_DSRV_Host -S Holder_Service -db MSS_DB_DSN -ld MSS_DB_INS -dt
    MSS -H MSS_DSRV_Host -S 5162 -DataService mssbroker1 -Dsrv SVUB,1
   ```

   The following list identifies the values used in the examples shown in Step 3:

   `MSS_DB_DSN`

   Identifies the ODBC Data Source name.

   `MSS_DB_INS`

   Identifies the MS SQL Server instance name.

   `MSS_Dsrv_host`

   Identifies the name of the MSS DataServer broker host.

   `5162`

   Identifies the NS1 NameServer default port where the MSS broker is registered.

`Holder_Service`

Identifies the service name used to server the database holder; however, it can also be a port number.

For details about the database connection parameters noted in each example in of this procedure, see Connecting a schema holder at startup on page 197.

Caution: Do not simultaneously run some DataServers for MS 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 (PSC) 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, PSC recommends that you reintroduce the NameServer into your configuration.

Once you have completely set up your environment, you can build the schema holder for your MS SQL Server database. See Creating a schema holder on page 177 for instructions.

Configuring from the command line

You can configure a remote DataServer broker process from the command line. You can use the `_probrkr.exe` executable or use the `mssman.bat` utility within OpenEdge Management or OpenEdge Explorer, the command line equivalent of using OpenEdge Management or OpenEdge 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 Configuring an ODBC driver and registering the data source on page 163 for details.

To configure from the command line:

- When using the OpenEdge 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` utility within OpenEdge Management or OpenEdge Explorer, 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 your environment variables in the same environment (DOS shell) from which you plan to run the DataServer.

The following table describes the environment variables that you must set.

Table 29: Environment variables for the remote DataServer

Variable	Description
`DSLOGDIR` (Optional)	The pathname of the log file that OpenEdge uses to keep track of DataServer processes and error messages. By default, OpenEdge writes to `%DSLOGDIR%\dataserv.lg` where `%DSLOGDIR%` is resolved to directory path of the log file `dataserv.lg`.

Variable	Description
MSSSRV	The pathname of the OpenEdge server executable. This variable is required only when the DataServer is spawned using the OpenEdge broker (_probrkr.exe). When using mssman, this setting is satisfied by the srvrExecFile setting in the broker properties file.
PROBRKR (Optional)	The pathname of the broker executable. When using the OpenEdge 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.
PROSTARTUP	The pathname of your default OpenEdge startup (.pf) file. To set the Message Buffer Size (-Mm) startup parameter for the broker to a value different from the 1024 default buffer size, you must set the PROSTARTUP environment variable equal to the path and name of the .pf file that contains the -Mm parameter. This approach ensures that your value is recognized and therefore used by the broker when it starts. Only Windows clients must use this technique. **Note:** If you use the command line utility and intend to set the -Mm startup parameter to a value other than 1024, you must use the .pf file with the PROSTARTUP variable set.
UNIQUE_DSLOG	Used to generate unique log file name on the server with dataserv_*pid*.lg, where *pid* is the process-id of server. When set, the designated DataServerlog log file name becomes a root name. Can be disabled by unsetting the environment variable.

Once you have completely set up your environment, you can build the schema holder for your MS SQL Server database and connect using the client executable. See Creating a schema holder on page 177 for instructions.

The ubroker.properties file

When using the Unified Broker framework, you configure the DataServer for MS SQL Server by editing the *OpenEdge-install-dir*\properties\ubroker.properties file. This file stores configuration definitions for instances of many OpenEdge products. For a complete list of products and a detailed discussion of the Unified Broker Framework, see *OpenEdge Getting Started: Installation and Configuration*.

When you use this file to configure the DataServer for MS SQL Server, you provide information that enables the host to start a broker that spawns the appropriate DataServer process (_msssrv.exe).

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 OpenEdge installation.

Establishing the TLS protocol in a DataServer broker instance of the Unified Broker

The following set of TLS properties are used to establish TLS protocol communication using OpenEdge networking from a unified broker configuration using the ubroker.properties file:

Table 30: TLS properties used to establish TLS protocol communication

Property	Default Value	Description
sslEnable	0	Enables TLS-secure connections to a DataServer broker instance from remote DataServer clients. To enable TLS security for incoming client connections, set the property value to *1*. While the `sslEnable` switch enables the infrastructure needed to make TLS connections, it does not initiate the start of an TLS connection dialogue.To start the dialog, the `srvrStartupParam` must be appended with the "-ssl" startup parameter.
keyAlias	default_server	Specifies the alias set for a private/public key. If this property is not set, the server certificate alias that is provided by OpenEdge is used.
keyAliasPasswd	20333c34252a2137	Specifies the encoded password set for the alias of the private/public-key to access the server Private Key / Digital Certificate key store entry. You can use the OpenEdge provided genpassword utility for encoding your passwords
keyStorePath	$DLC/keys	Specifies the key store location.
keyStorePasswd	NA	Specifies the password used for accessing the custom keystore. This password is expected to be encoded. You can use the OpenEdge provided genpassword utility for encoding your passwords.
noSessionCache	0	0 - enable TLS session caching (stores TLS information) 1 - disable TLS session caching (no TLS information persisted)
srvrStartupParam	*-svub -S X -N TCP -U X -P X -hs 0 -s 40*	Specifies server/agent startup parameters that each process uses when the broker starts it. The default startup parameters can be appended with the "-ssl" parameter in order to initiate server/agents in TLS secure connection mode as follows:-svub -S X -N TCP -U X -P X -hs 0 -s 40 –ssl. When agents are started with the -ssl switch, they can only accept TLS-enabled client connections. This property can be dynamically updated. Dynamic changes affect only new agents that are started after this value is changed.
sslCiphers	AES128-SHA256	The TLS algorithm list to use in arbitrating an TLS session with an TLS client. Also referred as server Ciphers
certStorePath	@{Startup\DLC}\certs\	The path to the trusted certificate store. The root certificate for the certificate authority that was used to sign the server certificate should be specified by this path. Client can mention Cert Store path on the command line i.e. _progres -b -p client.p -certstorepath $RDLQA/tests/SHA2/$alg/certs

When a DataServer instance of the Unified Broker is configured for TLS, the server/agent does not attempt to auto-correct the client's connection attempt when the client connects in non-TLS mode. Instead, the connection request is rejected by the server/agent and an exception is logged. Client users must explicitly initiate their server connection in TLS mode in order to connect to an TLS-enabled DataServer broker instance of the Unified Broker.

Note: This is in contrast to the ProBroker configuration where a DataServer client connection attempted in non-TLS mode receives a return message from the TLS-enabled ProBroker that auto-corrects the connection request and retries it in TLS mode.

Editing the ubroker.properties file

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.

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 following table describes the sections in the `ubroker.properties` file that apply to the DataServer for MS 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 31: DataServer for MS SQL Server sections of the ubroker.properties file

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.
DataServer	`UBroker.MS`	The parent entity of DataServer for MS SQL Server brokers. It defines default property settings for all of these broker instances.

Product	Section	Description
DataServer	`UBroker.MS.mssbroker1`	A sample property entity of a DataServer for MS 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. • `@{Startup\DLC}\bin\_msssrv.exe`, the default, which runs the default DataServer from your OpenEdge install path. • `srvrStartupParam`: Specify the OpenEdge 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 Unified Broker framework. • `srvrDSLogfile`: To specify the name of dataserver log file. If not specified default name as `dataserv.lg` used. • `srvrminport`: To specify maximum port number for server. • `srvrmaxport`: To specify minimum port number for server.
DataServer	`Environment.mssbroker1`	A sample environment entity of a DataServer for MS 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 Configuring a remote DataServer on page 164 for the environment variable settings required to configure the DataServer for MS SQL Server.

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

```
#
#  Default properties for broker instances serving MSS DataServers
#
[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
    portNumber=4446
    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=172.18.103.53:32:e031b1e7bc:-7d29
     description=A sample MSS DataServer Broker

##
Environment for MSS Dataserver Broker: mssbroker1
##
#[Environment.mssbroker1]
#
# Sample MSS DataServer Broker definition using SSL
#
[UBroker.MS.mssSSLbroker1]
     srvrExecFile="@{Startup\DLC}\bin\_msssrv.exe"
     srvrStartupParam=-svub -S X -N TCP -U X -P X -hs 0 -s 40
     srvrLogFile=@{WorkPath}\mssSSLbroker1.server.log
     brokerLogFile=@{WorkPath}\mssSSLbroker1.broker.log
     portNumber=4447
     defaultService=0
     appserviceNameList=mssSSLbroker1
     controllingNameServer=NS1
     environment=mssSSLbroker1
     uuid=172.18.103.53:32:e031b1e7bc:-7d29
     description=A sample MSS DataServer Broker using SSL

#
# Environment for MSS Dataserver Broker: mssSSLbroker1
#
[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.README file.

The ubroker.properties file is read on startup of the AdminService process. For changes in any used environment variables to take effect, the AdminService 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 MS SQL Server, you can validate the file by using the `mssconfig.bat` utility or the mergeprop utility. For more information about the MSSCONFIG utility, see Server Related Command Line Utilities and Startup Parameters on page 353. For details about the mergeprop utility, see *OpenEdge Getting Started: Installation and Configuration*.

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 MS 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 MS 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 Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190 for more information about connecting the DataServer to the NameServer and the broker. See *OpenEdge Getting Started: Installation and Configuration* for more information about the NameServer's role in a configuration.

Note: Do not simultaneously run some DataServers for MS 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 (PSC) 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, PSC recommends that you reintroduce the NameServer into your configuration.

Configuring a self-service DataServer

To confiure a self-service DataServer agent in the OpenEdge AppServer and the Progress Application Server for OpenEdge, perform the steps in the following sections.

Configuring the OpenEdge AppServer agent as a self-service DataServer session

To configure the OpenEdge AppServer agent as a self-service DataServer session:

1. Make sure that the AdminService is running.

   Note: For more information on using the PROADSV tool, see the *OpenEdge Application Server: Administration guide*.

2. Add the following property in the ubroker.properties file under`UBroker.AS.asbroker1 section`:

   ```
   srvrStartupParam= -pf "$WRKDIR/conn.pf"
   ```

   where AS1 is the AppServer.

3. Start the Application Server broker using the following command:

   ```
   asbman -i asbroker1 -start
   ```

   .

   Note: For more information on using asbman commands and its parameters, refer to the *OpenEdge Application Server: Administration*.

4. Open an ABL session and execute the client.p to connect to the self-service DataServer through an OpenEdge Appserver instance as shown in the following example:

   ```
   client.p
   ----------
   DEFINE VARIABLE hServer AS HANDLE NO-UNDO.
   DEFINE VARIABLE i AS logical.
   CREATE SERVER hServer.
   /*hServer:CONNECT('-AppService asbroker1 -S 5162 -H <hostname> -nsClientMinPort 5000
   -nsClientMaxPort 50100') NO-ERROR.*/
   hServer:CONNECT('-AppService asbroker1').
   i = hServer:Connected().
   message i.
   MESSAGE "Error:" ERROR-STATUS:ERROR.
   IF ERROR-STATUS:ERROR = FALSE THEN
   DO:
                 RUN server.p ON hServer .
                 hServer:DISCONNECT().
   END.
   ELSE
   MESSAGE ERROR-STATUS:GET-MESSAGE(1).
   DELETE OBJECT hServer NO-ERROR.
   output close.
   quit.
   ```

Configuring the Progress Application Server for OpenEdge agent as a self-service DataServer session

To configure the Progress Application Server for OpenEdge agent as a self-service DataServer session:

1. Create a schema holder.
2. Specify the MS SQL Server instance username (-U) and password (-P) connection parameters for the schema holder.
3. Create a Progress Application Server instance. For example:

```
pasman create -p 8811 -P 8812 -s 8813 $WRKDIR/pasds
```

Note: For more information on how to create a Progress Application server instance, see the *Progress Application Server for OpenEdge for OpenEdge: Administration Guide*.

4. Set the value of agentStartupParam in the openedge.properties file as follows to start the agent and connect to the schema holder:

```
agentStartupParam=-T "${catalina.base}/temp" -pf $WRKDIR/mss.pf
```

Note: The openedge.properties file is located in the `$WRKDIR/pasds/conf` folder. You can also connect the schema holder and database

5. Start the tomcat server using **proenv** as follows:

```
$WRKDIR/pasds/bin/tcman.bat start
```

Note: The agent log is located in $WRKDIR/pasds/logs/pasds.agent.log

6. Connect to the server code using client.p as shown below:

```
DEFINE  VARIABLE            hServer AS           HANDLE      NO-UNDO.
                                    CREATE SERVER hServer.
                                    hServer:CONNECT("-URL http://localhost:8811/apsv
-AppService nxgas -sessionModel       Session-Free").
                                    IF NOT   hServer:CONNECTED()    THEN     DO:
                                                  MESSAGE "Failed to connect to
AppServer".
                                                  RETURN.
                                    END.
                                    run server.p        on          hServer.
                                    hServer:DISCONNECT().
                                    DELETE OBJECT hServer.
```

Creating a schema holder

A schema holder contains a schema image of the MS SQL Server database that your DataServer application accesses. The schema image contains the schema for your data source—that is, its data definitions—in an OpenEdge format. Although OpenEdge 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.

Before you begin

Consider your business needs and deployment strategy when you decide where to build your schema holder. 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 Windows 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.

General schema holder creation tasks

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

1. Establish the appropriate server permissions needed for pulling the schema image into the schema holder.
2. Perform some preparatory steps.
3. Create and then connect an empty OpenEdge 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.

Authorization and authentication

When you use the DataServer to create a schema image in a schema holder, it accesses the associated MS SQL Server Database. To connect a schema holder for an MS SQL database, you must provide a valid user ID and password combination for MS SQL at connection time. Use the User ID (`-U`) parameter to provide the user ID and the Password (`-P`) parameter to provide the password. The allowed length for User ID and Password is 32 and 128 characters respectively.

Once you have set up the schema holder, the MS SQL Server privileges required for your foreign data source connection vary among users depending on their applications. For example, the user running an OpenEdge application that queries, but does not update, a table in the MS SQL Server database must connect to the MS SQL Server database with a user and password that provides at least `SELECT` privileges on the table.

In summary, the user ID and password combination required to run an application depends on:

- The MS SQL Server database tables the application accesses
- The type of access required on those tables

Note: A database administrator (DBA) must establish all user ID and password combinations within MS SQL Server.

Database authentication during foreign database connect

Users connecting from a DataServer application need to provide their foreign data source credentials to the make a connection.

Secure connection to a foreign database can be made in three different ways:

- Wait for the Data Administration tool to prompt you for your data source user ID and password
- Provide a -U and -P parameter during OpenEdge client start up
- Add user ID and password in the Connection Parameters field of your logical database and they will be picked up for processing with the foreign data source connection

The User Id and Password that is used to connect to the foreign database from OpenEdge can be encrypted by the user. The following example demonstrates a self-encryption method available to OpenEdge applications that uses an agreed upon encryption key. The user encrypts the message and the DataServer detects the encryption by virtue of an encryption key prefix that informs the DataServer to decrypt just prior to connection with the foreign data source. The ABL syntax to encrypt a connection string is shown in the following example:

```
DEFINE INPUT PARAMETER cUserID AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cPasswd AS CHARACTER NO-UNDO.
CONNECT <schema-holder> -1 NO-ERROR.
/* self service */
CONNECT <dsn> -ld <logical-database> -dt MSS
VALUE( " -U " + cUserID +
" -P " + "oech1::" +
AUDIT-POLICY:ENCRYPT-AUDIT-MAC-KEY(cPasswd)) NO-ERROR.
/* client/server */
CONNECT <dsn> -ld <logical-database> -dt MSS
VALUE( " -U " + cUserID +
" -P " + "oech1::" +
AUDIT-POLICY:ENCRYPT-AUDIT-MAC-KEY(cPasswd))
-H <host> -S <port> NO-ERROR.
```

User Id and Password are encrypted internally on behalf of the application so that clear test is never seen from the application, nor is it transmitted over a connection in clear test. This security policy is enabled by default.

Users are still able to store clear text credentials in the dictionary if they chose to do so. However, if they self-encrypt credentials placed in the dictionary, as described in the above example, the credentials will remain encrypted while in schema holder storage and will be automatically decrypted by the DataServer when authentication is needed for the foreign data source connection.

Note: The use of encryption requires that both the OpenEdge client and server are of the same OpenEdge version. Connecting a new OpenEdge 11.4 client with an old pre-11.4 server will fail to connect. Backward compatibility is lost if the new client utilizes encryption during connection to an old server.

Permissions for creating schema holder with constraints

In OpenEdge 11.0, we introduced support for **Foreign Constraints** in the DataServers for MSS and Oracle. However, from the MSS side, a part of this support involves creating a new procedure known as `_CONSTRAINT_INFO` in the foreign database during a schema pull operations (e.g. via the Create DataServer Schema utility). This procedure is created and dropped during the schema pull process.

Creating a schema holder from existing MS SQL Server database requires a user to be able to view meta-schema of the MS SQL Server Database for tables, stored procedures and views. The minimum privileges required for a user to have read-only privileges to the meta-schema is **VIEW**Definition on the Database scope. This will allow a user to create schema holder from a MS SQL Server database. For the schema pull operation to be successful, the MS SQL Database user must have **EXECUTE**, **ALTER ON SCHEMA**, and **CREATE PROCEDURE** privilages. Apart from this, the user should also have **db_datareader** and **db_datawriter** roles membership on the MS SQL server.

Once the schema holder creation is complete, revoke the **EXECUTE**, **ALTER ON SCHEMA**, and **CREATE PROCEDURE** privileges.

Note: If the SQL Server user performing the pull has limited privileges, then the pull can fail with one or more of the following errors:

- [SQL Server]The EXECUTE permission was denied on the object `_Constraint_Info` , database <test>, schema <dbo>.
- [SQL Server]The specified schema name <dbo> either does not exist or you do not have permission to use it.
- [SQL Server]Could not find stored procedure '_Constraint_Info'.

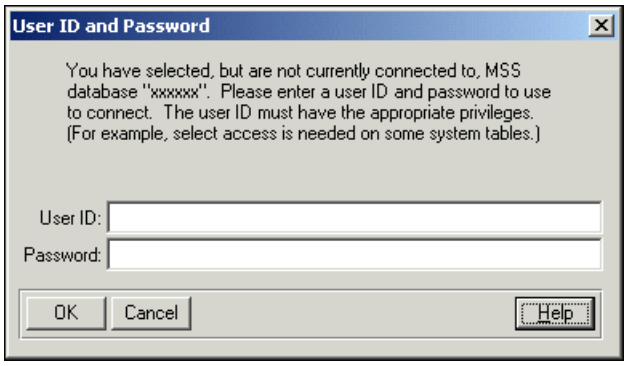
Permissions for creating or updating a schema holder

You must have privileges to use the `SELECT` statement on certain data source objects to perform certain tasks. In this chapter, these privileges are referred to as `SELECT` privileges.

Permissions for connecting a schema holder

When you connect to a schema holder, you can also choose to connect to your MS 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. The following figure shows an example of the **User ID and Password** dialog box.

Figure 10: User ID and Password dialog box



Note that value **xxxxxx** that appears in the figure above is only a placeholder for the actual MS SQL Server database name to which you intend to connect.

Application-specific permissions

In addition to the MS 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 an OpenEdge 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 select, 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

Preparing to create the schema holder

This section addresses the steps required to create the schema holder.

To prepare to create the schema holder:

1. Verify that your MS SQL Server database is accessible and that you can connect to it. Transact-SQL and the MS 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 DataServer. From your DSN configuration, select to test the connection to MS SQL Server using the ODBC interface.
4. Start the DataServer as described in either Starting a local DataServer on page 189 or Starting a remote DataServer on page 190 .
5. Open the **Data Administration** or the character **Data Dictionary**.

Creating a new empty OpenEdge database

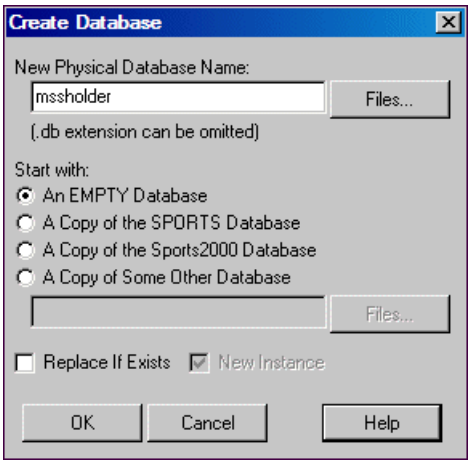
The DataServer uses the empty OpenEdge database as a holder for the schema for your MS SQL Server database. The simplest way to create an empty OpenEdge database is to do it from the Data Administration, as described in this section. For information on other ways to create an empty OpenEdge database, see *OpenEdge Data Management: Database Administration*.

If you create a new OpenEdge application to be Unicode-enabled, ensure that you create the database from a Unicode source. A copy of the empty OpenEdge UTF-8 database can be obtained from `$DLC/prolang/utf/empty`. If you choose UTF-8 as your schema image code page, your schema holder's code page must also be UTF-8. If you create a new OpenEdge application to be Unicode-enabled, ensure that you create the database from a Unicode source. A copy of the empty OpenEdge UTF-8 database can be obtained from `$DLC/prolang/utf/empty`. If you choose UTF-8 as your schema image code page, your schema holder's code page must also be UTF-8.

To create and connect an empty OpenEdge database from the OpenEdge Data Administration tool:

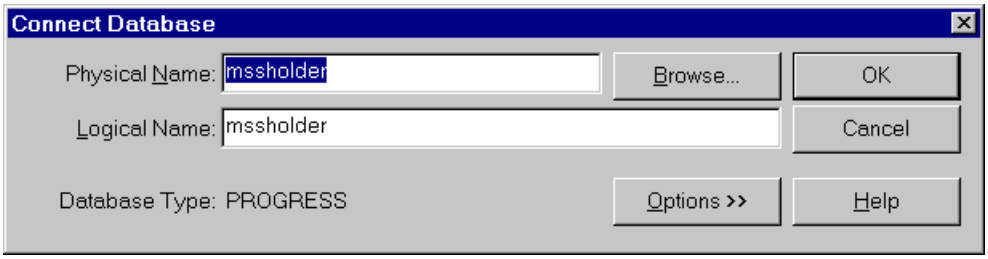
1. Start the **Data Administration** tool.
2. Select the **Database** > **Create** option.

   The following 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, as shown:



   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 OpenEdge database and return to the **Data Administration** main window.

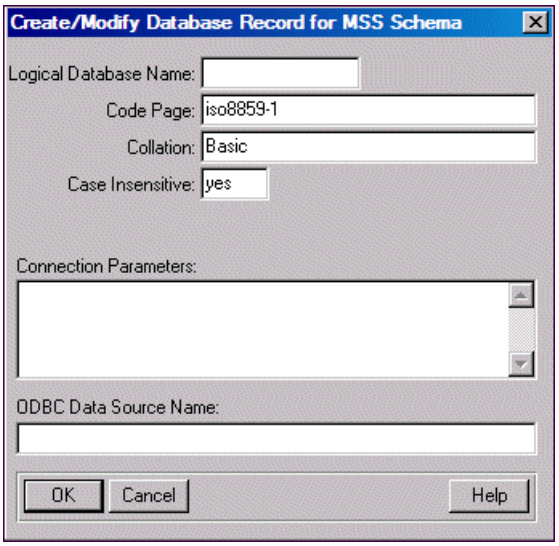
Creating a schema holder

This section addresses the steps needed to create a schema holder.

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 *OpenEdge Getting Started: ABL Essentials*.

   Note: If you place the schema from a second MS 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 OpenEdge name for the code page that the data source uses. The default is iso8859-1. If you choose UTF-8 as your schema image code page, your schema holder's code page must also be UTF-8.

   The following table lists the most common MS SQL Server database code pages and the equivalent OpenEdge names.

   Table 32: MS SQL Server and OpenEdge code pages

MS SQL Server Code page	OpenEdge equivalent
iso_1	iso8859-1 (default schema-holder code page)
cp850	ibm850

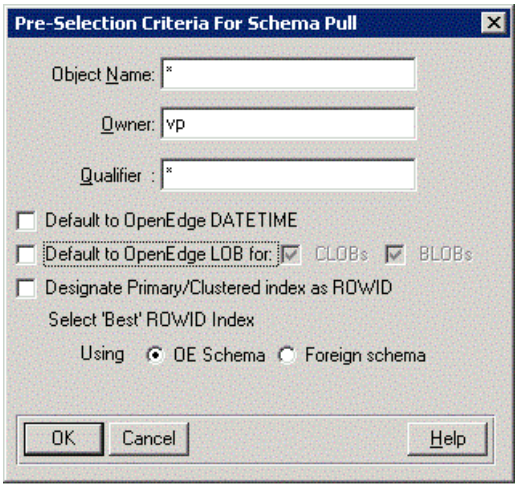
   If you use a code page that OpenEdge does not support, you must supply a conversion table that translates between the OpenEdge client code page and the code page that your data source uses. For a complete discussion of code pages, see *OpenEdge Development: Internationalizing Applications*.

4. In the **Collation** field, enter the name of the collation rule to use. The default is Basic. See Code pages on page 44 for a discussion of collation issues to consider.
5. In the **Case Insensitive** field, the default value **yes** indicates that MS SQL Server's case insensitivity feature is in use. To change this value, type **no**.
6. Type the connection parameters in the **Connection Parameters** field.

   See Connecting the DataServer on page 189 for a description of the required and optional connection parameters.

7. In the **ODBC Data Source Name** field, type the name that you used when you registered the data source with the ODBC administration tool.
8. Choose **OK**. The utility prompts you for your data source user ID and password. If they are required by the MS SQL Server data source and you did not provide the user ID and password in the **Connection Parameters** field (see Step 6 on page 182), enter a valid data source user ID and password combination now. For more information, see Authorization and authentication on page 177.
9. Choose **OK**. When the DataServer connects to the MS 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/library information, and qualifier. For example, you can specify A* in the **Object Name** field to list all the tables whose names begin with A or a.

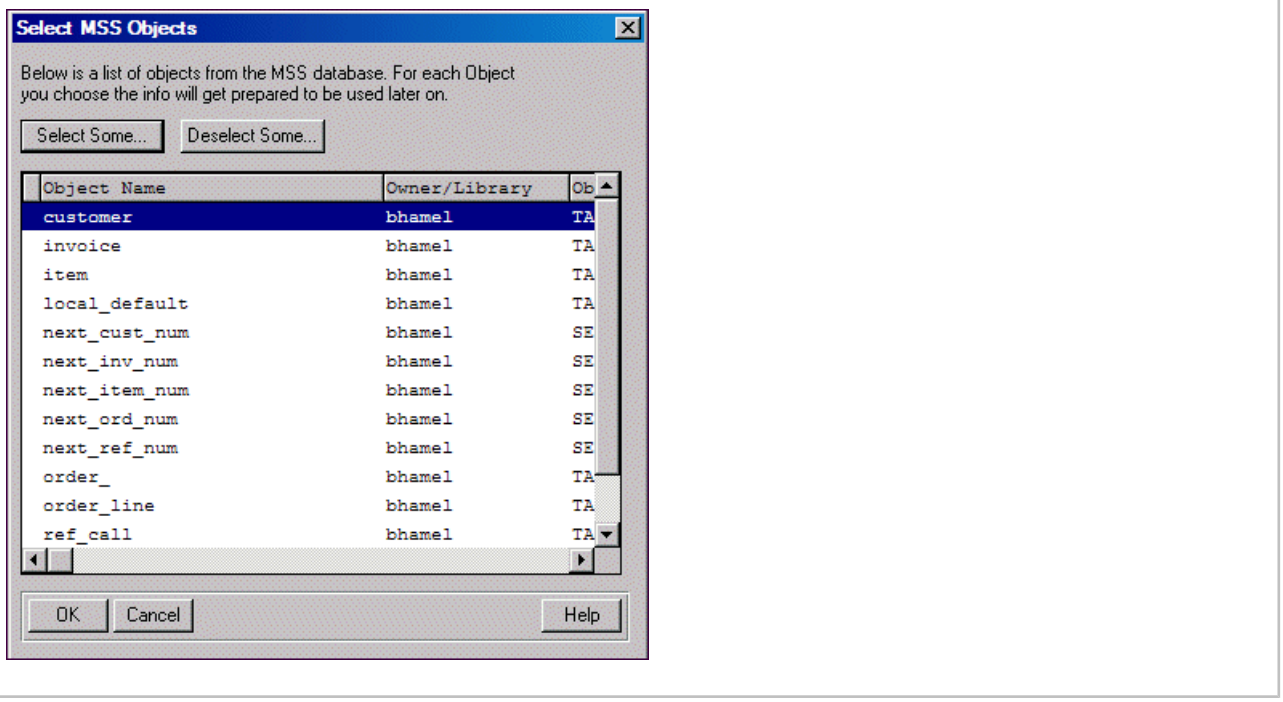
Note: Progress Software Corporation recommends that you do not specify an entry that consists exclusively of wild cards for each of the three entry fields in the dialog box. An entry that consists exclusively of wild cards might degrade the performance of the database when you perform a schema pull.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

Check **Default to OpenEdge DATETIME** to automatically map MS SQL Server data types to the associated OpenEdge data type. If you have modified your client application to handle `LOB` data types, check **Default to OpenEdge LOB for: CLOBs** and/or **BLOBs** to map the OpenEdge `LOB` data type to MS SQL Server `VARBINARY (MAX)`, `IMAGE` and `FILESTREAM` data types. For more information on mapping OpenEdge and MS SQL Server data types, see Support for OpenEdge ABL BLOB data type on page 72 . If you select **Designate Primary/Clustered index as ROWID**, you will get the warning "Existing ROWID designations may be overwritten by selecting this option". If selected, and a given table has a clustered index and does not have the `PROGRESS_RECID` field defined for it, it will be used as ROWID if it qualifies for ROWID selection. Qualification requires that the index be unique. This designation for ROWID takes precedence over any other options selected for the schema pull operation but does not overwrite a ROWID designation that is based on the presence of `PROGRESS_RECID`. When the **Select 'Best' ROWID Index is** selected, this option provides a legacy option for selecting ROWID when the `PROGRESS_RECID` column is not present in the table. When selected in conjunction with new process flow options for ROWID migration, this option plays a secondary role in the designation of ROWID indexes deferring to the **Designate Primary/Clustered index as ROWID** option as the first choice. This option searches for a viable index for ROWID when an `PROGRESS_RECID` column does not exist and other process flow options do not render a viable index.

For more information on mapping OpenEdge and MS SQL Server data types, see Support for OpenEdge ABL BLOB data type on page 72 and Support for OpenEdge ABL CLOB data type on page 71 .

10. Choose **OK**.

    OpenEdge displays a list of the data source objects that you can include in the schema holder, as shown:



    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.

11. Click the option appropriate to the action you want to perform:

    - **Select Some** — Displays the **Select by Pattern Match** dialog box on which you can specify object information used to select objects.
    - **Deselect Some** — Displays the **Deselect by Pattern Match** dialog box on which you can specify object information used to deselect objects.

    You can also elect to select and deselect individual objects by clicking and double-clicking on an object. An asterisk appears next to an object that has been selected; double-click an object to remove the asterisk and identify that the object is now deselected.

12. Choose **OK** after you have identified all the objects you want to include in the schema holder. 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 selected, the DataServer attempts to select an index to support the OpenEdge `ROWID`. If an appropriate index does not exist, the DataServer issues the warning, **Please check errors, 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 Data Dictionary. See Defining the ROWID on page 312 for instructions. For additional information, see Indexes and sorting on page 49.

Maintaining a schema holder

The Data Administration provides a set of DataServer utilities that you can use to maintain a schema holder. The DataServer Tutorial on page 227 describes these utilities.

Note: Keep in mind that you must have SELECT privileges on certain data source objects to update a schema holder. See Authorization and authentication on page 177 for details.

If you make changes to a MS 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.

Updating a deployed schema holder

The guidelines and techniques that apply to deploying an OpenEdge database also apply to deploying a schema holder for a MS 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.

To update a deployed DataServer application:

1. Run an SQL script to make changes to the MS 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 PROCOPY utility to distribute them.

Typical configuration for a remote client to connect to a remote DataServer

A typical configuration to connect a remote client to a remote DataServer requires the following general tasks:

- Configuring an ODBC data source on page 186
- Starting a broker on page 186
- Starting the schema holder to accept remote client connections on page 186
- Connecting remote clients to the MS SQL Server DataServer broker on page 187

This section highlights the basic procedure associated with each of these tasks.

Configuring an ODBC data source

An ODBC data source allows you to store information about how to connect to a data provider. You use the ODBC Data Source Administrator to configure an ODBC source. Note that the DataServer license must be installed on the machine where the ODBC data source is configured.

To configure an ODBC data source:

1. In Windows, choose**Start** > **Control Panel** > **Administrative Tools** > **Data sources (ODBC)** and select **Systems DSN**.

   The **System DSN Data Source Administrator** appears.

2. Choose **Add** to add a data source with either the MS SQL Server driver or an OpenEdge branded ODBC Wire Protocol driver for MS SQL Server. Proceed through the windows as they are presented to configure the data source, ensuring that the final test connection is successful.

Starting a broker

You can start the DataServer broker, using various methods supported by the Unified Broker Framework, including:

- OpenEdge Management/OpenEdge Explorer or `mergeprop` utility (for a Unified Broker)
- The command line (for a probroker)

The following example shows how to perform this step from the command line. For information about using the mergeprop utility, see *OpenEdge Getting Started: Installation and Configuration*. For information about using OpenEdge Management or OpenEdge Explorer, see Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190, and see *OpenEdge Getting Started: Installation and Configuration*.

To start a DataServer broker from the command:

1. Choose **All Programs** > **OpenEdge** > **Proenv**.

   The **proenv** prompt appears.

2. Type the following example command line to start the DataServer broker:

```
_probrkr -H hostname -S brokerservice -N tcp
```

   The values for *`hostname`* and *`brokerservice`* must be unique.

   For more details about using either OpenEdge Management/OpenEdge Explorer or the command line to start a unified broker process for the DataServer, see Configuring a remote DataServer on page 164.

Starting the schema holder to accept remote client connections

The connection between the schema holder and a remote client allows you to access data definitions stored in the schema holder. The following example shows how to service the schema holder to accept a remote client connection.

Using the same Progress PROENV shell as you did in Step 4 of Starting a broker on page 186, type the following example command line to start the DataServer connection:

```
proserve schemaholdername -H hostname -S databaseservice -N tcp
```

The value for *databaseservice* must be unique to the connection you are making to the schema holder.

Connecting remote clients to the MS SQL Server DataServer broker

The activity described in this section concludes setting up a typical configuration for a remote client to connect to a remote DataServer.

Note: For a description of each parameter identified in this section, see the table in Connecting a schema holder at startup on page 197.

Type the following command line entries for a remote Character or a remote UNIX client machine:

```
_progres schemaholdername -H hostname -S databaseservice -N tcp -db dsn_name -dt MSS
-ld logical-datasrc-name -H hostname -S brokerservice -N tcp -U userID -P password
```

Type the following command line entries for a remote GUI client machine:

```
prowin32 schemaholdername -H hostname -S databaseservice -N tcp -db dsn_name -dt MSS
-ld logical-datasrc-name -H hostname -S brokerserice -N tcp -U userID -P password
```

For details about the database connection parameters noted in each example in this procedure, see Connecting a schema holder at startup on page 197.

6

Connecting the DataServer

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

- OpenEdge Management or OpenEdge Explorer (in Windows and UNIX platforms)
- Mergeprop utilities (command-line alternative to OpenEdge Management or OpenEdge Explorer)
- OpenEdge Management or OpenEdge Explorer Command line utilities (MSSMAN and MSSCONFIG)
- Command line statements (`_probrkr.exe`)

This chapter describes how to connect the DataServer using these methods.

For details, see the following topics:

- Starting a local DataServer
- Starting a remote DataServer
- Connection guidelines
- Starting ProBroker distributed DataServer broker with TLS-enabled Progress Networking
- Connecting a schema holder
- Connection failures and OpenEdge responses

Starting a local DataServer

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

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

Use the following steps to start the client:

1. Make sure that your ODBC drivers are installed and configured properly.
2. Start your MS SQL Server database.
3. Set any environment variables required for your configuration as described in the relevant section of Configuring the DataServer on page 161 (for example, `DSLOGDIR`). You must set them in the environment from which you are starting the DataServer.

   Note: Note that if you change the values of any environment variables, you must shut down the DataServer processes and restart them.

4. Start the client by entering the following command:

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

For example, the following command starts OpenEdge with the local DataServer, connects a local schema holder named mssholder in single-user mode, and connects the MS SQL Server demo data source with the user `bob` whose password is `bobpass`:

```
prowin32 -1 mssholder -db demo -dt MSS -ld demo -U bob -P bobpass
```

Starting a remote DataServer

This section describes how to start and stop a remote DataServer for MS SQL Server. Starting a remote DataServer involves starting the processes that make up the remote DataServer.

To start a remote DataServer:

1. Start the DataServer broker process on your host machine. For details, see:
   a) Starting and stopping a broker process from the command line on page 193
   b) Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190
2. Start an OpenEdge client process on a UNIX machine or in a PC running Windows that connects to the schema holder and the MS SQL Server database.

Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client

You can use OpenEdge Management or OpenEdge Explorer to administer the server module (`_msssrv.exe`) of the DataServer in Windows and UNIX.

Performing from the Windows host

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

To start and stop the DataServer from OpenEdge Management or OpenEdge Explorer, see the OpenEdge Management or OpenEdge Explorer online help.

Performing on the client

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

- Include the `-Dsrv SVUB,1` parameter. This parameter allows you to connect to the broker administered by OpenEdge Management or OpenEdge 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.

  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` or add the `-DirectConnect` parameter as follows:

  ```
  -DataService none
  ```

  Or

  ```
  -DirectConnect
  ```

- 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 OpenEdge Management or OpenEdge 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 OpenEdge Management or OpenEdge Explorer (when the `-DataService` value is `none`)
- Set the `-H` parameter to the name of the 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 OpenEdge broker and the `-H` and `-S` parameters are used to locate a ProBroker 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.
```

Starting and stopping a broker process using MSSMAN

You can use the MSSMAN utility to start a DataServer broker on a Windows host. Prior to using MSSMAN, verify that the AdminService is running.

Use the following steps to start and shut down the DataServer:

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.

   For more information about the MSSMAN utility and a description of all the command options, see Server Related Command Line Utilities and Startup Parameters on page 353.

Starting and stopping a broker process from the command line

This section addresses how you can start and stop a broker process from the command line.

To start and stop a broker process from the command line:

1. Set the environment variable `MSSSRV` to the name of the executable (including the path) of the DataServer for MS 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 Windows host machine. Select a value for *`service-name`* from the list of available services in your `\`*`OpenEdge-install-dir`*`\system32\drivers\etc\services` file:

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

   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
   ```

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

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

After you start the Windows broker process, you are ready to start an OpenEdge client on a PC running in Windows or on a UNIX machine. See Starting the Windows client process on page 193 or Starting the UNIX client process on page 194 for instructions.

Starting the Windows client process

Start the OpenEdge 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 OpenEdge client process. On the command line for the program icon, enter the following information:

- The executable
- The schema holder name
- The connection parameters required by the remote DataServer configuration

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

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

Create a program icon for each unique set of connection parameters. See Connecting a schema holder at startup on page 197 for command-line information and more examples.

Starting the UNIX client process

After starting the remote broker, you start the OpenEdge client process on a 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 OpenEdge client, connects a read-only schema holder named `mssholder`, and connects the MS SQL Server demo database using the `mssdemo` DSN with the user `bob` whose password is `bobpass`:

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

See Connecting a schema holder at startup on page 197 for descriptions of the required command line.

Connection guidelines

When you create a schema holder, you specify certain connection information, such as the physical and logical names for your MS SQL Server database and the user and password information. OpenEdge stores this information so that you do not have to provide it each time you connect. After you consider your security requirements, proceed to connect the schema holder and then the data source.

Connections and security in Windows

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

- **Standard** — Requires that a client pass a user ID and password that the MS 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 OpenEdge client or WebSpeed agent passes this information with the User ID (`-U`) and Password (`-P`) connection parameters.
- **Integrated** — Relies on the Windows operating system to validate user information. The MS 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 MS 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 MS SQL Server database accepts the connection.

- If the connection is nontrusted, the OpenEdge client must provide the user ID and password.

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

- Configure a MS SQL Server database to use standard or mixed security if you are using remote OpenEdge clients.
- If you are using mixed security, always have the clients specify the `-U` and `-P` connection parameters.

Starting ProBroker distributed DataServer broker with TLS-enabled Progress Networking

The schema holder can run on the client or can optionally be located on any other host that is accessible on your network.The network can be configured to use a TLS or a non-TLS connection irrespective of remote or local DataServer configuration.The schema holder can also be established using TLS network protocol by using “-ssl” command line parameter as shown in the following example:

```
_mprosrv -S service-name -ssl
```

The dataServer client should also be instantiated with the “-ssl” parameter if you also wish to establish a DataServer broker connection over a TLS enabled Progress networking, as follows:

```
prowin32 -1 SHDB -S 12345 -ssl
db my_dsn -ld ldb -dt MSS -U mb -P mb -S ccall7sv1 -ssl
```

Note: The following TLS connections are established using the above instruction:

- Schema holder DB and Progress Client
- OE DataServer Client and DataServer Server

The following instructions use default executable names. If you built custom executables, substitute those names. When creating a schema holder, the startup syntax is the same whether you are running a local DataServer or using MS SQL Server Networking.

Non-default TLS certificates

The Unified Broker Framework has the “certStorePath” TLS property to specify the path to the trusted certificate store which is the root certificate for the certificate authority that was used to sign the server certificate. Servers using this customized certificate can connect clients instantiated with a matching certificate specified using `-certstorepath` command line parameter of the client.

However, in a Probroker configuration, responsibility for spawning a TLS or a non-TLS DataServer servers rests with the Probroker based on whether it was instantiated with a TLS or a non-TLS command line parameter. A TLS-enabled ProBroker can include the following startup parameters to customize the brokers use of certificates and key stores.

Table 33: DataServer connection parameters

Parameter	Description
-certstorepath	The path to the trusted certificate store. The root certificate for the certificate authority that was used to sign the server certificate should be specified by this path.
-keystorepath	Specifies the key store location.
-keyalias	Specifies the alias set for a private/public key. If this parameter is not provided, the server certificate alias that is provided by OpenEdge is used.
-keyaliaspasswd	Specifies the encoded password set for the alias of the private/public-key to access the server Private Key / Digital Certificate key store entry. You can use the OpenEdge provided genpassword utility for encoding your passwords

For example a Probroker can be started and customized in TLS mode as follows:

```
_probrkr -S ccall7sv2 –ssl
-certstorepath
E:\\users\\<currentuser>\\par\\SHA224\\certs
-keystorepath
E:\\users\\<currentuser>\\par\\SHA224\\keys
-keyalias localhost
-keyaliaspasswd 20333c34252a2137
```

The above Probroker can establish connection to a client which uses a matching certificate:

```
prowin32 -pf parametersFile.pf
-certstorepath E:\\users\\ashukla\\par\\SHA224\\certs
-nohostverify
```

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, OpenEdge provides the following techniques for connecting to a schema holder:

- Use the Data Dictionary or Data Administration. 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 MS SQL Server database as your working database and connect to it. You cannot use the **Utilities** > **Edit OpenEdge Auto-Connect** option to connect to a MS SQL Server data source.
- Use the ABL `CONNECT` statement (see its reference entry in *OpenEdge Development: ABL Reference*). A `CONNECT` statement must first list the schema holder and related connection parameters, then the MS SQL Server database and related parameters.

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

```
CONNECT mssholder -1 -db mssdemo -dt MSS
-Dsrv qt_debug,EXTENDED
```

- Use connection parameters when you start OpenEdge. You can include these parameters on the program item command line or in a parameter file that you specify when you start OpenEdge. A parameter file is portable and easy to maintain. For information on creating a parameter file, see *OpenEdge Deployment: 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 OpenEdge `CONNECT` statement. If the proper connect information is stored in the schema holder, referencing a table in the MS SQL Server database will cause the DataServer to issue the connect statement automatically. Any combination of connection techniques works, as long as you first connect to the schema holder and then the MS SQL Server data source.

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

If you connect to the schema holder and the MS 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 MS SQL Server database connection after the `-db` parameter.

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

Connecting a schema holder at startup

OpenEdge supports connection parameters that you can use to connect both the OpenEdge schema holder and a MS 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.

The following table describes the database connection parameters that you use when you connect to a schema holder and a MS SQL Server database through the DataServer.

Table 34: DataServer connection parameters

Parameter	Status	Description
Database Type MSS (`-dt MSS`)	Optional	Specifies that the type of the target data source is MS SQL Server. If not specified, the server will resolve the database type.
Physical Database Name (`-db`)	Required	Indicates the name by which OpenEdge recognizes the MS 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 MS SQL Server database.

Parameter	Status	Description
Logical Database Name (-ld)	Optional	Specifies the logical name of the MS 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 MS SQL Server demo database as mydemo. In this case, the physical name is demo, and the logical name is mydemo. If not specified, the logical database name is assumed to be the same as the physical name. An error will be generated if the physical name and logical name do not match when this parameter is omitted.
Host Name (-H)	Required for remote DataServer	Indicates the name of the Windows 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 OpenEdge Management or OpenEdge Explorer, specify the service name or port number on the host machine where the NameServer resides. If you are using ProBroker or OpenEdge Management/OpenEdge Explorer without a NameServer, specify the service name or port number on the host machine where the broker resides.
User ID (-U)	Required	Supplies the login name that the DataServer uses to log into the MS SQL Server database.
Explorer Redirection (-Dsrv SVUB,1)	Required for OpenEdge Management or OpenEdge Explorer connections	Redirects connection logic to OpenEdge Management or OpenEdge Explorer instead of ProBroker. This parameter must be set if the -DataService connection parameter is to also be used for connectivity. For more information see Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190. Also refer to Using MS SQL Server and DataServer options on page 340.
Data Service (-DataService)	Required for OpenEdge Management or OpenEdge Explorer connections	Specifies the data service the NameServer uses. For more information, see Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190.

Parameter	Status	Description
Direct Connect (-DirectConnect)	Optional. For use with OpenEdge Management or OpenEdge Explorer connections	Instructs the client to bypass the controlling NameServer and connect directly to the MS SQL Server DataServer Broker. If not specified, the connection is made through the NameServer.
Password (-P)	Required	Supplies the password that the DataServer uses to log into the MS 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. If not specified, -RO or remote connection parameters are required.
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. If not specified, the database is opened for update as well as read access.
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. If not specified, no cache file is used.

Parameter	Status	Description
DataServer (`-Dsrv`)	See note	Specifies options with which you control your ODBC Driver and DataServer environment. See DataServer options on page 337 and Using MS SQL Server and DataServer options on page 340 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 the query of an OpenEdge database. Use `-nojoinbysqldb` at startup time. If not specified, the server will perform joins, but results may not be consistent with identical joins run against an OpenEdge database.

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 *OpenEdge Deployment: Startup Command and Parameter Reference*.

Using a local DataServer configuration

Use the following general command syntax to start OpenEdge:

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

```
prowin32 schema-holder-name -1 -db datasrc-name -dt MSS
  -ld mss-logical-dbname -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 OpenEdge in a local DataServer configuration:

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

The examples in the first part of this section show how to use specific command syntax to start OpenEdge in the following modes:

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

Note the following command syntax used in a Windows client:

Synrax

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

Note the following command used on a UNIX client:

Syntax

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

The examples in the second part of this section allow you to start OpenEdge in a remote DataServer configuration:

- 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 user ID is bob.
- The password is bobpass.

Note the following command used in a Windows client:

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

Note the following command used on a UNIX client:

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

Unsupported connection parameters

You cannot use the following OpenEdge connection parameters when connecting to a MS SQL Server database through the DataServer. If you do specify them, OpenEdge 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)

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 DataServer for MS SQL Server passes the connection string specified by the PRGRS_CONNECT option directly through to the ODBC driver without modification. For more information and a complete list of keywords, refer to the Microsoft ODBC programming reference documentation. 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 (;). There can be no spaces within *connection-string*. If any of the name/value pairs does include a space, substitute the value &^ for the space.

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;UID=name;PWD=password;.
```

  For *datasrc-name*, supply the name of the MS 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 MS SQL Server database whose name has a blank space, which is not allowed by OpenEdge, substitute the characters &^ for the illegal characters in the data source name. OpenEdge 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 MS 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,;.
```

To simplify your PRGRS_CONNECT string, you can specify a File DSN. When you configure access to your MS SQL Server database, specify a File DSN and then reference that file in your connect string. For example:

```
CONNECT datasrc-name -ld logical-name -dt MSS
   -Dsrv PRGRS_CO,FILEDSN=dsn-filename;.
```

Binding with connection and startup parameters

Binding settings determine how data is pulled from the data source by the DataServer. When binding is turned on, query results are copied directly into the DataServer's memory, eliminating the need to do post-query calls to retrieve the data. When binding is not enabled, queries run independent of the data retrieval, and subsequent calls to the data source are required to retrieve the data.

The default setting is to use binding for both NO-LOCK and EXCLUSIVE-LOCK queries. You can override the default using the -Dsrv BINDING, nconnection parameter. The syntax is:

Syntax

```
-Dsrv BINDING,n;
```

The following table describes the BINDING options that you can specify with the -Dsrv parameter.

Table 35: Connection BINDING options

BINDING value	Description
0	Indicates that binding is not enabled at all.
1	Use binding for NO-LOCK queries only.
2	Use binding for EXCLUSIVE-LOCK queries only.
3	Use binding for both NO-LOCK and EXCLUSIVE-LOCK queries. This is the default value.

Query tuning with connection and startup parameters

You can control aspects of DataServer query handling not only programmatically within ABL 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 Query tuning on page 145 for more information.

You override query-tuning defaults with the DataServer (-Dsrv) connection parameter when you connect to a MS 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:

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

The following table describes the query-tuning options that you can specify with the -Dsrv parameter.

Table 36: Connection query-tuning options

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 record the SQL sent to the ODBC driver in the dataserv.lg file. Note that this SQL contains place holders for values which will be bound to the statement when sent to the data source • 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 Using qt_debug to Analyze Performance on page 369. For information on Enhanced Logger options that replace the qt_debug options, see Analyzing application execution with Enhanced Logger on page 206.
qt_lookahead qt_no_lookahead	Specifies whether the DataServer uses lookahead or standard cursors. To generate efficient queries, qt_lookahead is the default in the following cases: • Statements that use NO-LOCK • Statements that use SHARE-LOCK with transaction isolation level set to read uncommitted Specify qt_no_lookahead for query behavior that is consistent with an OpenEdge database.

Option	Description
qt_separate_connection qt_no_separate_connection	Specifies whether each cursor should use a separate connection to the MS SQL Server database. The default is qt_no_separate_connection, which provides behavior that is consistent with an OpenEdge database. 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: 10,000 when block cursors are enabled (the default). 30,000 when block cursors are disabled.

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, as shown:

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

OpenEdge 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 OpenEdge 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 queries run against an OpenEdge database. See Initial Programming Considerations on page 41 for more information.

Analyzing application execution with Enhanced Logger

The Enhanced Logger provides a means for either standard reporting of runtime activity or logging diagnostic data for troubleshooting problems based on a set of logging characteristics. The DataServer can be configured to write information to a log file. You control the type and the degree of detail logged using a combination of log entry types and logging levels. Log entry types are categories of information written to the log file. Logging levels control the degree of logging detail for any given log entry type.

The legacy qt_debug logging infrastructure only logged server information but the Enhanced Logger framework for DataServers offers two logging contexts:

- Server context for Server-specific log entries
- Client context, which shares logging information with OpenEdge clients and AppServer agents.

Note: Both the server and the client contexts for DataServers include some log entry types that work in concert with one another. You can yield a detailed understanding of the DataServer operations by merging the contexts into one view of DataServer log activity.

LOG-MANAGER system handle attributes and methods

The LOG-MANAGER system handle controls logging settings for the OpenEdge client and DataServer client contexts. The LOG-MANAGER attributes and methods allow you to programmatically control the logging levels and log entry types during the execution of your program.

The following table lists the function of each LOG-MANAGER system handle attribute.

Table 37: LOG-MANAGER system handle attributes

Attribute	Access	Function
ENTRY-TYPES-LIST	Readable	Returns a list of all valid entry types.
LOG-ENTRY-TYPES	Readable/Writable	Returns/Writes a list of one or more types of log entries.
LOGFILE-NAME	Readable/Writable	Returns/Writes the name of the log file.
LOGGING-LEVEL	Readable/Writable	Returns/Writes the level at which log entries are written.
LOG-THRESHOLD	Readable	Returns the file size threshold of log files. When the current log file becomes equal to or greater than the specified size, OpenEdge renames and saves the log file and creates a new log file.
NUM-LOG-FILES	Readable	Returns the number of rolled over log files to keep on disk at any one time, for OpenEdge session, including the current log file.
HANDLE	Readable	Returns a handle.
INSTANTIATING-PROCEDURE	Readable	Returns a procedure handle for the first, or instantiating, procedure.
TYPE	Readable	Returns the type of handle.

The following table lists the function of each LOG-MANAGER system handle method.

Table 38: LOG-MANAGER system handle methods

Method	Function
CLEAR-LOG( )	Clears all messages existing in the current client log file and leaves the file open for writing

Method	Function
WRITE-MESSAGE( )	Writes user messages to the current log file
CLOSE-LOG( )	Stops an interactive or batch OpenEdge client from writing messages to the current client log file and closes the log file

Note: The CLOSE-LOG( ) method is valid only for interactive and batch clients. WebSpeed agents and AppServer servers write a message to the server log file indicating that it is invalid to use the CLOSE-LOG( ) method to close a WebSpeed or AppServer server log file. DataServer connections established by Webspeed or AppServer agents all share the same server log file. So, you cannot execute the CLOSE-LOG( ) method unless there is only one agent with an open DataServer connection. In this case, the method returns FALSE.

For more information on the LOG-MANAGER system handle or its attributes and methods, see the LOG-MANAGER system handle reference entry in *OpenEdge Development: ABL Reference*.

DSLOG-MANAGER system handle attributes and methods

The DSLOG-MANAGER system handle is similar to the LOG-MANAGER system handle. Is is used to control the logging settings for the server context of the OpenEdge DataServer logging infrastructure, while the LOG-MANAGER controls logging settings for OpenEdge Client and DataServer client context. Although both provide attributes and methods for enabling, disabling, and changing logging capabilities, the DSLOG-MANAGER applies only to DataServer components.

The DB-CONTEXT attribute must be set before any other DSLOG-MANAGER attribute can be accessed.

The following table lists the DSLOG-MANAGER system handle attributes.

Table 39: DSLOG-MANAGER system handle attributes

Attribute	Access	Function
DB-CONTEXT[38]	Readable/ Writable	Specifies the logical database name of a valid and connected data source for a supported db-type. It set the system handle to a specific server context to enable access to other DSLOG-MANAGE attributes and methods. You can also set it to the alias of a connected data source.
ENTRY-TYPES-LIST	Readable	Returns a list of all valid entry types.
LOG-ENTRY-TYPES	Readable/ Writable	Returns/Writes a list of one or more types of log entries. The default value returned by the LOG-ENTRY-TYPES attribute are the logentrytypes specified in the -Dsrv parameter when the foreign data source was connected.
LOGFILE-NAME[39]	Readable	Returns the name of the log file. By default, the DSLOG-MANAGER uses the dataserv.lg log file. You can change the file name by using the -dslog startup parameter from the OpenEdge client or the srvrDSLogFile UBROKER property from the DataServer broker instance. The server context cannot work without a log file opened so it cannot be changed at runtime. This would risk having no log file opened in the event that the new log file name failed to open.

Attribute	Access	Function
LOGGING-LEVEL	Readable/ Writable	Returns/Writes the level at which log entries are written. The default value returned by the LOGGING-LEVEL attribute is the logginglevel specified in the -Dsrv parameter when the foreign data source was connected.
LOG-THRESHOLD[40]	Readable	Returns the file size threshold of log files. When the current log file becomes equal to or greater than the specified size, OpenEdge renames and saves the log file and creates a new log file.
NUM-LOG-FILES[41]	Readable	Returns the number of rolled over log files to keep on disk at any one time, for OpenEdge session, including the current log file.
HANDLE	Readable	Returns a handle.
INSTANTIATING-PROCEDURE	Readable	Returns a procedure handle for the first, or instantiating, procedure.
TYPE	Readable	Returns the type of handle.

By default, the DSLOG-MANAGER uses the dataserv.lg log file. You can change the file name by using the -dslog startup parameter from the OpenEdge client or the srvrDSLogFile UBROKER property from the DataServer broker instance.

The following example tests for an active connection before setting the DSLOG-MANAGER to that connected DataServer context and getting a list of log entry types and setting a specific message inside the application at runtime.

```
IF CONNECTED("mymss") THEN DO:
    DSLOG-MANAGER:DB-CONTEXT = 'mymss'.
    MESSAGE DS-LOGMANAGER:LOG-ENTRY-TYPES.
    DSLOG-MANAGER:WRITE-MESSAGE ("Some message after connection").
END.
```

For more information on the DSLOG-MANAGER system handle or its attributes, see the DSLOG-MANAGER system handle reference entry in *OpenEdge Development: ABL Reference*.

[38] Each connection to a DataServer is its own entity. The DB-CONTEXT attribute enables you to provide different logging settings for each data source connection. The DB-CONTEXT attribute must be set to the logical database name or alias which is currently connected, prior to setting any other attributes. The attribute defaults to the unknown value. If DB-CONTEXT does not contain the value of a valid logical database name, any attempt to access other attributes of DSLOG-MANAGER fails.

[39] The value returned by the LOGFILE-NAME is the default name dataserv.lg or; the name specified by the -dslog client startup parameter for self-service clients; the name specified by the -dslog startup parameter when ProBroker is started; the name specified by the srvrDSLogFile attribute in the ubroker.properties file when a Unified Broker instance is started for DataServer connections.

[40] The value returned by the LOG-THRESHOLD is one of the following:the name specified by the -logthreshold client startup parameter for self-service clients; the name specified by the -logthreshold startup parameter when ProBroker is started; the name specified by the srvrLogThreshold attribute in the ubroker.properties file when a Unified Broker instance is started for DataServer connections.

[41] The value returned by the NUM-LOG-FILES is one of the following: the name specified by the -numlogfiles client startup parameter for self-service clients; the name specified by the -numlogfiles startup parameter when ProBroker is started; the name specified by the srvrNumLogFiles attribute in the ubroker.properties file when a Unified Broker instance is started for DataServer connections.

The following table lists the function of each DSLOG-MANAGER system handle method.

Table 40: DSLOG-MANAGER system handle methods

Method	Function
CLEAR-LOG( )	Clears all messages existing in the current client log file and leaves the file open for writing
WRITE-MESSAGE( )	Writes user messages to the current log file

Note: The CLOSE-LOG( ) is not a supported DSLOG-MANAGER system handle method. You cannot execute the CLOSE-LOG( ) method unless there is only one brokered DataServer server spawned by the Unified Broker or by ProBroker.

Log entry types

Log entry types represent categories of information you can write to a log file. You can categorize the information into server and client contexts. The log entry types in the client context are applicable to DataServer clients, ABL clients, AppServer agents and WebSpeed agents. However, the log entry types in the server context are applicable only to DataServer brokers.

The degree of detail written to the log file is determined by the logging level. For more information on logging levels, see Logging levels on page 215.

The following table lists the DataServer log entry types in the client context.

Table 41: DataServer Log entry types (Client context)

Log entry type	Description
DS.QryInfo	Logs messages about client processing for SQL queries. By default, setting the DS.QryInfo log entry type in the client context will automatically set the SQL log entry type in the server context.
DS.Cursor	Logs client-side cursor details. By default, setting the DS.Cursor log entry type in the client context will automatically set the Cursor log entry type in the server context.
4GLTrans	Logs client-side transaction details. Level 4 of the entry type writes client context information to the log file specific to DataServers. Output from the 4GLTrans log entry type can supplement output of the TRANS log entry type in the server context.
DB.Connects	Logs client-side connection details. Level 3 of this log entry type logs the -Dsrv connection switch values in the server context.

Log entry type	Description
DS.Performance	Logs client-side performance details. By default, setting the DS.Performance log entry type in the client log context sets the Performance log entry type in the server context at the corresponding logging level. Level 3 of DS.Performance log entry type reports the performance summary for the interfaces invoked in a client session when a user disconnects from the logical database. Level 4 logs the detail performance in the client log context. Level 2 is an aggregate of the performance summary for the calls made at DataServer client interface layer. **Note:** The time measurement of performance parameters reported in the dataserv.lg is in microseconds (denoted by **us**).
DS.Interface	Logs client-side interface details. By default, setting the DS.Interface log entry type in the client log context sets the Interface log entry type in the server context at the corresponding logging level. Level 3 of DS.Interface log entry type reports the interface summary when user disconnects from the logical database. Level 4 log entry type logs **each** interface call in the client log context. Level 2 is an aggregate of the interface summary for the calls made at DataServer client interface layer.

For details about log entry types of the DataServer client context and other log entry types specific to the client context, refer to *OpenEdge Development: Debugging and Troubleshooting*.

The following table lists the DataServer log entry types in the server context.

Table 42: DataServer Log entry types (Server context)

Log entry type	Description
SQL	Logs information about SQL queries. Similar to the legacy qt_debug,SQL option. • Level 2 (Basic) — Logs details of SQL query preparation and execution. • Level 3 (Verbose) — Adds bound values when prepared and/or executes SQL utilize bind parameters.
Cursor	Logs information about cursor details. Similar to the legacy qt_debug,Cursor option. • Level 2 (Basic) — Logs basic cursor details such as table number, client and server cursor identifiers.

Log entry type	Description
`Trans`	Logs information about the transaction details. • Level 2 (Basic) — Logs messages when a SQL transaction begins or ends on the server or when a transaction is undone against the MS SQL Server and Oracle database. • Level 3 (Verbose) — Adds messages for the beginning and ending of sub-transactions and when a transaction is undone. • Level 4 (Extended) — Logs details about records locked in a transaction including the type of lock, lock upgrades, etc.
`Connects`	Logs details of connection attributes and related information. Mapped to the connection information formerly output with `legacy qt_debug`,`SQL` and `qt_debug,Cursor`. • Level 2 (Basic) — Logs details about connection parameters set before and after connection. Logs connection pooling information for firehose cursors. Assigns a connection ID for connection identification in the log file.

Log entry type	Description
`Performance`	Logs details of query performance. • Level 2 (Basic) — Logs a summary of the measure of performance in terms of time spent in the DataServer server layer versus time spent on the database side of the ODBC Application Program Interface. • Level 3 (Verbose) — Aggregates the measure of performance by accumulating totals for all the calls of an individual interface. The logs are of relatively higher detail than the logs generated by Level 2 (Basic). • Level 4 (Extended) — Logs messages for every individual round trip, that is, every time the DataServer client calls the server in a single session or during the time period in which dynamically logging was enabled. **Note:** Performance and Interface totals complement each other at each logging level. **Note:** The time measurement of performance parameters reported in the `dataserv.lg` is in microseconds (denoted by **us**).
`Interface`	Logs function call details. • Level 2 (Basic) — Logs the aggregate of the interface parameter's measure for the calls made within an Interface layer on the server. **Note:** Performance and Interface totals complement each other at each logging level. • Level 3 (Verbose) — Logs messages for every function call with the recurrence count in the server context in an interface. • Level 4 (Extended) — Logs messages for every function call with the recurrence count in the server context in an interface. The logs are of relatively higher detail than the logs generated by Level 3 (Verbose).

The following table identifies the logging levels and entry types for client and server contexts.

Table 43: Setting logging levels and log entry types

Operation Mode	Logging Option	OE Client	MSS DataServer Server
Startup / connection	Logging level	`-logginglevel, parameter`	`-Dsrv logginglevel, parameter`
	Log entry types	`-logentrytypes, parameter`	`-Dsrv logentrytypes, parameter`
	ubroker.properties[42,43]	`-logginglevel, parameter -logentrytypes, parameter`	`-Dsrv logginglevel -Dsrv logentrytype`
Runtime setting	Logging level	`LOG-MANAGER: LOGGING-LEVEL attribute`	`DSLOG-MANAGER: LOGGING-LEVEL attribute`
	Logging level	`LOG-MANAGER: LOG-ENTRY-TYPES attribute`	`DSLOG-MANAGER: LOG-ENTRY-TYPES attribute`

The following code examples shows how logging levels and log entry types are set using the client startup switch:

```
-logentrytypes DS.Cursor:3
```

```
-logentrytypes DS.Cursor -logginglevel 3
```

In the following example, logging levels and the log entry type are set using the `LOG-MANAGER` system handle:

```
LOG-MANAGER:LOG-ENTRY-TYPES = "DS.Cursor:3"
```

[42] `logginglevel` and `logentrytypes` specified in the `ubroker properties` file are for AppServer agents in the client context only.

[43] To specify `logginglevel` and `logentrytypes` at startup for the DataServer's server context, continue to use the `-Dsrv` Startup/connection options. `logginglevel` and `logentrytypes` specified in the `ubroker properties` file for a DataServer broker instance are ignored. Specifying the `-Dsrv` options within the srvrStartupParam of a DataServer broker instance is also ignored.

In the next example, the logging level and log entry types are defined using the `-Dsrv` switch.

```
-Dsrv logentrytypes,SQL:3
```

```
-Dsrv logentrytypes,SQL,logginglevel,3
```

The following example demonstrates the use of a `DSLOG-MANAGER` system handle with a DataServer `DB-CONTEXT` attribute.

```
DSLOG-MANAGER:LOG-ENTRY-TYPES = "SQL:3"
```

Logging levels

You can choose one of five logging levels, as shown in the following table.

Table 44: Logging levels

Logging level	Description
0 (None)	Logs no entries. This is equivalent to turning logging off.
1 (Errors)	Logs OpenEdge error messages. This includes all error messages and is unrelated to the entry types specified. Errors continue to be logged at all higher levels.
2 (Basic)	Log entry type determines the logged information. Each log entry type generates at least some output. This is the default.
3 (Verbose)	Logs more verbose levels of information than the basic level.
4 (Extended)	Extends the verbose logging level and, typically, the log information includes minute details or expands on the level of information provided by lower logging levels.

Note: All the log entry types may not log information at all the different logging levels. For information about the different log entry types and the logging levels associated with them, see Log entry types on page 210.

Setting the DataServer logging levels and log entry types

Client context log settings can be set using:

- Start-up parameters at client startup
- `LOG-MANAGER` system handles in client applications

Server context settings can be set using:

- The `-Dsrv` switch at client startup or during a database connection.
- `DSLOG-MANAGER` system handle in client applications

Setting all other DataServer log settings (clientlog, dslog, numlogfiles, logthreshold and logappend)

- Start-up parameters at client startup. These settings are shared with the server context when DataServers are connected self-service.

  Note: Of these, -dslog is not shared but simply passed onto the server.

- Ubroker.properties specified for agents at AppServer startup.

  Note: AppServer client context does not include the ability to forward a DataServer log file name onto the server context.

- LOG-MANAGER system handles in client applications.

  Note: Client context does not include the ability to forward a DataServer log file name onto the server context)

Server context settings can be set using:

- ProBroker startup parameters (-dslog, -numlogfiles, -logthreshold, -logappend)
- Unified-Broker DataServer instance attributes (srvrDSLogFile, srvrNumLogFiles, srvrLogThreshold, srvrLogAppend)

Note: DSLOG-MANAGER system handle can be used in client applications to return the above settings but cannot modify them.

For a details on LOG-MANAGER and DSLOG-MANAGER startup parameters, refer to the LOG-MANAGER and DSLOG-MANAGER entries in *ABL Reference* and *Troubleshoot ABL Applications*.

Aggregate logging using performance and interface log entry types at run-time

You can diagnose ABL code at run-time using the PERFORMANCE and INTERFACE log entry types. Collecting information with logging level 3 (Verbose) using the log entry type PERFORMANCE is called **aggregate logging**. Placing the PERFORMANCE log entry type around a section of ABL code helps you measure performance for that particular section of code.

When you have to monitor the performance or interface of a particular section of ABL code, you must use the PERFORMANCE or INTERFACE log entry type respectively and assign it the highest logging level so that detailed logs are logged for that particular section of ABL code. For the rest of ABL code that you do not have to monitor, use the PERFORMANCE or INTERFACE log entry type and assign it a relatively lower logging level. For more information on log entry types and logging levels, see Log entry types on page 210 and Logging levels on page 215.

The following use case shows you how to set the PERFORMANCE log entry type in ABL code for aggregate logging. The log entry type set at run-time through DSLOG-MANAGER overrides any log entry type set at startup or at connection level. For more information on DSLOG-MANAGER, see DSLOG-MANAGER system handle attributes and methods on page 208.

Suppose you have an ABL program with the following code for which you want aggregate performance logs for the FOR EACH section of the ABL code and less detailed logs for the rest of the ABL code, then, for the FOR EACH block of the ABL code you use the performance logging level, 3 (Verbose). After the FOR EACH block ends, you lower the logging level to 2 (basic) as you do not need detailed logs for the rest of the ABL code.

```
. . .
DSLOG-MANAGER:DB-CONTEXT = 'logical-database-name'.
DSLOG-MANAGER:LOG-ENTRY-TYPES = "PERFORMANCE:3".

FOR EACH mytable.
END.
DSLOG-MANAGER:LOG-ENTRY-TYPES = "PERFORMANCE:2".
. . .
```

Note: Dynamic (run-time) logging is enabled only for the server context of the enhanced logger.

Dynamic performance and interface logging does not affect the aggregate performance and interface logging at the session level.

Server Context Log File Naming

When no log file name is specified for the server context, its default name continues to be `dataserv.lg` in the working directory, just as it was using the legacy `qt_debug` logging options.

OpenEdge client startup and ProBroker startup both allow the server context log file to be specified with the `-dslog` parameter. For a DataServer broker instance specified to the Unified Broker, the `srvrDSLogFile` attribute in the `ubroker.properties` file specifies the server context log file name.

If the server context log file name is specified without any path, it is placed in the working directory. If it is specified with a relative path name, it searches for that relative path under the absolute path of the working directory unless the `$DSLOGDIR` environment variable is set. When `$DSLOGDIR` is set, it overrides the working directory as the root path for placing the server log file. The relative path name of the specified log name is combined with this absolute path to place the server context file. If you specify a full absolute path for the server context log file name, it is used exclusively to set the log file path.

The values for DSLOGDIR and -dslog can be set as described in the following table:

Table 45: Parameters for creating log file

DSLOGDIR value	-dslog value	Result
An existing directory	dslog name	dslog file is created in the DSLOGDIR
An existing directory	Qualified path and dslog name	dslog file is created in the dslog path
An existing directory	Not set	Log file with default name is created in DSLOGDIR
A new/nonexistent directory	dslog name	dslog file is created in working directory

DSLOGDIR value	-dslog value	Result
Not set	Qualified path and dslog name	dslog file is created in the dslog path
Not set	Not set	Log file with default name is created in working directory
Not set	A new/nonexistent path and dslog name	Log file with ds_ prefix is created in working directory
Not set	dslog name	dslog file is created in working directory

Creating a unique log file per connected client

By default the log file started in the server context is shared by all client/server connections started under a specific broker.

You can optionally set another environment variable at startup called `$UNIQUE_DSLOG` if you wish to have a unique log file name for each client session.

Note: When multiple clients are connecting by specifying a non-zero -logthreshold value as a start up parameter, if the `$UNIQUE_DSLOG` environment variable is not set and -logthreshold value has been specified, each client generates a unique log file name.

When the `$UNIQUE_DSLOG` environment variable is set on, the log file name starts with `ds_` followed by the log file name specified for the server context followed by the server's process ID number. The file extension continues to be `.lg`. The full name syntax is as follows:

Syntax

```
ds_<-dslog or srvrDSLogFile name>_pid.lg.
```

Enabling ABL to SQL Correlation in Logging

The Enhanced Logger allows ABL to SQL correlation so that you can compare ABL database requests with SQL directives that are executed on the server.

In the DataServer log file, ABL to SQL correlation includes information from the client context, the ABL module and line number, along with the server context SQL that was generated based on the ABL code and run against the foreign data source. The correlation is enabled using the `-Dsrv PRGRS_LOG_ABL,`*n* switch where *n* ranges from `0` to `2`, as demonstrated in the following example:

```
-Dsrv logentrytypes,SQL:2,PRGRS_LOG_ABL,2
```

For a DataServer broker instance, the `SQL` log entry type can be set by using the `-Dsrv logentrytypes,SQL` switch in the '**`srvrStartupParam`**' property under the **UBroker.MS** section in OpenEdge Management/OpenEdge Explorer or `ubroker.properties` files or using the parameter file `.pf`.

The following table describes the PRGRS_LOG_ABL connect option values:

Table 46: PRGRS_LOG_ABL connect option values

Option Value	Description
0	Disables the ABL to SQL correlation feature. No correlation information will be logged.
1	Enables the ABL to SQL correlation feature for SQL-related log entry types. Every SQL request derived from the log entry type output is preceded by ABL filename and line number which can be correlated to a compiled listings file.
2	Enables a superset of value 1 which, along with correlation information, includes visual delimiters marking the start and end of each ABL file module.

Note: The SQL log entry type is specifically designed to output SQL-related logging information and therefore would be used most commonly in conjunction with the PRGRS_LOG_ABL -Dsrv switch. However, the PRGRS_LOG_ABL switch can be enabled to acquire module and line number information with any log entry type that writes SQL request information to the log.

The ABL to SQL correlation is logged in the dataserv.lg file where the log entries will contain the ABL module and the line number where the ABL procedure started. It will also contain information about the SQL query executed on the foreign data source and the visual delimiters.

The following ABL constructs log SQL query information:

- FIND statements
  - FIND FIRST
  - FIND NEXT
  - FIND LAST
  - FIND PREV
  - FIND CURRENT
  - FIND rid
- Dynamic FIND statements
  - FIND-BY-ROWID
  - FIND-CURRENT
  - FIND-FIRST
  - FIND-LAST
  - FIND-UNIQUE
- FOR statements
  - FOR EACH
  - FOR FIRST
  - FOR LAST
- QUERY statements
  - OPEN
  - GET-LAST
  - GET-NEXT
  - GET-PREV
  - GET-FIRST
  - REPOSITION-TO-ROWID
  - REPOSITION-BACKWARD
  - REPOSITION-FORWARD
  - REPOSITION-TO-ROW
  - QUERY-OPEN method
- CREATE statement
- UPDATE statement
- DELETE statement
- IMPORT statement
- EXPORT statement
- RUN STORED-PROCEDURE statement
- SQL Pass-thru statements
- RELEASE statement
- VALIDATE statement
- ASSIGN statement
- END statement
- CURRENT-VALUE statement
- NEXT-VALUE statement
- SET-VALUE statement
- DYNAMIC-CURRENT-VALUE statement
- DYNAMIC-NEXT-VALUE statement
- CAN-FIND function
- RECID function
- ROWID function
- ProDataSet FILL, SAVE-ROW-CHANGES, MERGE-CHANGES methods.
- DO Transaction

The following example shows how the ABL to SQL correlation is logged in the dataserv.lg file, when the FOR EACH statement is compiled.

Using PRGRS_LOG_ABL switch to log ABL to SQL correlation

```
FOR EACH Customer:
  DISP NAME.
END.
```

The log output shown below, includes the ABL module name and the line number where the ABL procedure started in correlation to the SQL queries executed on the foreign data source. Visual delimiters are also logged for clarity in the log files.

```
[11/10/27@04:53:27.462-0400] P-007852 T-015312 1 MSS    -- MSS version: Microsoft SQL Server
10.00.1600 (4203)
[11/10/27@04:53:27.712-0400] P-007852 T-015312 2 MSS    SQL       --------->{START} [abl2sql.p]
[11/10/27@04:53:27.712-0400] P-007852 T-015312 2 MSS    SQL
------------------------------------------------------
[11/10/27@04:53:27.728-0400] P-007852 T-015312 2 MSS    SQL        ABL statement reference:
abl2sql.p line 0 (15223)
[11/10/27@04:53:27.728-0400] P-007852 T-015312 2 MSS    SQL        SELECT * FROM
"vp"."vp"."customer" WHERE 1 = 2
[11/10/27@04:53:27.728-0400] P-007852 T-015312 2 MSS    SQL        Execute on 0x1151348 Master
 Connection direct: SELECT * FROM "vp"."vp"."customer" WHERE 1 = 2  (15069)
[11/10/27@04:53:27.759-0400] P-007852 T-015312 2 MSS    SQL        ABL statement reference:
abl2sql.p line 2 (15223)
[11/10/27@04:53:27.759-0400] P-007852 T-015312 2 MSS    SQL       SELECT "custnum", "country",
 "name", "address", "address2", "city", "state", "postalcode", "contact", "phone", "salesrep",
 "creditlimit", "balance", "terms", "discount", "comments", "fax", "emailaddress",
PROGRESS_RECID, PROGRESS_RECID_IDENT_ FROM "vp"."vp"."customer" ORDER BY "custnum"
[11/10/27@04:53:27.759-0400] P-007852 T-015312 2 MSS    SQL       Execute on 0x11522a8 Firehose
 Connection Level 1 new: SELECT "custnum", "country", "name", "address", "address2", "city",
"state", "postalcode", "contact", "phone", "salesrep", "creditlimit", "balance", "terms",
"discount", "comments", "fax", "emailaddress", PROGRESS_RECID, PROGRESS_RECID_IDENT_ FROM
"vp"."vp"."customer" ORDER BY "custnum"  (15069)
[11/10/27@04:53:27.821-0400] P-007852 T-015312 2 MSS    SQL       <---------{END} [abl2sql.p]
```

For more information about the PRGRS_LOG_ABL switch, see *Troubleshoot ABL Applications*.

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 ABL SAVE CACHE COMPLETE statement creates a binary file that contains the entire schema for an OpenEdge 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 MS SQL Server database whose data source name is `sqlbdb` with the schema `sqlhold` and tells OpenEdge to use the cache file `sqlcache`:

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

If you make any changes to a schema holder, you must create a new cache file for it. For more information, see the "SAVE CACHE Statement" reference entry in *OpenEdge Development: ABL Reference*.

Connection failures and OpenEdge responses

The following table lists circumstances under which a connection might fail and describes how OpenEdge responds.

Table 47: Failure responses

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.
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 Data Dictionary	The Data Dictionary displays an error message and returns to the main window.
During an attempt to connect a connected MS 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.

Failure circumstance	Result
During an attempt to connect a connected MS 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 MS SQL Server database whose logical name is already in use by a connected MS SQL Server database	The system responds with a run-time error condition and you cannot connect to the second database.

Connection troubleshooting

Here are some reasons why a connection attempt to an MS SQL Server database might fail:

- The schema holder is not connected.
- The OpenEdge or MS SQL Server required environment variables are not set correctly when using the DataServer and a broker. For environment-variable information, see Configuring the DataServer on page 161
- The data source is not registered properly for ODBC client connectivity. See your Microsoft SQL Server documentation for information on configuring a user, system, or file DSN to properly connect to your data source.
- You have an outdated version of an ODBC DLL; for example, `ODBC16.DLL`, which runs on 16-bit machines only. This prevents OpenEdge 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 Troubleshooting on page 335.

Managing connections to an MS SQL Server database

Typically, the DataServer maintains one connection to an MS SQL Server 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 each request. Depending on the capabilities of the ODBC driver being used, the following cases may be candidates for using additional connections to accommodate additional cursors:

- Running multiple stored procedures
- Running a stored procedure and a `send-sql-statement` simultaneously
- 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(SEPARATE-CONNECTION)` option 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.
END.
```

Accessing the DataServer log

OpenEdge 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.

Naming the log file

The DataServer log file name, the number of log files, the log threshold, and the log append switch settings are derived from their respective client startup switches: `-dslog`, `-numlogfiles`, `-logthreshold`, and `-logappend`. For example:

- In **self-service mode**, `-dslog`, `-numlogfiles`, `-logthreshold`, and `-logappend` startup switches apply to both client and DataServer log files
- In **client-server mode**, the `-dslog`, `-numlogfiles`, `-logthreshold`, and `-logappend` settings are derived from the broker startup
- In **ProBroker**, these are the `-dslog`, `-numlogfiles`, `-logthreshold`, and `-logappend` startup parameters from the `_probrkr` command line
- In the **Unified Broker**, these are `srvrDSLogFile`, `srvrNumLogFiles`, `srvrLogThreshold`, and `srvrLogAppend` ubroker properties

All the above attributes are static in nature. They are set at startup and cannot be modified once established. They are read-only attributes when requested from the `DSLOG-MANAGER` runtime system handle.

Use the `-dslog` client startup parameter to specify the DataServer log file name. If `-dslog` is not specified, then the DataServer chooses the default log file name of `dataserv.lg`.

Setting the log file location

You can specify the absolute path of the log file name with `-dslog` startup switch to set the location of the log file. The absolute path is obtained from the `$DSLOGDIR` environment variable. So for example, setting `$DSLOGDIR="C:\OE\Wrk\logs"` and setting `-dslog` to `mydatasrv.lg`, yields the target server log file name of `C:\OE\Wrk\logs\mydatasrv.lg`. If `$DSLOGDIR` is not set and just the log file name is specified, then the log file is created in the directory where the OpenEdge session (client or agent) is started.

The path for the log file name is derived in the following order of preference.

1. Usage precedence is for `-dslog` with an absolute path name.
2. On specifying `-dslog` with relative path then path name, the `-dslog` specified is relative to the `$DSLOGDIR` path.
3. If `-dslog` is a relative path and no value has been specified for `$DSLOGDIR`, then the log file will located in the working directory where the DataServer server component is started. This can be:
    1. The current directory where the OpenEdge client is started, if using an OpenEdge client with a self-service DataServer.
    2. The working directory of your AppServer broker instance when using a self-service DataServer in the Unified Broker framework.
    3. The working directory of your DataServer broker instance when using a client-server DataServer in the Unified Broker framework.
    4. The current directory where you started your traditional broker using a client-server DataServer with ProBroker.

If neither `-dslog` nor `$DSLOGDIR` are specified, the default log file name `dataserv.lg` is used with the same directory path order of precedence as previously mentioned.

Handling log file creation errors

If creation of the DataServer log file fails for any reason, an error is reported and the DataServer continues to execute. If an error occurs, the DataServer will derive its log file name as `ds_`*`pid`*`.lg` where pid is the process id of the server process. The DataServer continues to write log messages to `ds_`*`pid`*`.lg` log file. For example, if you don't have write privilege on the `$DSLOGDIR` directory, the `ds_`*`pid`*`.lg` file is created in the directory where the client or broker agent is started. This behavior is optionally available by default by setting environment variable `$UNIQUE_DSLOG` at startup.

Restrictions on dynamic changes to log file

When the client and server share the same log file in self-service mode, the `LOG-MANAGER` system handle is available to allow the log file to be managed. However, when self-service DataServers don't share the same log file with the client, or if the DataServer is run in client-server mode, the server log file is likely to exist among other servers writing to the same log simultaneously. For this reason, dynamic changes to the log file, such as renaming or closing the log file, are restricted for all server log file contexts. The one exception is the `CLEAR-LOG(_)` function which is allowed only if no other process than the one initiating the `CLEAR-LOG(_)` is writing to the log file simultaneously.

Using the $UNIQUE_DSLOG environment variable

You can use the $UNIQUE_DSLOG environment variable to isolate the log output of a given server instance and prevent the sharing of log files resources. When the $UNIQUE_DSLOG variable is set, the designated DataServer log file name becomes a root name. Specifically, it becomes the prefix of a log file name that is followed by the process ID `pid` of the running server process - which is then suffixed with the familiar `.lg` extension if the server name is `dataserv`. This feature can be disabled by unsetting the `UNIQUELOG` environment variable.

With a unique name, the server executable can manage its own log file copy. However, the `DSLOG-MANAGER` system handle continues to disallow changes to the log file name or closing of the file at runtime. This restriction ensures that the DataServer always has an open log file to which it can write information. The legacy `qt_debug` logging mechanism relies on this being the case.

For information on managing log settings, see Analyzing application execution with Enhanced Logger on page 206.

Accessing the log file using the host machine

To obtain access to the DataServer log file using 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, OpenEdge writes the information to the `dataserv.lg` file. If OpenEdge 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.

7

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 DataServer for MS SQL Server demonstration databases, the tutorial describes how to perform the tasks.

For details, see the following topics:

- Demonstration databases for DataServer for MS SQL Server
- Preparing to create demonstration databases
- DataServer utilities
- Prime to ROWID identification
- Creating a schema holder
- Updating a schema holder
- Verifying a schema holder
- Changing connection information in a schema holder
- Changing a code page in a schema holder
- Deleting a schema holder
- Managing server attributes
- Migrating an OpenEdge database to MS SQL Server
- Column width
- Adjust schema utility

- Delta df to MS SQL Server Incremental Schema Migration utility
- Modifying a schema holder
- Independent OpenEdge MS SQL Server Schema Pull
- Adding extended ABL support

Demonstration databases for DataServer for MS SQL Server

The demonstration databases for the DataServer for MS SQL Server allow you to run the tutorials from this chapter, code examples, or your own procedures.

The demonstration databases are not part of the OpenEdge installation media. To create and initialize a demonstration data source, you run the OpenEdge DB to MS SQL Server utility, which migrates the sports database to your data source and then creates a schema holder. (See Preparing to create demonstration databases on page 228 for detailed instructions.) After you do this, you are ready to run the tutorial exercises or your own 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

Preparing to create demonstration databases

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 MS SQL Server **Utilities** menu.

The following lists identifies the basic preliminary steps you must complete to create demonstration databases:

- Install and start your data source. This specific step depends on your unique environment. As necessary, refer to your data source documentation.
- Install your DataServer drivers.
- Install the OpenEdge client. As necessary, refer to the OpenEdge client-related documentation.

Use the following steps to create the demonstration database and schema holder using the OpenEdge DB to MS SQL Server utility:

1. Start the ODBC Data Source Administrator tool for your data source. Refer to Configuring an ODBC driver and registering the data source on page 163.
2. Select the appropriate ODBC driver for the schema holder.
3. Select the target ODBC data source for the schema holder. Refer to Configuring a local DataServer on page 163 and Configuring a remote DataServer on page 164.
4. Make any additional selections required by your data source.
5. Specify **mysport** as the data source name. OpenEdge recognizes the name that you use to register a data source.

You are not required to specify the data source name as **mysport**. However, identifying the data source name, using this suggested data source name or another one you choose, can help you to distinguish your tutorial data source from all other data sources you set up.

6. From the **Start** menu on the task bar, select **OpenEdge** and select the specific product you want to use to begin your OpenEdge session or type the following command line in Windows:

```
install-path\dlc\bin\prowin32
```

Note: Starting a local OpenEdge session also automatically starts the local DataServer.

7. Access the Data Administration. Create a copy of the sports database and connect to it.
8. Select **DataServer** > **MS SQL Server Utilities** > **Schema Migration Tools** > **OpenEdge DB to MS SQL Server** to start the OpenEdge DB to MS SQL Server utility.
9. Specify OpenEdge DB to MS SQL Server parameter values as described in the following table.

Table 48: OpenEdge-to-MS SQL Server Conversion utility

Interface element	Description
Original OpenEdge Database	Accept the default value; it is the name of the connected data source. For example, if you copied the sports database to create a demonstration database, the default value **sports** displays.
Connect parameters for OpenEdge	Accept the default value, which specifies the current working database.
Name of Schema holder Database	Type **myholder**, or another schema holder name of your choice.
Logical Database Name	Type the name that OpenEdge uses to identify the MS SQL Server database.
ODBC Data Source Name	Type the name you used when registering the data source. For example, if you accepted the ODBC Data Source Name suggested earlier in Preparing to create demonstration databases on page 228, type **mysports**. Otherwise, type the ODBC Data Source name you previously set up.
Username	Type the user ID for MS SQL Server.
User's Password	Type the password of the user.
Connect parameters	See Connecting the DataServer on page 189 for connection parameters.

Interface element	Description
MaximumVarchar Length	8000 characters. The server LOB data types (text and varchar(max)) can be greater than 8000 bytes. If you are using a single byte character set and your character field display format or SQL width is less than 8000, a VARCHAR data type will be used. If the value is greater than 8000, then a server LOB type will be used. NOTE: When **Use Unicode Types** is selected, the Unicode variable character type, NVARCHAR, is used when the maximum length is 4000 or less. The maximum length must now be set at 4000, the adjusted default, or less. If your character field display format or SQL width is greater than 4000, then a Unicode server LOB data type is used. Keep in mind that because UTF-8 character encoding can be up to four bytes, only 2000 characters are guaranteed to fit in a Unicode field.
Codepage	Accept the default code page value.
Collation	Accept the default value.
Insensitive	Accept the default value to retain case-insensitivity.
Load SQL	This check box is select by default, to specify that the generated SQL script loads into the data source. Leave it selected.
Move Data	Select the check box to dump and load data from OpenEdge to MS SQL Server.
Advanced	Click **Advanced** to open the **OpenEdge DB to MS SQL Server Conversion Advanced Options** dialog box. The purpose of this dialog box is to reduce clutter on the main dialog box and provide logical grouping of fields on it. See the next table for the details you are prompted to enter.

Table 49: OpenEdge DB to MS SQL Server Conversion Advanced Options

Interface Elements	Description
Migrate Constraints	By default, the **Migrate Constraints** option is selected. This enables you to migrate constraint definitions defined using the procedures defined in Overview of foreign database constraint definitions on page 255.
ROWID sub-section	The ROWID sub-section groups selections for specifying ROWID and RECID. See ROWID sub-section of MS SQL Server Conversion on page 235 for more information.
Map to MSS Datetime Type	Selected by default. Uncheck box to map to datetime data types with time zone components. For details, see DataServer for MS SQL Server support for datetime data types on page 61 .

Interface Elements	Description
Create Shadow Columns	MS SQL Server is case-insensitive by default and, when used with a case-insensitive code page, this box is unmodifiable. When SQL Server is configured with a case-sensitive code page, this box is available for selection. Selecting it provides case-insensitivity compatible with the behavior of an OpenEdge database.
Use Revised Sequence generator	Select to enable revised generator. Clear this option to retain older sequence generator, particularly if SET-VALUE sequence statement is used extensively in your program.
Try Native Sequence	Select to migrate OpenEdge Sequences to MS SQL sequences. **Note:** This option selection will be successful only if the target MS SQL databse is MS SQL 2012 or newer.
Cache size	If the **Try Native Sequence** option is selected, then the **Cache Size** option is made available for entry. This determines the size of the cache that should be used on the server for the cache size. The following three values are applicable: • 0 = server (the cache size is determined by the server based on server criteria) • "?" = No Cache("NO CACHE" option is appended to the native sequence creation on the server) • Positive integer(1 or >1) - The number of sequences stored in the cache(Can range between 1 to the maximum value of the data type of the native sequence)
Use Unicode Types	Changes default code page to UTF-8 and directs the schema migration to convert all character data types in OpenEdge database to Unicode data types.
Expand width (utf-8)	Doubles the size of character fields based on specified field width criteria.

Interface Elements	Description
For field widths use:	When pushing fields to a foreign data source, you can select one of two primary field format options: • **Width** — Uses the value of the _**width** field in the _field record. **Note:** Recommended for Unicode implementations. For more information on column width, see Column width on page 294. • **4GL Format** — Compiles with the current default width specified. (default) If you select the **4GL Format** option, you have an additional setting to define: • **Expand x(8) to 30** — This option is selected on by default to indicate that the format for the character fields defined as x(8) will be created as 30 characters. **Note:** You cannot use the **Expand x(8) to 30** setting with the **Width** option.
Apply Uniqueness as	The **Apply Uniqueness as** option allows to you specify how uniqueness is defined. The default, **Indexes**, provides backward compatibility. The **constraints** option defines constraint definitions for uniqueness, but existing constraint definitions always take precedence over any constraint definition generation from this migration option.
Include Defaults	The **Include Defaults** includes an **Apply Default as** option that allows you to specify if you want your defaults defined as **Field Attributes** (default) or **constraints**. The **Field Attributes** option provides backward compatibility. The **Constraints** option only defines constraint definitions if a constraint definition doesn't already exist. If you select **Constraints**, then after the migration, all the initial values defined for migration become default constraint definitions on the server.

Note: For a complete description for running while in batch mode, see Running the OpenEdge DB to MS SQL Server utility on page 268.

Running the utility creates and connects a schema holder and the MS SQL Server database. It operates as follows:

1. SQL script is generated.
2. SQL that creates the schema is sent to the MS SQL Server data manager.
3. The schema is pulled back to the schema holder.

4. The schema holder and the OpenEdge database are compared and all information needed by OpenEdge is applied to the schema holder.
5. Data is loaded if selected.
6. The schema holder is disconnected.
7. 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.

DataServer utilities

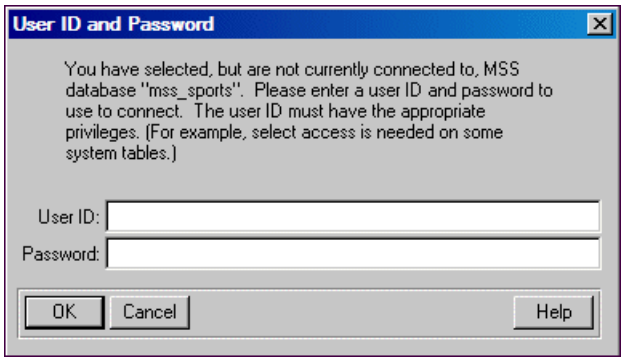
OpenEdge supports a number of utilities that allow you to create and maintain a MS SQL DataServer.

To see all available DataServer utilities, choose **DataServer** > **MS SQL Server Utilities**. The following table describes the DataServer utilities.

Table 50: DataServer utilities

Utility	Description
Create DataServer Schema...	Creates a schema image in the schema holder for the MS 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 MS SQL Server database
Change DataServer Schema Code Page...	Changes the code page in the schema holder associated with the MS SQL Server database
Server Attributes	Provides an interface to create, view, and maintain server attributes, and define constraint definitions in your OpenEdge database that can be migrated to the server. Constraint definitions are applied to the columns and indexes of the foreign database during migration and become metadata in your OpenEdge schema holder
Delete DataServer Schema...	Deletes a schema image from the schema holder
Schema Migration Tools	Accesses utilities for migrating an OpenEdge database to a MS SQL Server database, for incrementally migrating a schema to MS SQL Server, and adjusting the schema image

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:



To accept the user ID and password combination that you supplied in the **User ID and Password** dialog box, choose **OK**. 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 Authorization and authentication on page 177 for information on privileges.

Note that the value **msssports** that appears in the previous **User ID and Password** dialog box identifies the MS SQL Server database to which a connection will be attempted. The actual database name that appears will depend on the specific MS SQL Server database to which you are trying to connect.

Prime to ROWID identification

New migration capabilities help control database construction with respect to ROWID. The goal is to improve on ROWID performance by providing a direct, more naturalized, database-centric, approach to ROWID selection that reduces overhead to the database and can improve on overall transactional performance.

Based on inputs during migration, a processing algorithm attempts to generate purposed, intelligent selections for the primary, clustered, and ROWID indexes, in that order.

In this process, a primary key is typically "elected", the clustered index is "derived" out of the "push" phase of the migration, and the ROWID index is "derived" out of the "pull" phase of the migration. Through a combination of migration options and defaulting schemes, a best alternative for ROWID is chosen and loaded into the schema image and is then used at run-time to optimize server access for DataServer applications.

Ideally, the chosen ROWID index provides for the best overall application performance. While any unique index can make a good ROWID selection, as long as it contains the right properties and is a good "natural" key selection for your DataServer application's query criteria, some choices are better than others.

This criterion gives deference to natural key selection over the use of a surrogate key, since greater potential synergy can be gained between the ROWID index and the DataServer if the selected key also has business meaning. Progress recognizes that wide, highly mutable, excessively propagated natural key, or a key lacking significant business meaning cannot be a better alternative than a surrogate key. The ultimate decision on the best selection should go to those who understand their database objectives or have tested the overall application performance that any specific key selections can yield.

The following list identifies a general selection preference order for ROWID index criteria, in decreasing order of preference:

- A ROWID generated from a selected unique and mandatory index whose clustered and primary indexes are explicitly or implicitly that same and whose characteristics closely resemble the clustered index guidelines discussed in Recommendation for choosing best clustered index on page 238.
- A ROWID generated from a selected unique and mandatory index whose clustered and primary indexes are explicitly or implicitly that same.
- A ROWID generated from a selected index with uniqueness, but without mandatory components that would otherwise supply a SQL primary.

- A ROWID generated from a selected index whose uniqueness is appended to create ROWID candidacy.
- A ROWID generated from a generated IDENTITY or from choosing **Select Best** whose index can be made clustered.
- A ROWID generated from a generated IDENTITY or from choosing **Select Best**.

ROWID sub-section of MS SQL Server Conversion

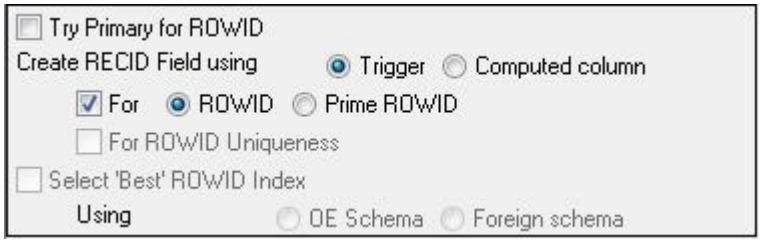
The following figure shows the sub-section of the OpenEdge to MS SQL Server conversion dialog boxes.

Note:

When you select the **Migrate Constraints** option above the **ROWID** sub-section and you define a primary constraint definition or clustered mock constraint definition, or both for a migrated table, **Migrate Constraints** takes precedence over any selection made in the **ROWID** sub-section for ROWID designation.

A clustered index is always implied when a primary key is specified and when an explicit clustered index is not specified. Also, by not setting the primary index to the clustered explicitly, you can specify separate indexes for one or other clusters without any conflicts.

Figure 11: ROWID sub-section



The behavior resulting from selecting the various elements of the figure above is as follows:

- **Try Primary for ROWID**
  - **Not selected** (default) — Do not use the primary index mapping of this feature. Left unselected, this option is backward compatible, or you can choose other options to determine index selections and ROWID designation.
  - **Selected** — Whenever available, use the OpenEdge primary index to establish index selections for primary and/or clustered index and attempt to make the ROWID designation from the same index if uniqueness can be obtained from the key. This option takes precedence, when you select this option along with other options in the **ROWID** sub-section.
- **Create RECID Field using**

  This option is selected by default. Select this option for backward compatibility with the defaults from previous OpenEdge releases. This option when used with the **Select 'Best' ROWID Index** option uniquify the the non-unique indexes. This option is enabled if any of the two mutually exclusive options, **Create RECID Field - For ROWID Uniqueness** and **Create RECID Field - For**, are selected.
  - **Trigger** (default) — Create `PROGRESS_RECID/PROGRESS_RECID_UNIQUE` using the legacy trigger mechanism. Select this option for backward compatibility.
  - **Computed Column** — Create `PROGRESS_RECID/PROGRESS_RECID_UNIQUE` using a computed column. This is preferred method for creating `PROGRESS_RECID/PROGRESS_RECID_UNIQUE` for

producing index uniqueness and ROWID capability, but comes with the restriction that no server-based INSERT trigger can exist on the table.

- **Create RECID For**

  Selected by default, this option is disabled when the **Create RECID For ROWID Uniqueness** is selected. The two options are mutually exclusive.

  Also, when this option is unselected, the legacy process of generating the PROGRESS_RECID field and assigning it to ROWID is removed.

  - **ROWID** (default) — When selected without the **Migrate Constraints** and the **Try Primary for ROWID** options selected, this option performs the legacy task of the **Create RECID Field** option from previous releases.
  - When selected in conjunction with **Migrate Constraints** and/or the **Try Primary for ROWID** options, this option becomes a secondary solution for generating ROWID for non-unique primary and/or clustered indexes that could not be designated for ROWID.
  - **Prime ROWID** — When selected without the **Migrate Constraints** and the **Try Primary for ROWID** options selected, this option performs the legacy task of the **Create RECID Field** option from previous releases.
  - When selected in conjunction with **Migrate Constraints** and/or the **Try Primary for ROWID** options, this option becomes a secondary solution for generating ROWID for non-unique primary and/or clustered indexes that could not be designated for ROWID. The ROWID index is optimized by designating the index as the primary and clustered as well.

- **Create RECID For ROWID Uniqueness**

  Unselected by default, this option is enabled when **Create RECID For** is unselected. The two options are mutually exclusive. This option works in conjunction with either of the two options, **Try Primary for ROWID** and **Select 'Best' ROWID Index**.

  - **Unselected** (default) — Does not compensate for missing uniqueness of the existing indexes by appending PROGRESS_RECID_UNIQUE identity column for ROWID designation purposes.
  - **Selected** — Whenever the targeted primary and/or clustered index is non-unique and therefore cannot be designated for the ROWID index, generate the IDENTITY column and append it as the last index component of the ROWID composite in order to create uniqueness. Effectively, this provides a DataServer replacement for the 4-byte internal RID that MS SQL Server otherwise requires on the primary and/or clustered indexes (but is inaccessible to DataServers). It also makes the designated primary and/or clustered index now eligible for ROWID candidacy during migration.

    Note: When the RECID field is used in this way, it is named PROGRESS_RECID_UNIQUE, and not PROGRESS_RECID. The name change allows it to be utilized by the trigger and/or computed column, and be populated on CREATE, but the name uniqueness prevents it from being identified as a ROWID candidate by itself from the dictionary utilities.

- **Select 'Best' ROWID INDEX**

  Unselected by default, this option is mutually exclusive with the **Create RECID Field For** options. If the **Create RECID Field For** options are unselected, this option becomes mandatory.

  When selected and used in conjunction with **Migrate Constraints** and/or **Try Primary for ROWID**, it becomes a secondary path in the search for the designated ROWID index. The primary and clustered selections associated with those options will take priority in ROWID selection.

Note: This is a schema "pull" option, not a database "push" option and is therefore unaffected by the **Create RECID Field - For ROWID Uniqueness** when that option is selected.

- **Using OE Schema** (default) — Locate an existing index to use as ROWID, if it qualifies, according to the requirements of the OE Schema. The criteria of the OE Schema is described in Best index selection on page 237. If a Best index selection can be found in the OE Schema, it is designated as ROWID. Selecting this option provides backward compatibility with earlier releases when those legacy migrations had the **Create RECID Field** unselected.

  In case none of the existing indexes are unique and you have selected the **For ROWID Uniqueness** option, then this option evaluates indexes for ROWID designation by making unique the existing indexes. The identity column will be appended to the 'best' index selected for ROWID designation.

- **Using Foreign schema** — Locate an existing index to use as ROWID from the ODBC API that evaluates best index on the server. If a Best Index selection can be found through the ODBC API, it is designated as ROWID.

Best index selection

The best index algorithm is used in the pull criteria when a primary index has not been found to support ROWID through both dictionary and DataServer methodologies that examine indexes in search of the optimal ROWID. The best index selection algorithm takes place during schema pull after all the server objects are created. Prior to Progress Version 7.3, a unique index with a single mandatory integer column was required to set RECID/ROWID. From Version 7.3 until OpenEdge Release 11.0, a single column index may be selected automatically if one or many columns exist that meet the criteria of being mandatory and unique.

The ranking protocol used to choose the first/best RECID/ROWID index is shown in the following table.

Table 51: Best index ranking criteria

Rank	Mandatory Unique Single- Component[44]	Integer	Date [45]	Float[46]	V7.3 Compatible ROWID	Auto-selected ROWID
1	?	?	—	—	?	?
2	?	—	—	—	—	?
3	?	—	?	—	—	?
4	?	—	?	?	—	?
5	—	?	—	—	?	—
6	—	—	—	—	—	—
7	—	—	?	—	—	—
8	—	—	?	?	—	—

Recommendation for choosing best clustered index

When selecting an index as the clustered index, consider the following guidelines:

- Choose a frequently searched index for query resolution, particularly by large queries, or as a foreign key on join relations (JOIN), or for sorting (GROUP BY, ORDER BY, etc.) and/or ROWID positioning inside the DataServer. For composite clustered keys, the first component column should be the most searched column of the key.
- Choose an index with a high degree of uniqueness.
- Choose an index that can be utilized in range queries to produce bracketed results.
- Choose an index that is often accessed sequentially. Rows are stored in the sorted order of this key, and the clustered index can act as a "covering" index, allowing data to be read out of the index itself rather than retrieving additional pages in a lookup operation to get the desired data values.
- Choose an index that is monotonic, incremental, and unique, such as an IDENTITY column, particularly if it is also meaningful to the application table content. An IDENTITY column is guaranteed to be incremental, and unique and the MS SQL Server optimizer can provide additional optimizations when an IDENTITY column is also the clustered index.
- Choose an index that has a narrowly sized key and preferably a single column, especially since non-clustered indexes use the clustered index to locate rows and hold the clustered key in their B-tree structure. A composite clustered key can provide benefits because the part of the table that is also the clustered index does not need additional space to create the clustered columns and its index.Choose an index that is reasonably static and does not undergo frequent changes. The primary key is a good clustered index candidate.

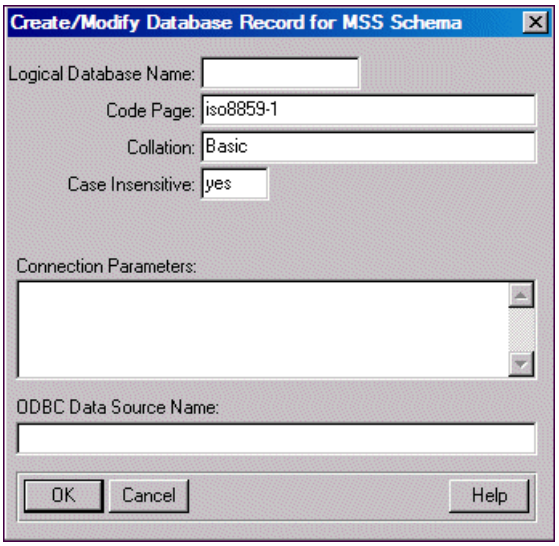
Creating a schema holder

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:

[44] Automatically selectable
[45] Issues sorting date fields, decreases rank
[46] Issues with equality in WHERE clauses, decreases rank



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 *OpenEdge Getting Started: ABL Essentials*.

   Note: If you place the schema from a second MS 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 OpenEdge name for the code page that the data source uses. The default is iso8859-1. If you choose UTF-8 as your schema image code page, your schema holder's code page must also be UTF-8.

   The following table lists the most common MS SQL Server database code pages and the equivalent OpenEdge names.

   Table 52: MS SQL Server and OpenEdge code pages

MS SQL Server Code page	OpenEdge equivalent
iso_1	iso8859-1 (default schema-holder code page)
cp850	ibm850

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

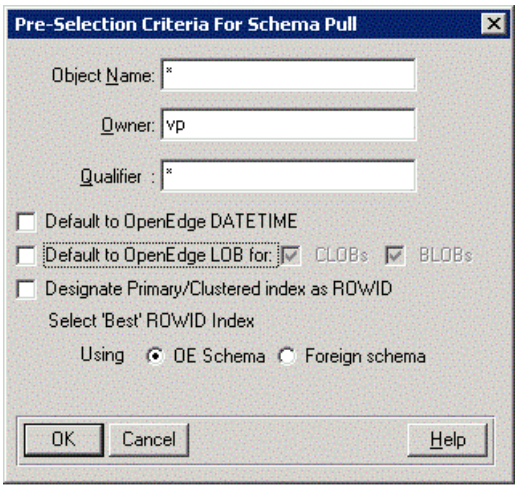
4. In the **Collation** field, enter the name of the collation rule to use. The default is Basic. See Code pages on page 44 for a discussion of collation issues to consider.
5. In the **Case Insensitive** field, the default value **yes** indicates that MS SQL Server's case insensitivity feature is in use. To change this value, type **no**.
6. Type the connection parameters in the **Connection Parameters** field.

   See Connecting the DataServer on page 189 for a description of the required and optional connection parameters.

7. In the **ODBC Data Source Name** field, type the name that you used when you registered the data source with the ODBC administration tool.
8. Choose **OK**.

   The utility prompts you for your data source user ID and password. If they are required by the MS SQL Server data source and you did not provide them in the **Connection Parameters** field (see 6 on page 239), enter a data-source user ID and password combination that has SELECT privileges for the system objects listed in Authorization and authentication on page 177 and read access to other database objects that the schema holder will include.

9. Choose **OK**. When the DataServer connects to the MS SQL Server database, it reads information about data source objects. The **Pre-Selection Criteria For Schema Pull** dialog box appears:



10. Provide the information described in the the following table:

Table 53: Pre-Selection Criteria For Schema Pull

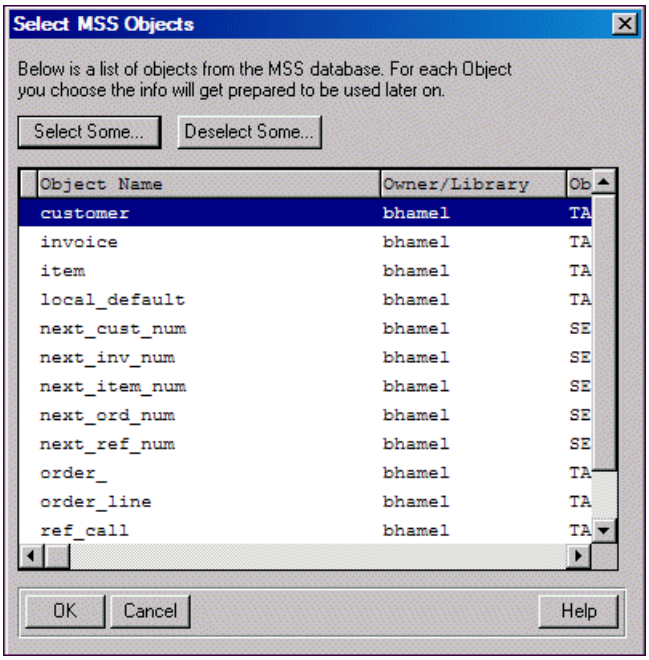
Interface element	Description
Object name	Specify an Object name qualifier for objects to be pulled for the server. The wild card default selects all objects. For example, you can specify A* in the **Object Name** field to list all the tables whose names begin with A or a. **Note:** You should not specify 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.)

Interface element	Description
Owner information	Specify an Object owner to be pulled for the server. The wild card default selects all objects. For example, you can specify `DS*` in the **Owner information** field to list all the tables whose names begin with A or a. **Note:** You should 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.)
Qualifier	Specify an Object's qualifying database name to be pulled for the server. The wild card default selects all objects. **Note:** You should 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.)
Default to OpenEdge DATETIME	Select **Default to OpenEdge DATETIME** to automatically map MS SQL Server timestamp data types to the associated OpenEdge equivalent DATETIME data type. If this check box is not selected, server timestamp data types map to DATE data type in OpenEdge for backward compatibility.
Default to OpenEdge LOB for:	If you have modified your client application to handle `LOB` data types, select **Default to OpenEdge LOB for: CLOBs** and/or **BLOBs** to map the OpenEdge `LOB` data type to MS SQL Server `VARBINARY(MAX)`, `IMAGE` and `FILESTREAM` data types. If you do not select this option, all server LOB data types map to the CHARACTER data type in OpenEdge for backward-compatible. For more information on mapping OpenEdge and MS SQL Server data types, see Support for OpenEdge ABL BLOB data type on page 72.

Interface element	Description
Designate Primary/Clustered index as ROWID	Specifying **Designate Primary/Clustered index as ROWID** for a table with a defined clustered index and without a defined PROGRESS_RECID field designates the clustered index as ROWID if it qualifies for ROWID selection. For the clustered index to qualify for ROWID selection, it must be unique. This designation for ROWID takes precedence over any other options selected for the schema pull operation but does not overwrite a legacy ROWID designation that is based on the presence of PROGRESS_RECID. **Note:** The **Designate Primary/Clustered index as ROWID** option may produce some or all of the "natural" key selections available from the foreign table's indexes. These can be considered alternatives to the PROGRESS_RECID column or other ROWIDs previously designated.
Select 'Best' ROWID Index	

Interface element	Description
	Selecting **Select 'Best' ROWID Index** during migration provides the legacy equivalent of ROWID designation behavior for the tables from prior versions of OpenEdge where the PROGRESS_RECID column was not present in a table. When selected in conjunction with new process flow options for ROWID migration, this option plays a secondary role in the designation of ROWID indexes deferring to the **Designate Primary/Clustered index as ROWID** option as the first choice. This option searches for a viable index for ROWID when an PROGRESS_RECID column does not exist and other process flow options that take precedence do not render a viable index. Select 'Best' ROWID Index using one of the following options: • **OESchema:** if you must locate an existing index to be used as ROWID that meets the requirements of the OpenEdge Schema. The OpenEdge Schema criteria is described in Best index selection on page 237. If a Best index selection can be found in the OpenEdge Schema, it is designated as ROWID. When this option is used in conjunction with the **Designate Primary/Clustered index as ROWID** option, this option specifies a secondary path in the search for the ROWID index. The primary and clustered selections associated with those options will take priority in ROWID selection as does the presence of a PROGRESS_RECID column. • **Foreign Schema:** if you must locate an existing index to be used as ROWID from the ODBC API that evaluates best index on the server. If a Best Index selection can be found through the ODBC API, it is designated as ROWID. If this option is selected in conjunction with the **Designate Primary/Clustered index as ROWID** option, then a warning is generated with respect to both options. When both options are selected, this option becomes a secondary path in the search for the ROWID index. The primary and clustered selections associated with those options takes priority in ROWID selection as does the presence of a PROGRESS_RECID column.

11. Choose **OK**. OpenEdge displays a list of the data source objects that you can include in the schema holder, as shown:



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.

12. Click the option appropriate to the action you want to perform:

    - **Select Some** — Displays the **Select by Pattern Match** dialog box on which you can specify object information used to select objects
    - **Deselect Some** — Displays the **Deselect by Pattern Match** dialog box on which you can specify object information used to deselect objects

    You can also elect to select and deselect individual objects by clicking and double-clicking on an object. An asterisk appears next to an object that has been selected; double-click an object to remove the asterisk and identify that the object is now deselected.

13. Choose **OK** after you have identified all the objects you want to include in the schema holder. 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 OpenEdge `ROWID`. If an appropriate index does not exist, the DataServer issues the warning, **Please check errors, 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 Data Dictionary. See Defining the ROWID on page 312 for instructions. For additional information, see Indexes and sorting on page 49 .

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.

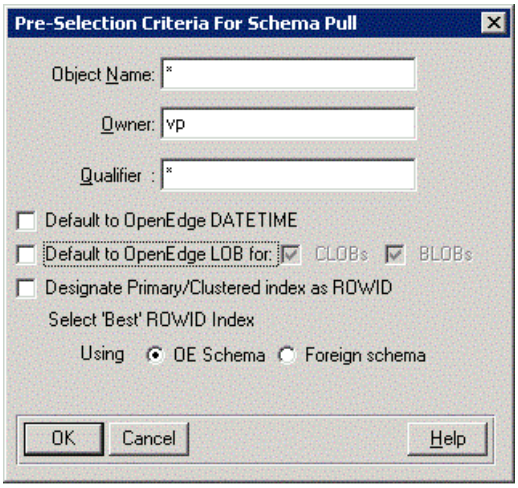
The Update/Add Table Definitions utility allows you to:

- 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. For example, an `integer` data type that is pulled from a MS SQL Server data source can be mapped to any of the following OpenEdge data types: `INTEGER`, `DECIMAL`, `LOGICAL`, or `INT64`. See Data types on page 56 for more information about support for MS SQL Server data types and their equivalent OpenEdge data type mapping options.

To update a schema holder from the Data Administration tool:

1. Access the **Data Administration**, 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 and select data type options as required. These values preselect the data source objects that the utility uses to update the schema holder. By default, the wild card symbol (*****) appears; it specifies that the utility uses all of the objects in the data source. For more information on data type options,

   ---

   Note:

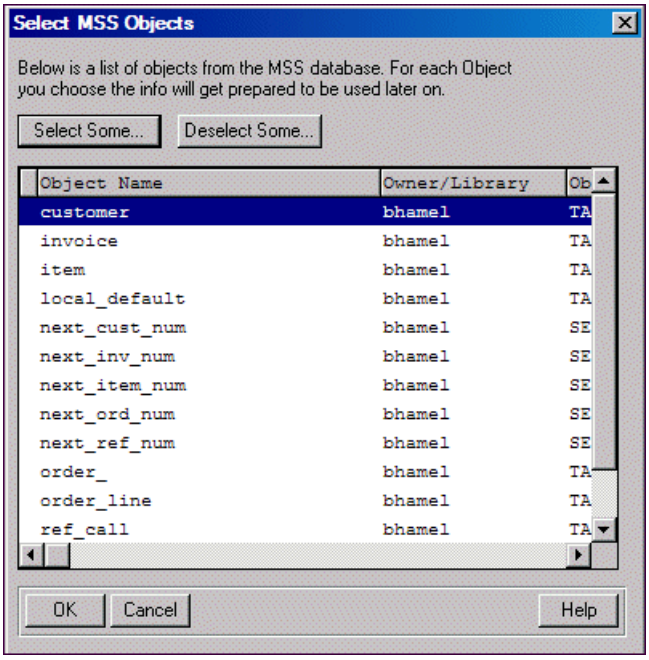
   You should not specify an entry that consists exclusively of wild cards for each of the three entry fields in the dialog box. An entry that consists exclusively of wild cards might degrade the performance of the database when you perform a schema pull. (It will include system catalog files from the data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

   Consider whether you will choose to perform a schema pull that consists only of wild cards. Doing so might degrade your database's performance as the activity will include system catalog files that are not typically included in a user database.

   Check **Default to OpenEdge DATETIME** to automatically map MS SQL Server data types to the associated OpenEdge data type. If you have modified your client application to handle `LOB` data types, check **Default to OpenEdge LOB** to map the OpenEdge `LOB` data type to MS SQL Server equivalent of `BLOB`, or `CLOB` data types. For more information on mapping OpenEdge and MS SQL Server data types, see Support for OpenEdge ABL BLOB data type on page 72 and Support for OpenEdge ABL CLOB data type on page 71.

3. Choose **OK**.

   A dialog box like the following example lists the objects and table information that you have preselected:



4. Select the objects that you want to update, then choose **OK**. When the update completes, you return to the **Data Administration** main window.

When the update completes, you are reminded 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, OpenEdge overwrites the old definition with the new one based on the current data source object. It also preserves the OpenEdge-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 OpenEdge-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`.

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 Table Definition 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.

The following table lists the differences that the Verify Table Definition utility detects.

Table 54: Verify utility report

Database object	Difference	Category
Table	Description[47]	Retained
Table	Foreign type	Severe
Table	Name in OpenEdge	Retained
Table	`ROWID` index	Retained
Index	Active	Minor
Index	Description[47]	Retained
Index	Name in OpenEdge	Retained
Index	Unique	Retained[48]
Index field	Abbreviated	Minor
Index field	Ascending	Severe
Index field	Order	Severe
Field	Case sensitivity	Retained
Field	Decimals	Retained[49]
Field	Description[47]	Retained
Field	Extent	Severe
Field	Initial value	Retained[49]
Field	Mandatory	Retained
Field	Name in OpenEdge	Retained
Field	Order	Retained
Field	OpenEdge data type	Retained[49]

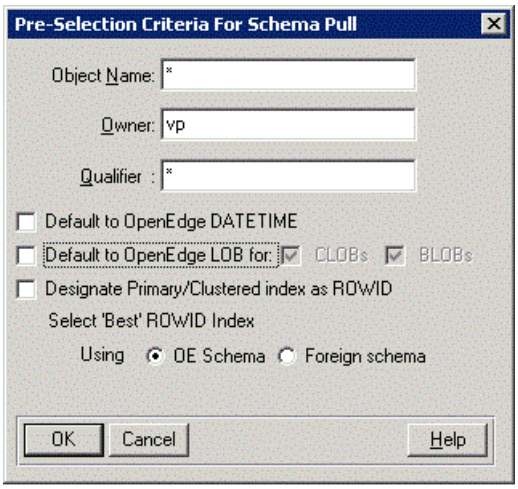
Database object	Difference	Category
Field	OpenEdge format	Retained[49]
Field	Shadow-column Name	Severe

Verifying a table

To verify a table from the Data Administration tool:

1. Access the **Data Administration**, if you are not already there, and select **DataServer** > **MS SQL Server Utilities** > **Verify Table Definition**.

   The following dialog box appears:



   Note: You should 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.)

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 wild card 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 check box **Verify only objects that currently exist in the schema holder**.
4. There is a 32K limit on the contents of the verify report. Select **Output differences to file** if you anticipate a large volume of differences, or wish to save the report. The file that is produced will be named, *ldbname*+.vfy and will be written to the working directory.

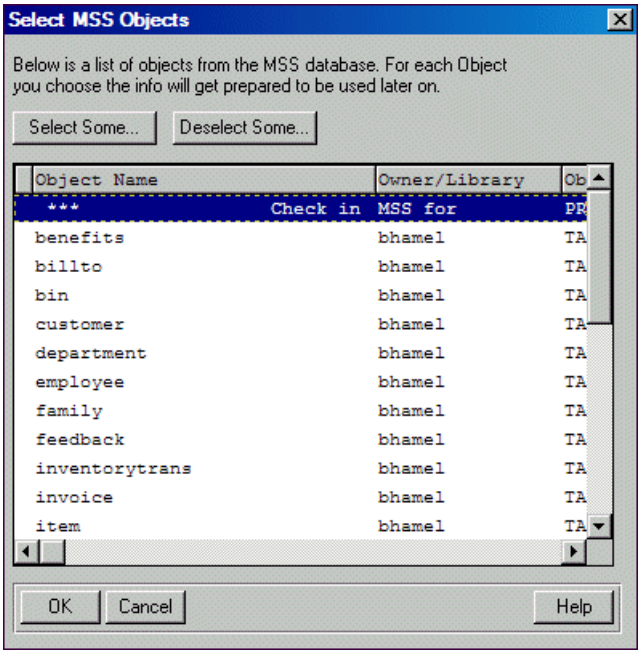
[47] The differences in descriptions between OpenEdge and MS SQL Server is not captured during the verification process because MS SQL Server, unlike OpenEdge, does not store descriptions for tables, fields, and indexes.

[48] When you update an index, it is identified as unique if it was defined as unique in the data source or the schema holder.

[49] 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 OpenEdge data type is CHARACTER, the data type information is not updated.

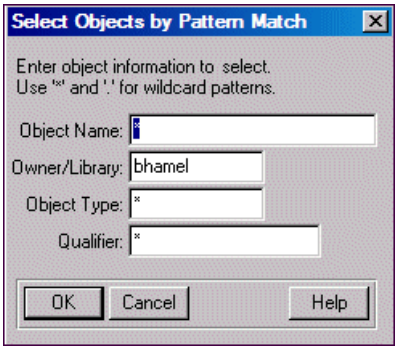
5. Choose **OK**.

   A dialog box like the following example lists the objects and table information that you preselected:



6. Select the objects that you want to update, then choose **OK**.
7. To select tables by matching a pattern, choose **Select Some**.

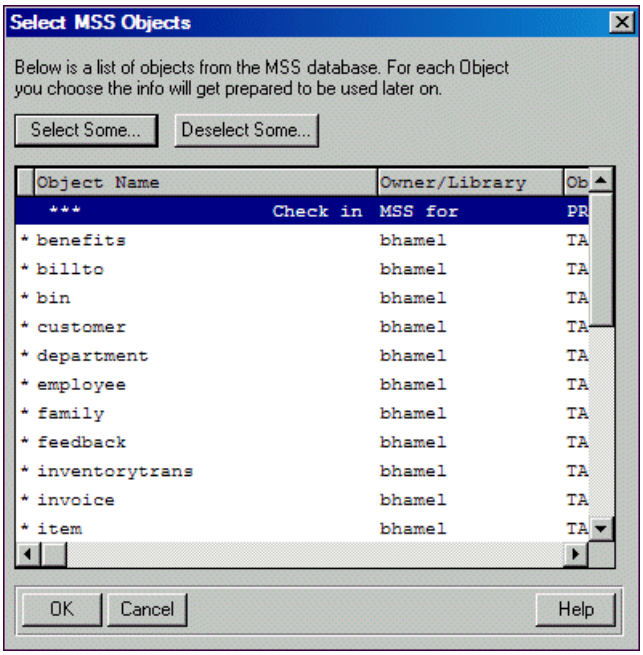
   The following dialog box appears:



Note: You can also choose **Deselect Some** to deselect objects, using the identical matching a pattern options to deselect objects so they are not included in the schema holder.

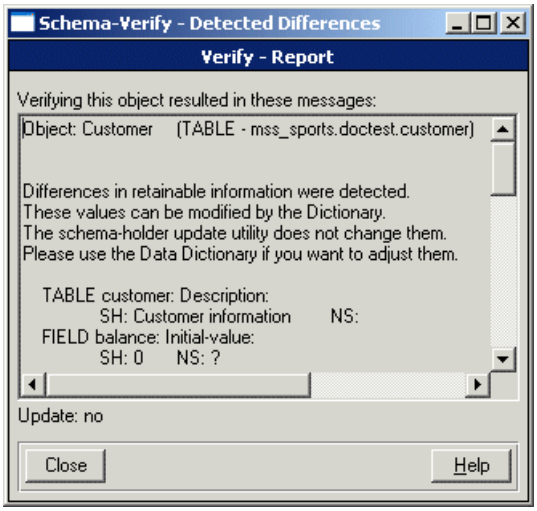
8. Type the pattern that you want to match, then choose **OK** to start the verification.

   If you did not choose to output the differences to a file, dialog boxes that list the objects and the verification results appears:



9. When reading the text of the report, **SH** indicates the value in the schema image; **NS** indicates the value in the MS SQL Server database.

   For example:



10. Choose **Close** to return to the **Schema Verify** dialog box.
11. The utility automatically selects objects with severe differences for updating. You can select or deselect all other objects as you wish.

    As there is no description column in the MS SQL Server database (**NS**), if the Schema image (**SH**) has objects with description column, then the utility discards these differences and are not shown in the report. Note that you **must** resolve retained differences manually. Retained differences appear in subsequent reports until you resolve them.

12. Choose **OK** to start the update or **Cancel** to quit the utility without updating the schema holder.

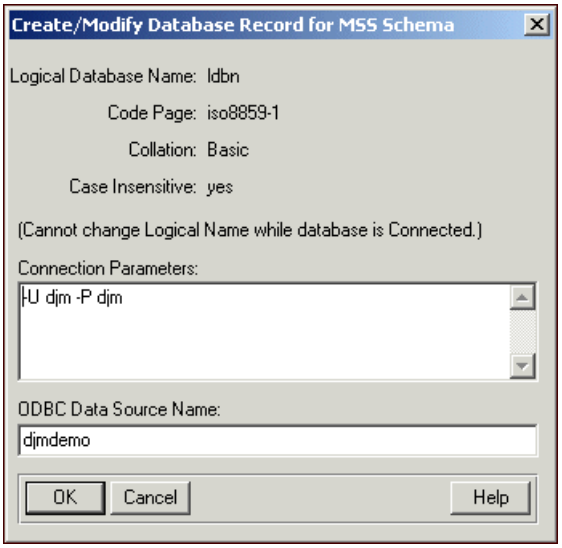
Changing connection information in a schema holder

The Edit Connection Information utility allows you to change a MS SQL Server database's connection information or logical name in the associated schema.

To change connection information for a schema holder from the Data Administration tool:

1. Access the **Data Administration**, 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, OpenEdge uses the new connection parameters.

For details on connection parameters, see Connecting the DataServer on page 189 and *OpenEdge Deployment: 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 could cause unpredictable run-time results.

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.

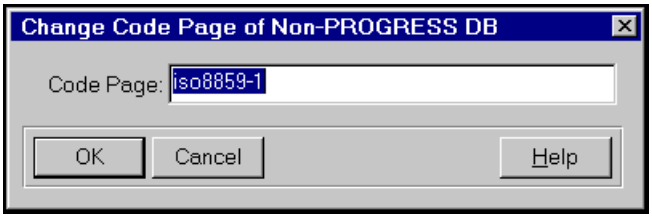
To change a code page in a schema holder from the Data Administration tool:

1. Access the **Data Administration**, 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. If you choose UTF-8 as your schema image code page, your schema holder's code page must also be UTF-8.

   See *OpenEdge Development: Internationalizing Applications* for a complete list of code pages that OpenEdge supports. If you are using an unsupported code page, you are allowed 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, OpenEdge disconnects you to make the change. The **Connect Database** dialog box appears to allow you to reconnect.

Deleting a schema holder

The Delete DataServer Schema utility allows you to delete the schema holder of a data source.

To delete a schema holder from the Data Administration tool:

1. Access the **Data Administration** and select **DataServer** > **MS SQL Server Utilities** > **Delete DataServer Schema**.

   A dialog box appears, prompting you to verify the deletion.

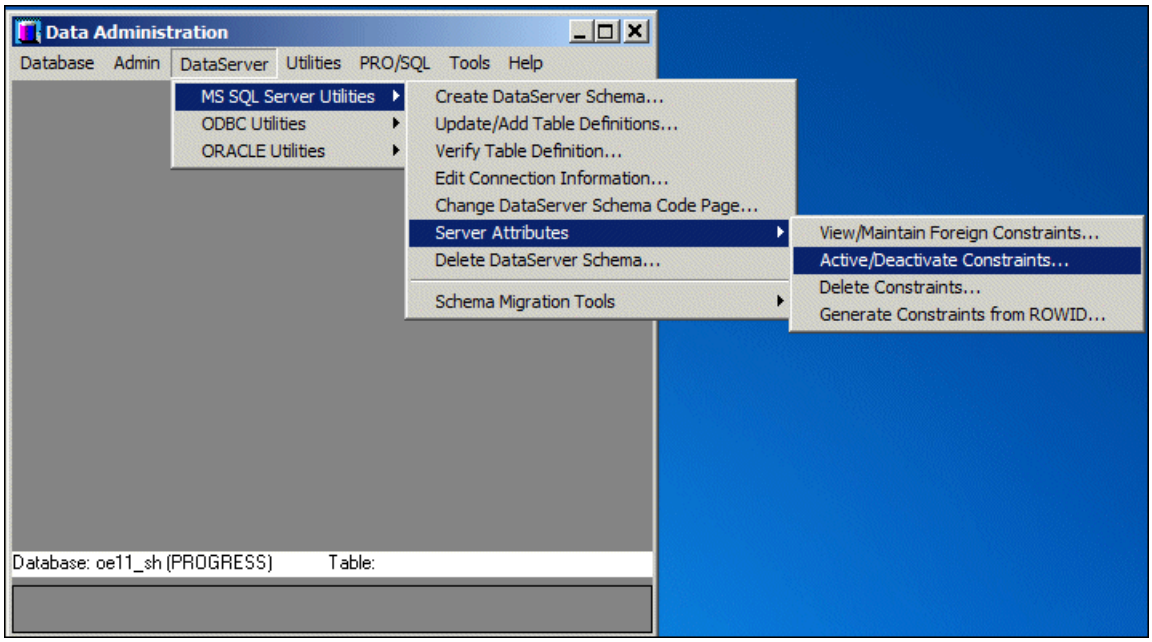
2. Choose **Yes** to verify your selection. After the schema holder is deleted, a confirmation message is displayed.
3. Choose **OK** to return to the **Data Administration** main window.

Managing server attributes

In OpenEdge Release 11, the DataServer for Microsoft SQL Server is enhanced to enable you to define foreign RDBMS constraint definitions within the meta schema of your OpenEdge database for use in migrating your database to a foreign data source. Once migrated to the foreign data source, constraint definitions can be pulled back into your OpenEdge schema holder.

To achieve this, OpenEdge provides a submenu for the MS SQL Server Utilities for creating constraint definitions. The following figure shows the Server Attributes submenu.

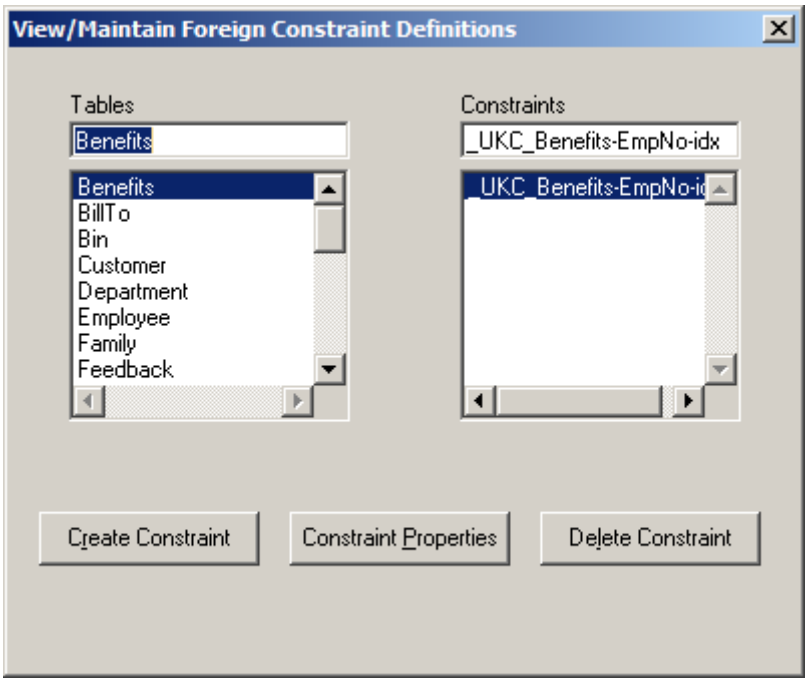
Figure 12: Server Attributes submenu



From the **Server Attributes** submenu, you can access dialog boxes that provide an interface to create, view, and maintain server attributes, and define constraint definitions applied to the data and indexes of the foreign database within your OpenEdge schema holder. The dialog boxes accessed from the **Server Attributes** submenu are as follows:

- **View/Maintain Foreign constraint definitions** — Use the **View/Maintain Foreign Constraint** Definitions dialog box, shown in the figure below, to define and delete constraint definitions as well as view constraint definition properties.

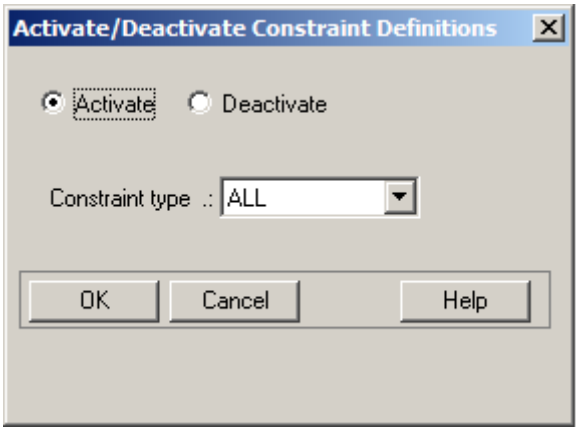
  Figure 13: View/Maintain Foreign Constraint Definitions dialog box



See Defining constraint definitions in Data Administration on page 255 for details on how to define constraint definitions. See Modifying an existing constraint definition on page 264 for details on how to maintain constraint definitions.

- **Activate/Deactivate constraint definitions** — Use the **Activate/Deactivate Constraint** Definitions dialog box, shown in the following figure, to activate or deactivate constraint definitions you defined.

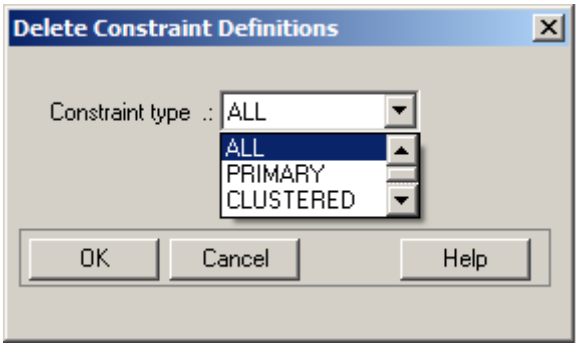
  Figure 14: Activate/Deactivate Constraint Definitions dialog box



  See Activating or deactivating a class of existing constraint definitions on page 265 for details on how to activate and deactivate constraint definitions.

- **Delete constraint definitions** — Use the **Delete Constraint Definitions** dialog box, shown in the following figure, to delete defined constraint definitions.

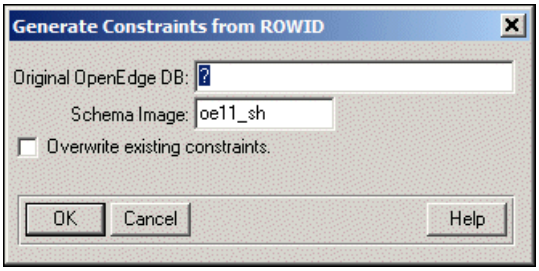
  Figure 15: Delete Constraint Definitions dialog box



  See Deleting one existing constraint definition on page 265 for details on how to delete constraint definitions.

- **Generate constraint definitions from ROWID** — Use the **Generate Constraints from ROWID** dialog box, shown in the following figure, to generate constraint definitions in an OpenEdge database from the ROWID designations that exist in the schema image of a connected schema holder database.

  Figure 16: Generate Constraint from ROWID dialog box



Overview of foreign database constraint definitions

A constraint definition is a property assigned to one column or a set of columns to guarantee data consistency and integrity. You can define the following constraint definitions in your OpenEdge meta schema which you want to migrate to a foreign data source:

- **Primary key constraint definition** — Ensures that each row in a table is uniquely identifiable. Primary keys require a value that is not NULL or NULL-capable. There can only be one primary key, and one primary key constraint definition per table.
- **Unique key constraint definition** — Enforces uniqueness in a column or set of columns. You can have multiple unique key constraint definitions per table.
- **Foreign key constraint definition** — Prevents record updates that break foreign key relationships between tables.
- **Check constraint definition** — Defines an evaluation expression for data validation during an update to a field or insertion of a new row.
- **Default constraint definition** — Defines an initial value for a field.
- **Clustered index "mock" constraint definition** — Defines the index that orders the way records are physically stored in the database. There can be only one clustered index constraint definition per table.This is not an actual constraint definition, and so it is termed a "mock" constraint definition that the definition of the clustered index is migrated to the MS SQL Server database like the other constraint definitions.

Constraint definition names

Within **Data Administration**, constraint definitions are given default names as defined in the following table.

Table 55: Default constraint definition names

Constraint definition type	Default constraint definition name
Primary key	`_PKC_` *`table-name-index-name-`* `idx`
Unique key	`_UKC` *`_table-name-index-name-`* `idx`
Foreign key	`_FKC` *`_parent-index-child index-`* `idx`
Check	`_CC` *`_table-name-field-name-`* `idx`
Default	`_DC` *`_table-name-field-name-`* `idx`
Clustered index	`_CKC` *`_table-name-index-name`* `-idx`

Defining constraint definitions in Data Administration

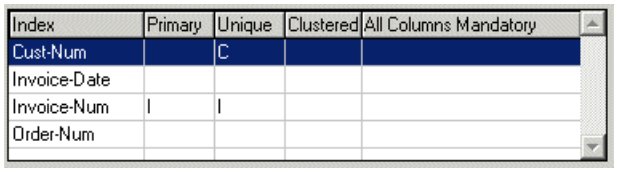
Use the procedures described in the following sections to define constraint definitions in **Data Administration**:

- Defining a primary key constraint definition on page 256
- Defining a unique key constraint definition on page 258

- Defining a foreign key constraint definition on page 259
- Defining a check constraint definition on page 261
- Defining a default constraint definition on page 263

The dialog boxes for defining primary, unique, and clustered index constraint definitions contain a common section that displays current settings for both OpenEdge properties and defined constraint definitions, a sample of which is shown in the following figure.

Figure 17: Index properties



Possible values for each field are:

- **C** — A **C** in a column indicates that a constraint definition of this type is currently defined. The constraint definition can be created in the foreign database if you configure your migration to migrate constraint definitions.
- **I** — An **I** indicates that the property is currently defined your OpenEdge database.

Defining a primary key constraint definition

The following procedure describes how to define a primary key constraint definition.

To define a primary key constraint definition:

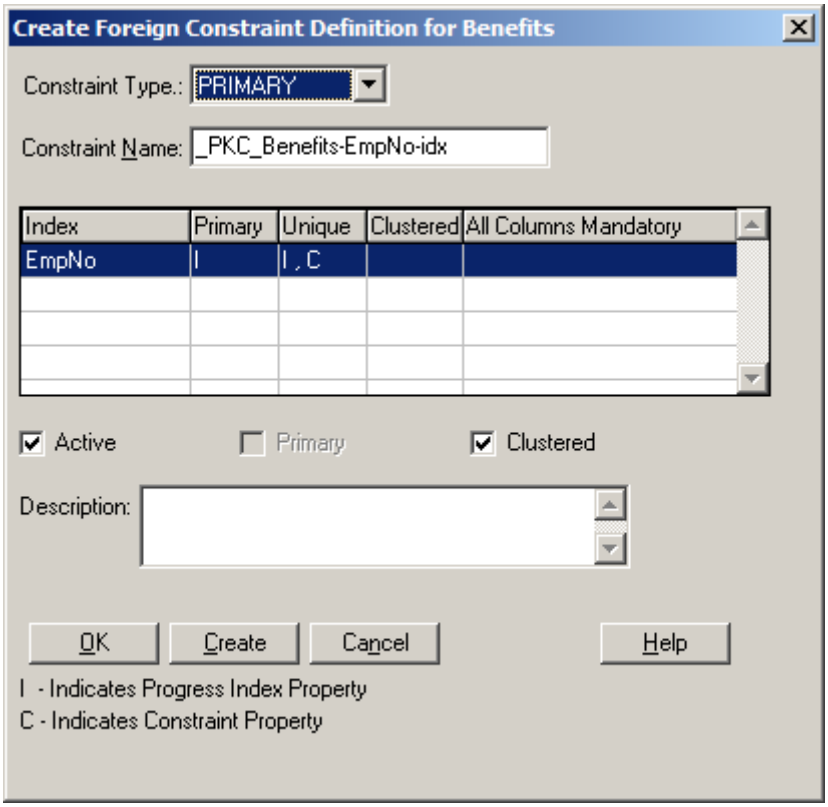
1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **View/Maintain Foreign Constraints**.

   The **View/Maintain Foreign Constraint Definitions** dialog box appears.

3. Select the table for which you are defining a primary key constraint definition and click **Create Constraints**.

   The **Create Foreign Constraint Definitions for *tablename*** dialog box appears, as shown in the following figure.

   Figure 18: Create Foreign Constraint Definitions for tablename dialog box



4. Accept the default constraint definition name, or enter a new one.
5. Select an index for the primary key constraint definition. Only indexes with an **I** in the **All Columns Mandatory** column are recommended for the primary key constraint definition. As long a column is unique, you can select as the primary constraint. Note, however, that the columns designated as the primary constraint are migrated as NOT NULL capable. That is, if you store many records in your OpenEdge database with the unknown value, they cannot be migrated to the foreign data source, given the defined primary constraint on that server object.
6. By default, the primary key constraint definition has the **Active** check box selected. Leave this box selected if you want the constraint definition you are defining to be available for migration.
7. For the DataServer for MS SQL Server, if no clustered index constraint definition is currently defined, the **Clustered** check box is selected. Leave selected if you want the clustered index constraint definition to also be applied to the index you are selecting for the primary key constraint definition.

   Note: Progress Software Corporation recommends defining primary and clustered constraint definitions for the same index whenever possible.

8. Optionally, add a description of the constraint definition in the **Description** box.
9. Click **OK** to save the constraint definition and exit the dialog box.
10. Click **Create** to save the constraint definition and remain in the dialog box and continue defining constraint definitions for the selected table.

If the primary constraint definition is incorrectly defined, a message is displayed and the constraint definition is not created.

The system fails to generate a constraint definition when an index is not defined as unique by the presence of either an OpenEdge unique attribute (indicated by an **I**) or by a unique constraint definition (indicated by a **C**) in the **Unique** column.

Defining a unique key constraint definition

The following procedure describes how to define a unique key constraint definition. Alternately, you can define unique constraint definitions during migration.

To define a unique key constraint definition:

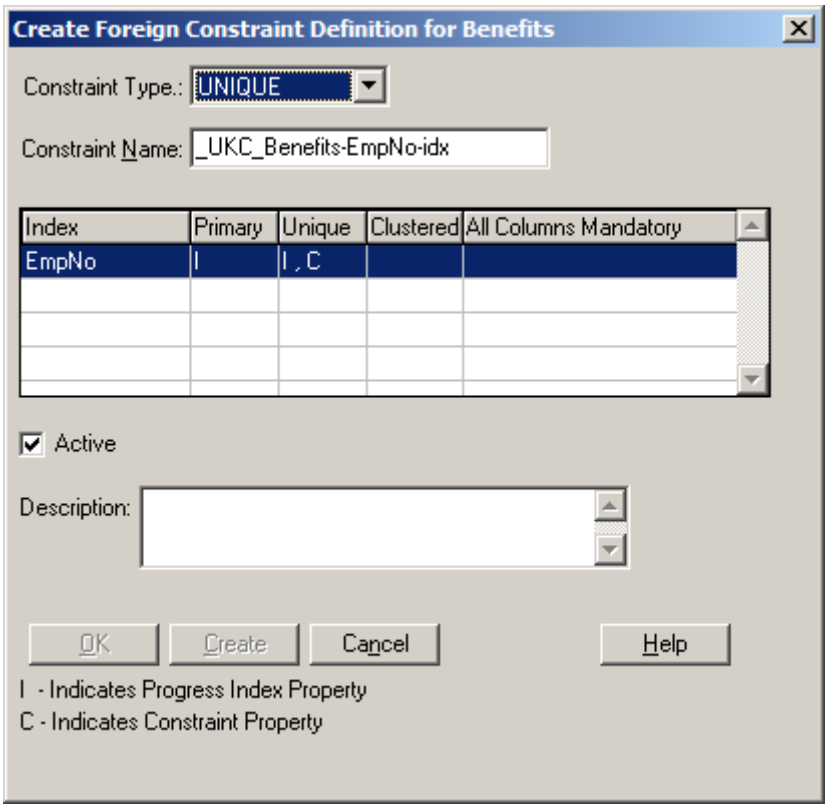
1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **View/Maintain Foreign Constraints**.

   The **View/Maintain Foreign Constraint Definitions** dialog box appears.
3. Select the table for which you are defining a unique key constraint definition and click **Create Constraints**.

   The **Create Foreign Constraint Definitions for *tablename*** dialog box appears, as shown in figure below.
4. Select **UNIQUE** from **Constraint Type.**

   The **Create Foreign Constraint Definitions for *tablename*** for a unique key dialog box, appears as shown in the following figure.

Figure 19: Create Foreign Constraint Definitions for tablename dialog box for unique constraint definitions



5. Accept the default constraint definition name, or enter a new one.
6. Select an index for the unique key constraint definition.
7. By default, the unique key constraint definition has the **Active** check box selected. Leave this box selected if you want the constraint definition you are defining to be available for migration.
8. Optionally, add a description of the constraint definition in the **Description** box.
9. Click **OK** to save the constraint definition and exit the dialog box.
10. Click **Create** to save the constraint definition and remain in the dialog box to define more constraint definitions for the selected table.

Defining a foreign key constraint definition

The following procedure describes how to define a foreign key constraint definition.

To define a foreign key constraint definition:

1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **View/Maintain Foreign Constraints**.

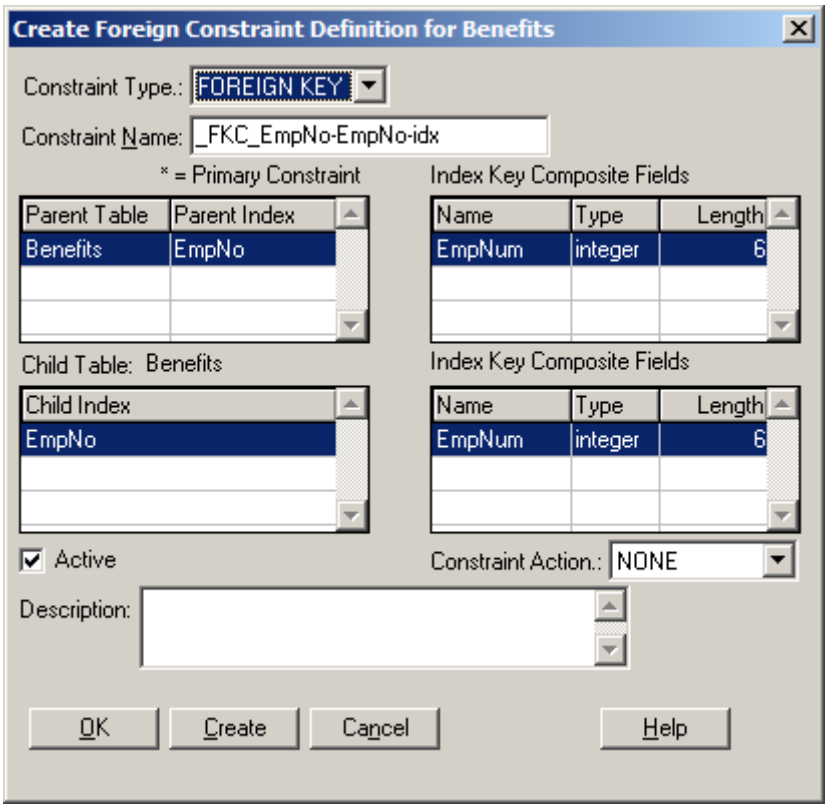
   The **View/Maintain Foreign Constraint Definitions** dialog box appears.
3. Select the table for which you are defining a foreign key constraint definition and click **Create Constraints**.

   The **Create Foreign Constraint Definitions for** *tablename* dialog box appears, as shown in figure below.

4. Select **FOREIGN KEY** from **Constraint Type.**

   The **Create Foreign Constraint Definitions for** *tablename* dialog box, appears as shown in the following figure.

   Figure 20: Create Foreign Constraint Definitions for tablename dialog box for foreign key constraint definitions



5. Accept the generated default constraint definition name, or enter a new one.
6. The **Parent Table/Parent Index** lists the indexes from all of the other tables defined in the OpenEdge database, available for selection. For the DataServer for MS SQL Server, the list shows primary and unique indexes. Below the parent index information is the **Child Index** list. The **Child Index** list shows the available indexes for the selected table. For both parent and child indexes, the **Index Key Composite Fields** shows the columns that comprise the indexes. Select an index from the parent and child lists with matching composite fields.
7. By default, the foreign key constraint definition has the **Active** check box selected. Leave this box selected if you want the constraint definition you are defining to be available for migration.
8. Select an appropriate action item from the **Constraint Action**.

   Constraint actions can be **NONE**, **CASCADE**, **SET NULL**, or **SET DEFAULT**. A typical constraint definition action ensures that the data integrity, in case you need to delete data in the rows of a referenced table.

   The **NONE** constraint action ensures data is not deleted from a row which exists in a foreign key table that references the value in the referenced table.

   The **CASCADE** constraint action ensures that whenever you delete rows in the master (referenced) table, the respective rows of the child (referencing) table with a matching foreign key column is deleted.

   The **SET NULL** constraint action ensures that the foreign key values in the referencing row are set to `NULL` when the referenced row is deleted.

The **SET DEFAULT** constraint action ensures that foreign key values in the referencing row are set to the column default when the referenced row is deleted.

9. Add a description of the constraint definition in the **Description** fill-in, if desired.
10. Click **OK** to save the constraint definition and exit the dialog box. Click **Create** to save the constraint definition and remain in the dialog box and continue defining constraint definitions for the selected table.

Defining a check constraint definition

A check constraint definition is a server-side validation. The following steps demonstrate how to define a check constraint definition.

To define a check constraint definition:

1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **View/Maintain Foreign Constraints**.

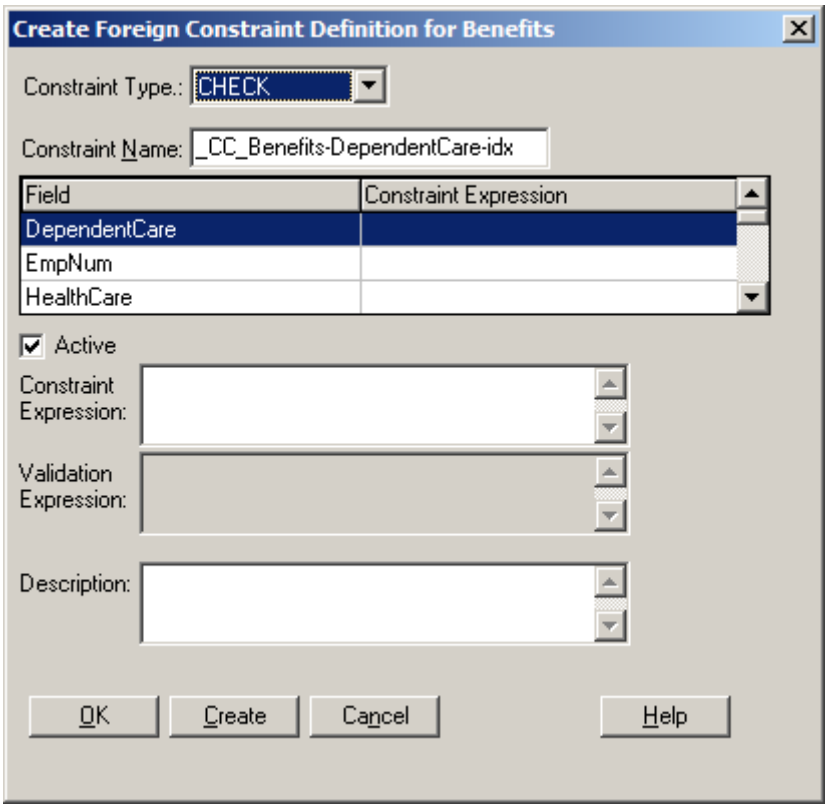
   The **View/Maintain Foreign Constraint Definitions** dialog box appears.
3. Select the table for which you are defining a check constraint definition and click **Create Constraints**.

   The **Create Foreign Constraint Definitions for** *tablename* dialog box appears, as shown in figure below.

4. Select **CHECK** from **Constraint Type.**

   The **Create Foreign Constraint Definitions for *tablename*** for a dialog box, appears as shown in the following figure.

   Figure 21: Create Foreign Constraint Definitions for tablename dialog box for check constraint definitions



5. Accept the default check constraint definition name, or enter a new one.
6. Select a field from the **Field** list.
7. By default, the check constraint definition has the **Active** check box selected. Leave this box selected if you want the constraint definition you are defining to be available for migration.
8. Enter an expression for server-side validation in the **Constraint Expression** field.

   Caution: The expressions entered are not checked for correct syntax. If the expression is not valid on the server, the check constraint definition is rejected when building check constraint definition objects on the server.

9. A validation expression is a client-side validation check that conforms to OpenEdge syntax requirements and only takes place during interactive screen I/O. This is in contrast to a check constraint, which is tied to the actual database column, not the screen dialog.

   Caution: The validation expression is a read-only value for reference. Use the **Field Properties** dialog box of the **Data Dictionary** to maintain validation expressions.

10. Optionally, add a description of the constraint definition in the **Expression Description** box.
11. Click **OK** to save the constraint definition and exit the dialog box. Click **Create** to save the constraint definition and remain in the dialog box and to define more constraint definitions for the selected table.

Defining a default constraint definition

The following procedure describes how to define a default constraint definition. Alternately, you can define default constraint definitions during migration.

To define a default constraint definition:

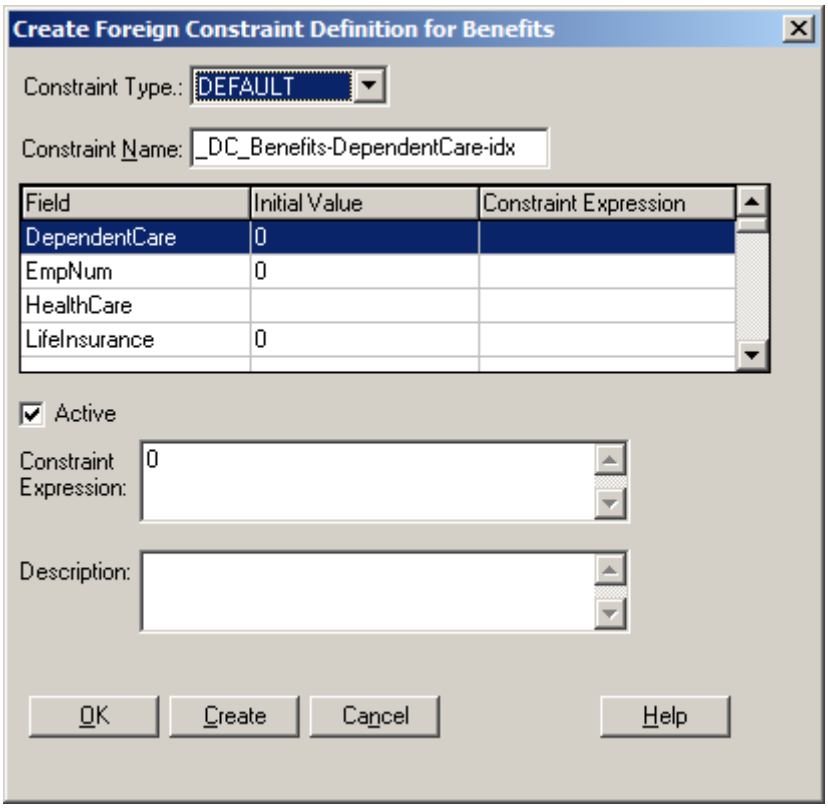
1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **View/Maintain Foreign Constraints**.

   The **View/Maintain Foreign Constraint Definitions** dialog box appears.
3. Select the table where you are defining a default constraint definition and click **Create Constraints**.

   The **Create Foreign Constraint Definitions for *tablename*** dialog box appears, as shown in Figure 18: Create Foreign Constraint Definitions for tablename dialog box on page 257.
4. Select **DEFAULT** from **Constraint Type.**

   The **Create Foreign Constraint Definitions for** *tablename* dialog box, is modified as shown in the following figure.

   Figure 22: Create Foreign Constraint Definitions for tablename dialog box for default constraint definitions



5. Accept the default constraint definition name, or enter a new one.

6. Select a field from the **Field** list.
7. By default, the default constraint definition has the **Active** check box selected. Leave this box selected if you want the constraint definition you are defining to be available for migration.
8. A default constraint definition is a server-side validation. Enter a value for server-side population of the field in the **Constraint Expression**.

   Caution: The expressions entered are not checked for correct syntax. If the expression is not valid on the server, the default constraint definition is rejected when building default constraint definition objects on the server.

   For fields with an **Initial Value**, the **Constraint Expression** is populated with the initial value. The **Constraint Expression** value for the Default constraint definition is not required to match the **Initial Value**, however, they should normally match as there may otherwise be conflicts regarding what value to use as the default when creating a server record. The initial value is used in a client-side template record when new records are created, and overrides default constraint definitions on the server side when records are created in an OpenEdge application. The default constraint definition is applied only when an initial value is not supplied by the template record of an OpenEdge application and the column is also left blank by the user when the record is created.

   Note: Since default constraints are tied to the server column, non-OpenEdge applications will get their defaults from the constraint migrated from the default constraint definition.

9. Optionally, add a description of the constraint definition in the **Description** box.
10. Click **OK** to save the constraint definition and exit the dialog box. Click **Create** to save the constraint definition and remain in the dialog box and continue defining constraint definitions for the selected table.

Modifying an existing constraint definition

Once you have defined a constraint definition, you can change a subset of a constraint definition's properties. The following steps demonstrate how to modify a constraint definition.

To modify a constraint definition:

1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select**DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **View/Maintain Foreign Constraint Definitions**.

   The **View/Maintain Foreign Constraint Definitions** dialog box appears.
3. Select the table and constraint definition you want to modify, and click **Constraint Properties**.

   The **Modify Foreign Constraint Definitions for *tablename*** dialog box appears.
4. Modify your constraint definition as required.

   The **Modify Foreign Constraints for *tablename*** dialog box is very similar to the **Create Foreign Constraint Definitions for *tablename*** dialog box, with the exception being that only modifiable fields are active.

Note: If the constraint definition property you want to change is not modifiable, you can delete the existing constraint definition and define a new one.

5. Click **OK** to save the changes to the constraint definition and exit the dialog box. Click **Create** to save the changes to the constraint definition and remain in the dialog box.

Activating or deactivating a class of existing constraint definitions

Once you have defined constraint definitions, you can activate or deactivate a whole class of constraint definitions at once. When a constraint definition is deactivated, it is unidentified and disabled during a subsequent migration.

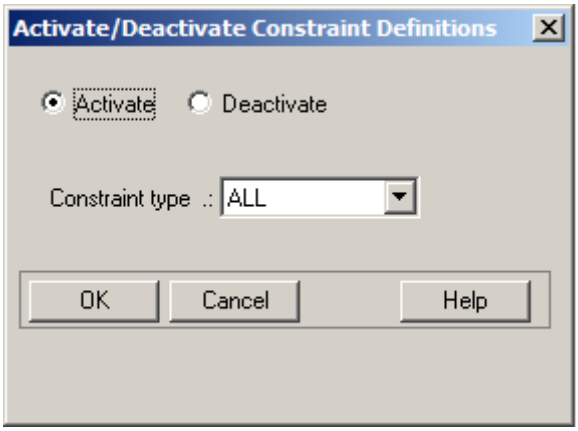
The following steps demonstrate how to activate or deactivate a class of constraint definitions.

To activate a class of constraint definitions:

1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **Activate/Deactivate Constraints**.

   The **Activate/Deactivate Constraint Definitions** dialog box appears, as shown in the following figure.

   Figure 23: Activate/Deactivate Constraint Definitions dialog box



3. Chose the **Activate** or **Deactivate** toggle.
4. Select a constraint definition type or ALL.
5. Click **OK** to activate or deactivate constraint definitions and exit the dialog box, or click **Cancel** to exit without changing the state of the constraint definitions.

Deleting contstraint definitions

Once you have defined a constraint definition, you can delete it. The following procedures demonstrate how to delete all constraint definitions or one class of constraint definitions.

Deleting one existing constraint definition

To delete one constraint definition:

1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **View/Maintain Foreign Constraints**.

   The **View/Maintain Foreign Constraint Definitions** dialog box appears.
3. Select the table and constraint definition you want to delete.
4. Click **Delete Constraints** to delete the selected constraint definition.

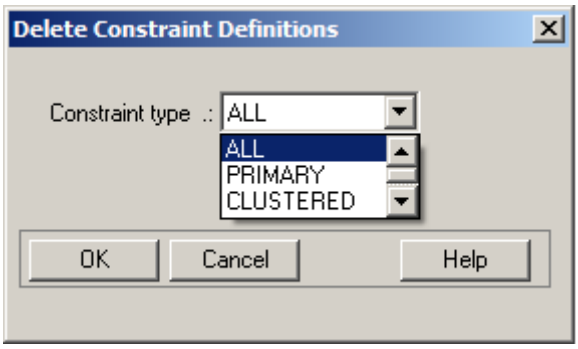
Deleting all existing constraint definitions

To delete all constraint definitions or all of a particular type of constraint definition:

1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Select **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **Delete Constraints**.

   The **Delete Constraints** dialog box appears, as shown in the following figure.

   Figure 24: Delete Constraint Definitions dialog box



3. Select the **Constraint type**, either **ALL** or a specific sub-set.
4. Click **OK** to delete the constraint definitions. You are prompted to confirm your decision. Click **Yes** to delete the constraint definitions or **Cancel** to exit without deleting the constraint definitions.

Generating constraint definitions from ROWID

During the pull phase of a schema migration, choosing **Select 'Best' ROWID Index** with **Foreign Schema**, defers to the server API to determine the "best" index choice. The best index choice is based on many factors specific to the server and may provide insight into the server benefits of a particular index selection for ROWID.

You can choose to re-migrate your OpenEdge schema based on the previously derived ROWID designations, by using one of the following:

- Results of the **Select 'Best' ROWID Index** migration option from your schema pull
- Results of other **ROWID** sub-section migration options from your schema pull
- Results of the **Designate Primary/Clustered index as ROWID** option from your schema pull

Choosing, the **Select 'Best' ROWID Index** option might remove the legacy `PROGRESS_RECID` columns from foreign tables of the schema and enable new ROWID designation possibilities.

To harness the performance benefits of wise ROWID choices, you must import back primary and clustered key constraint definitions from your current schema holder into your original OpenEdge database in the form of constraint definitions. The wise ROWID choices are derived either from one of the previous ROWID designations from previous migrations or from pull exercises, which also designate a primary or clustered key index.

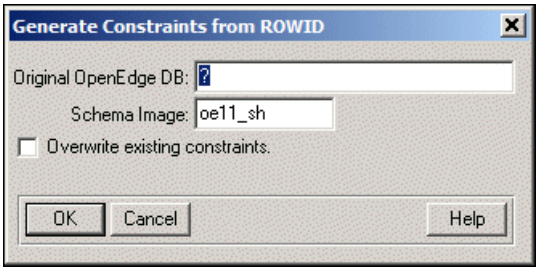
The performance benefits are accomplished by using the Generate constraints for ROWID option. Once the constraint definitions are back into the original OpenEdge database, they can be used for remigration to the server by using the Migrate Constraints option in the Advanced Options migration box. By doing so, the accumulation of designated primary and clustered indexes are repeated in the remigration, as constraint definitions always take priority over other ROWID designation options. Wherever the primary and clustered constraint definitions are found during the remigration, they are used to designate ROWID.

To generate constraint definitions from ROWID:

1. Start **Data Administration** and connect to the OpenEdge database you are migrating.
2. Choose **DataServer** > **MS SQL Server Utilities** > **Server Attributes** > **Generate Constraints from ROWID**.

   The **Generate Constraints from ROWID** dialog box appears, as shown in the following figure:

   Figure 25: Generate Constraints from ROWID dialog box



3. Specify your OpenEdge database name.
4. If you want to override any existing constraint definitions, check **Overwrite existing constraint definitions**.
5. Click **OK** to generate primary and mock clustered index constraint definitions from designated ROWID wherever primary and mock clustered index constraint definitions do not exist (or overwrites existing constraint definitions).

In this scenario, your original OpenEdge database is populated with primary and clustered constraint definitions based on the ROWID designations of the current schema image. The selecting the overwrite option indicates that it is permissible to replace existing constraint definitions with the definition derived in the schema holder.

After choosing to generate the constraint definitions based on ROWID, you can then consider a re-migrating to get the primary and clustered indexes constructed in the foreign schema by setting on the **Migrate Constraints** option on the re-migration, noting that you may need to clear existing ROWID designations in order to assign new ones.

Migrating an OpenEdge database to MS SQL Server

The DataServer for MS SQL Server supports the OpenEdge DB to MS SQL Server utility that allows you to migrate an OpenEdge database to a foreign data source through the use of the MS SQL Server DataServer. While the DataServer makes a data source conform to an OpenEdge database, this utility provides compatibility in the opposite direction. It copies an existing OpenEdge database schema into a target data source.

The OpenEdge DB to MS SQL Server 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 OpenEdge database

The MS 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 MS SQL Server features that are not supported by the migration utility.

The OpenEdge DB to MS SQL Server utility requires a local OpenEdge database.

Preparing a data source for the utility

The OpenEdge DB to MS SQL Server migration utility translates definitions for OpenEdge objects into an SQL DDL script that loads into your target data source. It automatically makes adjustments in the target data source in order to provide the functionality of the OpenEdge and target systems.

When you use the OpenEdge DB to MS SQL Server utility to make a MS SQL Server database compatible, it converts arrays by using one column in the MS SQL Server database for each extent of the OpenEdge 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, an OpenEdge 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`.

Note: OpenEdge BLOB must be set to NOT NULL in MS SQL, that is, any MS SQL application that inserts records into a table also initializes the record with EMPTY_BLOB() function as OpenEdge DataServer cannot access a NULL BLOB column.

Migrating defined constraint definitions to your foreign data source

Once your constraint definitions are defined in your OpenEdge database, you need to migrate them to your foreign data source. This migration is accomplished with the Schema push utilities, OpenEdge DB to MS SQL Server. constraint definition migration is enabled by default, so execution of a schema push will automatically migrate any constraint definitions. If you do not wish your constraint definitions to be migrated, uncheck the **Migrate Constraints** check box on the **Advanced** migration dialog box, or deactivate them prior to starting the migration. See Activating or deactivating a class of existing constraint definitions on page 265 for information on deactivating constraint definitions. The user interface of the schema push utilities has changed.

Constraint definitions first defined in your foreign data source can be added to your OpenEdge schema holder during a schema pull, but are read only. To extract the read-only constraints into your OpenEdge database for reusing on subsequent migrations, use the **Generating Constraints from ROWID** option. For information on generating constraint definitions from ROWID, see Generating constraint definitions from ROWID on page 266.

Running the OpenEdge DB to MS SQL Server utility

The OpenEdge DB to MS SQL Server utility runs in Windows with a DataServer accessing MS SQL Server through an ODBC driver.

You can run the utility interactively from Windows or UNIX, or in batch mode. All the logs are logged at `protomss.log`.

To run the OpenEdge DB to MS SQL Server utility interactively:

1. Create a target MS SQL Server. You must use an empty target data source when you run the OpenEdge DB to MS SQL Server utility.
2. Configure your ODBC driver to connect to your new target data source.
3. Start the OpenEdge client and connect to the OpenEdge 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 OpenEdge client. The values that you specify for these parameters must match the code page that the target data source uses.

4. Start the utility using one of the following methods:

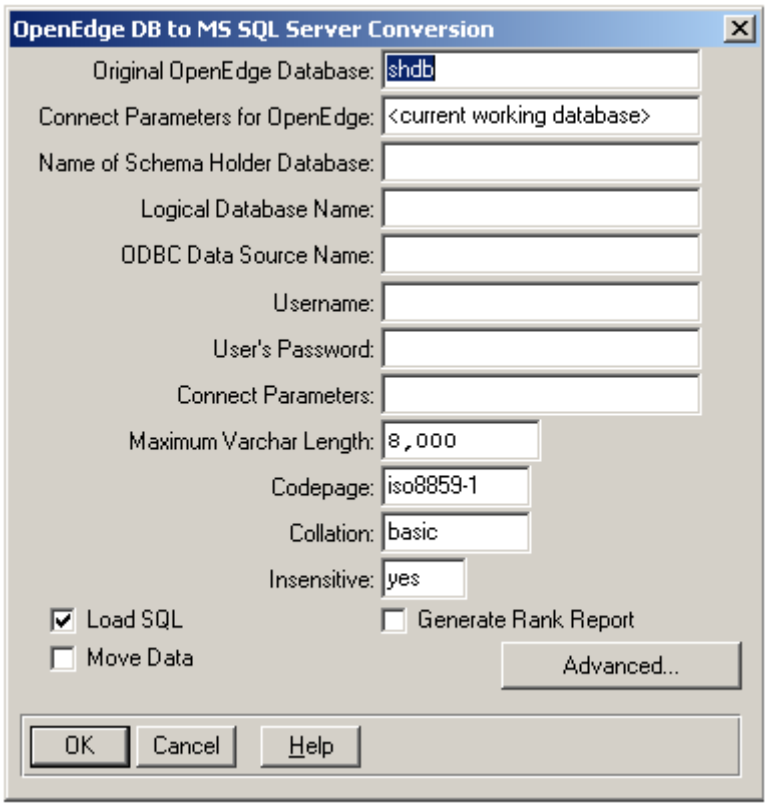
   - From the **Data Administration**, choose **DataServer** > **MS SQL Server Utilities** > **Schema Migration Tools** > **OpenEdge DB to MS SQL Server**.
   - Independently run the OpenEdge DB to MS SQL Server utility in the system prompt using the `protomss` utility:

     ```
     pro -p prodict/mss/protomss.p
     ```

Note: All the warnings, errors, or causes of termination in the conversion process can be found in the log file, `protomss.log`.

The following screen appears:

Figure 26: OpenEdge to MS SQL Server Conversion dialog box



5. Enter the information required.

   The following table describes the information you are prompted to enter.

Table 56: OpenEdge-to-MS SQL Server Conversion utility

Interface element	Description
Original OpenEdge Database	**Default**: <blank> Specifies a source OpenEdge database to be migrated. If this interface element is unset and the user has already set a current working database, the conversion utility uses the current working database to set the value for the original OpenEdge database. **Dependencies**: If this element is unset, and no other working database is pre-connected, the migration cannot proceed.

Interface element	Description
Connect parameters for OpenEdge	**Default**: current working database parameters Specifies startup parameters used to connect the Original OpenEdge Database. **Dependencies**: It is required that the Original OpenEdge Database is started in single-user or read-only mode. Therefore, if you provide your own connect parameters for OpenEdge, you must include settings to connect to the Original OpenEdge Database in the single-user mode or read-only mode to avoid connection failure.
Name of Schema Holder Database	**Default**: <blank> Specify the new schema-holder name, in which the resultant logical database will reside. **Dependencies**: If a schema holder database is not set, the migration cannot proceed.
Logical Database Name	**Default**: <blank> Specify the logical database name associated with the schema image of the foreign data. **Dependencies**: If the logical database name is not set, the migration cannot proceed. The **Logical Database Name** must be different from the **Name of the Schema Holder Database**.
ODBC Data Source Name	**Default**: <blank> Specify the ODBC Data Source name used to configure the OEDB connection to the foreign data source. **Dependencies**: If the **ODBC Data Source Name** is not set, the migration cannot proceed. The value for **ODBC Data Source Name** can be the same as Original OpenEdge Database name but must be different from the **Name of the Schema Holder Database**. The ODBC DNS must be properly specified in the ODBC Data Source Administrator of your system. If the **Load SQL** interface element is set to YES, ODBC DNS must also be properly configured to connect to the foreign Data Source.
Username	**Default**: <blank> Specify the user name for the target data source. **Dependencies**: If you do not set the user name required for authentication, the migration terminates with an error.

Interface element	Description
User's Password	**Default**: <blank> Specify the password of the user for the target data source. **Dependencies**: If you do not set the password required for authentication, the migration terminates with an error.
Connect parameters for the DataServer connection	**Default**: <blank> Specifies the startup parameters for the DataServer and the target foreign data source.For more information on connection parameters, see Connecting the DataServer on page 189 for connection parameters. **Dependencies**: Connect parameters are not mandatory. If supplied, they must be formatted as a comma-separated list of names or name-value pairs, without white-spaces, that comprise the `-Dsrv` parameter value passed to the DataServer connection. Some of the parameters are parsed into the DataServer run-time, while others in the list are forwarded onto the target foreign data source. If an invalid or improperly formatted connection parameter is parsed during migration, the migration terminates with an error.
MaximumVarchar Length	**Default**: The unconditioned default is 8000. It ranges from 4000 to 8000. Specify the maximum number of characters for a VARCHAR type. When a single-byte character set is utilized (as specified by the **Code Page** interface element), it must be a positive value less than or equal to 8000. The default threshold value of 8000 is the maximum number of characters that can be stored in a VARCHAR data type in MS SQL Server. When Unicode character data is utilized by the application, the default threshold value is limited to 4000. This value can also be overridden. During OpenEdge to MS SQL Server migration, whenever the column length of character data exceeds the value specified for `VARLENGTH`, the column is instead migrated as a Large Object (LOB) data type that can store data that far exceeds the data that can be stored in a VARCHAR data type. **Dependencies**: When `MSSCODEPAGE` is set to `utf-8`, the value set for `VARLENGTH` must be a positive value less than or equal to 4000 as it is the maximum number of characters that can be stored in a NVARCHAR data type in MS SQL Server due to UCS-2 character expansion. And, character lengths exceeding the threshold value set for `VARLENGTH` are converted to Unicode Large Object types. When an OpenEdge database stores UTF-8 data, each OpenEdge character can be as large as 4 bytes, potentially reducing the Maximum Varchar Length to 2000 characters on the server. But generally, MS SQL Server only supports the UCS-2 character set that allows 4000 UTF-8 characters in the UCS-2 range to be migrated into a Varchar column on the server.

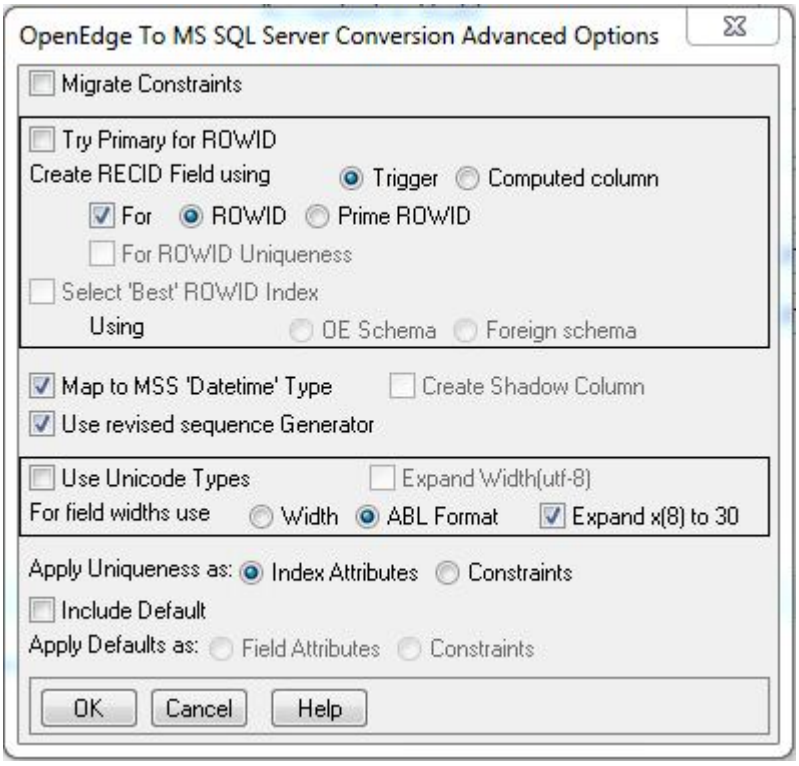
Interface element	Description
Codepage	**Default**: iso8859-1 when UNICODETYPES is disabled (that is, set to **NO**). Else, the default is utf-8. Specifies the corresponding OpenEdge name for the code page with which the MSS Database is compatible. Use UTF-8 if Unicode support is desired. If you use a code page that OpenEdge does not support, you must supply a conversion table that translates between the OpenEdge client code page and the code page that your data source uses. For a complete discussion of code pages, see *OpenEdge Development: Internationalizing Applications*. It is not mandatory to pass this value during batch migration. You can use **Change DataServer Schema Code Page** utility to add the code page information to the schema holder later but before you start using the DataServer to read and write data. For more information on changing code page, see Changing a code page in a schema holder on page 251. **Dependencies**: The specified codepage must exist in the `convmap.cp` file of your Progress environment in order to be used during the migration.
Collation	**Default**: Basic Specifies the collation name with which the OpenEdge client must collate and weigh code page values. **Dependencies**: The specified collation must exist in the `convmap.cp` file of your Progress environment in order to be used during the migration unless you are using a Unicode code page. In case you are using a Unicode code page, then the code page name can correspond to one of the International Components for Unicode (ICU) collations that provide linguistic sorting of Unicode data based on the Unicode Collation Algorithm.
Insensitive	**Default**: NO Specifies if the code page that you are matching on the foreign data source is or isn't case insensitive. Provide `YES` if your server code page is case insensitive, else retain the default value `NO`. **Dependencies**: By default, the code pages in MS SQL Server are case-insensitive. This matches the default sensitivity in OpenEdge. So, if you are using a case-insensitive code page in MS SQL Server, Progress recommends setting this value to `YES`.
Load SQL	**Default:** Not selected (NO) Allows you to specify whether you want the utility to execute the SQL script that is generated in order to create the schema in the target empty MS SQL Server database on the server that was configured through your ODBC Data Source Name. Specify **YES** to enable this behavior.

Interface element	Description
Move Data	**Default:** Not selected (NO) Allows you to specify whether to populate the foreign data source with your OpenEdge data after the schema has been migrated. 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. **Dependencies**: **Load SQL** must be selected in order to move data.
Genenerate Rank Report	**Default:** Not selected (NO) When selected, this option generates a ranking selection report, `rnkreppdb.out`. This report contains details of the selections you made in the **OpenEdge-to-MS SQL Server Conversion** utility.
Advanced	Click **Advanced** to open the **OpenEdge DB to MS SQL Server Conversion Advanced Options** dialog box. The purpose of this dialog box is to reduce clutter on the main dialog box and provide logical grouping of fields on it. See Table 57: OpenEdge DB to MS SQL Server Conversion Advanced Options on page 275 for the details you must enter.

6. Click **Advanced** to edit the advanced settings.

   The OpenEdge DB to MS SQL Server Conversion dialog box appears:

   Figure 27: OpenEdge DB to MS SQL Server Conversion Advanced Options



7. Enter the information required.

The following table describes the information you are prompted to enter.

Table 57: OpenEdge DB to MS SQL Server Conversion Advanced Options

Interface Elements	Description
Migrate Constraints	Selecting this option enables you to migrate any constraint definitions defined using the procedures defined in Overview of foreign database constraint definitions on page 255. Now, the **Migrate Constraints** box is not selected by default in order to maintain backward compatibility with previous releases. **Note:** In OpenEdge Version 11.0 through 11.2, **Migrate Constraints** was selected by default and could still maintain backward compatibility. OpenEdge now has a new migration algorithms may be invoked as soon as a non-default option is selected from the ROWID-related options of the **Advanced** menu.
ROWID sub-section	Select to specify ROWID and RECID. See ROWID sub-section of MS SQL Server Conversion on page 235 for more information. **Note:** While the **Migrate Constraints** option is not a direct mechanism for establishing ROWID and RECID, in contrast to the options in the **ROWID sub-section**, the presence of a Primary or Clustered constraint migrated from OpenEdge supersede the specifications for ROWID and RECID selected in the **ROWID sub-section** that might have otherwise derived a Primary and/or clustered index corresponding to a ROWID/RECID designation.
Map to MSS Datetime Type	Selected by default to provide backward compatibility to legacy DATETIME data types in MS SQL Server. Uncheck box to map to datetime server data types with time zone components that support time zone component with a stronger sub-section time precision. For details, see DataServer for MS SQL Server support for datetime data types on page 61.

Interface Elements	Description
Create Shadow Columns	When SQL Server is configured with a case sensitive code page, this box is available for selection. Selecting it provides case insensitivity compatible with the behavior of an OpenEdge database. **Note:** If **Create Shadow Columns** is not selected with a case-sensitive code page, then case-insensitivity in the OpenEdge schema supports the UPPER() function in run-time SQL translation. Ensure that you use this option appropriately because the inclusion of the UPPER() function in SQL translation might negatively impact server performance. **Dependencies**: MS SQL Server uses case insensitive by default, which is why the Insensitive option on the main menu of the Conversion options is YES by default. If the Insensitive option remains equal to YES, then the Create Shadow Column option is disables and ignored.
Use generator	Selected by default. Enables revised generator. Not selecting the option retains older sequence generator, particularly if SET-VALUE sequence statement is used extensively in your program.
Use Unicode Types	Maps OpenEdge character fields to MS SQL Server Unicode data types. **Dependencies**: Selecting this option changes the default code page to UTF-8 and directs the schema migration to convert all character and CLOB data types in the OpenEdge database to Unicode data types on the server.
Expand width (utf-8)	Doubles the size of character fields to accommodate Unicode expansion of single-byte data based on specified field width criteria.
For field widths use:	When pushing fields to a foreign data source, you can select one of two character field formatting options: • **Width** — Uses the value of the _**width** field in the _field record. **Note:** Recommended for Unicode implementations. For more information on column width, see Column width on page 294. • **4GL Format** — Compiles with the current default width specified. (default) If you select the **4GL Format** option, you have an additional setting to define: • **Expand x(8) to 30** — This setting is on by default to indicate that the format for the character fields defined as x(8) will be created as 30 characters. **Dependencies**: The **Width** and **4GL Format** options are mutually exclusive options and you cannot use the **Expand x(8) to 30** setting with the **Width** option.

Interface Elements	Description
Apply Uniqueness as	The **Apply Uniqueness as** allows you to specify how uniqueness is defined. The default, **Index Attributes**, provides backward compatibility by marking the UNIQUE attribute on a created index. The **Constraints** option defines named constraint definitions for uniqueness independent of the index definition. **Dependencies**: Existing UNIQUE constraints defined on the server always take precedence over any constraint definitions that would otherwise be redundantly generated using this option.
Include Defaults	The **Include Defaults** check box is indicates whether or not, as part of migration, you want to include OpenEdge initial values in the fields of the foreign data source's column schema. The **Apply Default as** option further enables you to decide how the initial values must be expressed in your database and application environment.
Apply Defaults as	The **Apply Default as** enables you to specify how defaults must be applied to the DataServer implementation. The **Include Defaults** check box is enhanced with an **Apply Default as** radio set that enables you to specify if your defaults must be defined as **Field Attributes** (default) or **Constraints**. The **Field Attributes** option provides backward compatibility. The **Constraints** option specifies constraint definitions if a constraint definition doesn't already exist. If the user selects **Constraints**, then after the migration, all the initial values defined for migration become named default constraint definitions on the server.

To perform a complete migration of your OpenEdge database to a target data source, you must enter information in all appropriate fields and select all appropriate check boxes.

The utility creates a schema holder, updates the empty target data source that you created to contain the objects stored in your OpenEdge 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 conversion in batch mode

Use the following steps to run the OpenEdge-to-MS SQL Server Conversion utility in batch mode:

1. Create a target MS SQL Server database. You must use an empty target data source when you run the OpenEdge DB to MS SQL Server utility.
2. Configure your ODBC driver to connect to your new target data source.

   On your client machine, export the environment variables listed the following tables:

   - Table 58: OpenEdge-to-MS SQL Server Conversion utility batch parameters on page 278
   - Table 60: OpenEdge-to-MS SQL Server Conversion Advanced options batch parameters on page 285

   Since the batch mode process cannot provide feedback to the user, any warnings, errors, or causes of termination can be found in `protomss.log`.

The following graphic is to help you map the environment variables with the migration options described in Table 58: OpenEdge-to-MS SQL Server Conversion utility batch parameters on page 278:

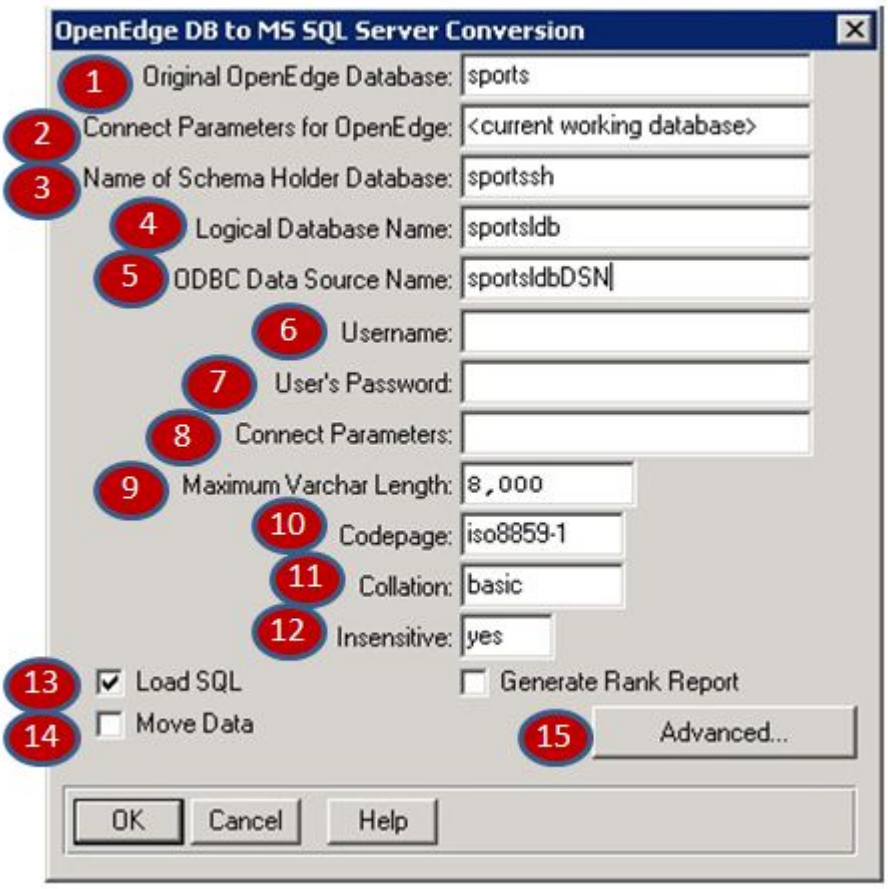


Table 58: OpenEdge-to-MS SQL Server Conversion utility batch parameters

Environment variable	Interface element	Description
PRODBNAME	**Original OpenEdge Database**	**Default:** <blank> Specify the source OpenEdge database name to be migrated. If this environment variable is not set and if the user has already set a current working database, the command line migration uses this as the value for the original OpenEdge database. **Dependencies:** If PRODBNAME is not set, and no working database is connected prior to running the conversion utility then the migration terminates, and an error is thrown in the logs.

Environment variable	Interface element	Description
PROCONPARMS	**Connect parameters for OpenEdge**	**Default:** Current working database parameters Specifies startup parameters for the original OpenEdge source database. If this environment variable is not set and if the user has already set a current working database, the command line migration uses the parameters that were associated with the startup of the original OpenEdge database specified by the PRODBNAME environment variable. You can specify your own startup parameters for the connection to the source OpenEdge database with this environment variable. **Dependencies:** The connection parameters associated with the source OpenEdge database, named by PRODBNAME, is specified by and connected with the PROCONPARMS environment variable value, along with the single-user mode (-1) parameter. In single-user mode, only one user can access the database. It is required that the PRODBNAME source database be started in a single-user mode for successful batch migration so that a snapshot of the database is used during the conversion. If you provide your own PROCONPARMS value, it must include settings to connect PRODBNAME in single-user mode or in read-only mode (-RO). Otherwise, the database connection fails with an error. For more information, see *OpenEdge Deployment: Startup Command and Parameter Reference*.
SHDBNAME	**Name of Schema holder Database**	**Default:** <blank> Specify the new schema-holder name, in which the resultant logical database will reside. **Dependencies:** If SHDBNAME is not set, the migration terminates with an error.
MSSDBNAME	**ODBC Data Source Name**	**Default:** <blank> Specify the ODBC Data Source name associated with the resultant schema image from the migration. **Dependencies:** If MSSDBNAME is not set, the migration terminates with an error. The ODBC DSN must be properly specified and located in the ODBC Data Source Administration of your system. If the **Load SQL** environment variable is set to **YES**, it must be properly configured to connect to the foreign data source.

Environment variable	Interface element	Description
MSSPDBNAME	**Logical Database Name**	**Default:** <blank> Specify the Logical Database name configured to connect the foreign data source. **Dependencies:** If MSSPDBNAME is not set, the migration terminates with an error. The value for MSSPDBNAME can be the same as PRODBNAME but must be different from SHDBNAME.
MSSUSERNAME	**Username**	**Default:** <blank> Specify the user name for the target data source. **Dependencies:** If MSSUSERNAME is not set, the migration terminates with an error.
MSSPASSWORD	**User's Password**	**Default:** <blank> Specify the password of the user for the target data source. **Dependencies:** If a password is required for authentication and MSSPASSWORD variable is not set, the migration terminates with an error. The error is logged in protomss.log.
MSSCONPARMS	**Connect parameters**	**Default:** <blank> Specifies the startup parameters for the DataServer and the target foreign data source. For more information on connection parameters, see Connecting the DataServer on page 189 for connection parameters. **Dependencies:** Connect parameters are not mandatory. If supplied, they must be formatted as a comma-separated list of names or name-value pairs, without white-spaces, that comprise the -Dsrv parameter value passed to the DataServer connection. Some of the parameters are parsed into the DataServer run-time, while others in the list are forwarded onto the target foreign data source.

Environment variable	Interface element	Description
VARLENGTH	**MaximumVarchar Length**	**Default:** The unconditioned default is 8000. It ranges from 4000 to 8000. Specify the maximum number of characters for a VARCHAR type. When a single-byte character set is utilized, as determined by the selected MSSCODEPAGE value, this must be a positive value less than or equal to 8000. The default threshold value of 8000 is the maximum number of characters that can be stored in a VARCHAR data type in MS SQL Server. When Unicode character data is utilized by virtue of a Unicode CODEPAGE environment variable selection, the default threshold value is limited to 4000. This value can also be overridden. During OpenEdge to MS SQL Server migration, whenever the column length of character data exceeds the value specified for VARLENGTH, the column is instead migrated as a Large Object (LOB) data type that can store data that far exceeds the data that can be stored in a VARCHAR data type. **Dependencies:** When MSSCODEPAGE is set to utf-8, the value set for VARLENGTH must be a positive value less than or equal to 4000 as it is the maximum number of characters that can be stored in a NVARCHAR data type in MS SQL Server due to UCS-2 character expansion. And, character lengths exceeding the threshold value set for VARLENGTH are converted to Unicode Large Object types. When an OpenEdge database stores UTF-8 data, each OpenEdge character can be as large as 4 bytes, potentially reducing the Maximum Varchar Length to 2000 characters on the server. But generally, MS SQL Server only supports the UCS-2 character set that allows 4000 UTF-8 characters in the UCS-2 range to be migrated into a Varchar column on the server.

Environment variable	Interface element	Description
MSSCODEPAGE	**Codepage**	**Default:** iso8859-1 when UNICODETYPES is disabled (that is, set to **NO**). Else, the default is utf-8. Specifies the corresponding OpenEdge name for the code page with which the MSS Database is compatible. Use UTF-8 if Unicode support is desired. If you use a code page that OpenEdge does not support, you must supply a conversion table that translates between the OpenEdge client code page and the code page that your data source uses. For a complete discussion of code pages, see *OpenEdge Development: Internationalizing Applications.* It is not mandatory to pass this value during batch migration. You can use **Change DataServer Schema Code Page** utility to add the code page information to the schema holder later but before you start using the DataServer to read and write data. For more information on changing code page, see Changing a code page in a schema holder on page 251. **Dependencies:** The specified codepage must exist in the `convmap.cp` file of your Progress environment in order to be used during the migration.
MSSCOLLNAME	**Collation**	**Default:** Basic Specifies the collation name with which the OpenEdge client must collate and weigh code page values. **Dependencies:** The specified collation must exist in the `convmap.cp` file of your Progress environment in order to be used during the migration unless you are using a Unicode code page. In case you are using a Unicode code page, then the code page name can correspond to one of the International Components for Unicode (ICU) collations that provide linguistic sorting of Unicode data based on the Unicode Collation Algorithm.
MSSCASESEN	**Insensitive**	**Default:** NO Specifies if the code page you are matching on the foreign data source is or isn't case insensitive. Provide YES if your code page is case insensitive, else retain the default value NO. **Dependencies:** By default, the code pages in MS SQL Server are case-insensitive. This matches the default sensitivity in OpenEdge. So, if you are using a case-insensitive code page in MS SQL Server, Progress recommends setting this value to YES.

Environment variable	Interface element	Description
LOADSQL	**Load SQL**	**Default:** NO Allows you to specify whether you want the utility to execute the SQL script that is generated in order to create the schema in the target empty MS SQL Server database on the server that was configured through your ODBC Data Source Name. Specify **YES** to enable this behavior.
MOVEDATA	**Move Data**	**Default:** NO Allows you to specify whether to populate the foreign data source with your OpenEdge data. 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. **Dependencies: Load SQL** must be selected in order to move data.

The hidden environment variables that affect the migration irrespective of whether the migration is performed in batch or in GUI are described in the following table.

Table 59: OpenEdge-to-MS SQL Server Conversion Hidden Environment Variables

Environment variable	Description
USE_OE_INIT_VALUE	This variable bypasses legacy migration behavior in which the OpenEdge default initial value for character fields, the NIULL string("") is ignored. Instead, you can set the USE_OE_INIT_VALUE to "Yes" to have a field with a null string("") as the OpenEdge initial value converted to the null string("") in SQL Server. Setting the value of USE_OE_INIT_VALUES to "Yes" impacts the migration only if **Include Defaults** is checked. **Note:** The OpenEdge Initial Value of a null string ("") is applied to any character server type that is created during the conversion, irrespective of whether it is a variable length character (VARCHAR) or an LOB type (VARCHAR(max)) because character strings of a larger size can also be converted during migration to CLOB data types.
_MSSBLANKDEFAULT	This variable translates an OpenEdge null string value("") to a server string with a single blank character(" "). **Note:** This environment variable causes a null string ("") to convert to a single-space string (" ") on the server regardless of whether the **Include Defaults** option is set to ON or OFF during migration. This environment variable is used to enable the MSS DataServer migration to emulate the way an Oracle DataServer migration handles the conversion of null strings. This provides for an application, whose queries depend upon the character field containing a single-space string by default, to behave the same way in MSS as it does in Oracle. The purpose of this environment variable is to provide data compatibility between MSS and Oracle for character default values.

Note: If the _MSSBLANKDEFAULT and USE_OE_INIT_VALUE variables are both set to YES, then _MSSBLANKDEFAULT overrides USE_OE_INIT_VALUE and sets anything with an OpenEdge Initial Value of the null string ("") to a single-spaced string (" ") on the server, irrespective of whether **Include Default** is set or not.

The following graphic is to help you map the environment variables with the Advanced migration options described in Table 60: OpenEdge-to-MS SQL Server Conversion Advanced options batch parameters on page 285:

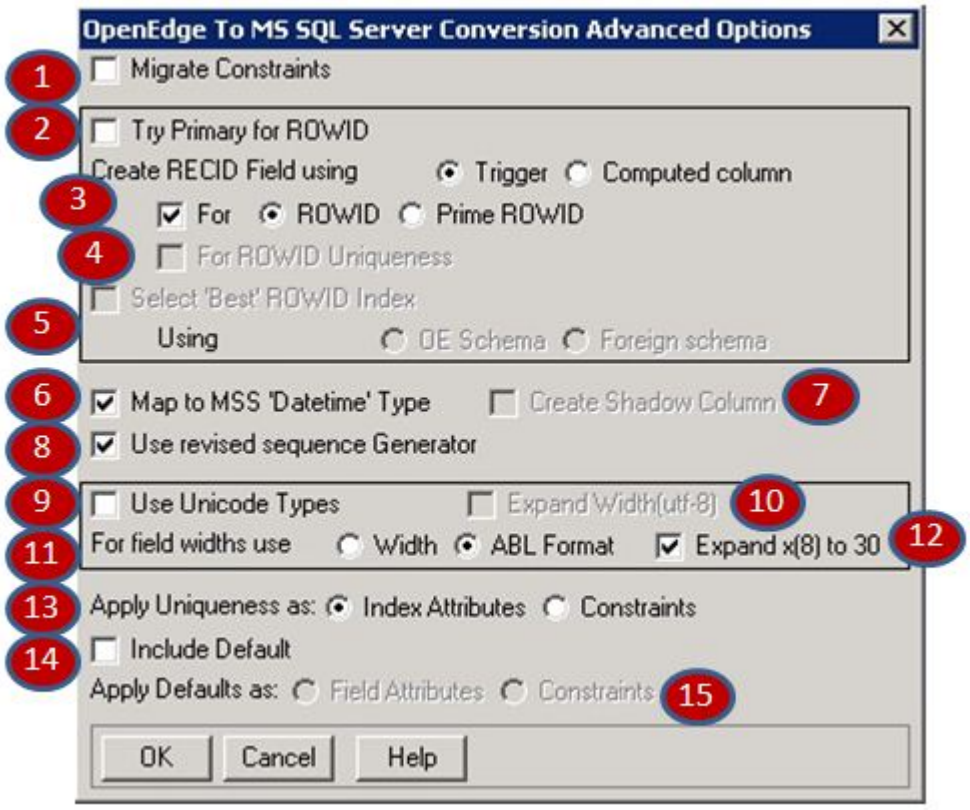


Table 60: OpenEdge-to-MS SQL Server Conversion Advanced options batch parameters

Environment variable	Interface element	Description
MIGRATECONSTR	**Migrate Constraints**	**Default:** NO Set to **YES** to migrate constraint definitions from the OpenEdge database definitions to real constraint objects in the foreign data source and constraint definitions in the schema image.
MAPOEPRIMARY	**Try Primary for ROWID**	**Default:** NO Determines if the OpenEdge primary should attempt to map itself to the SQL Server primary constraint and/or clustered index and/or the DataServer ROWID. **Dependencies**: If MIGRATECONSTR is set to **YES**, this option is superseded in the conversion process by existing constraints in determining the DataServer ROWID.

Environment variable	Interface element	Description
COMPATIBLE	**Create RECID Field using:** • **Trigger** • **Computed columns**	**Default:** Yes or 1 Determines if the migration must generate table definitions that provide compatibility with OpenEdge ROWID, RECID and with OpenEdge extent fields. When COMPATIBLE is set to NO, extent fields are not migrated, and array fields in OpenEdge databases are properly migrated to the server so that references to extend data types are properly mapped to the foreign data source and are handled the same as OpenEdge handles extents in the ABL. When COMPATIBLE and GENROWID are both set to NO, PROGRESS_RECID is not generated to support OpenEdge ROWID and RECID and a version of the **Select 'Best' ROWID Index** algorithm is instead used to try to find existing indexes to support ROWID. For the **Select 'Best' ROWID Index** algorithm to support RECID in a backward compatible way, COMPATIBLE, GENROWID, MIGRATECONSTR, MAPOEPRIMARY and GENUNIQROWID options must all be unset. If any of these ROWID options are set in the conversion, the **Select 'Best' ROWID Index** algorithm will try to produce an index to support ROWID but that index may or may not support RECID. When COMPATIBLE= 1 or Yes, it uses the legacy trigger behavior on the server to support PROGRESS_RECID. When COMPATIBLE= 2, it uses the computed column approach to supporting PROGRESS_RECID on the server. **Dependencies:** • In releases prior to OpenEdge 11, when COMPATIBLE=YES, ROWID and RECID selection was resolved by adding the PROGRESS_RECID field to the server table definitions. Alternatively in prior releases, when COMPATIBLE=NO, a legacy version of the **Select 'Best' ROWID Index** option, used starting in OpenEdge 11, was used to find an existing index capable of supporting both ROWID and RECID. Toggling the COMPATIBLE value on or off was equivalent to toggling the **Create RECID Field** option on or off in MS SQL Server DataServer versions prior to OpenEdge 11. Beginning in OpenEdge 11.1, the legacy COMPATIBLE option is extended by GENROWID, MIGRATECONSTR, MAPOEPRIMARY and GENUNIQROWID options that are used to generate OpenEdge-compatible ROWID solutions that may or many not be compatible with OpenEdge RECID.

Environment variable	Interface element	Description
		• You can simulate the behavior of OpenEdge releases prior to Version 11.0 by not setting the ROWID section options and by only setting the COMPATIBLE option to YES, 1 or 2. The GENROWID option is active by default so that the migration does take advantage of the new ROWID features (that are not backward-compatible) by default. Explicitly unset GENROWID and all other ROWID options (GENROWID, MIGRATECONSTR, MAPOEPRIMARY and MAPOEPRIMARY) if you want backward compatible behavior. • While behavior of the GENROWID option is similar to that of the COMPATIBLE option, the COMPATIBLE option also controls whether extent fields will be migrated and whether the trigger or computed column solution is used to support PROGRESS_RECID. Also, when COMPATIBLE is set to NO and all of the new ROWID options are unset (GENROWID, MIGRATECONSTR, MAPOEPRIMARY and MAPOEPRIMARY) a legacy version of the Select 'Best' ROWID Index option's algorithm for ROWID selection can be used. The Select 'Best' ROWID Index algorithm employed by the GENROWID option is advanced and is not backward compatible with the COMPATIBLE option as set in releases prior to OpenEdge 11.0. For a detailed explanation of how COMPATIBLE and GENROWID options work together, see the NOTE section at the end of the table.
GENROWID	**Create RECID Field using:** • **Trigger** • **Computed columns** **For:** • **ROWID** • **Prime ROWID**	**Default:** Yes or 1 Determines if the migration must generate table definitions that provide compatibility with OpenEdge ROWID and RECID. When COMPATIBLE and GENROWID are both set to no, PROGRESS_RECID is not generated to support OpenEdge ROWID and RECID. When GENROWID= 1 or Yes, it is the equivalent of COMPATIBLE= 1 or Yes with regard to support for OpenEdge ROWID and RECID; except that its value does not determine whether the trigger or computed column approaches should be used to support PROGRESS_RECID. Only COMPATIBLE determines that behavior. When GENROWID= 2, it is equivalent to setting GENROWID=1 except that when the PROGRESS_RECID index is created on the server and designated as the ROWID/RECID selection, it also creates a primary constraint over PROGRESS_RECID index key.

Environment variable	Interface element	Description
		Dependencies: • Whenever any of the OpenEdge 11 ROWID selection options are set along with or independent of the COMPATIBLE environment variable, the conversion process is incompatible with legacy conversion routines in terms of how the ROWID designation is selected. • If COMPATIBLE=NO and GENROWID=NO, then GETBESTROWID will be set to YES whether it was specified in the environment variables or not. A legacy version of GETBESTROWID is set to YES by default when COMPATIBLE=NO and is GETBESTROWID is set to NO when COMPATIBLE=YES, 1 or 2. Starting in OpenEdge 11.0 when GENROWID=NO, a new algorithm for GETBESTROWID is used by default. When GETBESTROWID=YES, it is mutually exclusive with the ROWID/RECID selection activities of both COMPATIBLE and GENROWID. But, the GETBESTROWID algorithm gets turned on and off like a toggle switch when COMPATIBLE is used for ROWID/RECID selection. Whereas, when the GETBESTROWID algorithm is on starting in OpenEdge 11.0, its operations can be complimentary to other ROWID options available starting in OpenEdge 11.0. Also, the existing index selected by the GETBESTROWID algorithm for ROWID starting in OpenEdge 11.0 may or may not be compatible with OpenEdge RECID functionality. Only when the ROWID index selected by the algorithm is a single component integer or Big integer key is it also compatible with RECID. For a detailed explanation of how COMPATIBLE and GENROWID options work together, see the NOTE section at the end of the table.
GENUNIQROWID	**Create RECID Field using:** • **For ROWID Uniqueness**	**Default:** NO Generates the PROGRESS_RECID_UNIQUE column for tables whose primary and/or clustered index selection does not qualify for ROWID designation because it is not unique. This option appends PROGRESS_RECID_UNIQUE to the end of the existing index component so that uniqueness and OpenEdge ROWID support can be obtained through that index.

Environment variable	Interface element	Description
		As there are potential performance benefits to finding and assigning the most appropriate ROWID candidate for OpenEdge as ROWID, adding uniqueness to non-unique indexes increases the ROWID eligibility of that key and the potential that the ROWID designation will benefit from both MS SQL Server and DataServer performance. When GENUNIQROWID=YES, non-unique indexes are considered for ROWID designation but none of the non-unique indexes is actually appended with a unique component unless the ROWID selection algorithm actually selects that non-unique index for ROWID designation. Only a non-unique index that is designated for ROWID is actually appended with a component that supplies uniqueness to the index. **Dependencies:** If you set the option, `GENROWID` to Yes, 1 or 2, then the system ignores the setting. The `PROGRESS_RECID_UNIQUE` column can make non-unique keys eligible for ROWID and can be used in combination with three other options that are used during migration to seek ROWID eligibility from indexes or constraints. If GENUNIQROWID is set to `YES` in conjunction with either the `MIGRATE_CONSTR`, `MAPOEPRIMARY` or `GETBESTFORROWID` ROWID options, then all non-unique indexes are considered for `PROGRESS_RECID_UNIQUE` supplementation when tying to establish index eligibility for ROWID key designation. **Note:** For detailed explanation of how GENROWID, GETUNIQROWID and COMPATIBLE options work together, see the note section at the end of the table.

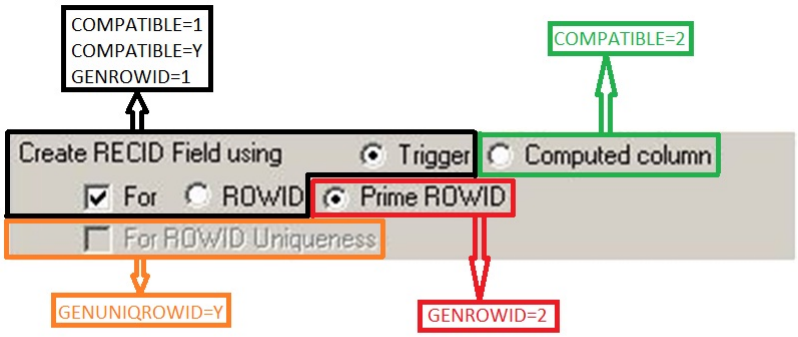
Environment variable	Interface element	Description
GETBESTROWID	**Select 'Best' ROWID Index using:** • **OE Schema** • **Foreign schema**	**Default:** NO Looks at existing indexes over the table and tries to identify the "Best" ROWID candidate based on index uniqueness, data types, number of components, and mandatory component characteristics. **Dependencies:** Select 'best' ROWID index is mutually exclusive to the GENROWID and COMPATIBLE options. So, when GETBESTROWID is set to YES, COMPATIBLE, and GENROWID must be set to NO. However, one of the following should be true at all times: • If COMPATIBLE and GENROWID are set to NO, then GETBESTROWID is set to YES. • If COMPATIBLE and GENROWID are set to on, then GETBESTROWID is set to off. • If GETBESTROWID is set to YES, then the default method for choosing the 'Best' ROWID index is used by locating the key through an investigation of the existing indexes in the OpenEdge Schema.
MAPMSSDATETIME	**Map to MSS 'Datetime' type**	**Default:** Yes Specifies whether DATE and DATETIME data types in OpenEdge must map to legacy DATETIME timestamp data types in MSS Server. If set to **Yes**, it provides backward-compatibility to earlier version of OpenEdge and MSS Server. However, if your database target is MSS 2008 or above and MAPSMSSDATETIME is set to **No**, more accurate mapping takes place between OpenEdge data and time data types and the new data and time data types in MSS Server.
SHADOWCOL	**Create Shadow Column**	**Default:** NO Determines if shadow columns must be added to the record layout to represent data in upper case form for OpenEdge fields marked case-insensitive. **Dependencies:** If MSSCASESEN is set to **YES**, the SHADOWCOL value is ignored.

Environment variable	Interface element	Description
MSSREVSEQGEN	**Use Generator**	**Default:** NO Determines if migration must use generator or the old sequence generator when migrating sequences to foreign data source. When MSSREVSEQGEN is set to **NO**, the legacy sequence generator is migrated instead of the revised version .
MSSSEQ	**Native Sequence Generator**	**Default:** NO Select to migrate OpenEdge Sequences to MS SQL sequences
SEQCACHESIZE	**Cache Size**	If the Try **Native Sequence** option is selected, then the **Cache Size** option is made available for entry. This determines the size of the cache that should be used on the server for the cache size. The following three values are applicable: • "0"= server (the cache size is determined by the server based on server criteria) • "?"= No Cache("NO CACHE" option is appended to the native sequence creation on the server) • Positive integer(1 or >1) - The number of sequences stored in the cache(Can range between 1 to the maximum value of the data type of the native sequence)
UNICODETYPES	**Use Unicode Types**	**Default:** NO Maps OpenEdge character fields to MSS Server unicode data types. **Dependencies:** Selecting this option changes the default code page to UTF-8 and directs the schema migration to convert all character and CLOB data types in the OpenEdge database to Unicode data types on the server.
UNICODE_EXPAND	**Expand Width (utf-8)**	**Default:** NO Set to **YES** to double the length of fields on conversion, and **NO** otherwise. If set to **YES**, the single-byte characters receive adequate size as double-byte characters in MSS UCS-2 format.

Environment variable	Interface element	Description
SQLWIDTH	**For field widths use:** • **Width** • **ABL Format**	**Default:** NO Specifies if the migration must use the _WIDTH field to calculate column size instead of using the format field. **Dependencies:** Specify YES to use the _width field in the column's schema image to calculate column size in place of the default format field, else **ABL Format** is used to size migrated column.
EXPANDX8	**Expand x(8) to 30**	**Default:** Yes Determines whether character fields set to the default format size of x(8) should be expanded to a 30 character default. This expands the default length on the foreign data source. **Dependencies**: The option only affects the batch conversion if SQLWIDTH is set to NO, which utilizes **4GL Format** for character size translation to the foreign data source. If SQLWIDTH is set to YES, this option is ignored.
UNIQUECONSTR	**Apply Uniqueness as:** • **Index Attributes** • **Constraints**	**Default:** NO Specify **YES** to create named Unique constraints for index uniqueness, otherwise retain the default of automatically using Index attributes to define index uniqueness for unique indexes migrated from an OpenEdge database.
CRTDEFAULT	**Include Default**	**Default:** NO Specify **YES** to include OpenEdge initial values in the schema image definitions of the foreign data source in the schema holder database with fields migrated to columns in the foreign data source. When CRTDEFAULT is off, the initial values are assigned out of the schema image to records created in the foreign data source by the OpenEdge DataServer. When CRTDEFAULT is YES, the initial values are actually transferred and become part of the server definitions, not just the schema image definitions.
DFLTCONSTR	**Apply Defaults as:** • **Field Attributes** • **Constraints**	**Default:** NO Specify **YES** to create named default constraints for initial values rather than setting the DEFAULT column attribute of the generated server column. **Dependencies:** If the OpenEdge initial value is going to be migrated to the foreign data source, CRTDEFAULT must be set to **YES**.

Note:

The following diagram maps environment variables, `COMPATIBLE`, `GENROWID`, and `GENUNIQROWID`, to its corresponding options in the OpenEdge migration dialog, and clarifies backward compatibility requirements and the usage of the different value indicators that the environment variables provide.



Prior to OpenEdge 11.1, the OpenEdge database to MS SQL Server migration dialog provided **Create RECID Field** option, which was mapped to the environment variable setting `COMPATIBLE=1`.

Beginning OpenEdge 11.1, the OpenEdge database to MS SQL Server migration dialog provides two options to create a RECID field, **Trigger** and **Computed column**. **Trigger** is mapped to the environment variable setting `COMPATIBLE=Y` and `COMPATIBLE=1`, and **Computed column** is mapped to the environment variable setting `COMPATIBLE=2` respectively.

In addition, **FOR** option provides three options, **ROWID**, **Prime ROWID**, and **ROWID Uniqueness**. **ROWID** is mapped to `GENROWID=1`, **Prime ROWID** is mapped to the environment variable setting `GENROWID=2`, and **ROWID Uniqueness** is mapped to the environment variable setting `GENUNIQROWID=Y`.

For more information on OpenEdge database to MS SQL Server migration, see Migrating an OpenEdge database to MS SQL Server on page 267.

For instance, you can pass the values for the environment variables at the system prompt:

```
PRODBNAME=db-name; export PRODBNAME
PROCONPARMS="-1 -i"
SHDBNAME=schema-holder-name; export SHDBNAME
.
.
.
pro -b -p prodict/mss/protomss.p
```

3. Execute `protomss.p`.

Column width

The OpenEdge-to-MS SQL Server migration utility uses either a field's format or width information (_width value in the _Field schema record) for the column length when it defines the field as an MS SQL Server column. Since OpenEdge allows a field to hold more data than the field's format can display, a field's format might not be the best indicator as to how large a column should be sized on the server. You can select **Width** in the **For Field widths use** migration option to create a column that is wider than the format indicates. For more information on **For Field widths use** option, Table 57: OpenEdge DB to MS SQL Server Conversion Advanced Options on page 275. If a column generated by the migration utility is not wide enough to hold data from the OpenEdge database, the utility backs out any data loaded and the data in the MS SQL Server database remains unchanged.

The default size of a column's width is twice the length set for the OpenEdge FORMAT field attribute. When you use the DBTOOL utility option, **SQL Width Scan w/Fix Option**, the utility calculates the column width based on the actual size of data in the columns to ensure that server data length requirements are satisfied during migration. Unless the default column width is greater than the actual data size obtained from the largest column row, the utility adjusts the column width based on its calculation. Otherwise, the default column width is utilized during migration to help prevent under-sized columns on the server from receiving over-populated column data.

When you use the DBTOOL utility to calculate a column size of an extent column, derived from the entire array of extent elements, the extent element with the maximum number of characters is identified, and all the extent elements defined on the server are re-sized to the column length of the extent element with the maximum number of characters. If the extent element with the maximum number of characters exceeds the migration threshold parameter **Maximum Varchar Length**, then each element of the array is re-sized as a LOB server column type. The OpenEdge record buffer may otherwise overflow, whereas a LOB locator takes a very small space in the record.

Prior to running the utility, you must determine which method of calculating column size will be used to ensure your data will fit. If necessary, use the **Adjust Field Width** tool in the Data Dictionary or the database administration tool (DBTOOL utility) to enlarge character, decimal, and array fields. For more information on the DBTOOL utility, see *OpenEdge Data Management: Database Administration*.

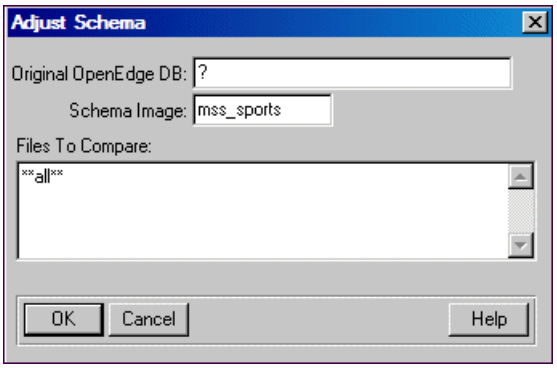
Adjust schema utility

The Adjust Schema utility allows you to compare your MS SQL Server Schema holder information with the original OpenEdge database, updating any OpenEdge Attributes in the schema holder. These attributes include field names, labels, help, validation expressions and messages. You must be connected to the OpenEdge database and the schema holder that you want to compare to before running this utility.

To adjust your schema:

1. Select the utility from the **Data Admin Tool** > **DataServer** > **MS SQL Server Utilities** > **Schema Migration Tools** > **Adjust Schema**.

   The following screen appears:



If you leave the ****all**** in the **Files To Compare** section, all objects are compared and the changes are made. 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 are compared will be displayed on screen as they are processed.

Delta df to MS SQL Server Incremental Schema Migration utility

The Incremental Schema Migration utility allows you to migrate schema changes from OpenEdge to a MS SQL Server database. For example, in the process of developing an application in ABL that you will migrate to MS SQL Server, you might want to make and test schema changes in the OpenEdge database that you want reflected in the MS SQL Server database. The utility reads an OpenEdge `delta.df` file that has been created using the standard incremental dump procedure. and creates a `delta.sql` file that contains the SQL DDL for making the changes and a `delta.df` file. You can then load the `delta.df` file into the schema holder. You can then apply the `delta.sql` file to the MS SQL Server database to complete the migration process.

Note: You do not make schema changes directly in the schema holder, which must remain synchronized with the MS SQL Server database. The utility uses the schema holder to determine what the MS SQL Server definitions are.

The following sections discuss how you can perform schema pull interactively or silently (that is, in batch mode).

Running the Incremental Schema Migration utility interactively

Use the following steps to run the Incremental Schema Migration utility interactively:

1. Generate delta.sql using one of the following methods:

   - From the **Data Admin** main menu, choose **DataServers** > **MS SQL Server Utilities** > **Schema Migration Tools** > **Generate Delta.sql OpenEdge to MSS**.

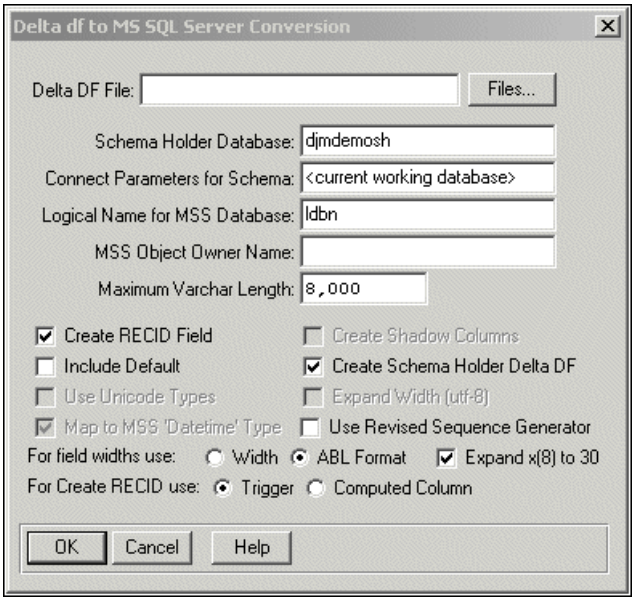
     Note: All the selected options, errors, and warning messages are are logged at `deltasqlutil.log`.

   - Execute the following command in the command prompt to Independently perform an incremental update to MS SQL Server:

     ```
     pro -p prodict/mss/deltasql.p
     ```

     Note: All the selected options, errors, and warning messages are logged at `deltasql.log`.

   The **Delta df to MS SQL Server Conversion** dialog box appears:



2. Provide the information described in the following table.

Table 61: Generate Delta.sql OpenEdge to MSS utility

Interface element	Description
Delta DF File	**Default:** <blank> Specify the Delta.sql file by selecting the file location.

Interface element	Description
Schema Holder Database	**Default:** <blank> Specify the new schema-holder name, in which the resultant logical database must reside. **Note:** If this is not set, the migration terminates with an error.
Connect parameters for Schema	**Default:** <blank> Specifies the startup parameters for the DataServer and the target foreign data source. For more information on connection parameters, see Connecting the DataServer on page 189 for connection parameters. **Note: Dependencies:** Connect parameters are not mandatory. If supplied, they must be formatted as a comma-separated list of names or name-value pairs, without white-spaces, that comprise the `-Dsrv` parameter value passed to the DataServer connection. Some of the parameters are parsed into the DataServer run-time, while others in the list are forwarded onto the target foreign data source.
Logical name for MSS Database	Specify the ODBC data source name configured to connect the foreign data source. **Dependencies:** If this option is not set, the migration terminates with an error. The ODBC DSN must be properly specified and located in the ODBC Data Source Administration of your system.
MSS Object Owner Name	**Default:** <blank> Specify the user name for the target data source. **Dependencies:** If this option is not set, the migration terminates with an error.

Interface element	Description
Maximum Varchar Length	**Default:** The unconditioned default is 8000. It ranges from 4000 to 8000. Specify the maximum number of characters for a VARCHAR type. When a single-byte character set is utilized, as determined by the selected MSSCODEPAGE value, this must be a positive value less than or equal to 8000. The default threshold value of 8000 is the maximum number of characters that can be stored in a VARCHAR data type in MS SQL Server. When Unicode character data is utilized by virtue of a Unicode CODEPAGE environment variable selection, the default threshold value is limited to 4000. This value can also be overridden. During OpenEdge to MS SQL Server migration, whenever the column length of character data exceeds the value specified for `VARLENGTH`, the column is instead migrated as a Large Object (LOB) data type that can store data that far exceeds the data that can be stored in a VARCHAR data type. **Dependencies:** When `MSSCODEPAGE` is set to `utf-8`, the value set for `VARLENGTH` must be a positive value less than or equal to 4000 as it is the maximum number of characters that can be stored in a NVARCHAR data type in MS SQL Server due to UCS-2 character expansion. And, character lengths exceeding the threshold value set for `VARLENGTH` are converted to Unicode Large Object types. When an OpenEdge database stores UTF-8 data, each OpenEdge character can be as large as 4 bytes, potentially reducing the Maximum Varchar Length to 2000 characters on the server. But generally, MS SQL Server only supports the UCS-2 character set that allows 4000 UTF-8 characters in the UCS-2 range to be migrated into a Varchar column on the server.
Create RECID field	**Default:** YES or 1 (**Trigger**) Set it to 1 to create PROGRESS_RECID field using trigger approach to RECID or ROWID construction. Select either 1 (**Trigger)** or 2 (**Computed Column**). Select **Trigger** if you want the migrated table to designate RECID and ROWID. Select **Computed Column** if you want the migrated table to make ROWID unique. **Computed Column** is preferred as it offers greater reliability, but it is only available for MS SQL Server 2005 and later .
Include Default	**Default:** NO Determines if shadow columns must be added to the record layout to represent data in upper case form for OpenEdge fields marked case-insensitive. Therefore, if `MSSCASESEN` is set to **YES**, the SHADOWCOL value is ignored.

Interface element	Description
Create Shadow Columns	**Default:** NO Specify **YES** to include OpenEdge initial values with fields migrated to columns in the foreign data source. **Dependencies:** When SQL Server is configured with a case sensitive code page, the OpenEdge fields marked as case insensitive are migrated to MSS as case insensitive records in MSS. In this process, a shadow column is added to the record.
Create schema holder delta df	**Default:** NO Specify **YES** if you want the utility to generate a `.df` file that includes the incremental schema information. You can then load this `.df` file into the schema holder. By default, this check box is selected.
Use Unicode Types	**Default:** NO Maps OpenEdge character fields to MSS Server unicode data types. Set to **YES** to create character columns with unicode specification, and **NO** otherwise.
Expand Width (utf-8)	**Default:** NO Set to **YES** to double the length of fields on conversion, and **NO** otherwise. If set to **YES**, the single-byte characters receive adequate size as double-byte characters in MSS UCS-2 format.
Map to MSS Datetime Type	**Default:** YES Specifies if OpenEdge DATE and DATATIME data types must be mapped to legacy DATETIME timestamp data types in MSS. Therefore, if set to **YES**, it provides backward compatibility to the previous versions of OpenEdge and MSS. Deselect the check box if your database target is MS SQL Server 2008 or above. For details, see DataServer for MS SQL Server support for datetime data types on page 61.
Use Generator	**Default:** YES Specifies weather to use the generator instead of the legacy sequence generator when migrating sequences to the foreign data source. When set to **NO**, the legacy sequence generator is migrated. Progress recommends that you use the revised generator as it provides better performance.

Interface element	Description
Expand x(8) to 30	**Default:** YES Set to **YES** to double the length of fields on conversion, and **NO** otherwise. If set to **YES**, the single-byte characters receive adequate size as double-byte characters in MSS UCS-2 format. Specify **NO** if you want the column size of the server calculated directly from the x(8) format value. **Dependencies:** This option is ignored if SQLWIDTH is set to **YES**.
For field widths use: • **Width** • **ABL format**	**Default:** NO Specifies if the migration must use the **Width** field to calculate column size instead of using the format field. Specify **YES** to use the **Width** field in the column's schema image to calculate column size in place of the default format field, else **ABL format** is used to size migrated column. **Dependencies:** You cannot use the **Expand x(8) to 30** setting with the **Width** option.

3. Choose **OK**. The Pre-selection utility generates a `delta.sql` file and, optionally, a `delta.df` file.

   When this step is performed for an MS SQL Server DataServer, and the incremental update procedure includes changing an existing field from `INTEGER` to `INT64` in the schema holder, the DataServer supports the update process through the `ALTER TABLE ALTER COLUMN` statement. If a column is part of an index, the incremental schema migration utility generates a script to drop any indexes that contain the column, change the column's data type to `BIGINT`, and then re-adds the indexes. During this process, the utility uses the index definitions defined in the schema holder.

4. After running the utility, you must apply the SQL that the utility generates to the MS SQL Server database and load the new delta.df file into the original schema holder so that it is synchronized with the modified MS SQL Server database.

Note: Note that if you updated an existing field from INTEGER to INT64, you also must confirm that none of the data is lost.

Running the Incremental Schema Migration utility silently (batch mode)

Use the following steps to run the Incremental Schema Migration utility silently (in batch mode):

1. Create or use an existing target MS SQL Server database
2. Configure your ODBC driver to connect to your target data source.
3. On your client machine, pass parameters to the utility by setting the environment variables listed in described in the table below that correspond to the **Delta df to MS SQL Server Conversion** dialog.

Note: To troubleshoot any errors or warnings, you can view the `deltasqlutil.log`.

Table 62: Generate Delta.sql OpenEdge to MSS utility

Environment variable	Interface element	Description
DELTADFNAME	**Delta DF File**	**Default:** <blank> Specify the Delta.sql file by selecting the file name and location.
SHDBNAME	**Schema Holder Database**	**Default:** <blank> Specify the new schema-holder name, in which the resultant logical database will reside. **Note: Dependencies:** If SHDBNAME is not set, the migration terminates with an error.
MSSCONPARMS	**Connect parameters for Schema**	**Default:** <blank>Specifies the startup parameters for the DataServer and the target foreign data source.For more information on connection parameters, see Connecting the DataServer on page 189 for connection parameters. **Note: Dependencies:** Connect parameters are not mandatory. If supplied, they must be formatted as a comma-separated list of names or name-value pairs, without white-spaces, that comprise the `-Dsrv` parameter value passed to the DataServer connection. Some of the parameters are parsed into the DataServer run-time, while others in the list are forwarded onto the target foreign data source.
MSSPDNAME	**Logical name for MSS Database**	Specify the logical database name configured to connect the foreign data source. **Dependencies:** If MSSPDBNAME is not set, the migration terminates with an error. The ODBC DSN must be properly specified and located in the ODBC Data Source Administration of your system.
MSSUSERNAME	**MSS Object Owner Name**	**Default:** <blank> Specify the user name for the target data source. **Dependencies:** If MSSUSERNAME is not set, the migration terminates with an error.

Environment variable	Interface element	Description
VARLENGTH	**Maximum Varchar Length**	**Default:** The unconditioned default is 8000. It ranges from 4000 to 8000. Specify the maximum number of characters for a VARCHAR type. When a single-byte character set is utilized, as determined by the selected MSSCODEPAGE value, this must be a positive value less than or equal to 8000. The default threshold value of 8000 is the maximum number of characters that can be stored in a VARCHAR data type in MS SQL Server. When Unicode character data is utilized by virtue of a Unicode CODEPAGE environment variable selection, the default threshold value is limited to 4000. This value can also be overridden. During OpenEdge to MS SQL Server migration, whenever the column length of character data exceeds the value specified for VARLENGTH, the column is instead migrated as a Large Object (LOB) data type that can store data that far exceeds the data that can be stored in a VARCHAR data type. **Dependencies:** When MSSCODEPAGE is set to utf-8, the value set for VARLENGTH must be a positive value less than or equal to 4000 as it is the maximum number of characters that can be stored in a NVARCHAR data type in MS SQL Server due to UCS-2 character expansion. And, character lengths exceeding the threshold value set for VARLENGTH are converted to Unicode Large Object types. When an OpenEdge database stores UTF-8 data, each OpenEdge character can be as large as 4 bytes, potentially reducing the Maximum Varchar Length to 2000 characters on the server. But generally, MS SQL Server only supports the UCS-2 character set that allows 4000 UTF-8 characters in the UCS-2 range to be migrated into a Varchar column on the server.
COMPATIBLE	**Create RECID field**	Default: Yes or 1 Determines if the migration must generate table definitions that provide compatibility with OpenEdge ROWID, RECID and with OpenEdge extent fields. When COMPATIBLE is set to NO, extent fields are not migrated, and array fields in OpenEdge databases are properly migrated to the server so that references to extend data types are properly mapped to the foreign data source and are handled the same as OpenEdge handles extents in the ABL.

Environment variable	Interface element	Description
		When COMPATIBLE and GENROWID are both set to NO, PROGRESS_RECID is not generated to support OpenEdge ROWID and RECID and a version of the **Select 'Best' ROWID Index** algorithm is instead used to try to find existing indexes to support ROWID. For the **Select 'Best' ROWID Index** algorithm to support RECID in a backward compatible way, COMPATIBLE, GENROWID, MIGRATECONSTR, MAPOEPRIMARY and GENUNIQROWID options must all be unset. If any of these ROWID options are set in the conversion, the **Select 'Best' ROWID Index** algorithm will try to produce an index to support ROWID but that index may or may not support RECID. When COMPATIBLE= 1 or Yes, it uses the legacy trigger behavior on the server to support PROGRESS_RECID. When COMPATIBLE= 2, it uses the computed column approach to supporting PROGRESS_RECID on the server. **Dependencies:** • In releases prior to OpenEdge 11, when COMPATIBLE=YES, ROWID and RECID selection was resolved by adding the PROGRESS_RECID field to the server table definitions. Alternatively in prior releases, when COMPATIBLE=NO, a legacy version of the **Select 'Best' ROWID Index** option, used starting in OpenEdge 11, was used to find an existing index capable of supporting both ROWID and RECID. Toggling the COMPATIBLE value on or off was equivalent to toggling the **Create RECID Field** option on or off in MS SQL Server DataServer versions prior to OpenEdge 11. Beginning in OpenEdge 11.1, the legacy COMPATIBLE option is extended by GENROWID, MIGRATECONSTR, MAPOEPRIMARY and GENUNIQROWID options that are used to generate OpenEdge-compatible ROWID solutions that may or many not be compatible with OpenEdge RECID. • You can simulate the behavior of OpenEdge releases prior to Version 11.0 by not setting the ROWID section options and by only setting the COMPATIBLE option to YES, 1 or 2.

Environment variable	Interface element	Description
		The GENROWID option is active by default so that the migration does take advantage of the new ROWID features (that are not backward-compatible) by default. Explicitly unset GENROWID and all other ROWID options (GENROWID, MIGRATECONSTR, MAPOEPRIMARY and MAPOEPRIMARY) if you want backward compatible behavior. • While behavior of the GENROWID option is similar to that of the COMPATIBLE option, the COMPATIBLE option also controls whether extent fields will be migrated and whether the trigger or computed column solution is used to support PROGRESS_RECID. Also, when COMPATIBLE is set to NO and all of the new ROWID options are unset (GENROWID, MIGRATECONSTR, MAPOEPRIMARY and MAPOEPRIMARY) a legacy version of the Select 'Best' ROWID Index option's algorithm for ROWID selection can be used. The Select 'Best' ROWID Index algorithm employed by the GENROWID option is advanced and is not backward compatible with the COMPATIBLE option as set in releases prior to OpenEdge 11.0. For a detailed explanation of how COMPATIBLE and GENROWID options work together, see the NOTE section at the end of Table 60: OpenEdge-to-MS SQL Server Conversion Advanced options batch parameters on page 285.
SHADOWCOL	**Create Shadow Columns**	**Default:** NO Determines if shadow columns must be added to the record layout to represent data in upper case form for OpenEdge fields marked case-insensitive. **Dependencies:** If MSSCASESEN is set to **YES**, the SHADOWCOL value is ignored.

Environment variable	Interface element	Description
CRTDEFAULT	**Include Default**	**Default:** NO Specify **YES** to include OpenEdge initial values in the schema image definitions of the foreign data source in the schema holder database with fields migrated to columns in the foreign data source. When CRTDEFAULT is off, the initial values are assigned out of the schema image to records created in the foreign data source by the OpenEdge DataServer. When CRTDEFAULT is YES, the initial values are actually transferred and become part of the server definitions, not just the schema image definitions.
CRTDELTADF	**Create schema holder delta df**	**Default:** NO Specify **YES** if you want the utility to generate a `.df` file that includes the incremental schema information. You can then load this `.df` file into the schema holder. By default, this check box is selected.
UNICODETYPES	**Use Unicode Types**	**Default:** NO Maps OpenEdge character fields to MSS Server unicode data types. **Dependencies:** Selecting this option changes the default code page to UTF-8 and directs the schema migration to convert all character and CLOB data types in the OpenEdge database to Unicode data types on the server.
UNICODE_EXPAND	**Expand Width (utf-8)**	**Default:** NO Set to **YES** to double the length of fields on conversion, and **NO** otherwise. If set to **YES**, the single-byte characters receive adequate size as double-byte characters in MSS UCS-2 format.

Environment variable	Interface element	Description
MAPMSSDATETIME	**Map to MSS Datetime Type**	**Default:** Yes Specifies whether DATE and DATETIME data types in OpenEdge must map to legacy DATETIME timestamp data types in MSS Server. If set to Yes, it provides backward-compatibility to earlier version of OpenEdge and MSS Server. However, if your database target is MSS 2008 or above and MAPSMSSDATETIME is set to No, more accurate mapping takes place between OpenEdge data and time data types and the new data and time data types in MSS Server.
MSSREVSEQGEN	**Use Generator**	**Default:** NO Determines if migration must use generator or the old sequence generator when migrating sequences to foreign data source. When MSSREVSEQGEN is set to **NO**, the legacy sequence generator is migrated instead of the revised version.
EXPANDX8	**Expand x(8) to 30**	**Default:** Yes Determines whether character fields set to the default format size of x(8) should be expanded to a 30 character default. This expands the default length on the foreign data source. **Dependencies**: The option only affects the batch conversion if SQLWIDTH is set to NO, which utilizes **4GL Format** for character size translation to the foreign data source. If SQLWIDTH is set to YES, this option is ignored.
SQLWIDTH	**For field widths use:** • **Width** • **ABL format**	**Default:** NO Specifies if the migration must use the _WIDTH field to calculate column size instead of using the format field. **Dependencies:** Specify YES to use the _width field in the column's schema image to calculate column size in place of the default format field, else **ABL Format** is used to size migrated column.
GENROWID	**For Create RECID use:**	**Default:** Yes or 1 Determines if the migration must generate table definitions that provide compatibility with OpenEdge ROWID and RECID. When COMPATIBLE and GENROWID are both set to no, PROGRESS_RECID is not generated to support OpenEdge ROWID and RECID.

Environment variable	Interface element	Description
		When GENROWID= 1 or Yes, it is the equivalent of COMPATIBLE= 1 or Yes with regard to support for OpenEdge ROWID and RECID; except that its value does not determine whether the trigger or computed column approaches should be used to support PROGRESS_RECID. Only COMPATIBLE determines that behavior. When GENROWID= 2, it is equivalent to setting GENROWID=1 except that when the PROGRESS_RECID index is created on the server and designated as the ROWID/RECID selection, it also creates a primary constraint over PROGRESS_RECID index key. **Dependencies**: • Whenever any of the OpenEdge 11 ROWID selection options are set along with or independent of the COMPATIBLE environment variable, the conversion process is incompatible with legacy conversion routines in terms of how the ROWID designation is selected. • If COMPATIBLE=NO and GENROWID=NO, then GETBESTROWID will be set to YES whether it was specified in the environment variables or not. A legacy version of GETBESTROWID is set to YES by default when COMPATIBLE=NO and is GETBESTROWID is set to NO when COMPATIBLE=YES, 1 or 2. Starting in OpenEdge 11.0 when GENROWID=NO, a new algorithm for GETBESTROWID is used by default. When GETBESTROWID=YES, it is mutually exclusive with the ROWID/RECID selection activities of both COMPATIBLE and GENROWID. But, the GETBESTROWID algorithm gets turned on and off like a toggle switch when COMPATIBLE is used for ROWID/RECID selection. Whereas, when the GETBESTROWID algorithm is on starting in OpenEdge 11.0, its operations can be complimentary to other ROWID options available starting OpenEdge 11.0. Also, the existing index selected by the GETBESTROWID algorithm for ROWID starting in OpenEdge 11.0 may or may not be compatible with OpenEdge RECID functionality. Only when the ROWID index selected by the algorithm is a single component integer or Big integer key is it also compatible with RECID.

Environment variable	Interface element	Description
		For a detailed explanation of how COMPATIBLE and GENROWID options work together, see the NOTE section at the end of Table 60: OpenEdge-to-MS SQL Server Conversion Advanced options batch parameters on page 285.

4. Execute the following command to set and export environment variables at the system prompt:

```
DELTADFNAME=FileName.df
SHDBNAME=MyDatabase
export  DELTADFNAME SHDBNAME
.
.
.
```

5. Execute the following command in the **Proenv** command prompt to perform an incremental update to MS SQL Server:

```
pro -b -p prodict/mss/deltasql.p
```

After running the utility, you must apply the SQL data that the utility generates to the MS SQL Server database and load the new `delta.df` file into the original schema holder so that it is synchronized with the modified MS SQL Server database.

Note: Note that if you updated an existing field from INTEGER to INT64, you must also confirm that none of the data is lost.

The Incremental Schema Migration utility generates SQL that will create objects in the MS SQL Server database that are compatible with OpenEdge.

The following table shows that the utility creates the same objects as the OpenEdge DB to MS SQL Server migration utility.

Table 63: MS SQL Server equivalents of OpenEdge objects

OpenEdge object	MS SQL Server equivalents
Array	One column for each extent of the OpenEdge array.
Table	For new tables, if the **Create RECID Field** option was selected, a `PROGRESS_RECID` column and associated triggers are created. This indexed column provides a unique key on the MS SQL Server table.

The utility ensures that the migrated objects have names that are unique to the MS SQL Server database. If you have given the object a name that is not unique, it drops characters from the end of the name and appends numbers until it creates a unique name.

Since MS SQL Server requires that index names be unique to the database, the utility appends the table name to the indexed column name to create a unique name.

Adjusting field widths during migration

Especially when migrating a Unicode UTF-8 OpenEdge database with Unicode data, Progress Software recommends that you select **Width** in the **For Field widths use** options during OpenEdge database migration or Incremental Schema Migration. Prior to migration, the DBTOOL database utility can be used to size your server columns based on the actual size of your CHARACTER, RAW and DECIMAL data rather than using the default format value in the schema to size your server columns.

The width value for each field is based on the maximum number of characters (not bytes) found in the database records of that column. For character data type, the size is calculated based on the code page defined in the database schema. You can also use the Adjust Field Width utility from the Options pulldown menu of the Data Dictionary to manually set a specific character width on an individual column prior to migration.

Updating the MS SQL Server database

You should review, and modify if necessary, the `delta.sql` file that the utility generates before applying it. You can apply the `delta.sql` file to the MS SQL Server database through SQL-based tools.

After applying the `delta.sql` file to the MS SQL Server database, you must update the original schema holder by loading the new `delta.df` file into the original schema holder so that the original schema holder reflects the modifications you made to the MS SQL Server database.

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 OpenEdge 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 OpenEdge information at the table and field levels in the schema holder. The following sections describe how to enter OpenEdge information at both levels. An additional section describes how to select an index to support the `ROWID` function.

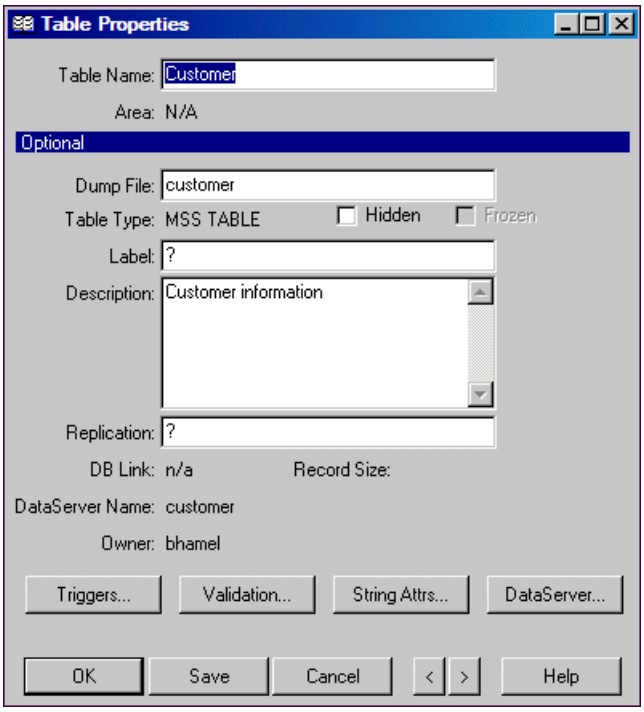
Modifying table-level information

You can modify table-level information.

To modify information in the schema holder at the table level:

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 **Validation**. 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.

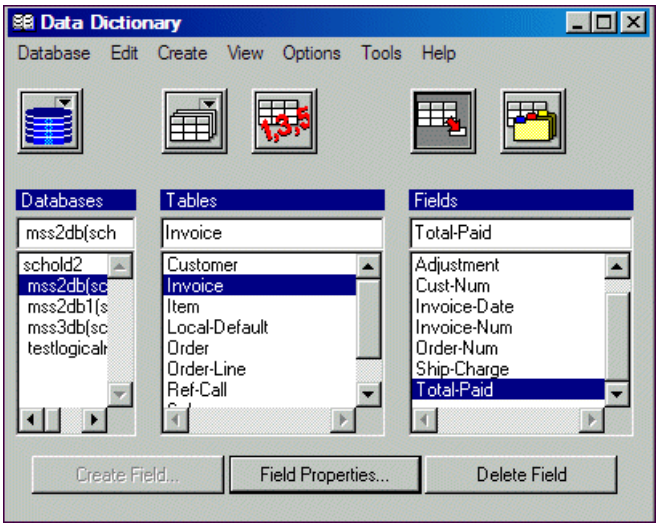
Modifying field-level information

You can modify field-level information.

To modify information in the schema holder at the field level:

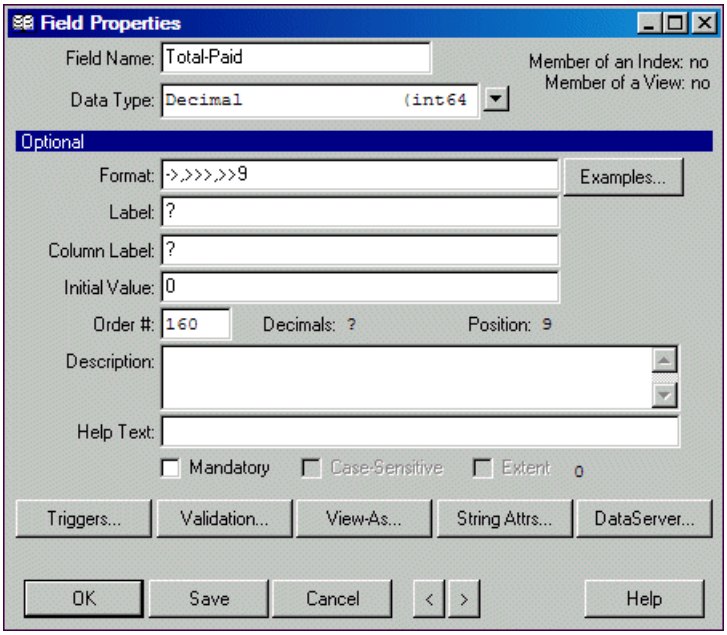
1. Access the **Data Dictionary**, if it is not already showing. Choose **Fields**.

   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 information at the field level, such as a validation expression or a validation message.

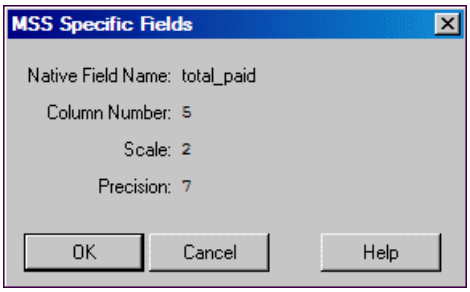
The Data Dictionary displays the standard MS SQL Server names for data types and not the native ODBC data source names. Using the Data Dictionary, you can make the following changes:

- Change the data type or the format in which OpenEdge displays the data. For example, choose the MS SQL Server `decimal` data type, which in turn defaults to the OpenEdge `INTEGER` data type mapping. However, you can change the `decimal` mapping to either the OpenEdge `INTEGER` or `INT64` data type instead. (The previous **Field Properties** dialog box shows the `DECIMAL` data type with the `INT64` data type option selected.) For more information about optional settings for data types, see Data types on page 56 and Data Type Details on page 365.
- 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 **DataServer** 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 `total_paid` field is named `total_paid` in the 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 ABL statement.

Defining the ROWID

When you create or update a schema holder, the DataServer uses the following guidelines to select a column in a 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. To create a `PROGRESS_RECID` column, see Using triggers to enable PROGRESS_RECID support of the ROWID function on page 330.
- 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 criteria:
  - The index must be defined as unique.
  - The index must be defined as mandatory, or at least treated as such by your application.

The index selection process proceeds, according to the following order:

1. If you select an index, then you must select a qualified index.
2. If you do not select an index, the MS SQL Server DataServer selects a qualified index based on a single-component integer.

   However, if the DataServer determines that more than one qualified index meets the additional criterion as a single-component integer, the DataServer selects the first index that fulfills the complete criteria.

3. If the MS SQL Server DataServer cannot find a qualified index based on a single-component integer, it seeks a qualified index that is multi-component and/or non-integer.

   However, if the DataServer determines that more than one qualified index meets the additional criterion as multi-component and/or non-integer, the DataServer selects the first index that fulfills the complete criteria.

4. If the MS SQL Server DataServer cannot find a qualified index that is a multi-component and/or non-integer, the task cannot be completed and `ROWID` support is not provided. The table will not be able to uniquely identify any records. As a result, implementing `FIND` statements and performing queries could lead to inaccuracies in locating records.

The indexes in this class are **not** mandatory; therefore, it is essential that you enforce the column supporting `ROWID` as mandatory at least through code if not through definitions.

Note: The MSS DataServer was enhanced to require less bandwidth for ROWID in networked messages containing data records. This improves progress networking performance for client/server DataServer applications. With this enhancement, the ROWID area stored in the record buffer is shorter in size. Reduced storage size of ROWID in the record buffer is enabled by default in the MSS DataServer. Due to this enhancement, the ROWID buffer size can now be set to as low as 13 bytes if the ROWID selected for tables is chosen as the PROGRESS_RECID surrogate key or any similar existing, single component, unique key, integer index.

The DataServer allocates a buffer to hold the key information for the selected `RECID` index. The buffer is 245 bytes by default. If the index you select for the `RECID` is a multi-component index, this may be inadequate. The size of the buffer can be modified with the `-Dsrv ZPRGRS_RECID_BUF_SIZE,`*n* option. See Old format RECID Buffer size: the ZPRGRS_RECID_BUF_SIZE,nnn Option on page 345 for more information on setting this option.

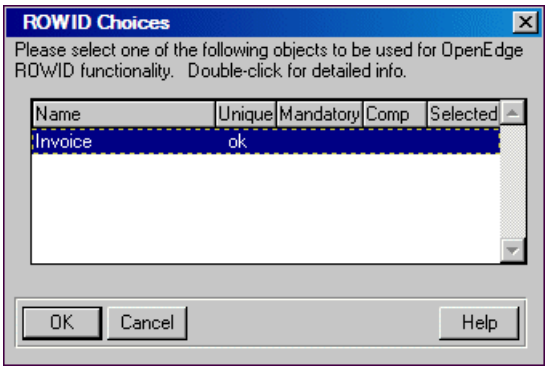
Note: An index that you select as a `ROWID` must be defined as a unique index. It must also be mandatory, if not by definition, then by means of the application code.

Selecting an index to support ROWID

To select an index to support the `ROWID` function, use the following procedure in the Data Dictionary with the schema holder connected (you do not have to connect to the MS SQL Server database).

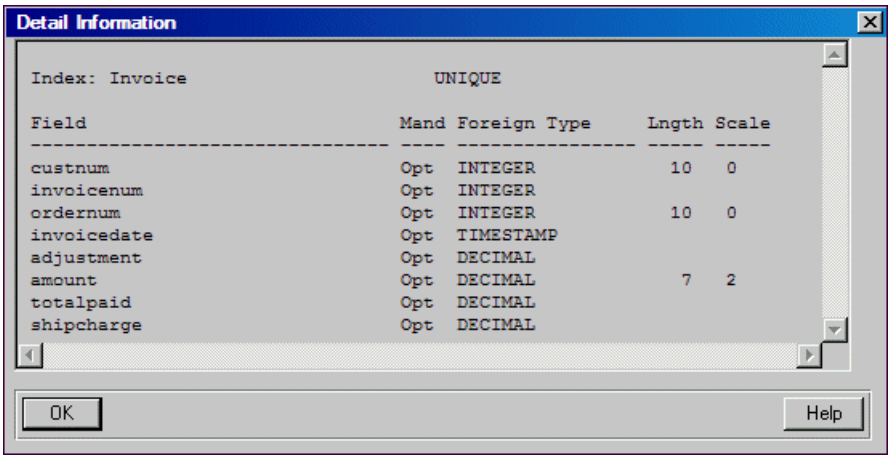
1. Choose **Tables mode**.
2. Select the table whose `ROWID` you want to modify.
3. Choose **Table Properties**.
4. Choose **DataServer**.

   The following dialog box appears:



5. **Double-click** an index name listed in the **Name** field to display detailed information about 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.

Dumping and Loading Schema Holder Information

Some of the Dictionary tools that can be used to modify OpenEdge features associated with your schema and and can change the schema's interaction with your OpenEdge applications. But, Dataserver schema holders contain not just OpenEdge attributes but server attributes that comprise what is known as Schema Image. A schema image is a reflection of the schema attributes of the foreign data source interpreted for OpenEdge. It is contained with a logical database of the schema holder.

It is possible to connect to this logical database and dump and load definitions from the schema image just as you might dump and load schema from an OpenEdge Database. But the foreign database information in that schema image is derived from the Schema Pull process (see Independent OpenEdge MS SQL Server Schema Pull on page 315). The attributes of the schema image, once dumped from the logical database, should not be modified as it would also manipulate the depiction of the foreign data source seen by the DataServer which uses this information to accurately communicate with the foreign data source. One example would be the foreign position of a column in a table(known in the schema dump as "FOREIGN-POS") . This tells the DataServer where to locate column information in a physical record layout. Manipulating FOREIGN-POS information in the definitions file and then loading such definitions back into a logical database that was then used to connect to the same foreign data source would likely lead to data corruption should the DataServer be asked to read or write data to the server by the ABL application.

Note: Attributes of the Schema Image that reflects the foreign data source should never be modified without an appropriate and corresponding change taking place at the foriegn data source that matches the attribute change reflected in the schema image. This cannot be done reliably by manipulating a .df file. The only reliable way to modify schema on a foreign data source and have it accurately reflected in your logical database is to perform a schema pull operation from the changed foreign objects back into your logical database.

Independent OpenEdge MS SQL Server Schema Pull

The utility connects to the MS SQL Server database and then establishes the schema pull criteria, and then imports objects from MS SQL Server to the schema holder. You use this Independent OpenEdge MS SQL Sever schema pull utility to update an OpenEdge schema holder from an SQL server database.

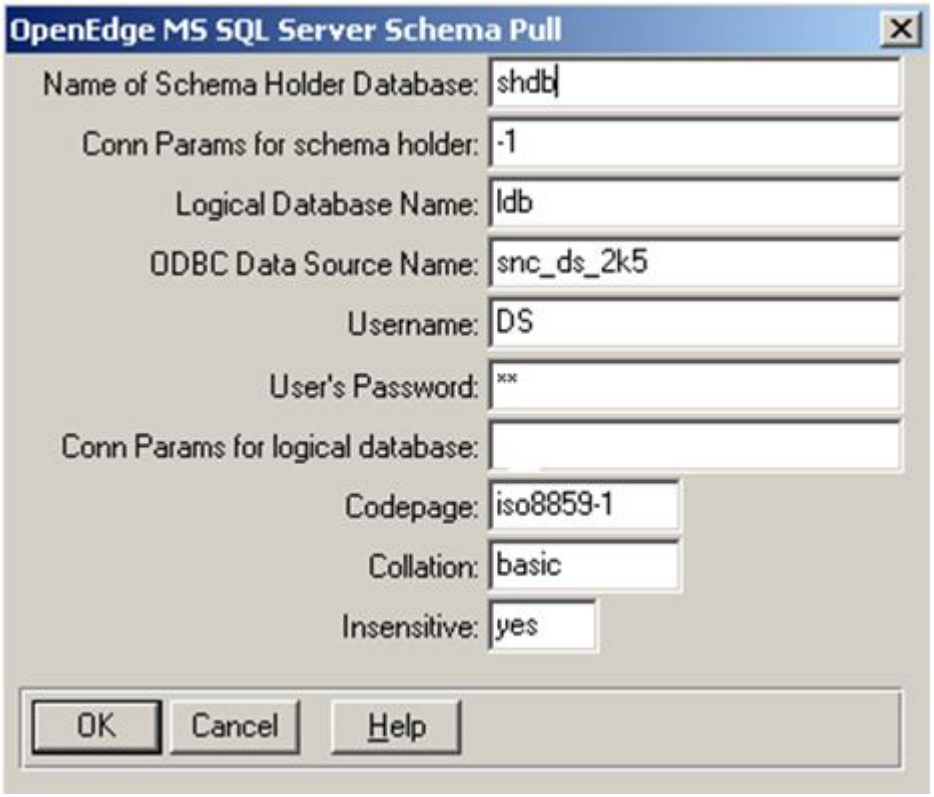
The following sections discuss how you can perform schema pull interactively or silently (that is, in batch mode). All the migration logs, errors and warning messages, are logged at `schpullmss.log`.

To run the Independent Schema pull utility interactively:

1. Execute the following command in the **Proenv** command-line utility:

```
pro -p prodict/mss/schpullmss.p
```

The **OpenEdge MS SQL Server Schema Pull** dialog appears:



Note: The fields in the dialog get populated with the preset environment variables of OpenEdge DataServers. You can either retain the environment variables or overwrite them by providing new values for the environment variables.

2. Provide the information described in the following table in the **OpenEdge MS SQL Server Schema Pull** dialog.

Table 64: OpenEdge MS SQL Server Schema Pull

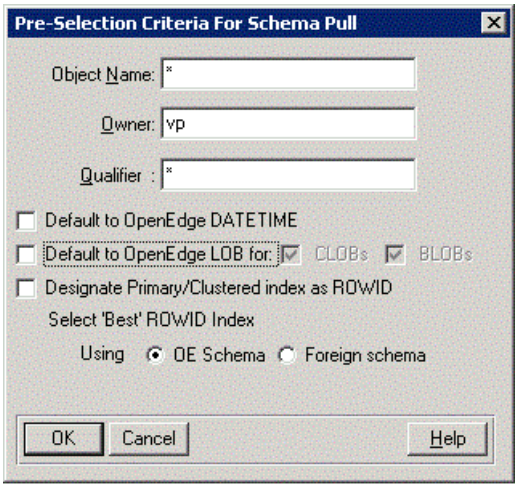
Interface element	Description
Name of Schema holder Database	**Default:** <blank> Specifies a schema holder name of your choice. **Dependencies:** If a schema holder database is not set, the pull operation cannot proceed.
Connect parameters for schema holder	**Default:** <Schema holder of current working database> Specifies the startup parameters for the schema holder pull process. For more information on connection parameters, see Connecting the DataServer on page 189 for connection parameters. **Dependencies:** By default, the schema holder is started in multi-user. Therefore, if you provide your own connect parameters for schema holder, you must include settings to connect the run the schema holder in the single-user mode or read-only mode to avoid connection failure.
Logical Database Name	**Default:** <blank> Specifies a MSS logical database associated with the schema image from the pull process. OpenEdge identifies the MS SQL Server database using the logical database name. **Dependencies:** If the logical database name is not set, the pull operation cannot proceed. The **Logical Database Name** must be different from the **Name of the Schema Holder Database**.

Interface element	Description
ODBC Data Source Name	**Default:** <blank> Specifies the ODBC Data Source Name you used when registering the data source. **Dependencies:** If the **ODBC Data Source Name** is not set, the pull operation cannot proceed. The value for **ODBC Data Source Name** must be different from the **Name of the Schema Holder Database**. The ODBC DNS must be properly specified in the ODBC Data Source Administrator of your system. If the **Load SQL** interface element is set to YES, ODBC DNS must also be properly configured to connect to the foreign Data Source.
Username	**Default:** <blank> Specifies the user ID for MS SQL Server.**Dependencies:** If you do not set the user name that is required for authentication, the pull operation terminates with an error.
User's Password	**Default:** <blank> Specifies the password of the user. **Dependencies:** If you do not set the password that is required for authentication, the pull operation terminates with an error.
Connect parameters for logical database	**Default:** <current working database> Specifies the startup parameters for the schema holder and the foreign data source connection. For more information on connection parameters, see Connecting the DataServer on page 189 for connection parameters. **Dependencies:** Connect parameters are not mandatory. If supplied, they must be formatted as a comma-separated list of names or name-value pairs, without white-spaces, that comprise the `-Dsrv` parameter value passed to the DataServer connection. Some of the parameters are parsed into the DataServer run-time, while others in the list are forwarded onto the target foreign data source. If an invalid or improperly formatted connection parameter is parsed during migration, the pull operation terminates with an error.

Interface element	Description
Codepage	**Default:** iso8859-1 when UNICODETYPES is disabled (that is, set to **NO**). Specifies the corresponding OpenEdge name for the code page with which the MSS Database is compatible. Use UTF-8 if Unicode support is desired. If you use a code page that OpenEdge does not support, you must supply a conversion table that translates between the OpenEdge client code page and the code page that your data source uses. For a complete discussion of code pages, see *OpenEdge Development: Internationalizing Applications*. **Dependencies:** The specified codepage must exist in the `convmap.cp` file of your Progress environment in order to be used during the pull operation.
Collation	**Default:** Basic Specifies the collation name with which the schema holder must collate and weigh code page values. **Dependencies:** The specified collation must exist in the `convmap.cp` file of your Progress environment in order to be used during the migration unless you are using a Unicode code page. In case you are using a Unicode code page, then the code page name can correspond to one of the International Components for Unicode (ICU) collations that provide linguistic sorting of Unicode data based on the Unicode Collation Algorithm.
Insensitive	**Default:** NO Specifies if the code page you use is or isn't case insensitivity. Provide YES if your code page is case insensitive, else retain the default value `NO`. **Dependencies:** By default, the code pages in MS SQL Server are case-insensitive. This matches the default sensitivity in OpenEdge. So, if you are using a case-insensitive code page in MS SQL Server, Progress recommends setting this value to YES.

3. Choose **OK**. When the DataServer connects to the MS SQL Server database, it reads information about data source objects.

   The **Pre-Selection Criteria For Schema Pull** dialog box appears:



4. Provide the information described in the following table:

Table 65: Pre-Selection Criteria For Schema Pull

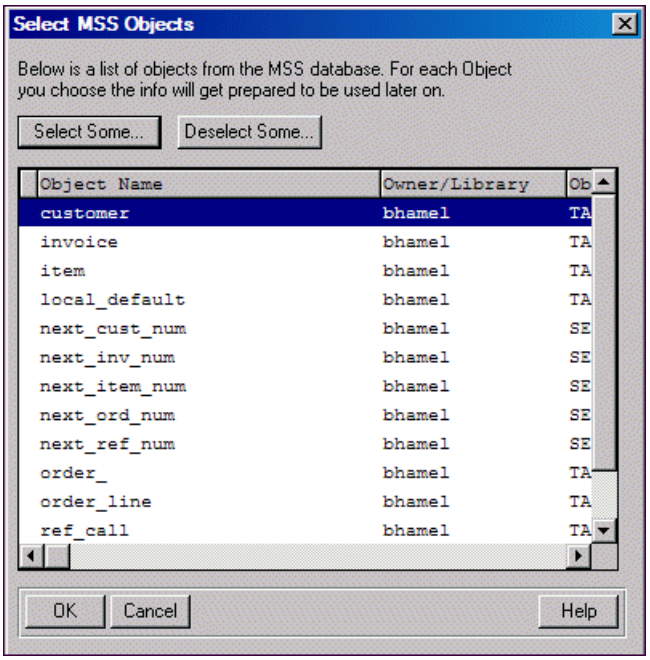
Interface element	Description
Object name	**Default:** * Specify an Object name qualifier for objects to be pulled for the server. The wild card default selects all objects. For example, you can specify A* in the **Object Name** field to list all the tables whose names begin with A or a. **Note:** You should 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.)
Owner	**Default:** *owner name of the connected user* Specify an Object owner to be pulled for the server. The wild card default selects all objects. For example, you can specify DS* in the **Owner**field to list all the tables whose names begin with A or a. **Note:** You should 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.)

Interface element	Description
Qualifier	**Default:** * Specify an Object's qualifying database name to be pulled for the server. The wild card default selects all objects. **Note:** You should 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.)
Default to OpenEdge DATETIME	**Default:** NO or unchecked When set to **YES**, this automatically maps MS SQL Server timestamp data types to the associated OpenEdge equivalent DATETIME data type. When set to **NO**, server timestamp data types map to DATE data type in OpenEdge for backward compatibility.
Default to OpenEdge LOB for:	**Default:** NO or unchecked By default, the unchecked box indicates that all server LOB data types will map to CHARACTER data types in OpenEdge for backward-compatibility. Check this box to enable the mapping of MS SQL Server LOB types to OpenEdge LOB's. **NOTE:** By checking this box you are requiring that your ABL application be coded specifically to handle LOB objects with OpenEdge LOB operations. Check the CLOBs checkbox in order to map MS SQL Server VARCHAR(MAX) and TEXT data types to OpenEdge CLOBS. Check the BLOBs checkbox in order to map MS SQL Server VARBINARY(MAX), IMAGE and FILESTREAM data types to OpenEdge BLOBs to MS SQL Server VARBINARY(MAX), IMAGE and FILESTREAM data types. Else, all server LOB data types map to the CHARACTER data type in OpenEdge for backward-compatible. **Caution:** If you select YES, it enables the use of LOB objects in the ABL language for LOBs in the foreign schema but also disables compatibility with the original OpenEdge database and will cause an Adjust Schema operation to fail. For more information on mapping OpenEdge and MS SQL Server data types, see Support for OpenEdge ABL BLOB data type on page 72.

Interface element	Description
Designate Primary/Clustered index as ROWID	**Default:** NO Specifying **YES** designates the clustered index as ROWID if it qualifies for ROWID selection for a table with a defined clustered index and without a defined PROGRESS_RECID field. **Dependencies:** For the clustered index to qualify for ROWID selection, it must be unique. For legacy purposes, if the server table is identified as having a PROGRESS_RECID field, this setting takes precedence over other settings for ROWID selection.
Select 'Best' ROWID Index	Selecting **Select 'Best' ROWID Index** during migration provides the legacy equivalent of ROWID designation behavior for the tables from prior versions of OpenEdge where the `PROGRESS_RECID` column was not present in a table. When selected in conjunction with new process flow options for ROWID migration, this option plays a secondary role in the designation of ROWID indexes deferring to the **Designate Primary/Clustered index as ROWID** option as the first choice. This option searches for a viable index for ROWID when an `PROGRESS_RECID` column does not exist and other process flow options that take precedence do not render a viable index. Select 'Best' ROWID Index using one of the following options: • **OESchema:** if you must locate an existing index to be used as ROWID that meets the requirements of the OpenEdge Schema. The OpenEdge Schema criteria is described in Best index selection on page 237. If a Best index selection can be found in the OpenEdge Schema, it is designated as ROWID. • **Foreign Schema:** if you must locate an existing index to be used as ROWID from the ODBC API that evaluates best index on the server. If a Best Index selection can be found through the ODBC API, it is designated as ROWID. **Dependencies:** For legacy purposes, if the server table is identified as having a PROGRESS_RECID field, this setting takes precedence over other settings for ROWID selection. Also, if clustered and primary index ROWID mapping is also selected via the SETROWID variable, then the SETROWID variable takes precedence over GETBESTROWID for ROWID selection.

5. Choose **OK**.

   OpenEdge displays a list of the data source objects that you can include in the schema holder, as shown:



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.

6. Click the option appropriate to the action you want to perform:

   - **Select Some** — Displays the **Select by Pattern Match** dialog box on which you can specify object information used to select objects
   - **Deselect Some** — Displays the **Deselect by Pattern Match** dialog box on which you can specify object information used to deselect objects

   You can also elect to select and deselect individual objects by clicking and double-clicking on an object. An asterisk appears next to an object that has been selected; double-click an object to remove the asterisk and identify that the object is now deselected.

7. Choose **OK** after you have identified all the objects you want to include in the schema holder. 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 OpenEdge `ROWID`. If an appropriate index does not exist, the DataServer issues the warning, **Please check errors, 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 Data Dictionary. See Defining the ROWID on page 312 for instructions. For additional information, see Indexes and sorting on page 49 .

Running the Independent schema pull utility in batch mode

Use the following steps to run the Independent schema pull utility in batch mode:

1. Create or use an existing target MS SQL Server database
2. Configure your ODBC driver to connect to your target data source.
3. On your client machine, pass parameters to the utility by setting the environment variables listed in the table that follows.

Note: There are additional parameters required for the batch pull that are not required for the Update/Add Table Definitions Pull Down option. This is because the batch mode pull can be used to perform the duties of **Create DataServer Schema** if a corresponding schema holder database and logical database contains the schema image that does not already exist or is not already connected when the batch schema pull is executed.

Table 66: Independent Schema pull utility

Environment variable	Corresponding Interface element	Description
SHDBNAME	**Schema Holder Database**	**Default:** <blank> Specify the new schema-holder name, in which the resultant logical database will reside. **Dependencies:** If SHDBNAME is not set, the migration terminates with an error.
MSSCONPARMS	**Connect parameters for Schema**	**Default:** <blank> Specifies the startup parameters for the DataServer and the target foreign data source. For more information on connection parameters, see Connecting the DataServer on page 189 for connection parameters. **Dependencies:** Connect parameters are not mandatory. If supplied, they must be formatted as a comma-separated list of names or name-value pairs, without white-spaces, that comprise the -Dsrv parameter value passed to the DataServer connection. Some of the parameters are parsed into the DataServer run-time, while others in the list are forwarded onto the target foreign data source.
MSSPDNAME	**Logical name for MSS Database**	Specify the logical database name configured to connect the foreign data source. **Dependencies:** If MSSPDBNAME is not set, the migration terminates with an error. The value for MSSPDBNAME can be the same as PRODBNAME but must be different from SHDBNAME.
MSSDBNAME	**ODBC Database Name**	**Default:** <blank> Specify the ODBC database name associated with the resultant schema image from the pull operation. **Dependencies:** If MSSDBNAME is not set, the pull operation terminates with an error. The ODBC DSN must be properly specified and located in the ODBC Data Source Administration of your system.

Environment variable	Corresponding Interface element	Description
MSSUSERNAME	**User Name**	**Default:** <blank> Specify the user name for the target data source. **Dependencies:** If MSSUSERNAME is not set, the pull operation terminates with an error.
MSSPASSWORD	**User's Password**	**Default:** <blank> Specify the password of the user for the target data source. **Dependencies:** If a password is required for authentication and MSSPASSWORD variable is not set, the pull operation terminates with an error.
SHCONPARMS	**Conn Params for logical database**	**Default:** *current working database* Specifies the startup parameters for the DataServer and the target foreign data source. For more information on connection parameters, see Connecting the DataServer on page 189 for connection parameters. **Dependencies:** Connect parameters are not mandatory. If supplied, they must be formatted as a comma-separated list of names or name-value pairs, without white-spaces, that comprise the -Dsrv parameter value passed to the DataServer connection. Some of the parameters are parsed into the DataServer run-time, while others in the list are forwarded onto the target foreign data source.

Environment variable	Corresponding Interface element	Description
MSSCODEPAGE	**Codepage**	**Default:** iso8859-1 when UNICODETYPES is disabled (that is, set to **NO**). Specifies the corresponding OpenEdge name for the code page with which the MSS Database is compatible. Use UTF-8 if Unicode support is desired. If you use a code page that OpenEdge does not support, you must supply a conversion table that translates between the OpenEdge client code page and the code page that your data source uses. For a complete discussion of code pages, see *OpenEdge Development: Internationalizing Applications.* It is not mandatory to pass this value during batch migration. You can use **Change DataServer Schema Code Page** utility to add the code page information to the schema holder later but before you start using the DataServer to read and write data. For more information on changing code page, see Changing a code page in a schema holder on page 251. **Dependencies:** The specified codepage must exist in the `convmap.cp` file of your Progress environment in order to be used during the pull operation.
MSSCOLLNAME	**Collation**	**Default:** Basic Specifies the collation name with which the OpenEdge client must collate and weigh code page values. **Dependencies:** The specified collation must exist in the `convmap.cp` file of your Progress environment in order to be used during the migration unless you are using a Unicode code page. In case you are using a Unicode code page, then the code page name can correspond to one of the International Components for Unicode (ICU) collations that provide linguistic sorting of Unicode data based on the Unicode Collation Algorithm.
MSSCASESEN	**Insensitive**	**Default:** NO Specifies if the code page you are matching on the foreign data source is or isn't case insensitive. Provide YES if your code page is case insensitive, else retain the default value `NO`. **Dependencies:** By default, the code pages in MS SQL Server are case-insensitive. This matches the default sensitivity in OpenEdge. So, if you are using a case-insensitive code page in MS SQL Server, Progress recommends setting this value to YES.

Environment variable	Corresponding Interface element	Description
MSSOBJNAME	**Object name**	**Default:** * Specify an Object name qualifier for objects to be pulled for the server. The wild card default selects all objects. For example, you can specify A* in the **Object Name** field to list all the tables whose names begin with A or a. **Note:** You should 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.)
MSSOBJOWNER	**Object owner**	**Default:** *owner name of the connected user* Specify an Object owner to be pulled for the server. The wild card default selects all objects. For example, you can specify DS* in the **Owner information** field to list all the tables whose names begin with A or a. **Note:** You should 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.)
MSSOBJQUALIFIER	**Object qualifier**	**Default:** * Specify an Object's qualifying database name to be pulled for the server. The wild card default selects all objects. **Note:** You should 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.)

Environment variable	Corresponding Interface element	Description
MAPOEDATETIME	**Default to OpenEdge DATETIME**	**Default:** Yes Specifies whether DATE and DATETIME data types in OpenEdge must map to legacy DATETIME timestamp data types in MSS Server. If set to Yes, it provides backward-compatibility to earlier version of OpenEdge and MSS Server. However, if your database target is MSS 2008 or above and MAPSOEDATETIME is set to No, more accurate mapping takes place between OpenEdge data and time data types and the new data and time data types in MSS Server.
MAPTOLOB	**Default to OpenEdge LOB for:** • **CLOB** • **BLOB**	**Default:** NO By default, all server LOB data types map to the CHARACTER data type in OpenEdge for backward compatibility. By changing this parameter from it's default value of NO, you are requiring that your ABL application be coded specifically to handle LOB objects using OpenEdge LOB operations. Specify a value of "B" to map MS SQL Server VARBINARY(MAX), IMAGE and FILESTREAM data types to OpenEdge BLOBs. Specify a value of "C" to map MS SQL Server VARCHAR(MAX) and TEXT data types to OpenEdge CLOBS. Specify a value of "L" (for Lobs) to map both BLOB server types to OpenEdge BLOBs and CLOB server types to OpenEdge CLOBs. For more information on mapping OpenEdge and MS SQL Server data types, see Support for OpenEdge ABL BLOB data type on page 72.

Environment variable	Corresponding Interface element	Description
MAPTOROWID	**Designate Primary/Clustered index as ROWID**	**Default:** NO Specifying **YES** designates the clustered index as ROWID if it qualifies for ROWID selection when the pulled server table has a defined clustered index and does not have a PROGRESS_RECID field defined in it. **Dependencies:** For the clustered index to qualify for ROWID selection, it must be unique. For legacy purposes, if any server table is identified as having a PROGRESS_RECID field, this ROWID designation takes precedence over any other method of determining a ROWID designation.
GETBESTROWID	**Select 'Best' ROWID Index using:** • **OE Schema** • **Foreign schema**	**Default:** Yes or 1 or OE Schema Uses either the OpenEdge Schema or Foreign Schema to determine the candidacy of index attributes for ROWID eligibility. Select 'Best' ROWID Index using one of the following options: • **Select 1 or YES** for **OESchema:** to try to locate an existing index to be used as ROWID that meets the requirements of the OpenEdge Schema. The OpenEdge Schema criteria is described in Best index selection on page 237. If a Best index selection can be found in the OpenEdge Schema, it is designated as ROWID. • **Select 2** for **Foreign Schema:** to try to locate an existing index to be used as ROWID from the ODBC API that evaluates best index on the server. If a best index selection can be found through the ODBC API, it is designated as ROWID. **Dependencies:** For legacy purposes, if the server table is identified as having a PROGRESS_RECID field, this designation for ROWID takes precedence over any other option for designating a ROWID selection for the schema image. Otherwise, if clustered and primary index ROWID mapping is also selected via the MAPTOROWID environment variable, then the MAPTOROWID environment variable will take precedence over the GETBESTROWID option for ROWID selection if GETBESTROWID is on. When COMPATIBLE and all the OpenEdge 11.0 ROWID selection options are turned off, a value of 1 or YES for OE Schema provides the legacy equivalent of ROWID designation behavior for the tables from prior versions of OpenEdge (assuming the PROGRESS_RECID field is not present in the table to supersede this designation).

4. Execute the following command to set and export the above environment variables at the system prompt:

```
PRODBNAME=e
SHDBNAME=sholder
MSSDBNAME=vp_snc10
MSSPDBNAME=lholder
MSSUSERNAME=vp
MSSPASSWORD=vp
SOURCEDB=e
export  PRODBNAME SHDBNAME MSSDBNAME MSSPDBNAME MSSUSERNAME MSSPASSWORD
.
.
.
```

5. Execute the following command in the **Proenv** command prompt to pull schema from MS SQL Server:.

```
pro -p prodict/mss/schpullmss.p -b
```

Adding extended ABL support

The DataServer provides support for arrays and the RECID/ROWID function when certain objects exist in a data source. You can modify your data source to enable support for these extended features. You can also add support when you create new tables.

The DataServer maintains the columns that you add to your data source tables to support the RECID/ROWID function. If non-OpenEdge applications are updating those data source tables, the applications must recognize the additional columns and update them appropriately. You can enable tables either through the use of triggers or through the use of computed columns. The use of computed columns is the preferred method as it offers greater efficiency and reliability, however it is only available when using MS SQL Server 2005 or later versions.

The following sections describe how to modify an existing data source so that it supports arrays, and OpenEdge record identifiers. See ROWID function on page 91 for alternate ways to support this function.

Modifying tables to support arrays

The DataServer allows you to extend OpenEdge's ability to support database arrays to your data source. To have access to this functionality, you must make changes to your data source table.

To modify a table to support arrays:

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. OpenEdge 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.

Using triggers to enable PROGRESS_RECID support of the ROWID function

If you want to use the ROWID function with a data source, you must select an index to associate with this function. Adding a new integer column named PROGRESS_RECID and defining it as a single-component index is the preferred method. This method can only be applied to a table that does not currently have an identity column defined. If your table already has an identity column, you can support ROWID by selecting an existing index. See Defining the ROWID on page 312 for instructions on selecting an existing index. The following procedure describes how to create the PROGRESS_RECID column to support ROWID. To use the ROWID function with a MS SQL Server database, you must make changes to your data source table.

Note: Starting in OpenEdge Release 10.1B, the default PROGRESS_RECID recognizes an INT64 value for a MS SQL Server DataServer. The unique key value in the PROGRESS_RECID will be derived by default from a 64-bit value. (For prior releases, the value was derived, by default, from the 32-bit INTEGER.)

To modify a table to support the ROWID function:

1. Perform these tasks:

   a) Add a column of the integer data type named PROGRESS_RECID. The new column must be able to contain null, as shown:

```
alter table table
    add PROGRESS_RECID integer null
```

   b) Add a column with identity characteristics named PROGRESS_RECID_IDENT_. The new column must have the integer data type, as shown:

```
alter table table
    add PROGRESS_RECID_IDENT_
integer identity
```

2. Create a trigger to maintain the PROGRESS_RECID column:

```
create trigger _TI_table ON table for insert as
 RAISERROR ('PSC-init',0,1)
 SET XACT_ABORT ON
 SET LOCK_TIMEOUT -1
    if  ( select PROGRESS_RECID from inserted) is NULL
    begin
        update t set PROGRESS_RECID = i.IDENTITYCOL
         from customer t  JOIN inserted i ON
         t.PROGRESS_RECID_IDENT_ = i.PROGRESS_RECID_IDENT_
        select convert (bigint, @@identity)
    end
 SET XACT_ABORT OFF
```

3. Create a non-unique index for the PROGRESS_RECID and a unique index for the PROGRESS_RECID_IDENT, respectively:

```
create index table#_#progress_recid on table (PROGRESS_RECID)
```

```
create unique index table#_#progress_recid_ident_on table (PROGRESS_RECID_IDENT_)
```

4. Change the nonunique indexes so that they include a PROGRESS_RECID column as the last component, as shown:

```
create index table#_#index on table (column, PROGRESS_RECID)
```

5. If you have already created your schema holder, delete and recreate it.

Using computed columns to enable PROGRESS_RECID support of the ROWID function

Use the following procedures to create PROGRESS_RECID to support the ROWID function. Note that computed column technology is available only in MS SQL Server 2005 and later versions.

To modify a table to support the ROWID function using a computed column:

1. Add new columns PROGRESS_RECID, PROGRESS_RECID_IDENT_, and PROGRESS_RECID_ALT_, as shown:

```
ALTER TABLE table ADD PROGRESS_RECID AS
     CASE WHEN PROGRESS_RECID_ALT_ is NULL
        THEN  PROGRESS_RECID_IDENT_
        ELSE  PROGRESS_RECID_ALT_
      END PERSISTED NOT NULL,
     PROGRESS_RECID_IDENT_ bigint identity,
     PROGRESS_RECID_ALT_ bigint null default NULL,
     CONSTRAINT table#_#table progress_recid UNIQUE(PROGRESS_RECID)
```

2. If you have already created your schema holder, delete and recreate it.

Migrating RECID-trigger mechanism to RECID computed column mechanism

Use the following procedure to convert a table that supports the ROWID function from one that is trigger-enabled to the more reliable computed column-enabled. Computed column enablement is available only through MS SQL Server 2005 and later.

Note: Prior to performing the following steps, ensure you have backed up the tables with which you will be working.

To adopt a table to support the ROWID function using computed column from trigger mechanism:

1. Drop the trigger on INSERT for PROGRESS_RECID column as:

```
DROP TRIGGER _TI_tbl-name
GO
```

2. Store the current identity value on the table as follows:

```
select ident_current('schemaname.tbl-name') as old_identity_value
go
```

3. Drop the indexes on PROGRESS_RECID, PROGRESS_REID_IDENT_ columns as:

```
drop index table#_#progress_recid on table
drop index table#_#progress_recid_ident_on table
```

4. Drop the other indexes which include PROGRESS_RECID column as:

```
drop index table#_#index on table
```

5. Rename the PROGRESS_RECID column to PROGRESS_RECID_bkp as:

```
EXEC sp_rename 'tbl-name.PROGRESS_RECID', 'PROGRESS_RECID_BKP' ,'COLUMN'
```

6. Drop the PROGRESS_RECID_IDENT_ column as:

```
alter table tbl-name drop column PROGRESS_RECID_IDENT_
```

7. Add new columns for computed column mechanism as:

```
ALTER TABLE tbl-name
ADD PROGRESS_RECID AS
          CASE WHEN PROGRESS_RECID_ALT_ is null
          THEN PROGRESS_RECID_IDENT_
          ELSE PROGRESS_RECID_ALT_
          END PERSISTED not null,
PROGRESS_RECID_IDENT_ bigint identity,
PROGRESS_RECID_ALT_ bigint null default NULL,
CONSTRAINT tbl-name#_#progress_recid UNIQUE(PROGRESS_RECID)
GO
```

8. Update the PROGRESS_RECID_ALT_ column with PROGRESS_RECID_BKP as:

```
update tbl-name set PROGRESS_RECID_ALT_ = PROGRESS_RECID_BKP
go
```

9. Reseed the altered table with previously stored identity value (in step-2)as:

```
DBCC CHECKIDENT ('tbl-name', RESEED, <old-identity-value>)
go
```

10. Drop the PROGRESS_RECID_BKP column as:

```
alter table tbl-name drop column PROGRESS_RECID_BKP
go
```

11. Re-create the dropped indexes during Step 4 as:

```
create index table#_#index on table (column, PROGRESS_RECID)
go
```

12. If you have already created your schema holder, delete and recreate it.

Note: It is recommended that you back-up database tables before following the steps to adopt a new computed column mechanism.

8

Troubleshooting

This chapter describes common problems and how to work around them. Specifically, it explains troubleshooting techniques for:

For information on troubleshooting DataServer connections, see Connection failures and OpenEdge responses on page 222, and Accessing the DataServer log on page 224.

For details, see the following topics:

- Tuning your environment with the -Dsrv startup parameter
- ODBC options

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 MS 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...
```

```
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 -U x -P y
   -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.

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 OpenEdge-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*. The following table lists the OpenEdge-supplied ODBC startup options and the corresponding startup options defined in the ODBC interface.

Table 67: ODBC options

OpenEdge-supplied option	ODBC-defined option[50]
ACCESS_MODE	SQL_ACCESS_MODE
ASYNC_ENABLE	SQL_ASYNC_ENABLE
AUTOCOMMIT	SQL_AUTOCOMMIT

OpenEdge-supplied option	ODBC-defined option[50]
LOGIN_TIMEOUT	SQL_LOGIN_TIMEOUT
MAX_LENGTH	SQL_MAX_LENGTH
MAX_ROWS	SQL_MAX_ROWS
MSS_PRESERVE_CURS	SQL_MSS_PRESERVE_CURS[51]
NOSCAN	SQL_NOSCAN
OPT_TRACE	SQL_OPT_TRACE
PACKET_SIZE	SQL_PACKET_SIZE
QUERY_TIMEOUT	SQL_QUERY_TIMEOUT
RESP_POLLCT	SQL_RESP_POLLCT
RESP_TIMEOUT	SQL_RESP_TIMEOUT
TXN_ISOLATION	SQL_TXN_ISOLATION

When you specify an OpenEdge-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 OpenEdge 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.

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. Most of the DataServer options have a name of the form PRGRS_*option-name*, to reflect its origin. The following table lists and describes these options.

[50] Refer to an ODBC application developer's guide for information on the ODBC-defined options.

[51] Cursor preservation allows server side cursors to be preserved beyond the transaction boundary. See Preserve cursors: the -Dsrv MSS_PRESERVE_CURS,1 option on page 344 for information and warnings.

Table 68: DataServer options

Option	Description
PRGRS_ALWAYS_INDEX	Specifies if the DataServer should always order result sets. The value is either 1 for YES or 0 for NO. If this option is set off, then results that do not **explicitly** describe a desired ordering and/or that do not **require** ordering will be returned unordered. (The default setting is 0.) **Note:** The driver and data source need to have the capability to preserve cursors beyond a transaction boundary in order for the PRGRS_ALWAYS_INDEX option to be available.
PRGRS_BLOCK_CURS	Specified if the DataServer should use block cursors for NO-LOCK queries. The value is either **1** for YES or **0** for NO. This option is set to 1 by default.
PRGRS_CACHE_CONN	Specifies a maximum value to indicate the number of ODBC connections that can be cached without having to close existing connections. The default is **5**.
PRGRS_CONNECT	Passes a connection string to the ODBC driver.
PRGRS_DATEPART_OVERRIDE	Specifies the date component used to override the current date when an MSS Time column is converted to an OpenEdge DATETIME data type.
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 ABL error-handling behavior occurs (that is, OpenEdge waits and retries rather than halting). Valid values are native ODBC data source error message numbers.
LOGGINGLEVEL	Specifies the level at which log entries are written. For more information, see Analyzing application execution with Enhanced Logger on page 206.
LOGENTRYTYPES	Specifies one or more types of log entries. For more information, see Analyzing application execution with Enhanced Logger on page 206.
PRGRS_MAX_BLOCKSIZE	Specifies the size in bytes of the total allowable memory for all block cursors. By default, this value is set to ODBC_DEF_MAXBLOCKSZ, or 1MB.

Option	Description
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, OpenEdge uses the default value. The optimal setting for PRGRS_MINBUF is the size of the largest record data size plus 500 bytes. This can prevent run-time record expansion during the retrieval of query results. **Note: Do not** use this option when the -Dsrv BINDING switch is set to 3. With the binding set to 3, the size of the data is known, and this switch will cause the allocation of unneeded additional memory.
PRGRS_NATIVE_LOCKWAIT	Specifies the server-based wait period in which the server will block wait for access to a locked resource before returning to a MSS DataServer client application. The default value is zero, meaning that control is returned to the client immediately upon receiving a lock condition from the server. **Note:** A number of milliseconds is recommended at startup for transactional applications that anticipate a higher contention for server resources. The optimal value is application-specific.
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.
PRGRS_TABLE_BLOCKSIZE	Specifies the total amount of memory allowed for block cursor use per table. The default value is ODBC_DEF_TABBLOCKSZ, **65,000** bytes.
PRGRS_PROC_TRAN	The value is either **1** for ON or **0** for OFF. Allows a stored procedure to run in a separate connection. This allows OpenEdge to issue concurrent, active requests for running a stored procedure.
PRGRS_STABLE_CURS	The value is either **1** for ON or **0** for OFF. Enabling this option forces the DataServer to act as though cursors have stability across transaction boundaries. **Note:** This option can cause unpredictable query results if not used properly. See Cursor characteristics: the PRGRS_STABLE_CURS option on page 343 for details on the restrictions.

Option	Description
PRGRS_WAIT_DELAY	Specifies the number of seconds to wait for a lock response from the driver when using the driver in asynchronous mode and NO-WAIT has been specified in an ABL statement that requires a database lock. The default value is 2. **Note:** This option has no affect when running in synchronous mode.
QT_FIREHOSE	Specifies that the firehose cursor must be implemented for processing NO-LOCK queries in the current session whenever connection resources are available to do so. **Note:** By default, all sessions will attempt to use firehose cursors for faster cursor processing. For information on cursors, see Cursors on page 87.
QT_NO_FIREHOSE	Specifies that the firehose cursor must **not** be implemented for processing cursors in the current session. **Note:** This option disables the default processing of NO-LOCK queries using the firehose cursors. For information on cursors, see Cursors on page 87.
PRGS_JOIN_NOKEYCACHE	Disables key-cache population for the scenario where the first GET statement is called with EXCLUSIVE LOCK. Users can override this switch for a particular ABL query by setting the QUERY-TUNING(KEYCACHE-JOIN) switch to ABL query. The default value for this connection switch is OFF.

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 Using MS SQL Server and DataServer options on page 340 for information on when and how to use these options.

Using MS SQL Server and DataServer options

This section provides information on using various ODBC and DataServer options.

Transaction Management: the AUTOCOMMIT option

To avoid using the MS SQL Server transaction manager, specify `-Dsrv AUTOCOMMIT,1`. This option exercises the OpenEdge client's local before-image mechanism to manage transactions. This creates a transaction from each SQL statement that the DataServer issues. OpenEdge 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: OpenEdge does not guarantee crash recovery when it uses the local before-image mechanism to emulate transactions.

Query Result Order: the PRGRS_ALWAYS_INDEX option

Some queries can gain a performance advantage when the result set from the query does not use an index to order the results. In older versions of the DataServer for Microsoft SQL Server, results sets were always indexed because queries and their results were gathered in segments with new cursors being generated for each query segment. However, most advanced drivers and data sources now allow a single cursor to be retained for the life of a given query, even across a transaction boundary. Drivers that can preserve cursors are allowed to execute unindexed queries if the `PRGRS_ALWAYS_INDEX` option is set to 0 (off). When this option is off and the user does not specify an index or `BY` clause for the results of their ABL statement and the particular ABL being executed does not require scrolling capability in the OpenEdge client, the query result will remain unindexed by the DataServer.

Note: In an unindexed query, the DataServer will not send an `ORDER BY` clause to the data source. However, the DataServer has no control over whether or not the actual data source utilizes an index in order to generate a result set.

Concurrent query execution: the PRGRS_CACHE_CONN option

It is possible to run read-only queries in separate connections through the driver to the MS SQL Server. Opening a separate connection to run a query or stored procedure can provide better performance, although this is not guaranteed. Having too many open connections can degrade performance. The `PRGRS_CACHE_CONN` option allows you to set a limit for the maximum number of server connections available in the DataServer session. If the session attempts to exceed the maximum threshold, the session will need to wait for an existing connection to complete before an additional connection can be made.

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 OpenEdge; 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=engine-login-name;PWD=engine-login-pass;
UIDDBMS=dblogin;PWDDBMS=dblogin-pass
```

For more information and syntax examples, see Special connection issues on page 202.

Key-buffer size: the PRGRS_IDBUF option

The `PRGRS_IDBUF` option sets the size of the keys buffer. Generally, a default of 25 keys is sufficient. If the ODBC driver being used to access the MS SQL Server database has preserved cursors enabled across a transaction boundary, the keys buffer is used with all non-lookahead cursors. If the driver does have preserved cursors enabled, the `PROGRS_IDBUF` value and the keys buffer are unused.

Locking error messages: the PRGRS_LOCK_ERRORS option

DataServer for MS 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 an OpenEdge database 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_ERRORS,error-number1,error-number2.
```

Large rows: the PRGRS_MINBUF option

Some data rows can be very large; for example, in a MS 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.

The optimal setting for PRGRS_MINBUF is the size of the largest record data size plus 500 bytes. This can prevent run-time record expansion during the retrieval of query results.

Note:

Do not use this option when the -Dsrv BINDING switch is set to 3. With the binding set to 3, the size of the data is known, and this switch will cause the allocation of unneeded additional memory.

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 MS 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.

Concurrent procedure results: the PRGRS_PROC_TRAN option

The DataServer allows only one active request for running a stored procedure. However, you can process results form several stored procedures concurrently if you set the PRGRS_PROC_TRAN switch to 1 (ON.) When switched on, this option will cause a separate connection to be used for each stored procedure request, up to the maximum number of connections specified by the PRGRS_CACHE_CONN option.

Caution: When procedures run in separate connections of the same DataServer session, the scope of their respective transactions is isolated from one another. If one active procedure attempts to update the same record used by another active procedure in the same session, a lock timeout or even a deadlock could occur.

Cursor characteristics: the PRGRS_STABLE_CURS option

Enabling this switch indicates to the DataServer that it should assume that all cursors are stable. Normally, the ODBC driver and MS SQL Server determines whether a cursor is stable during the commit or rollback of a transaction and if the cursor can persist beyond a single transaction scope. The DataServer normally resolves cursor characteristics by interrogating the driver and setting the run-time environment accordingly.

Progress Software Corporation does not recommend bypassing normal operation under most circumstances, but under very limited circumstances, you can improve performance by overriding the derived cursor setting by setting the PRGRS_STABLE_CURS option to **1** (ON). Your application must comply to one of the following restrictions to safely enable this option:

- All ABL run in your DataServer session is without transactions.
- ABL run in your DataServer session has transactions, but all ABL queries and resultant data sets are fully processed on one side of any existing transaction boundary.

Note: This is a session-level switch, which means that all ABL run in the session must comply with the listed restrictions.

Wait time for asynchronous connections: the PRGRS_WAIT_DELAY option

The PRGRS_WAIT_DELAY switch allows you to determine the number of seconds you will allow the DataServer to delay further execution while waiting for the ODBC driver to respond to a request that may initiate a lock on a database resource. The delay is initiated when the NO-WAIT option is used in your ABL and the DataServer is awaiting a response to a database request made through the ODBC driver to MS SQL Server. The PRGRS_WAIT_DELAY switch is ignored unless you are running in ODBC asynchronous mode. Check your driver and OpenEdge connection requirements to determine if you are allowed to run in asynchronous mode.

Preserve cursors: the -Dsrv MSS_PRESERVE_CURS,1 option

The -Dsrv MSS_PRESERVE_CURS,1 startup option allows server side cursors in Microsoft SQL Server to be preserved beyond a transaction boundary. Under many circumstances, preserved cursors can result in a significant performance improvement. Enabling this feature can result in enhanced performance, but performance gains are not guaranteed and in limited circumstances performance degrades. For this reason, this option is disabled by default. To test if preserve cursors benefit your application, enable this option by toggling the switch on with -Dsrv MSS_PRESERVE_CURS,1.

RECID/ROWID format: the variable length option

ROWID/RECID — With the variable length ROWID/RECID format, the area stored in the record buffer for ROWID/RECID is variable. Reduced ROWID/RECID size causes the DataServer to use smaller network bandwidth while passing records between client and server.This might improve network performance. While the DataServer runtime uses the new ROWID/RECID algorithm by default, it allows the old format ROWID/RECID (OpenEdge 11.4 and before) to be maintained.

You can expand the ROWID/RECID buffer size to a maximum of 1000 bytes. The minimum size is 44 bytes. The algorithm to determine minimum adjusted size is as follows:

- Header +
- Table of rpos of key components +
- Table of offsets to key values +
- 1 byte null indicator + 1st comp key value +
- 1 byte null indicator + 2nd comp key value +
- ...

Note: With the introduction of variable length ROWID/RECID buffer, the ZPRGRS_RECID_BUF_SIZE,nnn Option is obsolete. To use the old ROWID format, provide -Dsrv PRGRS_ROWID_VER,0 during DataServer connection.

Old format RECID Buffer size: the ZPRGRS_RECID_BUF_SIZE,nnn Option

The DataServer for MS SQL Server tries to select a unique index over each table defined in your schema holder to support the Progress RECID/ROWID functions. RECID functionality enables backward and forward scrolling in the DataServer product. The RECID buffer is used to sort key information about the unique index selected to support RECID. The default RECID buffer size is 245 bytes. The space is used dynamically to allocate the RECID buffer for a given record. The RECID buffer size needs to be large enough to contain all the key information regarding the unique index. If the unique index selected is a composite of many columns or contains large column names, the RECID key information might exceed the RECID buffer size and issue error message 2090. If you exceed the RECID buffer limit, Progress Software Corporation first recommends that you try to find an alternative unique index with a smaller number of key fields. This should help to improve performance during record access as well. You can change the RECID index in the Data Dictionary by selecting the **DataServer** button from the **Table Properties** of the table in your schema holder.

If it is not possible to change the selected RECID unique index for the table that is causing the buffer overflow, you can expand the area by setting the startup option as follows:

```
-Dsrv ZPRGRS_RECID_BUF_SIZE,nnn
```

Where *nnn* is the new size of the area in bytes. The range limits for *nnn* are inclusive of values between 44 to 1024.

You can expand the RECID buffer size to a maximum of 1000 bytes. The minimum size is 44 bytes. The algorithm to determine minimum adjusted size is as follows:

- 9 + Length of first Index column name + Length of first column data + 17
- + Length of second Index column name + Length of second column data + 22
- + Length of third Index column name + Length of third column data + 22
- + ...

SVUB,x: the Unified Broker Server Option

When x=0 (or when this parameter is unspecified) and the DataServer is connected client/server (i.e., the -S and/or -H parameters are specific), the client attempts to connect to a DataServer server through the traditional ProBroker brokering environment.

When x=1 and the DataServer is connected client/server (i.e., the -S and/or -H parameters are specific), the client's broker connection is redirected away from the traditional PROBRKR broker environment. Instead the client adopts the protocol used by the Unified Broker framework to make the broker connection. You must ensure that you have started a DataServer broker instance in the OpenEdge Management or OpenEdge Explorer (or via the ubroker.properties files) before attempting to connect a client with the -Dsrv SVUB,1 parameter. There are additional configuration parameters to also consider for connectivity in the Unified Broker framework. See Table 30 for DataServer connection parameters and see Starting and stopping a broker process from OpenEdge Management/OpenEdge Explorer and connecting a client on page 190 for additional connectivity information.

Logging options

When connecting to a client with the -Dsrv switch, you can specify logging options. Use logginglevel to define the granularity of your log reports, and use logentrytypes to specify the types of entries to be included in the log. For more information on logging, see Analyzing application execution with Enhanced Logger on page 206.

The following example demonstrates the use of the `logginglevel` parameter:

```
CONNECT data-source-name -ld logical-name -dt mss
-Dsrv logginglevel,3.
```

This next example demonstrates the use of the `logentrytypes` parameter:

```
CONNECT data-source-name -ld logical-name -dt mss
-Dsrv logentrytypes,SQL:3,Cursor:4,Trans.
```

In the above example, the `Trans` log entry type inherits the default logging level of 2 unless the logginglevel `-Dsrv` switch is also specified as a startup parameter - in which case that logging level would be set generically for all log entry types: `SQL`, `Cursor` and `Trans`.

Using the block cursor switches

The DataServer provides one switch to enable or disable block cursors, and two switches to regulate memory consumption by block cursors. Block cursors also interact with the query tuning cache size allocations.

Block cursors: the -Dsrv PRGRS_BLOCK_CURS,1 option

Block cursors are ON (**1**) by default. To disable block cursors for the entire connection, set this option to OFF (**0**). The connection-level setting can be overridden for a specific query with the query-tuning option: `QUERY-TUNING(NO-LOOKAHEAD).`

Block cursor allocation by table: the -Dsrv PRGRS_TABLE_BLOCKSIZE,nnn option

The `-Dsrv PRGRS_TABLE_BLOCKSIZE,` *nnn* option where *nnn* is the size in bytes, indicates the total allowable memory available to cursors of a given table. This value puts a cap on the amount of memory all the open cursors against a given table can consume. It does not determine an amount to be allocated. This upper limit on memory allocation ensures that memory is not overutilized.

`ODBC_DEF_TABBLOCKSZ` is the default value for `PRGRS_TABLE_BLOCKSIZE`. It is currently set at 65,000 bytes. At this rate approximately 18 tables open and utilizing all available cursor space for block cusrors would utilize just 1 MB of memory. If record sizes are large or the typical row count of your result sets is large **and** you have plenty of available memory **and** your open table count is not too large, you might consider increasing this size. If you are using served OpenEdge clients, keep in mind that the memory requirments of the DataServer server executable on the server machine are compounded by the number of users attached to the database through the same broker.

Total Block cursor allocation: the -Dsrv PRGRS_MAX_BLOCKSIZE,nnn option

The `-Dsrv PRGRS_MAX_BLOCKSIZE,`*nnn* option where *nnn* is the size in bytes, indicates the total allowable memory available to all cursors established as block cursors. This value limits the total memory available to all block cursors irrespective of the table to which it is allocated. `ODBC_DEF_MAXBLOCKSZ` sets the default size which is currently 1MB. The overall memory allocated to block cursors by the DataServer connection cannot exceed the value set for `PRGRS_MAX_BLOCKSIZE`. Any block cursor that will place memory consumption over this threshold will be rejected as a block cursor and instead be established as a lookahead cursor.

Impact of block cursors on cache size: the -Dsrv QT_CACHE_SIZE,nnn option

The `-Dsrv QT_CACHE_SIZE,`*nnn* where *nnn* is the size in bytes of the cache to be allocated for a cursor's result set. This is a connection level default that can be overridden at the query level. If the connection level cache size is not overridden at the query level, then the query cache size times the number of open cursors cannot exceed the maximum block areas for the table set by the `-Dsrv PRGRS_TABLE_BLOCKSIZE` switch at any given time. The accumulation of query cache sizes from each block cursor that has been allocated cannot exceed the total maximum block areas available to block cursors as set by the `-Dsrv PRGRS_MAX_BLOCKSIZE` switch. If either block cursor limit is reached, cursors will downgrade to lookahead cursoring. There is no minimum for this value, however if two or more records cannot be returned to the established block, a lookahead cursor is used. The query tuning cache size should be set higher than the maximum record size times two to prevent cursor downgrades.

`ODBC_DEF_BLKCACHESZ` is the default value for `QT_CACHE_SIZE` when block cursors are enabled. This value is currently set to 10,000 bytes.

`ODBC_DEF_LHDCACHESZ` is the default value of `QT_CACHE_SIZE` when lookahead cursors are enabled. It is currently set at 30,000 bytes.

The value of `QT_CACHE_SIZE` represents an upper limit for the row space available to the lookahead cache, **not** the amount of space a lookahead cache will actually use. This highlights a key distinction between block and lookahead cursors. The "cache size" for block cursors is preallocated before results are retrieved so this value represents an actual allocated amount. For lookahead cursors, memory is accrued as rows are read back from the result set and added to the cache. So the "cache size" specifies an upper limit on the number of cached rows allowable, not an exact amount of space that will be allocated as is the case with block cursors.

Adjusting values

With the default size, approximately 6.5 open cursors and result sets per table can be established at a given time before the maximum block memory area (`PRGRS_TABLE_BLOCKSIZE`) is exceeded. At 10,000 bytes, 104 new block allocations can be established before the maximum block memory (`PRGRS_MAX_MBLOCKSIZE`) is exceeded. If record sizes are large or the typical row counts for your results sets are large, you can consider increasing one or both of these sizes. Conversely, if record sizes are small or the typical row counts for your results are small, you can consider decreasing theses sizes. Consider setting these tuning parameters based on your typical requirements and then override them for specific queries that are the exceptions using the query tuning option.

ODBC driver problems

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 utilities or by restarting Windows.

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

Migration Issues

This appendix discusses migration issues. It explains how to migrate from the DataServer for ODBC to the DataServer for MS SQL Server, and also discusses a trigger issue with upgrades from Version 9.1 of the MS SQL Server DataServer.

For details, see the following topics:

- Creating a schema holder
- Modifying your application
- Modifying your insert trigger

Creating a schema holder

The MS SQL Server Utilities are located under the **DataServer** menu option in the **Data Administration**. Progress Software Corporation 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 MS SQL Server instead of ODBC. After completing the schema pull, the schema needs to be adjusted to conform to OpenEdge behavior.

Use the following steps to adjust the schema after completing the pull:

1. Connect to the original OpenEdge database.
2. Select the **DataServer** > **MS SQL Server Utilities** > **Schema Migration Tools** > **Adjust Schema** option to fix the OpenEdge attributes needed for your application to run. You can specify the entire schema holder or specific objects.

Modifying your application

Note the following considerations and plan, as needed:

- The DataServer for MS 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 MS SQL Server.
- Determine your application needs and goals before changing INTEGER data types to INT64 data types in applications. Not all INTEGER data types will necessarily require the additional bit size, nor will wholesale changes in data types necessarily work efficiently with other interfaces to ABL. To review additional considerations about determining when to use INT64 versus INTEGER, see *Release 10 ABL Data Types* Web paper available at the Web paper category of the PSDN Web site located at http://communities.progress.com/pcom/community/psdn.

Modifying your insert trigger

Progress Version 9.1D modified the insert trigger that it places on a table in the foreign database when the PROGRESS_RECID support is selected. The modification reduces the level of contention between multiple users performing inserts simultaneously. The reduction in contention can produce performance improvement.

If your database was created using the OpenEdge DB to MS SQL Server utility and PROGRESS_RECID support is enabled, Progress Software Corporation recommends that you re-create your database by dumping your data, executing the OpenEdge DB to MS SQL Server migration again, and then reloading your data. If this is not possible, then you can manually change your existing MS SQL Server database by doing the following on each table that has been defined with PROGRESS_RECID support from your MS SQL Server Migration:

1. Drop the unique index created over the PROGRESS_RECID field. There should be one index for each table created during the initial migration. The syntax is:

```
DROP index database.owner.tablename#_#progress_recid.
```

2. Drop the old insert trigger. The name typically looks like _TI_*tablename*. The syntax is:

```
DROP trigger database.owner._TI_tablename
```

3. Add the new trigger. The syntax is:

```
create trigger _TI_tablename ON tablename for insert as
 RAISERROR ('PSC-init',0,1)
 SET XACT_ABORT ON
 SET LOCK_TIMEOUT -1
    if  ( select PROGRESS_RECID from inserted) is NULL
    begin
        update t set PROGRESS_RECID = i.IDENTITYCOL
         from customer t  JOIN inserted i ON
         t.PROGRESS_RECID_IDENT_ = i.PROGRESS_RECID_IDENT_
        select convert (bigint, @@identity)
    end
 SET XACT_ABORT OFF
 RAISERROR ('PSC-end',0,1)
 RAISERROR ('PSC-end',0,1)
```

Sample code for the `db.owner.Customer` table is:

```
SET XACT_ABORT ON
 SET LOCK_TIMEOUT -1
    if  ( select PROGRESS_RECID from inserted) is NULL
    begin
        update t set PROGRESS_RECID = i.IDENTITYCOL
         from customer t  JOIN inserted i ON
         t.PROGRESS_RECID_IDENT_ = i.PROGRESS_RECID_IDENT_
        select convert (bigint, @@identity)
    end
 SET XACT_ABORT OFF
 RAISERROR ('PSC-end',0,1)
```

4. Create a new **nonunique** index for each table naming the new index the same name as was dropped earlier in Step 1 on page 350. Make sure it is not unique. The syntax is:

```
CREATE INDEX database.owner.table#_#progress_recid ON
tablename(PROGRESS_RECID)
```

A second index must be created for the PROGRESS_RECID_IDENT field. This index must be created as **unique** and named *tablename*#_#progress_recid_ident. The syntax is:

```
CREATE UNIQUE INDEX tablename#_#progress_recid_ident_ ON
tablename(PROGRESS_RECID_IDENT_
```

An example of the two indices that must be created for the `customer` table is as follows:

```
CREATE INDEX customer#_#progress_recid ON CUSTOMER (PROGRESS_RECID)
CREATE UNIQUE INDEX customer#_#progress_recid_ident_ ON CUSTOMER(PROGRESS_RECID_IDENT_)
```

B

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.

See *OpenEdge Getting Started: Installation and Configuration* for additional information about the utilities and their role and relationship to other system administration facilities. See *OpenEdge Deployment: Startup Command and Parameter Reference* for additional information about syntax and usage.

For details, see the following topics:

- OpenEdge Management or OpenEdge Explorer command line utilities for the DataServer
- Other command line utilities for the DataServer
- DataServer startup parameters

OpenEdge Management or OpenEdge 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. The utilities are:

- MSSCONFIG utility on page 354
- MSSMAN utility on page 355

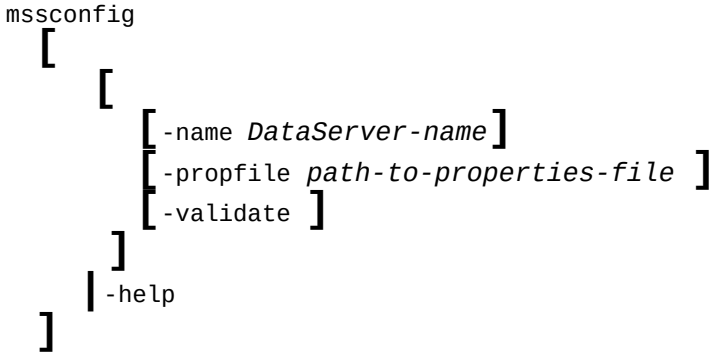
- NSCONFIG utility on page 357
- NSMAN utility on page 359

MSSCONFIG utility

Use the MSSCONFIG utility to help you debug existing DataServer for MS 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 MS SQL Server configuration, and checks that the syntax and values are valid.

The MSSCONFIG utility runs locally, on the machine on which the AdminService is running. The utility does not run across the network.

Syntax

Operating system	Syntax
Windows	

Parameters

`-name` *`DataServer-name`*

Specifies which existing configuration to examine. The name must match the name of an existing DataServer for MS 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 MS SQL Server 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`.

`-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, and property settings for each product instance. OpenEdge Management/OpenEdge 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 OpenEdge installation directory. For example, `%DLC%\properties\ubroker.properties` in Windows.
- 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 MS SQL Server configurations in `ubroker.properties` can include:

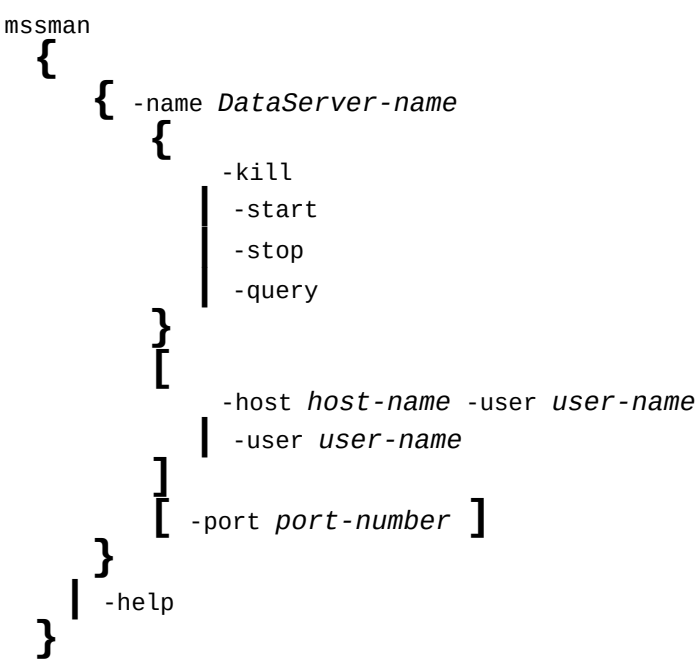
Configuration entity name	Configuration entity function
`[UBroker]`	Defines default property settings for all NameServer, AppServer, DataServer, and WebSpeed Transaction Server brokers.
`[UBroker.MS]`	Defines default property settings for all instances of DataServers for MS SQL Server.
`[UBroker.MS.product-instance-name]`	Defines property settings for this instance of a DataServer for MS SQL Server. The `ubroker.properties` file can 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 can edit the `ubroker.properties` file using a text editor such as 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 MS SQL Server. The utility allows you to start a broker, query its status, start and stop additional DataServer servers, and shut down the broker.

Syntax

Operating system	Syntax
Windows	

Parameters

`-name` *`DataServer-name`*

Specifies the name of a broker. This parameter is required.

`-kill`

Stops and removes the DataServer broker from memory, no matter what it is doing.

`-start`

Starts the DataServer broker.

`-stop`

Tells the DataServer broker to stop itself.

`-query`

Queries the DataServer brokers and servers for status.

`-host` *`host-name`*

Specifies the name of the machine where the AdminService 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 AdminService 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 supports three different formats:

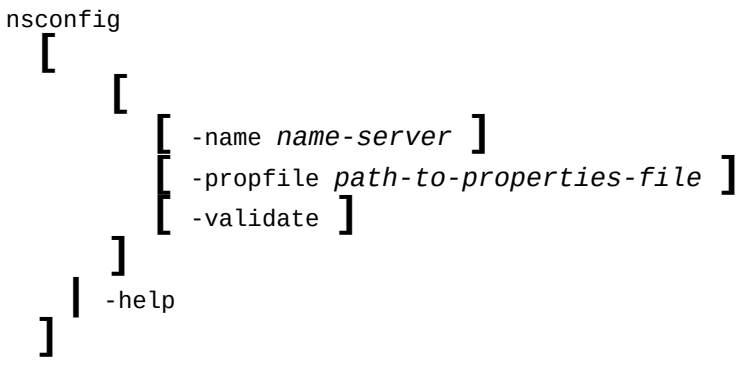
- A user name as a simple text string, such as "`mary`," implies a local user whose user account is defined on the local Windows server machine, which is the same machine that runs the AdminService.
- A user name as an explicit local user name, in which the user account is defined on the same machine that runs the AdminService except the user name explicitly references the local machine domain, for example "`.\mary`".
- A user name as a user account on a specific Windows 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 Windows Server, including the one where the AdminService 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 AdminService is running. The utility does not run across the network.

Syntax

Operating system	Syntax
Windows	

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` in Windows
- `$DLC/properties/ubroker.properties` on UNIX

`-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 OpenEdge Management or OpenEdge 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, and property settings for each product instance. OpenEdge Management/OpenEdge 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 OpenEdge installation directory. For example, `%DLC%\properties\ubroker.properties` in Windows.

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.

The NameServer configurations in `ubroker.properties` can 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 a NameServer. The `ubroker.properties` file can contain several of these entities each with a unique *`product-instance-name`*.

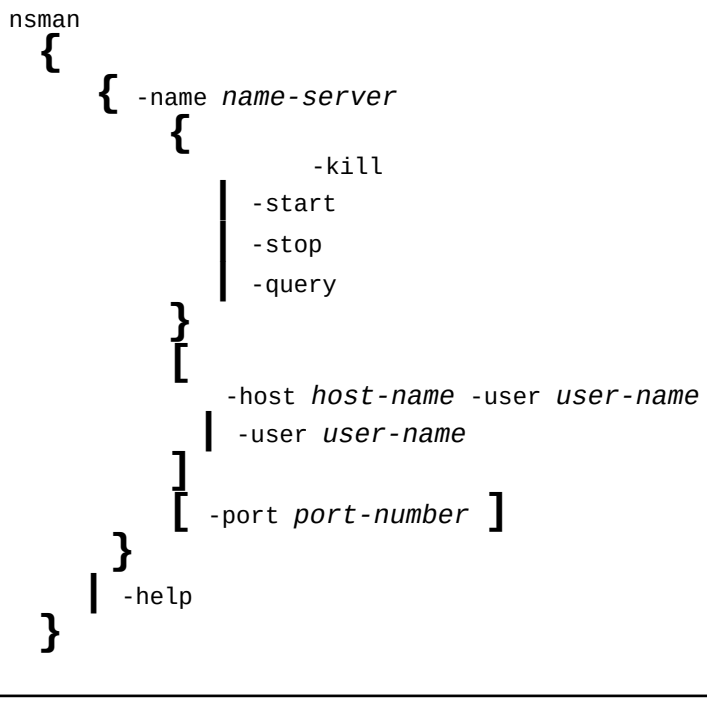
Parent entities provide default values for all of their child entities. For example, the parent `[UBroker]` contains a set of definitions that can be inherited by its child `[NameServer]`, and then again by its child `[NameServer.`*`product-instance-name`*`]`. However, at any child level, a redefinition of any value supersedes the default value of its parent. All children from the redefinition level down inherit this new value.

Optionally, you can edit the `ubroker.properties` file using a text editor such as 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 NSCONFIG 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. Unlike OpenEdge Management or OpenEdge Explorer, the NSMAN utility does not support a means to view log files or delete configured NameServer instances.

Syntax

Operating system	Syntax
Windows	

Parameters

`-name` *`name-server`*

Specifies the name of the NameServer. This parameter is required.

`-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 AdminService 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 AdminService 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 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 Windows server machine, which is the same machine that runs the AdminService.
  - A user name as an explicit local user name, in which the user account is defined on the same machine that runs the AdminService except the user name explicitly references the local machine domain, for example "`.\mary`".
  - A user name as a user account on a specific Windows 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 Windows Server, including the one where the AdminService is running.

Other command line utilities for the DataServer

This section describes the command line utilities you use to start and stop a DataServer that are not related to OpenEdge Management or OpenEdge Explorer. It discusses the purpose, syntax, and primary parameters for each operating system. The utilities are presented in alphabetical order. The utilities are:

- PROBRKR command on page 361
- PROSHUT command on page 362
- DataServer startup parameters on page 363

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.exe -S` *`service-name`* **[** `-H` *`host-name`* **]**

Parameters

`service-name`

Specifies the name of the broker process service on the host machine.

`host-name`

Specifies the name of the machine where the DataServer broker is installed. The default value is the local host.

Notes

- See *OpenEdge Deployment: Startup Command and Parameter Reference* for more details on the Server Name (`-S`), Host Name (`-H`) startup parameters.
- You can use any of the startup parameters with the PROBRKR command. See *OpenEdge Deployment: 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 MS SQL Server. Be sure to set this variable on the host machine. Also, in the same environment, make sure you have set all MS SQL Server environment variables required to connect to the data source. See Configuring the DataServer on page 161 for examples of required variables.
- To set the Message Buffer Size (-Mm) startup parameter for the broker to a value different from the default buffer size of 1024, you must set the `PROSTARTUP` environmentvariable equal to the path and name of the .pf file that contains the -Mm parameter and value that you are passing before you start the broker. This approach ensures that your value is recognized and therefore used by the broker when it starts. Only Windows clients must use this technique.
- Start the broker on a server that is locally connected to the disk containing the data source.

PROSHUT command

Shuts down the OpenEdge database server and individual OpenEdge processes. Before you shut down the server, have all application users quit their OpenEdge sessions. If necessary, you can disconnect users by using the PROSHUT command's Disconnect a User or Unconditional Shutdown parameters.

Syntax

Operating system	Syntax
Windows	`proshut { db-name | -Gw } [ -S service-name ]`

Parameters

db-name

Specify the name of the schema holder to shut down its server, if it was started in multi-user mode.

`-Gw`

For DataServers, specifies that the DataServer broker is to be shut down.

`-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.

Notes

For a complete listing of PROSHUT parameters and their functions, see *OpenEdge Data Management: Database Administration*.

DataServer startup parameters

The following table lists the parameters that you use with the command line utilities to start a DataServer for MS SQL Server.

Table 69: DataServer parameters

Parameter	Syntax
DataServer Logging	`-dslog filename`
DataServer	`-Dsrv keyword,value`
DataServer minimum port number	`-dsminport port-num`
DataServer maximum port number	`-dsmaxport port-num`
Database Type	`-dt db-type`
Host Name	`-H host-name`
Service name	`-S service-name`
Data Source User Name	`-U user-name`

Parameter	Syntax
Data Source User Name Password	`-P` *`password`*
Server Join	`-nojoinbysqldb`

C

Data Type Details

This appendix supplements the data type information presented in Data types on page 56. Specifically, this appendix presents each supported MS SQL Server data type, its OpenEdge data type equivalences, and any notes related to using each data type.

For details, see the following topics:

- Data type conversion details

Data type conversion details

The table below identifies each supported DataServer for MS SQL Server data type, its SQL-ODBC equivalent data type, and its supported OpenEdge equivalent data types. In the OpenEdge -equivalent data types column, the initial data type identified is the default. The data types in parentheses are alternative data types that you can specify in the schema holder for your MS SQL Server data source.

Note that the SQL-ODBC data types are presented as information only; you do not need to know nor use the SQL-ODBC data types to work with the DataServer for MS SQL Server.

Table 70: MS SQL Server data type details

MS SQL Server data type	SQL-ODBC data type[52]	OpenEdge-equivalent data types
integer	SQL_INTEGER	INTEGER[53] (DECIMAL) (INT64)[54]

MS SQL Server data type	SQL-ODBC data type[52]	OpenEdge-equivalent data types
bigint	SQL_BIGINT	INT64[55]
smallint	SQL_SMALLINT	INTEGER (DECIMAL or LOGICAL)
tinyint	SQL_TINYINT	INTEGER (DECIMAL or LOGICAL)
decimal	SQL_DECIMAL	DECIMAL (INTEGER) (INT64)[54]
numeric[56]	SQL_DECIMAL	DECIMAL (INTEGER) (INT64)[54]
float[57]	SQL_FLOAT	DECIMAL (INTEGER) (INT64)[54]
double precision	SQL_DOUBLE	DECIMAL (INTEGER) (INT64)[54]
real	SQL_REAL	DECIMAL (INTEGER) (INT64)[54]
char[58,59]	SQL_CHAR	CHARACTER
nchar[58,59]	SQL_NCHAR	CHARACTER
varchar[58,59]	SQL_VARCHAR	CHARACTER
varchar(max)	SQL_VARCHAR SQL_LONGVARCHAR	CHARACTER CLOB
nvarchar[59]	SQL_NVARCHAR	CHARACTER

MS SQL Server data type	SQL-ODBC data type[52]	OpenEdge-equivalent data types
text[60]	SQL_LONGVARCHAR	CHARACTER CLOB
ntext	SQL_LONGVARCHAR	CHARACTER CLOB
nvarchar(max)	SQL_VARCHAR	CHARACTER CLOB
money	SQL_DECIMAL	DECIMAL (INTEGER)
smallmoney	SQL_DECIMAL	DECIMAL (INTEGER)
datetime	SQL_TIMESTAMP	CHARACTER[61] (DATE)[62]
smalldatetime	SQL_TIMESTAMP	CHARACTER[62] (DATE)[63]
binary[58]	SQL_BINARY	CHARACTER
varbinary	SQL_VARBINARY	CHARACTER
varbinary(max)	SQL_VARBINARY SQL_LONGVARBINARY	CHARACTER BLOB
image	SQL_LONGVARBINARY	CHARACTER[62] BLOB
bit	SQL_BIT	LOGICAL
timestamp[63]	SQL_VARBINARY	CHARACTER
identity,[63,64]	Depends on underlying type	Depends on underlying type
uniqueidentifier[63]	SQL_VARBINARY	CHARACTER
compounded columns[63]	Depends on underlying type	Depends on underlying type

[52] The SQL-ODBC data types demonstrate the mapping of native MS SQL Server data types to the ODBC standard.

[53] Existing or new 32-bit INTEGER data types have a maximum internal field size of 64-bits. The internal field expansion supports larger values defined as INTEGER data types.

[54] Numeric expressions supported include DECIMAL, INTEGER, and INT64.

[55] The OpenEdge INT64 data type enables the MS SQL Server DataServer BIGINT data type to be successfully pulled into a schema holder.

[56] The DataServer truncates values in MS 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.

[57] Do not use the FLOAT or REAL data types in joins, in primary keys, or with the equality operator.

[58] When you define a binary or char column to allow nulls, MSS stores the data type definitions as varbinary and varchar respectively. This does not affect how the DataServer maps the MSS data types to OpenEdge data types.

[59] When migrating an OpenEdge database to SQL Server, character fields whose format is less than x(8000) are defined as VARCHAR by default if "ABL format" is used for field widths during migration. Fields whose SQL widths are less than 8000 are defined as varchar by default if "width" is used for fields widths during migration, and larger character fields are defined as VARCHAR(MAX). When the database uses unicode data types, the 800 byte record limit in SQL Server must be expressed in 4000 characters, so the above limits are halved for unicode data.

If there are several large character fields in one record that each individually fall under the maximum size limits so as to be defined as varchar, but in sum, exceed the maximum record size for SQL server, it will be necessary to define the fields as a server CLOB type, such as varchar(max) in order to successfully create the records.

The default server type mapping for character data that exceeds the SQL Server record size limit is changed from Text/ntext to varchar(max)/nvarchar(max) in SQL Server 2005 and later.

[60] The amount of data that can be accessed in a field defined to be CHARACTER by an OpenEdge application is limited by the maximum size of a record that can be passed to the client. The maximum record size for DataServer applications is 32,000 bytes. Keep this in mind if you decide to map server CLOB type to CHARACTER rather than CLOB. If you map server LOB data to CHARACTER data types but you don't process the data in these fields on the returned records, it is recommended that you construct field lists to exclude them from your query results.

[61] By default, the initial value of a MS SQL Server datetime or smalldatetimecolumn is the Unknown value (?). The default initial values for binary and varbinary are also the Unknown value (?). The MS SQL Server datetime and smalldatetime data types contain both date and time information. The DataServer maps these to the OpenEdge 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).

[62] When mapping of the MS SQL Server datetime or smalldatetime data types is to the OpenEdge DATE data type, OpenEdge truncates the time portion of the date.

[63] The DataServer considers this data type to be a hidden value. Its value is set by the server rather than the application. However, you can still access a MS SQL Server table that contains this type of column.

[64] Identity columns are limited by SQL Server to one per table. The PROGRESS_RECID identity column can be auto-generated in a migrated table in order to represent ROWID for OpenEdge for that table. Your SQL Server table can define an alternative identity column but a user-defined identity column cannot be used like PROGRESS_RECID to represent ROWID. See ROWID function on page 91 for more information.

D

Using qt_debug to Analyze Performance

The qt_debug option of the DataServer (-Dsrv) startup parameter can be used as a supplement to the OpenEdge Enhanced Logger to provide you with the means to perform diagnostics. Information in this appendix describes options used to enable logging with the qt_debug option. For information on using the Enhanced Logger as an alternative to analyzing application activity and performance, see Analyzing application execution with Enhanced Logger on page 206 and *Troubleshoot ABL Applications*.

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 MS SQL Server database. The following table lists the diagnostic capabilities of qt_debug.

Table 71: Diagnostic options

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 OpenEdge maximum cursor settings or cursor "leaks" that your application might have.

Option	Description
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.
qt_debug,CALL_SUMMARY	Prints information on cursors and timing. This information is supplied in summary form as an OpenEdge data (.d) file. Contact Progress Software Corporation 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 -Dsrv qt_cache_size,32000,qt_debug,PERFORMANCE.
```

Index

4GLTrans log entry type *210*
64-bit sequences
  support for *56*

A

ABL differences
  OpenEdge databases and MS SQL Server data sources *97*
ABL to SQL Correlation
  PRGRS_LOG_ABL switch *218*
ABL transaction logging
  log entry types *210*
Aggregates
  in views *53*
Arrays
  in joins *151*, *153*, *157*
  manually creating support *329*
  supporting in data source *329*
Auto-commit *341*
Auto-connect
  failure response *222*

B

BEGINS function *97*
Binding *204*
Brokers
  DataServer *361*
  starting from MS-DOS *167*, *193*
  Unified Broker *186*
Buffers
  allocating dynamically *342*
  defining for stored procedures *117*
BY option
  FOR EACH statement *49*
  joins *151*, *153*, *157*

C

Cache files
  changes to schema holders *221*
  creating *197*
  local schema cache *221*
Caching
  schema *25*
Case-insensitive indexes
  supporting in data source *329*
Case-sensitive indexes *51*
character sets
  defined *44*
Character sets
  changing in a schema holder *251*
Clear dynamic temp-table
  defined *123*
CLOSED STORED-PROCEDURE statement
  complete syntax reference *106*
  description *102*, *109*
  retrieving parameter values *115*
Code pages
  changing in a schema holder *251*
  collation table *46*
  conversion table *46*
  setting in schema holder *181*, *238*
COMPILE statement *96*
Configuration examples
  local DataServer *200*
  remote DataServer *201*
CONNECT statement
  failure response *222*
  NO-ERROR option *222*
Connecting schema holders
  at startup *197*
Connection BINDING Options *204*
Connection pool
  defined *139*
Connection pooling
  considerations *139*
  disabling *140*
  enabling *140*
  impact on commits *141*
  ODBC *140*
Connection to data source
  blank space in data source name *202*
  DataServer opens multiple connections *223*
  failure response due to existing connection *222*
  failure response due to logical name *222*
  failures with SQL Server *222*
  guidelines *194*
  interactive connection *202*
  non-standard connections *202*
  security on NT *194–195*
  troubleshooting *223*, *341*
Connection-string
  defined *202*
Connects log entry type *210*
CONTAINS operator *97*
COUNT-OF function *97*
CREATE statement *75*, *97*
Creating a temp-table layout plan
  dynamic temp-table in an prepared state *124*
  dynamic temp-table in an unprepared state *123*

Creating a temp-table layout plan *(continued)*
options *123*
Creating OpenEdge databases *180*
Cursor
defined *87*
Cursor log entry type *210*
Cursors
cursor downgrades *144*
monitoring cursor and connection use *143*
multiple *223*
statement cache *145*

D

Data Dictionary
changing data types *310*
connection failure response *222*
modifying schema holders *309*
Data output and retrieval options
results retrieved from a database
defining a special view on the MSS data source *113*
OpenEdge-supplied proc-text-buffer *113*
return codes *112*
values of output parameters you define when creating a procedure *112*
data source
defined *42*
Data sources
adding objects to schema holder *181*, *238*, *315*
code page *181*, *238*
data types *56*
database objects *42*
keys *49*
locking behavior *78*
logical name *181*, *238*
login name *97*, *180*
naming conventions *43*
null value *74*
password *97*, *180*
permissions *177*
physical name *181*, *238*
registering *163*
security levels *26*
triggers *54*
views *53*
zero-length character strings *74*
Data types
changing *310*
conversion details *365*
display format *310*
overflow checking *86*
stored procedures
INT64 in static and dynamic temp tables *122*
support for INT64 *102*
supported by DataServer *56*
Data types *(continued)*
user-defined *73*
Database conversion
OpenEdge-to-MS SQL Server utility *268*
Database Type (-dt) startup parameter *197*
Databases
OpenEdge objects *42*
SQL Server data-source objects *42*
triggers *54*
DataServer
$UNIQUE_DSLOG environment variable *226*
architecture *22*
architecture components *22*
broker
on host machine *164*
compiling applications *96*
components *162*
configurations *27*
configuring for distributed applications
using probrkr *30*
configuring with OpenEdge Management or OpenEdge Explorer on NT *164*
connection failure *222*
creating schema holders *181*, *238*
diagnostic options *369*
error handling *85*
internal logic *24*
local configuration *28*
log file *167*, *224*
log file creation error handling *225*
log file location setting *225*
logging level and log entry types setting *215*
MS SQL Server utilities *22*
Naming log file *224*
on host machine *164*
options *337*
record creation *75*
remote configuration *29*
Restrictions on changes to log file *225*
scrolling *87*
security considerations *25*
Server context log file naming *217*
starting
UNIX client *194*
Windows client *193*
starting in OpenEdge Management or OpenEdge Explorer *190*
utilities *26*
DataServer (-Dsrv) startup parameter
alternate syntax *204*
DataServer options *337*
ODBC options *336*
PRGRS_CONNECT option *202*
qt_ debug option *204*
qt_ separate_connection option *204*
qt_loookahead option *204*

DataServer (-Dsrv) startup parameter *(continued)*
qt_no debug option *204*
qt_no lookahead option *204*
qt_no separate_connection option *204*
query-tuning *204*
stored procedures *102*
tuning your environment *335*
DataServer for SQL Server *22*
DataServers
networks *36*
starting a broker *361*
DB.Connects log entry type *210*
DBE DataServer
startup parameters *268*
DBRESTRICTIONS function *97*
Deadlock
defined *81*
stored procedures *102*
Default result set
defined *141*
DEFINE BROWSE statement *93*
DEFINE BUFFER statement *115*
Defining a view to use as a buffer *117*
Defining Constraint
unique key constraint definition *258*
Defining constraints
activating or deactivating constraint *265*
check constraint definition *261*
default constraint definition *263*
deleting constraint *265–266*
foreign key constraint definition *259*
modifying constraint *264*
primary key constraint definition *256*
Deleting schema holders *252*
Demonstration databases
requirements for use *228*
Diagnostic options
qt_debug *369*
Display formats *310*
Distributed DataServer
configured using probrkr *30*
Documentation
suggested reading list *39*
DS.Cursor log entry type *210*
DS.Interface log entry type *210*
DS.Performance log entry type *210*
DS.QryInfo log entry type *210*
DSLOG-MANAGER system handle attributes and methods *208*
DSLOGDIR environment variable *167*, *224*
Dummy indexes
for non-indexed data sources *50*
Dynamic temp-table
error conditions *123*

E

Enhanced Logger
Log entry types *210*
Logging levels *215*
Enterprise DataServer for SQL Server *22*, *162*
Environment variables
DSLOGDIR *167*
MSSSRV *167*
PROBRKR *167*
Error handling *85*
Error messages
connection failures *222*
locking problems *342*
Error messages related to using the LOAD-RESULT-INTO phrase *130*
Exclusive locks *79*, *81*
EXCLUSIVE-LOCK option
in joins *151*, *153*, *157*
record locking *78*
expression
defined *108*
Extended 4GL
supported features *38*
Extended ABL
adding support to data source *329*

F

Fast Forward-Only cursors *143*
field extends
defined *74*
Field extents *74*
Field formats *310*
Field lists *93*
Field properties
modifying *310*
FIELDS option *93*
FIND statement
LAST option *95*
PREV option *95*
Firehose cursors *141*
FOR statement
EACH option *49*, *97*
Foreign keys *49*

H

Handling errors
Error messages related to using the LOAD-RESULT-INTO phrase *130*
Holding locks *79*
Host Name (-H) startup parameter *197*

I

Incremental schema changes
  support for changing INTEGER to INT64 using ALTER TABLE ALTER COLUMN statement *296*
Indexes
  case-sensitive *51*
  dummy *50*
  for stored procedure results *53*
  for views *53*
  initial value *52*
  problems importing *342*
  selecting for views *50*
  simulated *50*
  support for large key entries *51*
  unique
    FIND statements *50*
    for browsing *50*
INPUT option
  stored procedures *108*
INPUT-OUTPUT option
  retrieving parameter values *115*
  stored procedures *108*
INT64 data type
  MS SQL Server data type equivalents *365*
Interface log entry type *210*
Internal Code Page (-cpinternal) startup parameter
  OpenEdge-to-MS SQL Server utility *268*
Isolation levels *79*

J

Join by SQLDB
  defined *151*, *153*, *157*
  option *151*, *153*, *157*
JOIN-BY-SQLDB option *151*
Joins
  controlling by SQLDB behavior *151*, *153*, *157*

K

Key buffer
  troubleshooting *342*
Keys
  foreign *49*
  primary *49*

L

LOAD-RESULT-INTO phrase
  description *107*
Loading result sets into temp-tables
  creating a temp-table layout plan *120*
  Using a temp-table handle with an unprepared dynamic temp-table *123*
Local DataServer
  starting *189*
LocalCache (-cache) startup parameter *197*
Locks
  deadlock *81*
  page-level *79*
  record locking *78*
  stored procedures *102*
  table-level *79*
  upgrading *79*
Log entry types
  4GLTrans *210*
  Connects *210*
  Cursor *210*
  DB.Connects *210*
  DS.Cursor *210*
  DS.Interface *210*
  DS.Performance *210*
  DS.QryInfo *210*
  Interface *210*
  Performance *210*
  SQL *210*
  Trans *210*
Log files
  accessing *224*
LOG-MANAGER handle
  attributes and methods *207*
Logical database name
  changing *251*
Logical Database Name (-ld) startup parameter *197*
Login names
  SETUSERID function *97*

M

Maintaining schema holders
  recompiling code *184*
Managing server attributes
  Foreign database constraints *255*
MATCHES function *97*
mergeprop utility *34*
Microsoft SQL Server
  data types *56*
Mixed array
  defined *122*
Modifying schema holders *309*
MSSCONFIG Utility *354*
MSSMAN Utility
  DataServer processes on UNIX *192*
MSSSRV environment variable *167*

N

Naming conventions
  keywords in object names *43*
  non-unique table names *43*

Naming conventions *(continued)*
reused OpenEdge keywords *43*
NO-ERROR option
description *108*
NO-JOIN-BY-SQLDB option *151*, *153*, *157*
NO-LOCK option
record locking *78*
NO-WAIT option *83*
Non-unique table names *43*
NSCONFIG *357*
NSMAN *359*
NULL search condition *97*
Null value
in query *74*
ROWID *91*

O

ODBC
Application Programming Interface (API) *24*
compliance *21*
driver
data source *22*
interactive connections *202*
NT *163*
troubleshooting *348*
unstable DLLs *348*
Windows *163*
Driver Manager *22*
options *336*
ODBC connection pooling *140*
OF option
FIND statement *49*
FOR EACH statement *49*
OPEN QUERY statement *97*
OpenEdge
data types *56*
database objects *42*
database security levels *26*
locking behavior *78*
naming conventions *43*
triggers *54*
Unknown value (?) *74*
zero-length character strings *74*
OpenEdge Management or OpenEdge Explorer
configuring the DataServer on NT *164*
definition *34*
starting DataServer on NT *190*
stopping DataServer on UNIX *192*
OpenEdge Management or OpenEdge Explorer tool
definition *169*
OpenEdge sequences
64-bit sequences *56*
OpenEdge-to-MS SQL Server utility
migrating OpenEdge database to SQL Server data source *267*
OpenEdge-to-MS SQL Server utility *(continued)*
parameters *228*, *246*, *268*
running in batch mode *268*
running interactively *268*
ORACLE
password *177*
user ID *177*
OUTPUT option
retrieving parameter values *115*
stored procedures *108*
Overflow checking
defined *86*
potential conflicts *86*

P

PARAM option
retrieving parameter values *115*
stored procedures *108*
Param phrase
defined *102*, *108*
PARAM phrase
description *102*, *108*
Parameter (.pf) files
connection requirements for DataServer *197*
Parameters
abbreviating names *335*
Passing ODBC-specific information
PRGRS_CONNECT option *202*
Password (-P) startup parameter *197*
Passwords
SETUSERID function *97*
Performance
caching records *150*
joins *151*, *153*, *157*
modifying run-time schema check *158*
OpenEdge query tips *145*
query tuning *145*
writing queries *157*
Performance log entry type *210*
Permissions
data source security *26*
establishing *177*
requirements *177*
Physical database name *181*, *238*
Physical Database Name (-db) startup parameter *197*
PRGRS_CONNECT option
DataServer (-Dsrv) startup parameter *341*
PRGRS_IDBUF option
DataServer (-Dsrv) startup parameter *342*
PRGRS_LOCK_ERROR option
DataServer (-Dsrv) startup parameter *342*
PRGRS_MNBUF option
DataServer (-Dsrv) startup option *342*
PRGRS_PREPCACHE option
DataServer (-Dsrv) option *343*

Primary keys *49*
Prime to ROWID
Best index selection *237*
Create RECID Field using *235*
Create RECID For *235*
Create RECID For ROWID Uniqueness *235*
Select 'Best' ROWID INDEX *235*
Try Primary for ROWID *235*
pro executable
UNIX example *194*
PROBRKR command *361*
PROBRKR environment variable *167*
PROC-HANDLE function *117*
PROC-HANDLE phrase
description *102*, *107*
PROC-STATUS function *114*
PROC-STATUS phrase
description *102*, *107*
Procedures
compiling *96*
Processing considerations
32-bit and 64-bit data types *61*
caution in changing INTEGER data types *61*
requiring additional action
bigint data type and pulling a table *61*
data definition changes *61*
r-code references *61*
ProDataSet support *135*
PROGRESS_RECID
and INT64 data type *330*
PROSHUT command *362*

Q

qt_debug
CALL_SUMMARY option
DataServer (-Dsrv) startup parameter *204*
CURSOR option
DataServer (-Dsrv) startup parameter *204*
EXTENDED option
DataServer (-Dsrv) startup parameter *204*
PERFORMANCE option
DataServer (-Dsrv) startup parameter *204*
SQL option
DataServer (-Dsrv) startup parameter *204*
VERBOSE option
DataServer (-Dsrv) startup parameter *204*
qt_debug option
CALL SUMMARY option *369*
CURSOR option *369*
EXTENDED option *369*
PERFORMANCE option *369*
SQL option *369*
VERBOSE option *369*
qt_lookahead option
DataServer (-Dsrv) startup parameter *204*
qt_no_debug option
DataServer (-Dsrv) startup parameter *204*
qt_no_lookahead option
DataServer (-Dsrv) startup parameter *204*
qt_no_separate_connection option
DataServer (-Dsrv) startup parameter *204*
managing data sources *223*
qt_separate_connection option
DataServer (-Dsrv) startup parameter *204*
managing data sources *223*
Query tuning
precedence of methods *204*
startup parameters *204*
Query-tuning options
ARRAY-MESSAGE *145*
CACHE-SIZE *145*
DEBUG SQL *145*
DEBUG-EXTENDED *145*
JOIN-BY-SQLDB *145*
LOOKAHEAD *145*
NO-ARRAY-MESSAGE *145*
NO-DEBUG *145*
NO-JOIN-BY-SQLDB *145*
NO-LOOKAHEAD *145*
NO-SEPARATE-CONNECTION *145*
SEPARATE-CONNECTION *145*
QUERY-TUNING phrase *145*

R

R-code
size *96*
RDBMS stored procedure basics *102*
Read-only (-RO) startup parameter *196–197*
Read-only schema holders *197*
RECID function
replacing with ROWID *91*
Recompiling code
schema changes *184*
Record caching
lookahead cursors *150*
standard cursors *150*
Record creation *75*, *97*
Record locking
data sources *78*
exclusive locks *79*, *81*
EXCLUSIVE-LOCK option *78*
NO-LOCK option *78*
SHARE-LOCK option *78*
shared locks *79*
Recovery
on data sources *83*
Register
defined *22*
Registering
data sources *163*

RELEASE statement 75
Remote databases
starting a DataServer broker 361
Remote DataServer
starting 190
Requirements
software 37
result 102
Result set
definition 102
Retrieving output parameter values 115
Retrieving result sets using proc-text-buffer 115
Retrieving return codes 114
Roll back
on data sources 83
ROWID characteristics 131
ROWID function
defining supporting index 312
listing objects not supported 181, 238, 315
manually creating support 330
reposition functions, INTEGER data type, and INT64 data type 91
supporting in data source 329
ROWID support
additional ProDataSet support 135
ROWID characteristics 131
using ROWID with RUN STORED-PROCEDURE and LOAD-RESULT-INTO 131
Rows
troubleshooting large rows 342
Run Stored-Procedure statement
details 106
handling errors 127
without LOAD-RESULT-INTO phrase 109
RUN STORED-PROCEDURE statement
complete syntax reference 106
description 102
retrieving parameter values 115
send-sql-statement option
with LOAD-RESULT-INTO option 112
without LOAD-RESULT-INTO option 111
using the send-sql statement option 111
with LOAD-RESULT-INTO phrase 110
with LOAD-RESULT-INTO phrase syntax 110
without LOAD-RESULT-INTO phrase syntax 109

S

-S shutdown parameter
PROSHUT command 362
SAVE CACHE COMPLETE statement 197, 221, 369
Schema
defined 25
loading process 25
Schema caching
defined 25
Schema holder
before you begin 177
defined 25
Schema holders
changing logical name 251
connecting 196
connection failure 222
connection permissions 179
creating 177
deleting 252
maintaining 184, 228
modifying 309
setting code page 181, 238
system administration permissions 233
updating 185, 244
verifying 246
where best to build 177
Schema images
security 177
Scrolling
backward 87
forward 87
Security
guideline considerations 25
send-sql-statement option
RUN-STORED-PROCEDURE statement 111
Sequence generator
defined 54
Server Join (-nojoinbysqldb) startup parameter 145, 151, 153, 157, 204
Service Name (-S) shutdown parameter
PROSHUT 362
Service Name (-S) startup parameter 197
SETUSERID function 97
SHARE-LOCK option
record locking 78
Shared locks 79
Shutdown commands 362
Simulated indexes
for non-indexed data sources 50
Single-user Mode (-1) startup parameter 197
Skip schema verification 159
smalldatetime data type
on Microsoft SQL Server 56, 365
Software requirements 37
SQL log entry type 210
SQL ORDER BY option
OpenEdge USE-INDEX option 49
SQL Server
Administrator 22
creating schema holders 181, 238
SQL statement cache
troubleshooting 343
SQL*DBA 177
Starting a DataServer broker 361

Startup parameters
Database Type (-dt) 197
DataServer (-Dsrv) 197, 204
Host Name (-H) 197
Internal Code Page (-cpinternal) 44
LocalCache (-cache) 197
Logical Database Name (-ld) 197
Password (-P) 197
Physical Database Name (-db) 197
Read-only (-RO) 196–197
Server Join (-nojoinbysqldb) 197, 204
Service Name (-S) 197
Single-user Mode (-1) 197
unsupported 202
User ID (-U) 197
Statement level lock upgrade for Join Query 153, 157
Static temp-table 107
Stored procedure
defined 101
implementing and running points 102
language elements 102
Stored procedures
and dataserver performance improvement 101
basics 102
browsing results 50
data types 102
defining buffers 117
INPUT option 108
INPUT-OUTPUT option 108
joining database results 117
OUTPUT option 108
PARAM option 108
returning status values 109
unique indexes for results 50
Stream Code Page (-cpstream) startup parameter 268

T

Table properties
modifying 309
ROWID 312
Table validation 309
Temp-table
and INT64 data type 122
dynamic 107
loading results into using the LOAD-RESULT-INTO phrase 113
static 107
that support result sets 107
Terminology
differences between data sources 42
Trans log entry type 210
Transaction scoping
OpenEdge 83
Transactions 83
Triggers
defined 54
Troubleshooting 335

U

ubroker.properties file
hierarchy 354, 357
Unicode
support for 47
Unified Broker Framework
and the mergeprop utility 34
configuring DataServer applications 32
using the mergeprop utility 186
using the tool 186
Using the Unified Broker Framework 32
Unique indexes
stored procedures results 50
Unknown value (?)
in query 74
ROWID 91
Unsupported ABL statements 97
Updating schema holders 244
USE-INDEX option
FOR EACH statement 49
User ID
SETUSERID function 97
startup parameter (-U) 197
User IDs 177
User-defined data types 73
USING phrase
joins 151, 153, 157

V

VALIDATE statement 75
Validation
expressions 309–310
messages 309
Verifying schema holders 246
View
defined 53
Views
aggregates 53
browsing 53

W

Word index
CONTAINS operator 97

Z

Zero-length character strings 74