

# PSQL v13

---

## *PSQL Data Provider for .NET*

**Guide to Developing Applications with Data Provider for .NET**



## *disclaimer*

ACTION CORPORATION LICENSES THE SOFTWARE AND DOCUMENTATION PRODUCT TO YOU OR YOUR COMPANY SOLELY ON AN “AS IS” BASIS AND SOLELY IN ACCORDANCE WITH THE TERMS AND CONDITIONS OF THE ACCOMPANYING LICENSE AGREEMENT. ACTION CORPORATION MAKES NO OTHER WARRANTIES WHATSOEVER, EITHER EXPRESS OR IMPLIED, REGARDING THE SOFTWARE OR THE CONTENT OF THE DOCUMENTATION; ACTION CORPORATION HEREBY EXPRESSLY STATES AND YOU OR YOUR COMPANY ACKNOWLEDGES THAT ACTION CORPORATION DOES NOT MAKE ANY WARRANTIES, INCLUDING, FOR EXAMPLE, WITH RESPECT TO MERCHANTABILITY, TITLE, OR FITNESS FOR ANY PARTICULAR PURPOSE OR ARISING FROM COURSE OF DEALING OR USAGE OF TRADE, AMONG OTHERS.

## *trademarks*

Btrieve, Client/Server in a Box, and Pervasive are registered trademarks of Action Corporation. Built on Pervasive Software, DataExchange, MicroKernel Database Engine, MicroKernel Database Architecture, Pervasive.SQL, Pervasive PSQL, Solution Network, Ultralight, and ZDBA are trademarks of Action Corporation.

Apple, Macintosh, Mac, and OS X are registered trademarks of Apple Inc.

Microsoft, MS-DOS, Windows, Windows 95, Windows 98, Windows NT, Windows Millennium, Windows 2000, Windows 2003, Windows 2008, Windows 7, Windows 8, Windows 10, Windows Server 2003, Windows Server 2008, Windows Server 2012, Windows XP, Win32, Win32s, and Visual Basic are registered trademarks of Microsoft Corporation.

NetWare and Novell are registered trademarks of Novell, Inc. NetWare Loadable Module, NLM, Novell DOS, Transaction Tracking System, and TTS are trademarks of Novell, Inc.

Oracle, Java, all trademarks and logos that contain Oracle, or Java, are trademarks or registered trademarks of Oracle Corporation.

Progress and DataDirect are registered trademarks of Progress Software Corporation.

All other company and product names are the trademarks or registered trademarks of their respective companies.

© Copyright 2018 Action Corporation. All rights reserved. Reproduction, photocopying, or transmittal of this publication, or portions of this publication, is prohibited without the express prior written consent of the publisher.

This product includes software developed by Powerdog Industries. © Copyright 1994 Powerdog Industries. All rights reserved. This product includes software developed by KeyWorks Software. © Copyright 2002 KeyWorks Software. All rights reserved. This product includes software developed by DUNDAS SOFTWARE. © Copyright 1997-2000 DUNDAS SOFTWARE LTD., all rights reserved. This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

This product uses the free unixODBC Driver Manager as written by Peter Harvey ([pharvey@codebydesign.com](mailto:pharvey@codebydesign.com)), modified and extended by Nick Gorham ([nick@easysoft.com](mailto:nick@easysoft.com)), with local modifications from Action Corporation. Action Corporation will donate their code changes to the current maintainer of the unixODBC Driver Manager project, in accordance with the LGPL license agreement of this project. The unixODBC Driver Manager home page is located at [www.unixodbc.org](http://www.unixodbc.org). For further information on this project, contact its current maintainer: Nick Gorham ([nick@easysoft.com](mailto:nick@easysoft.com)).

A copy of the GNU Lesser General Public License (LGPL) is included on the distribution media for this product. You may also view the LGPL at [www.fsf.org/licenses/lgpl.html](http://www.fsf.org/licenses/lgpl.html).

# Contents

<b>Preface</b>	<b>ix</b>
What Are the PSQL Data Providers?	x
Using This Book.	xi
Typographical Conventions	xii
<b>1 Quick Start</b>	<b>1</b>
Supported .NET Framework Versions	2
Defining Basic Connection Strings	3
Connecting to a Database	4
Example: Using the Provider-Specific Objects	4
Example: Using the Common Programming Model.	6
Example: Using the PSQL Common Assembly.	6
Using the ADO.NET Entity Framework Data Provider	8
<b>2 Using the Data Providers</b>	<b>9</b>
About the Data Providers	10
Using Connection Strings	11
Guidelines.	11
Using the PSQL Performance Tuning Wizard	12
Stored Procedures.	13
Using IP Addresses	14
Transaction Support	15
Using Local Transactions	15
Thread Support	16
Unicode Support	17
Isolation Levels	18
SQL Escape Sequences	19
Event Handling	20
Error Handling	21
Using .NET Objects.	22
Developing Applications for .NET	23
<b>3 Advanced Features</b>	<b>25</b>
Using Connection Pooling.	26
Creating a Connection Pool	26
Adding Connections to a Pool	26
Removing Connections from a Pool.	27
Handling Dead Connection in a Pool	28
Tracking Connection Pool Performance	28
Using Statement Caching	29
Enabling Statement Caching	29
Choosing a Statement Caching Strategy.	29
Using Connection Failover.	30
Using Client Load Balancing.	31
Using Connection Retry	32
Configuring Connection Failover.	33
Setting Security	34
Code Access Permissions	34
Security Attributes	34

Using PSQL Bulk Load . . . . .	35
Use Scenarios for PSQL Bulk Load. . . . .	35
PSQL Common Assembly . . . . .	36
Bulk Load Data File . . . . .	36
Bulk Load Configuration File. . . . .	37
Determining the Bulk Load Protocol . . . . .	37
Character Set Conversions . . . . .	38
External Overflow File. . . . .	38
Bulk Copy Operations and Transactions . . . . .	38
Using Diagnostic Features . . . . .	39
Tracing Method Calls . . . . .	39
PerfMon Support. . . . .	40
Analyzing Performance With Connection Statistics. . . . .	41
Enabling and Retrieving Statistical Items . . . . .	41
<b>4 The ADO.NET Data Provider. . . . .</b>	<b>43</b>
About the PSQL ADO.NET Data Provider . . . . .	44
Namespace . . . . .	44
Assembly Name . . . . .	44
Using Connection Strings with the ADO.NET Data Provider. . . . .	45
Constructing a Connection String . . . . .	45
Performance Considerations. . . . .	46
Connection String Options that Affect Performance . . . . .	46
Properties that Affect Performance . . . . .	47
Data Types. . . . .	48
Mapping PSQL Data Types to .NET Framework Data Types. . . . .	48
Mapping Parameter Data Types . . . . .	50
Data Types Supported with Stream Objects. . . . .	51
Using Streams as Input to Long Data Parameters. . . . .	53
Parameter Markers . . . . .	54
Parameter Arrays . . . . .	55
<b>5 The ADO.NET Entity Framework Data Provider . . . . .</b>	<b>57</b>
About the ADO.NET Entity Framework Data Provider . . . . .	59
Namespace . . . . .	59
Assembly Names. . . . .	59
Configuring Entity Framework 6.1 . . . . .	60
Configuration File Registration . . . . .	60
Code-Based Registration . . . . .	60
Using Multiple Entity Framework Versions Against the Same Database . . . . .	61
Using Connection Strings with the PSQL ADO.NET Entity Framework Data Provider . . . . .	62
Defining Connection String Values in Server Explorer . . . . .	62
Changes in Default Values for Connection String Options . . . . .	62
Code First and Model First Support . . . . .	63
Handling Long Identifier Names. . . . .	63
Using Code First Migrations with the ADO.NET Entity Framework. . . . .	64
Using Enumerations with the ADO.NET Entity Framework . . . . .	65
Mapping Data Types and Functions . . . . .	66
Type Mapping for Database First . . . . .	66
Type Mapping for Model First . . . . .	67
Type Mapping for Code First. . . . .	69
Mapping EDM Canonical Functions to PSQL Functions . . . . .	70
Extending Entity Framework Functionality . . . . .	74

Enhancing Entity Framework Performance . . . . .	75
Limiting the Size of XML Schema Files . . . . .	75
Using Stored Procedures with the ADO.NET Entity Framework . . . . .	76
Providing Functionality . . . . .	76
Using Overloaded Stored Procedures . . . . .	77
Using .NET Objects. . . . .	78
Creating a Model. . . . .	79
For More Information . . . . .	84
<b>6 Using the PSQL Data Providers in Visual Studio . . . . .</b>	<b>85</b>
Adding Connections . . . . .	86
Adding Connections in Server Explorer. . . . .	86
Adding Connections with the Data Source Configuration Wizard. . . . .	94
Using the PSQL Performance Tuning Wizard. . . . .	96
Using Provider-Specific Templates . . . . .	98
Creating a New Project . . . . .	98
Adding a Template to an Existing Project. . . . .	99
Using the PSQL Visual Studio Wizards . . . . .	100
Creating Tables With the Add Table Wizard . . . . .	100
Creating Views With the Add View Wizard . . . . .	104
Adding Components from the Toolbox . . . . .	107
Data Provider Integration Scenario . . . . .	108
<b>7 Using the Microsoft Enterprise Libraries . . . . .</b>	<b>111</b>
<i>Using the Data Access Application Blocks in Applications</i>	
Data Access Application Blocks . . . . .	112
When Should You Use the DAAB? . . . . .	112
Should You Use Generic or Database-specific Classes? . . . . .	112
Configuring the DAAB . . . . .	112
Using the DAAB in Application Code. . . . .	114
Logging Application Blocks . . . . .	115
When Should You Use the LAB? . . . . .	115
Configuring the LAB . . . . .	115
Adding a New Logging Application Block Entry. . . . .	116
Using the LAB in Application Code . . . . .	116
Additional Resources . . . . .	118
<b>A .NET Objects Supported . . . . .</b>	<b>119</b>
<i>Using the .NET Objects</i>	
.NET Base Classes. . . . .	120
Data Provider-Specific Classes. . . . .	121
PsqlBulkCopy. . . . .	121
PsqlBulkCopyColumnMapping . . . . .	121
PsqlBulkCopyColumnMappingCollection . . . . .	121
PsqlCommand Object . . . . .	122
PsqlCommandBuilder Object . . . . .	124
PsqlConnection Object . . . . .	125
PsqlConnectionStringBuilder Object . . . . .	127
PsqlCredential Object . . . . .	135
PsqlDataAdapter Object. . . . .	136
PsqlDataReader Object . . . . .	137
PsqlError Object . . . . .	138

PsqlErrorCollection Object . . . . .	138
PsqlException Object . . . . .	138
PsqlFactory Object . . . . .	139
PsqlInfoMessageEventArgs Object . . . . .	140
PsqlParameter Object . . . . .	140
PsqlParameterCollection Object . . . . .	141
PsqlTrace Object . . . . .	142
PsqlTransaction Object . . . . .	142
PSQL Common Assembly Classes . . . . .	144
CsvDataReader . . . . .	144
CsvDataWriter . . . . .	145
DbBulkCopy . . . . .	146
DbBulkCopyColumnMapping . . . . .	146
DbBulkCopyColumnMappingCollection . . . . .	146
<b>B Getting Schema Information . . . . .</b>	<b>149</b>
<i>Finding and Returning Metadata for a Database</i>	
Columns Returned by the GetSchemaTable Method . . . . .	150
Retrieving Schema Metadata with the GetSchema Method . . . . .	152
MetaDataCollections Schema Collections. . . . .	152
DataSourceInformation Schema Collection . . . . .	152
DataTypes Collection . . . . .	153
ReservedWords Collection . . . . .	155
Restrictions Collection . . . . .	155
Additional Schema Collections . . . . .	157
Columns Schema Collection . . . . .	157
ForeignKeys Schema Collection . . . . .	158
Indexes Schema Collection . . . . .	159
PrimaryKeys Schema Collection . . . . .	160
ProcedureParameters Schema Collection . . . . .	161
Procedures Schema Collection . . . . .	162
TablePrivileges Schema Collection. . . . .	163
Tables Schema Collection . . . . .	164
Views Schema Collection . . . . .	164
<b>C SQL Escape Sequences for .NET . . . . .</b>	<b>167</b>
Date, Time, and Timestamp Escape Sequences . . . . .	168
Scalar Functions . . . . .	169
Outer Join Escape Sequences . . . . .	170
<b>D Locking and Isolation Levels . . . . .</b>	<b>171</b>
Locking . . . . .	172
Isolation Levels . . . . .	173
Locking Modes and Levels . . . . .	175
<b>E Designing .NET Applications for Performance Optimization . . . . .</b>	<b>177</b>
Retrieving Data . . . . .	178
Understanding the Architecture . . . . .	178
Retrieving Long Data . . . . .	178
Reducing the Size of Data Retrieved . . . . .	178
Using CommandBuilder Objects. . . . .	178
Choosing the Right Data Type . . . . .	179

Selecting .NET Objects and Methods . . . . .	180
Using Parameter Markers as Arguments to Stored Procedures. . . . .	180
Designing .NET Applications . . . . .	181
Managing Connections . . . . .	181
Opening and Closing Connections . . . . .	181
Using Statement Caching . . . . .	182
Using Commands Multiple Times . . . . .	182
Using Native Managed Providers . . . . .	182
Updating Data. . . . .	183
Using the Disconnected DataSet . . . . .	183
Synchronizing Changes Back to the Data Source. . . . .	183
<b>F Using an .edmx File . . . . .</b>	<b>185</b>
Code Examples . . . . .	186
<b>G Bulk Load Configuration Files . . . . .</b>	<b>191</b>
Sample Bulk Data Configuration File. . . . .	192
XML Schema Definition for a Bulk Data Configuration File. . . . .	193
<b>H IANA Code Page Mappings . . . . .</b>	<b>195</b>
<b>I Glossary . . . . .</b>	<b>197</b>
.NET Architecture . . . . .	197
ADO.NET. . . . .	197
ADO.NET Entity Framework . . . . .	197
assembly. . . . .	197
bulk load . . . . .	198
client load balancing. . . . .	198
code access security (CAS) . . . . .	198
common language runtime (CLR) . . . . .	198
connection failover. . . . .	198
connection pooling . . . . .	198
connection retry . . . . .	198
Data Access Application Block (DAAB). . . . .	198
destination table . . . . .	198
entity. . . . .	198
global assembly cache (GAC). . . . .	198
isolation level . . . . .	199
load balancing . . . . .	199
locking level. . . . .	199
Logging Application Block (LAB) . . . . .	199
managed code . . . . .	199
namespace . . . . .	199
Performance Monitor . . . . .	199
stream . . . . .	199
schema collection . . . . .	200
strong name. . . . .	200
unmanaged code . . . . .	200





# *Preface*

---

This guide covers the PSQL ADO.NET data provider and the ADO.NET Entity Framework data provider.

---

## What Are the PSQL Data Providers?

The PSQL ADO.NET data provider and the PSQL ADO.NET Entity Framework data provider are managed data providers, built with 100% managed code. The data providers are native wire protocol providers, which means that the data provider does not have to call out to unmanaged code – code outside of the .NET Framework – in the form of a database client, unless your application enlists in Microsoft Distributed Transaction Coordinator (MS DTC) coordinated transactions.

The PSQL ADO.NET data provider enables you to connect to PSQL. It works with both 32-bit and 64-bit .NET and is supported on all PSQL supported Windows platforms.

See also [Supported .NET Framework Versions](#).

---

## Using This Book

This book assumes that you are familiar with your operating system and its commands, the definition of directories, and accessing a database through an end-user application.

This book contains the following information:

- [Quick Start](#) provides information about connecting to a database with your .NET data provider.
- [Using the Data Providers](#) provides information about using .NET applications with the PSQL data provider and provides information about developing .NET applications in the .NET environment.
- [Advanced Features](#) describes advanced features of the data providers, including connection pooling, statement caching, configuring security, and using PSQL Bulk Load.
- [The ADO.NET Data Provider](#) describes connection string options, data types, and other information for the PSQL Entity Framework data provider.
- [The ADO.NET Entity Framework Data Provider](#) describes features of the ADO.NET Entity Framework data provider. It explains how to create a Entity Data Model for the PSQL ADO.NET Entity Framework data provider.
- [Using the PSQL Data Providers in Visual Studio](#) describes how to use the PSQL data provider and the Performance Wizard from within Visual Studio.
- [Using the Microsoft Enterprise Libraries](#) describes how to configure the Data Access Application Block and Logging Application Block, and use them in your application code.
- [.NET Objects Supported](#) provides the .NET public objects, properties, and methods supported by the PSQL data provider.
- [Getting Schema Information](#) describes the schema collections supported by the PSQL data provider.
- [SQL Escape Sequences for .NET](#) describes the scalar functions supported for the PSQL data provider. Your data store may not support all of these functions.
- [Locking and Isolation Levels](#) discusses locking and isolation levels and how their settings can affect the data you retrieve.
- [Designing .NET Applications for Performance Optimization](#) provides recommendations for improving the performance of your applications by optimizing its code.
- [Using an .edmx File](#) explains the necessary changes to an .edmx file in order to provide Extended Entity Framework functionality to the EDM layer.
- [Bulk Load Configuration Files](#) provides samples of the files created during bulk load operations.
- [IANA Code Page Mappings](#) maps the most widely used IBM code pages to IANA code page names.
- [Glossary](#) defines terms used in this guide.

**Note:** This book refers the reader to web URLs for more information about specific topics. Because it is the nature of web content to change frequently, we can guarantee only that the URLs referenced in this book were correct at the time of publishing.

---

## Typographical Conventions

The documentation uses the following typographical conventions.

Convention	Explanation
<b>bold</b>	Bold typeface usually indicates elements of a graphical user interface, such as menu names, dialog box names, commands, options, buttons, and so forth. Bold typeface is also applied occasionally in a standard typographical use for emphasis.
<i>italics</i>	Italics indicate a variable that must be replaced with an appropriate value. For example, <i>user_name</i> would be replaced with an actual user name. Italics is also applied occasionally in a standard typographical use for emphasis, such as for a book title.
cAsE	Uppercase text is used typically to improve readability of code syntax, such as SQL syntax, or examples of code. Case is significant for some operating systems. For such instances, the subject content mentions whether literal text must be uppercase or lowercase.
monospace	Monospace text is used typically to improve readability of syntax examples and code examples, to indicate results returned from code execution, or for text displayed on a command line. The text may appear uppercase or lowercase, depending on context.
' , " , and “ ”	Straight quotes, both single and double, are used in code and syntax examples to indicate when a single or double quote is required. Curly double quotes are applied in the standard typographical use for quotation marks.
	The vertical rule indicates an OR separator to delineate items for which you must choose one item or another. See explanation for angle brackets below.
[ ]	Square brackets indicate optional items. Code syntax not enclosed by brackets is required syntax.
< >	Angle brackets indicate that you must select one item within the brackets. For example, <yes   no> means you must specify either “yes” or “no.”
. . .	Ellipsis indicates that the preceding item can be repeated any number of times in succession. For example, [ <i>parameter</i> . . .] indicates that <i>parameter</i> can be repeated. Ellipsis following brackets indicate the entire bracketed content can be repeated.
::=	The symbol ::= means one item is defined in terms of another. For example, a::=b means that item “a” is defined in terms of “b.”
%string%	A variable defined by the Windows operating system. <i>String</i> represents the variable text. The percent signs are literal text.
\$string	An environment variable defined by the Linux operating system. <i>String</i> represents the variable text. The dollar sign is literal text.

# Quick Start

---

*chapter*

*1*

The following basic information enables you to connect to a database using the PSQL ADO.NET data providers immediately after their installation:

- [Supported .NET Framework Versions](#)
- [Defining Basic Connection Strings](#)
- [Connecting to a Database](#)
- [Using the ADO.NET Entity Framework Data Provider](#)

To take full advantage of PSQL ADO.NET data provider features, we recommend that you also read other ADO.NET topics documented [here](#).

## Supported .NET Framework Versions

The following table lists the PSQL data providers supported for use with Microsoft .NET Framework and Microsoft Entity Framework. Each row of the table represents the compatible combinations of versions of these three products.

PSQL Provider	Microsoft .NET Framework	Microsoft Entity Framework
ADO.NET Data Provider 4.2	2.0, 3.0, 3.5, 3.5 SP1, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2	—
ADO.NET Data Provider 4.3	2.0, 3.0, 3.5, 3.5 SP1, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2	—
ADO.NET Entity Framework Provider 4.2	4.5, 4.5.1, 4.5.2, 4.6.1, 4.6.2	5.0
ADO.NET Entity Framework Provider 4.3	4.5, 4.5.1, 4.5.2, 4.6.1, 4.6.2	5.0
ADO.NET Entity Framework Provider 4.2.0.6	4.5, 4.5.1, 4.5.2, 4.6.1, 4.6.2	6.0, 6.0.1, 6.0.2
ADO.NET Entity Framework Provider 4.3.0.6	4.5, 4.5.1, 4.5.2, 4.6.1, 4.6.2	6.0, 6.0.1, 6.0.2, 6.1, 6.1.1, 6.1.2

The PSQL ADO.NET Entity Framework data provider supports three versions of the Microsoft Entity Framework: 5.0, 6.0, and 6.1.

All versions listed apply to both 32- and 64-bit versions of .NET Framework.

If you are using ADO.NET without customization, then code written for earlier versions of the .NET Framework and of the PSQL data provider is compatible with PSQL data provider 4.2 and 4.3.

To use PSQL ADO.NET Entity Framework 4.2 and 4.3, your applications must target .NET Framework 4.5 or later.

Microsoft Entity Framework 6.1, 6.1.1, are 6.1.2 are compatible only with PSQL ADO.NET Entity Framework Provider 4.3.0.6.

## Defining Basic Connection Strings

The data provider uses a connection string to provide information needed to connect to a specific database server. The connection information is defined by connection string options.

The ADO.NET Entity Framework data provider can specify an existing connection in the Entity Framework Wizard, or can define a new connection. The ADO.NET Entity Framework uses information contained in connection strings to connect to the underlying ADO.NET data provider that supports the Entity Framework. The connection strings also contain information about the required model and mapping files. The data provider uses the connection string when accessing model and mapping metadata and connecting to the data source.

The connection string options have the form:

```
"option name=value"
```

Each connection string option value pair is separated by a semicolon. For example,

```
"Server DSN=DEMODATA;UID=test;PWD=test;Host=localhost"
```

See Table 27 for detailed information about all of the supported connection string options.

### Notes

- The spaces in the option names are optional.
- All connection string option names are case-insensitive. For example, User ID is the same as user id. However, the values of some options, such as User ID and Password, might be case-sensitive.
- If the connection string does not specify a port number, the data provider uses 1583, the default port number.

Table 1 gives the name and description for each option required for a minimum connection to a PSQL server.

*Table 1 Minimum Connection String Options Required*

Option	Description
Server DSN	Specifies the name of the data source on the server to which you want to connect, for example, DEMODATA.
Host	Specifies the name or the IP address of the PSQL server to which you want to connect. For example, you can specify a server name such as Accountingserver or an IP address such as 199.226.22.34 (IPv4) or 1234:5678:0000:0000:0000:0000:9abc:def0 (IPv6).  The initial default value is localhost.

## Connecting to a Database

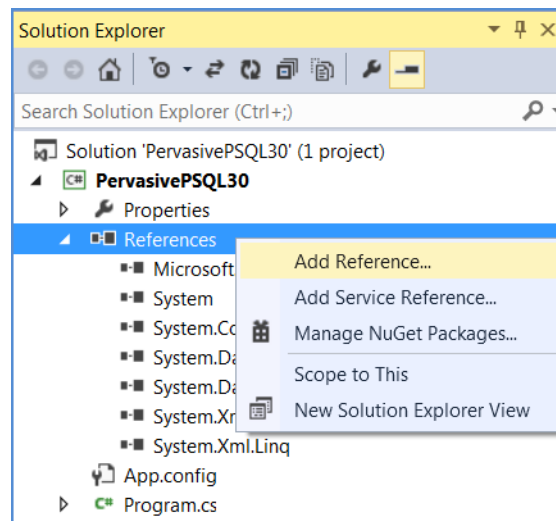
Once your data provider is installed, you can connect from your application to your database with a connection string. See Table 27 for a list of the connection string options.

**Note:** If your application uses the ADO.NET Entity Framework, you can use the Entity Data Model Wizard to create a new connection or use an existing connection. See [Creating a Model](#) for more information.

### *Example: Using the Provider-Specific Objects*

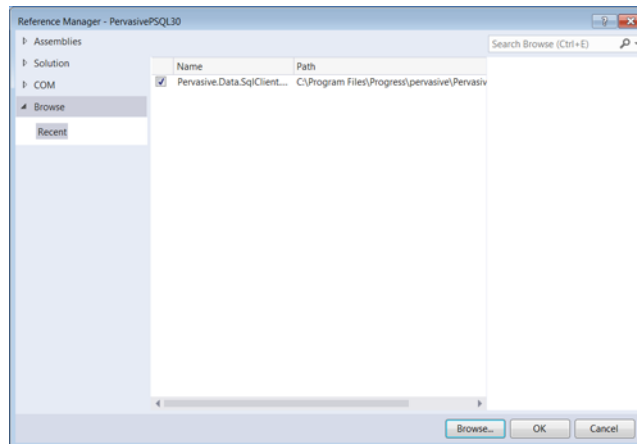
The following example uses the provider-specific objects to connect to a database using the ADO.NET data provider from an application developed in Visual Studio using C#.

- 1 In the Solution Explorer, right-click **References** and then select **Add Reference**.

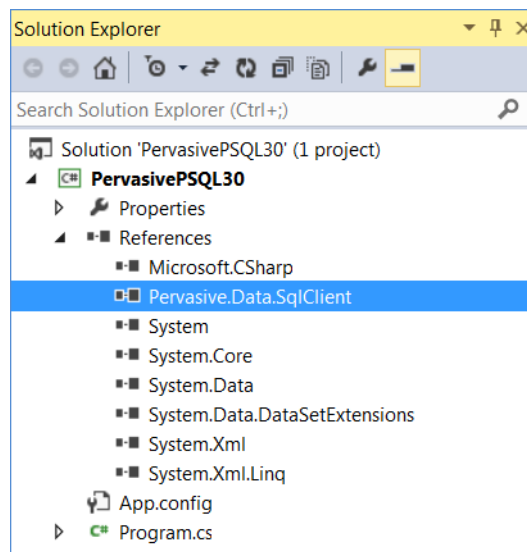


- 2 In the **Reference Manager** wizard, click the **Browse** button and navigate to the folder that contains the PSQL data provider assembly.
- 3 Select **Pervasive.Data.SqlClient.dll** and click **Add**. The **Browse** tab of the Reference Manager wizard lists the PSQL data provider assembly in the **Recent** items.





- 4 Select it and click **OK**. The Solution Explorer now includes the PSQL data provider.



- 5 Add the PSQL data provider's namespace to the beginning of your application, as shown in the following C# code fragment:

```
// Access PSQL
using System.Data;
using System.Data.Common;
using Pervasive.Data.SqlClient;
```

- 6 Add exception handling code and the connection information for your server:

```
PsqlConnection DBConn = new PsqlConnection("Server DSN=DEMODATA;Host=localhost");
try
{
    // Open the connection
    DBConn.Open();
    Console.WriteLine("Connection Successful!")
}
```

```

    }
    catch (PsqlException ex)
    {
        // Connection failed
        writer.WriteLine(ex.Message);
    }

```

## 7 Close the connection.

```

// Close the connection
DBConn.Close();

```

## **Example: Using the Common Programming Model**

The following example illustrates connecting to a PSQL database from an application developed in Visual Studio using C# and the Common Programming Model.

### 1 Check the beginning of your application. Ensure that the ADO.NET namespaces are present.

```

// Access PSQL using factory
using System.Data;
using System.Data.Common;

```

### 2 Add the connection information of your server and exception handling code and close the connection.

```

DbProviderFactory
factory=DbProviderFactories("Pervasive.Data.SqlClient");
DbConnection Conn = factory.createConnection();
Conn.CommandText = "Server DSN=DEMODATA;Host=localhost;";
try
{
    Conn.Open();
    Console.WriteLine("Connection successful!");
}
catch (Exception ex)
{
    // Connection failed
    Console.WriteLine(ex.Message);
}
// Close the connection
Conn.Close();

```

## **Example: Using the PSQL Common Assembly**

You can optionally include the PSQL Common Assembly if you want to use features such as PSQL Bulk Load in an application that conforms to the Common Programming Model. See [Using PSQL Bulk Load](#) for information about how to use PSQL Bulk Load with your application.

The following example illustrates how to use the PSQL Common Assembly in an application developed in Visual Studio using C# and the Common Programming Model.

- 1 Check the beginning of your application. Ensure the .NET Framework and PSQL data provider namespaces are present.

```
// Access PSQL using factory
using System.Data;
using System.Data.Common;
using Pervasive.Data.Common;
```

- 2 Add the connection information of your server and exception handling code and close the connection.

```
// This code does a bulk copy operation from one
// PSQL database to another
DbProviderFactory Factory =
    DbProviderFactories.GetFactory("Pervasive.Data.SqlClient");
DbConnection sourceConnection = Factory.CreateConnection();
sourceConnection.ConnectionString = "Host=localhost;Server DSN=DEMODATA;";

sourceConnection.Open();

DbCommand command = sourceConnection.CreateCommand();
command.CommandText = "SELECT * FROM test";
DbDataReader reader = command.ExecuteReader();

DbConnection destinationConnection = Factory.CreateConnection();
destinationConnection.ConnectionString =
    "Host= ntsl2003b;Server DSN=DEMODATA";
destinationConnection.Open();

DbBulkCopy bulkCopy = new DbBulkCopy(destinationConnection);
bulkCopy.DestinationTableName = "test";
try
{
    bulkCopy.WriteToServer(reader);
} //end try
catch (DbException ex)
{
    Console.WriteLine( ex.Message );
} //end catch
finally
{
    reader.Close();
    MessageBox.Show("done");
} //end finally
```

## Using the ADO.NET Entity Framework Data Provider

The Entity Data Model wizard asks questions that help you to define the components in your Entity Data Model (EDM). The wizard then creates a model of your data in Visual Studio, and automatically sets values for the components in the model. See [Using an .edmx File](#) for information about using the wizard to create an EDM.

Alternatively, you can use other tools in Visual Studio to define values and connection strings manually.

*Provider* is an attribute of the Schema element in the storage model file of an EDM. The storage model file is written in the store schema definition language (SSDL).

The Entity Data Model wizard assigns the value when you select the PSQL ADO.NET Entity Framework data provider. If you choose to manually define an Entity Data Model, assign the string `Pervasive.Data.SqlClient` to the *Provider* attribute of the Schema element, as shown in the following example:

```
<Schema Namespace="AdventureWorksModel.Store" Alias="Self"
Provider="Pervasive.Data.SqlClient" ProviderManifestToken="PSQL"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl">
```

# *Using the Data Providers*

---

The PSQL data providers provide data access to any .NET-enabled application or application server. The data providers delivers high-performance point-to-point and *n*-tier access to industry-leading data stores across the Internet and intranets. Because they are optimized for the .NET environment, the data providers allow you to incorporate .NET technology and extend the functionality and performance of your existing system.

See [Advanced Features](#) for information on advanced features such as connection pooling, statement caching, configuring security, PSQL Bulk Load, and diagnostic support.

See [The ADO.NET Data Provider](#) for information about using the ADO.NET data provider in the standard ADO.NET environment.

See [The ADO.NET Entity Framework Data Provider](#) for information about using the data provider with the ADO.NET Entity Framework.

## **About the Data Providers**

The PSQL data providers are compliant with the Microsoft .NET Framework 2.0, 3.0, 3.5, 3.5 SP1, 4.0, 4.5, 4.5.1, 4.5.2, 4.6.1, and 4.6.2. The data providers are built with 100% managed code; they can run and connect to the database entirely within the common language runtime (CLR).

Code that runs in the native operating system, such as client libraries and COM components, is called unmanaged code. You can mix managed and unmanaged code within a single application. However, unmanaged code reaches outside the CLR, which means that it effectively increases complexity, reduces performance, and opens possible security risks.

## Using Connection Strings

You can define the behavior of a connection using a connection string or the properties of the `PsqlConnection` object.

However, values set in the connection string cannot be changed by the connection properties.

The basic format of a connection string includes a series of keyword/value pairs separated by semicolons. The following example shows the keywords and values for a simple connection string for the PSQL data provider:

```
"Server DSN=SERVERDEMO;Host=localhost"
```

### Guidelines

Use the following guidelines when specifying a connection string:

- The spaces in the connection string option names are required.
- All connection string option names are case-insensitive. For example, Password is the same as password. However, the values of options such as User ID and Password may be case-sensitive.
- To include values that contain a semicolon, single quote, or double quotes, enclose the value in double quotes. If the value contains both a semicolon and double quotes, use single quotes to enclose the value.
- You can also use single quotes when the value starts with a double quote. Conversely, double quotes can be used if the value starts with a single quote. If the value contains both single quotes and double quotes, the character used to enclose the value must be doubled every time it occurs within the value.
- To include leading or trailing spaces in the string value, the value must be enclosed in either single quotes or double quotes. Any leading or trailing spaces around integer, Boolean, or enumerated values are ignored, even if enclosed in single or double quotes. However, spaces within a string literal keyword or value are preserved. Single or double quotes can be used within a connection string without using delimiters (for example, Data Source= my'Server or Data Source= my"Server), unless it is the first or last character in the value.
- Special characters can be used in the value of the connection string option. To escape special characters, surround the value in single or double quotes.
- The Equals character (=) can also be repeated within the connection string. For example:  

```
Initialization String=update mytable set coll == 'foo'"
```
- If the connection string contains invalid connection string options, the connection attempt returns an error. For example, an error is returned if you specify a value for Load Balancing when Alternate Servers has not been defined.
- If the connection string contains duplicated connection string options, the data provider uses the connection string option that appears last in the connection string. For example, Connection Timeout appears twice in the following connection string, with different values. The data provider uses the second value and waits 35 seconds before terminating an attempted connection:

```
"Server DSN=SERVERDEMO;Host=localhost;Connection Timeout=15;Min Pool  
Size=50;Connection Timeout=35"
```

See Table 27 for a list of the supported connection string options.

### ***Using the PSQL Performance Tuning Wizard***

You can use the Performance Wizard to select the optimal connection string options for both the ADO.NET data provider or the ADO.NET Entity Framework data provider.

See [Using the PSQL Performance Tuning Wizard](#) for more information.



---

## Stored Procedures

To enable stored procedures in the application, do the following:

- Set the `CommandText` property in the `PsqlCommand` object to the stored procedure name.  
`MyCommand.CommandText = "GetEmpSalary";`
- Set the `CommandType` property in the `PsqlCommand` object to `StoredProcedure`.  
`MyCommand.CommandType = CommandType.StoredProcedure;`
- Specify parameter arguments, if needed. The application should add the parameters to the parameter collection of the `PsqlCommand` object in the order of the arguments to the stored procedure. The application does not need to specify the parameter markers in the `CommandText` property of the `PsqlCommand` object.

To retrieve the return value from a stored procedure, the application should add an extra parameter to the parameter collection for the `PsqlCommand` object. This parameter's `ParameterDirection` property should be set to `ParameterDirection.ReturnValue`. The return value parameter can be anywhere in the parameter collection because it does not correspond to a specific parameter marker in the `Text` property of the `PsqlCommand` object.

If the stored procedure does not produce a return value, parameters bound with the `ParameterDirection` property as `ReturnValue` are ignored.

If the stored procedure returns a `ReturnValue` from the database and the application has not bound a parameter for it, the data provider discards the value.

**Note for ADO.NET Entity Framework Users:** The `PsqlConnection` object includes properties and methods that provide enhanced statistics functionality. The methods and properties are standard in the ADO.NET data provider but are not available at the ADO.NET Entity Framework layer. Instead, the ADO.NET Entity Framework data provider exposes the same functionality through "pseudo" stored procedures. See [Using Stored Procedures with the ADO.NET Entity Framework](#) for more information.

## Using IP Addresses

The data providers support Internet Protocol (IP) addresses in IPv4 and IPv6 formats. If your network supports named servers, the server name specified in the data source can resolve to an IPv4 or an IPv6 address.

The EnableIPv6 connection string option, when set to True, allows a client with IPv6 protocol installed to connect to the server using either an IPv4 address or an IPv6 address. For more information about IPv6 formats, see [IPv6](#) in *Getting Started with PSQL*.

## Transaction Support

The data provider uses only 100% managed code to support the transactions, which are implemented entirely within the .NET Framework.

### ***Using Local Transactions***

Local transactions use the internal transaction manager of the underlying database.

The application creates a `PsqlTransaction` object by calling `BeginTransaction` on the `PsqlConnection` object. Subsequent operations, such as committing or aborting the transaction, are performed on the `PsqlTransaction` object.

## **Thread Support**

The PsqlConnection object is thread-safe. Multiple PsqlCommand objects, each accessed on a separate thread, can simultaneously use a single connection.

Accessing other public and data provider-specific objects simultaneously on separate threads is not thread-safe.

---

## Unicode Support

The data provider supports Unicode as specified in the .NET Framework SDK. Effectively, this means that the data provider uses Unicode UTF-16 encoding to represent characters.

The data provider converts UTF-16 characters to the format used by the database, and returns .NET Framework strings to the application. For example, if a PSQL database code page is in extended ASCII format, the data provider uses extended ASCII to represent characters sent to the database. The data provider then converts the extended ASCII characters returned before sending them back to the application.

For more information about the .NET Framework implementation of Unicode and international characters, refer to the Microsoft .NET Framework SDK documentation.

## **Isolation Levels**

PSQL supports the ReadCommitted and Serializable isolation levels. It supports record-level locking. See [Locking and Isolation Levels](#) for details.

## SQL Escape Sequences

See [SQL Escape Sequences for .NET](#) for information about the SQL escape sequences supported by the PSQl data provider.

## Event Handling

The event handler receives an argument of type `PsqlInfoMessageEventArgs`, which contains data relevant to an event. See [PsqlInfoMessageEventArgs Object](#) for more information.

This event is defined as:

```
public event PsqlInfoMessageEventHandler InfoMessage;
```

Clients that want to process warnings and informational messages sent by the database server should create an `PsqlInfoMessageEventHandler` delegate to listen to this event.

You can use these events to capture failures that can occur when creating packages, stored procedures, or stored functions, which all create commands. If PSQL encounters errors when compiling a command created by a package, stored procedure, or stored function, the object is created, even though it is not valid. An event will be sent, indicating the failure.

The following code fragment defines a delegate that represents the method that handles the `InfoMessage` event of a `PsqlConnection` object:

```
[Serializable]
public delegate void PsqlInfoMessageEventHandler(
    object sender
    PsqlInfoMessageEventArgs e
);
```

where *sender* is the object that generated the event and *e* is an `PsqlInfoMessageEventArgs` object that describes the warning. For more information on Events and Delegates, refer to the .NET Framework SDK documentation.



## **Error Handling**

The `PsqlError` object collects information relevant to errors and warnings generated by the PSQL server. See [PsqlError Object](#) for more information.

The `PsqlException` object is created and thrown when the PSQL server returns an error. Exceptions generated by the data provider are returned as standard run time exceptions. See [PsqlException Object](#) for more information.

## **Using .NET Objects**

The data provider supports the .NET public objects, exposing them as sealed objects.

See [.NET Objects Supported](#) for more information.

---

## Developing Applications for .NET

Developers of data consumer applications must be familiar with the Microsoft .NET specification and object-oriented programming techniques.

Microsoft also provides extensive information about ADO.NET on its World Wide Web site, including the following articles:

- *Upgrading to Microsoft .NET: ADO.NET for the ADO Programmer*  
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/upgradingtodotnet.asp>
- *Using .NET Framework Data Providers to Access Data*  
[http://msdn2.microsoft.com/en-us/library/s7ee2dwt\(vs.71\).aspx](http://msdn2.microsoft.com/en-us/library/s7ee2dwt(vs.71).aspx)
- *Generic Coding with the ADO.NET 2.0 Base Classes and Factories*  
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/vsgenerics.asp>
- *Security Policy Best Practices*  
[http://msdn.microsoft.com/en-us/library/sa4se9bc\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/sa4se9bc(v=vs.100).aspx)
- *Writing Serviced Components*  
[http://msdn2.microsoft.com/en-us/library/3x7357ez\(vs.71\).aspx](http://msdn2.microsoft.com/en-us/library/3x7357ez(vs.71).aspx)
- *Creating and Using DataSets*  
[http://msdn.microsoft.com/en-us/library/ss7fbaez\(vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ss7fbaez(vs.110).aspx)
- *XML and the DataSet*  
[http://msdn.microsoft.com/en-us/library/84sxtbxh\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/84sxtbxh(v=vs.110).aspx)



# *Advanced Features*

---

This chapter includes the following sections that describe advanced features of the data provider:

- [Using Connection Pooling](#)
- [Using Statement Caching](#)
- [Using Connection Failover](#)
- [Using Client Load Balancing](#)
- [Using Connection Retry](#)
- [Setting Security](#)
- [Using PSQL Bulk Load](#)
- [Using Diagnostic Features](#)

## Using Connection Pooling

Connection pooling allows you to reuse connections rather than creating a new one every time the data provider needs to establish a connection to the underlying database. The data provider automatically enables connection pooling for your .NET client application.

You can control connection pooling behavior by using connection string options. For example, you can define the number of connection pools, the number of connections in a pool, and the number of seconds before a connection is discarded.

Connection pooling in ADO.NET is not provided by the .NET Framework. It must be implemented in the ADO.NET data provider itself.

### ***Creating a Connection Pool***

Each connection pool is associated with a specific connection string. By default, when the first connection with a unique connection string connects to the database, the connection pool is created. The pool is populated with connections up to the minimum pool size. Additional connections can be added until the pool reaches the maximum pool size.

The pool remains active as long as any connections remain open, either in the pool or used by an application with a reference to a Connection object that has an open connection.

If a new connection is opened and the connection string does not match an existing pool, a new pool must be created. By using the same connection string, you can enhance the performance and scalability of your application.

In the following C# code fragments, three new SqlConnection objects are created, but only two connection pools are required to manage them. Note that the first and second connection strings differ only by the values assigned for User ID and Password, and by the value of the Min Pool Size option.

```
DbProviderFactory Factory =
DbProviderFactories.GetFactory("Pervasive.Data.SqlClient");

DbConnection conn1 = Factory.CreateConnection();

conn1.ConnectionString = "Server DSN=DEMODATA;User ID=test;
Password=test;Host=localhost;"
conn1.Open();
// Pool A is created.

DbConnection conn2 = Factory.CreateConnection();
conn2.ConnectionString = "Server DSN=DEMODATA2;User ID=lucy;
                        Password=quake;Host=localhost;"
conn2.Open();

// Pool B is created because the connection strings differ.
DbConnection conn3 = Factory.CreateConnection();
conn3.ConnectionString = "Server DSN=DEMODATA;User ID=test;
                        Password=test;Host=localhost;"
conn3.Open();
// conn3 goes into Pool A with conn1.
```

### ***Adding Connections to a Pool***

A connection pool is created in the process of creating each unique connection string that an application uses. When a pool is created, it is populated with enough connections to satisfy the minimum pool size

requirement, set by the Min Pool Size connection string option. If an application is using more connections than Min Pool Size, the data provider allocates additional connections to the pool up to the value of the Max Pool Size connection string option, which sets the maximum number of connections in the pool.

When a Connection object is requested by the application calling the Connection.Open(...) method, the connection is obtained from the pool, if a usable connection is available. A usable connection is defined as a connection that is not currently in use by another valid Connection object, has a matching distributed transaction context (if applicable), and has a valid link to the server.

If the maximum pool size has been reached and no usable connection is available, the request is queued in the data provider. The data provider waits for the value of the Connection Timeout connection string option for a usable connection to return to the application. If this time period expires and no connection has become available, then the data provider returns an error to the application.

You can allow the data provider to create more connections than the specified maximum pool size without affecting the number of connections pooled. This may be useful, for example, to handle occasional spikes in connection requests. By setting the Max Pool Size Behavior connection string option to SoftCap, the number of connections created can exceed the value set for Max Pool Size, but the number of connections pooled does not. When the maximum connections for the pool are in use, the data provider creates a new connection. If a connection is returned and the pool contains idle connections, the pooling mechanism selects a connection to be discarded so that the connection pool never exceeds the Max Pool Size. If Max Pool Size Behavior is set to HardCap, the number of connections created does not exceed the value set for Max Pool Size.

**IMPORTANT:** Closing the connection using the Close() or Dispose() method of the SqlConnection object adds or returns the connection to the pool. When the application uses the Close() method, the connection string settings remain as they were before the Open() method was called. If you use the Dispose method to close the connection, the connection string settings are cleared, and the default settings are restored.

## ***Removing Connections from a Pool***

A connection is removed from a connection pool when it either exceeds its lifetime as determined by the Load Balance Timeout connection string option, or when a new connection that has a matching connection string is initiated by the application (SqlConnection.Open() is called).

Before returning a connection from the connection pool to an application, the Pool Manager checks to see if the connection has been closed at the server. If the connection is no longer valid, the Pool Manager discards it, and returns another connection from the pool, if one is available and valid.

You can control the order in which a connection is removed from the connection pool for reuse, based on how frequently or how recently the connection has been used, with the Connection Pool Behavior connection string option. For a balanced use of connections, use the LeastFrequentlyUsed or LeastRecentlyUsed values. Alternatively, for applications that perform better when they use the same connection every time, you can use the MostFrequentlyUsed or MostRecentlyUsed values.

The ClearPool and ClearAllPools methods of the Connection object remove all connections from connection pools. *ClearPool* clears the connection pool associated with a specific connection. In contrast, *ClearAllPools* clears all of the connection pools used by the data provider. Connections that are in use when the method is called are discarded when they are closed.

**Note:** By default, if discarding an invalid connection causes the number of connections to drop below the number specified in the Min Pool Size attribute, a new connection will not be created until an application needs one.

### ***Handling Dead Connection in a Pool***

What happens when an idle connection loses its physical connection to the database? For example, suppose the database server is rebooted or the network experiences a temporary interruption. An application that attempts to connect using an existing Connection object from a pool could receive errors because the physical connection to the database has been lost.

The PSQL ADO.NET Data Provider handles this situation transparently to the user. The application does not receive any errors on the Connection.Open() attempt because the data provider simply returns a connection from a connection pool. The first time the Connection object is used to execute a SQL statement (for example, through the Execute method on the Command object), the data provider detects that the physical connection to the server has been lost and attempts to reconnect to the server *before* executing the SQL statement. If the data provider can reconnect to the server, the result of the SQL execution is returned to the application; no errors are returned to the application. The data provider uses the connection failover options, if enabled, when attempting this seamless reconnection. See [Using Connection Failover](#) for information about configuring the data provider to connect to a backup server when the primary server is not available.

**Note:** Because the data provider can attempt to reconnect to the database server when executing SQL statements, connection errors can be returned to the application when a statement is executed. If the data provider cannot reconnect to the server (for example, because the server is still down), the execution method throws an error indicating that the reconnect attempt failed, along with specifics about the reason the connection failed.

This technique for handling dead connections in connection pools allows for the maximum performance out of the connection pooling mechanism. Some data providers periodically ping the server with a dummy SQL statement while the connections sit idle. Other data providers ping the server when the application requests the use of the connection from the connection pool. Both of these approaches add round trips to the database server and ultimately slow down the application during normal operation of the application is occurring.

### ***Tracking Connection Pool Performance***

The data providers install a set of PerfMon counters that let you tune and debug applications that use the data provider. See [PerfMon Support](#) for information about using the PerfMon counters.



## Using Statement Caching

A statement cache is a group of prepared statements or instances of Command objects that can be reused by an application. Using statement caching can improve application performance because the actions on the prepared statement are performed once even though the statement is reused multiple times over an application's lifetime. You can analyze the effectiveness of the statements in the cache (see [Analyzing Performance With Connection Statistics](#)).

A statement cache is owned by a physical connection. After being executed, a prepared statement is placed in the statement cache and remains there until the connection is closed.

Statement caching can be used across multiple data sources and can be used beneath abstraction technologies such as the Microsoft Enterprise Libraries with the Data Access Application Blocks.

### Enabling Statement Caching

By default, statement caching is not enabled. To enable statement caching for existing applications, set the Statement Cache Mode connection string option to Auto. In this case, all statements are eligible to be placed in the statement cache.

You can also configure statement caching so that only statements that you explicitly mark to be cached are placed in the statement cache. To do this, set the StatementCacheBehavior property of the statement's Command object to Cache and set the Statement Cache Mode connection string option to ExplicitOnly.

Table 2 summarizes the statement caching settings and their effects.

Table 2 Summary of Statement Cache Behavior

Behavior	StatementCacheBehavior	Statement Cache Mode
Explicitly add the statement to the statement cache.	Cache	ExplicitOnly (the default)
Add the statement to the statement cache. If necessary, the statement is removed to make room for a statement marked Cache.	Implicit (the default)	Auto
Specifically exclude the statement from the statement cache.	DoNotCache	Auto or ExplicitOnly

### Choosing a Statement Caching Strategy

Statement caching provides performance gains for applications that reuse prepared statements multiple times over the lifetime of an application. You set the size of the statement cache with the Max Statement Cache Size connection string option. If space in the statement cache is limited, do not cache prepared statements that are used only once.

Caching all of the prepared statements that an application uses might appear to offer the best performance. However, this approach may come at a cost of database memory if you implement statement caching with connection pooling. In this case, each pooled connection has its own statement cache that may contain all of the prepared statements used by the application. All of these pooled prepared statements are also maintained in the database's memory.

## Using Connection Failover

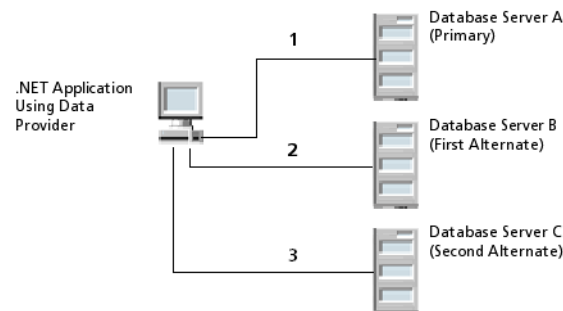
*Connection failover* allows an application to connect to an alternate, or backup, database server if the primary database is unavailable, for example, because of a hardware failure or traffic overload.

Connection failover ensures that the data on which your critical .NET applications depend is always available.

You can customize the data provider for connection failover by configuring a list of alternate databases that are tried if the primary server is not accepting connections. Connection attempts continue until a connection is successfully established or until all of the alternate databases have been tried the specified number of times.

For example, Figure 1 shows an environment with multiple database servers. Database Server A is designated as the primary database server, Database Server B is the first alternate server, and Database Server C is the second alternate server.

Figure 1 Connection Failover



First, the application attempts to connect to the primary database, Database Server A (1). If connection failover is enabled and Database Server A fails to accept the connection, the application attempts to connect to Database Server B (2). If that connection attempt also fails, the application attempts to connect to Database Server C (3).

In this scenario, it is probable that at least one connection attempt would succeed, but if no connection attempt succeeds, the data provider can retry the primary server and each alternate database for a specified number of attempts. You can specify the number of attempts that are made through the *connection retry* feature. You can also specify the number of seconds of delay, if any, between attempts through the *connection delay* feature. For more information about connection retry, see [Using Connection Retry](#).

The data provider fails over to the next alternate server only if it cannot establish communication with the current alternate server. If the data provider successfully establishes communication with a database and the database rejects the connection request because, for example, the login information is invalid, then the data provider generates an exception and does not try to connect to the next database in the list. It is assumed that each alternate server is a mirror of the primary and that all authentication parameters and other related information are the same.

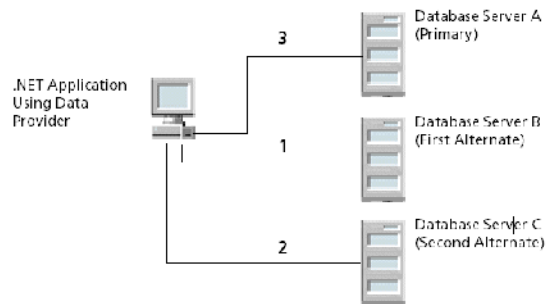
Connection failover provides protection for new connections only and does not preserve states for transactions or queries. For details on configuring connection failover for your data provider, see [Configuring Connection Failover](#).

## Using Client Load Balancing

*Client load balancing* works with connection failover to distribute new connections in your environment so that no one server is overwhelmed with connection requests. When both connection failover and client load balancing are enabled, the order in which primary and alternate databases are tried is random.

For example, suppose that client load balancing is enabled as shown in Figure 2:

Figure 2 Client Load Balancing Example



First, Database Server B is tried (1). Then, Database Server C may be tried (2), followed by a connection attempt to Database Server A (3); subsequent connection attempts use this same sequence. In contrast, if client load balancing was not enabled in this scenario, each database would be tried in sequential order, primary server first, then alternate servers based on their entry order in the alternate servers list.

For details on configuring client and load balancing for your data provider, see [Configuring Connection Failover](#).

## Using Connection Retry

*Connection retry* defines the number of times that the data provider attempts to connect to the primary, and, if configured, alternate database servers after the first unsuccessful connection attempt. Connection retry can be an important strategy for system recovery. For example, suppose you have a power failure scenario in which both the client and the server fail. When the power is restored and all computers are restarted, the client may be ready to attempt a connection before the server has completed its startup routines. If connection retry is enabled, the client application can continue to retry the connection until a connection is successfully accepted by the server.

Connection retry can be used in environments that only have one server or can be used as a complementary feature in connection failover scenarios with multiple servers.

Using connection string options, you can specify the number of times the data provider attempts to connect and the time in seconds between connection attempts. For details on configuring connection retry, see [Configuring Connection Failover](#).

## Configuring Connection Failover

*Connection failover* allows an application to connect to an alternate, or backup, database server if the primary database server is unavailable, for example, because of a hardware failure or traffic overload.

See [Using Connection Failover](#) for more information about connection failover.

To configure connection failover to another server, you must specify a list of alternate database servers that are tried at connection time if the primary server is not accepting connections. To do this, use the Alternate Servers connection string option. Connection attempts continue until a connection is successfully established or until all the databases in the list have been tried once (the default).

Optionally, you can specify the following additional connection failover features:

- The number of times the data provider attempts to connect to the primary and alternate servers after the initial connection attempt. By default, the data provider does not retry. To set this feature, use the Connection Retry Count connection string option.
- The wait interval, in seconds, used between attempts to connect to the primary and alternate servers. The default interval is 3 seconds. To set this feature, use the Connection Retry Delay connection option.
- Whether the data provider will use load balancing in its attempts to connect to primary and alternate servers. If load balancing is enabled, the data provider uses a random pattern instead of a sequential pattern in its attempts to connect. The default value is not to use load balancing. To set this feature, use the Load Balancing connection string option.

You use a connection string to direct the data provider to use connection failover. See [Using Connection Strings](#).

The following C# code fragment includes a connection string that configures the data provider to use connection failover in conjunction with all of its optional features – load balancing, connection retry, and connection retry delay:

```
Conn = new PsqlConnection Conn = new PsqlConnection();
Conn = new PsqlConnection("Host=myServer;User ID=test;Password=secret;
    Server DSN=SERVERDEMO;Alternate Servers="Host=AcctServer, Host=AcctServer2";
    Connection Retry Count=4;Connection Retry Delay=5;Load Balancing=true;
    Connection Timeout=60")
```

Specifically, this connection string configures the data provider to use two alternate servers as connection failover servers, to attempt to connect four additional times if the initial attempt fails, to wait five seconds between attempts, and to try the primary and alternate servers in a random order. Each connection attempt lasts for 60 seconds, and uses the same random order that was established on the first retry.

## Setting Security

The data provider supports Encrypted Network Communications, also known as wire encryption, on connections. By default, the data provider reflects the server's setting. See Table 27 for more information on encryption settings.

The level of encryption allowed by the data provider depends on the encryption module used. With the default encryption module, the data provider supports 40-, 56-, and 128-bit encryption.

Data encryption may adversely affect performance because of the additional overhead, mainly CPU usage, required to encrypt and decrypt data.

In addition to encryption, the PSQL ADO.NET Data Provider implements security through the security permissions defined by the .NET Framework.

### **Code Access Permissions**

The data provider requires the FullTrust permission to be set in order to load and run. This requirement is due to underlying classes in System.Data that demand FullTrust for inheritance. All ADO.NET data providers require these classes to implement a DataAdapter.

### **Security Attributes**

The data provider is marked with the AllowPartiallyTrustedCallers attribute.

## Using PSQL Bulk Load

PSQL Bulk Load offers a one-stop approach for all of your bulk load needs, with a simple, consistent way to do bulk load operations for PSQL and for all of the DataDirect Connect products that support this bulk load feature. This means that you can write your bulk load applications using the standards-based API bulk interfaces, and then, just plug in the database data providers or drivers to do the work for you.

Suppose you need to load data into PSQL, Oracle, DB2, and Sybase. In the past, you probably had to use a proprietary tool from each database vendor for bulk load operations, or write your own tool. Now, because of the interoperability built into PSQL Bulk Load, your task is much easier. Another advantage is that PSQL Bulk Load uses 100% managed code, and requires no underlying utilities or libraries from other vendors.

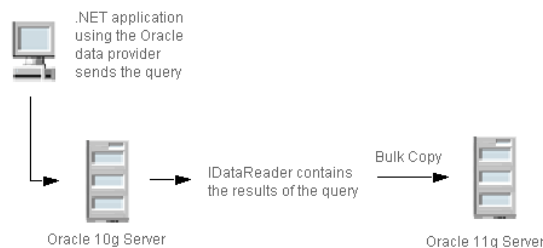
Bulk load operations between dissimilar data stores are accomplished by persisting the results of the query in a comma-separated value (CSV) format file, a bulk load data file. The file can be used between the PSQL ADO.NET Data Provider and any DataDirect Connect for ADO.NET data providers that support bulk load. In addition, the bulk load data file can be used with any DataDirect Connect product driver or data provider that supports the Bulk load functionality. For example, the CSV file generated by the PSQL data provider can be used by a DataDirect Connect for ODBC driver that supports bulk load.

### Use Scenarios for PSQL Bulk Load

You can use PSQL Bulk Load with the PSQL ADO.NET Data Provider in two ways:

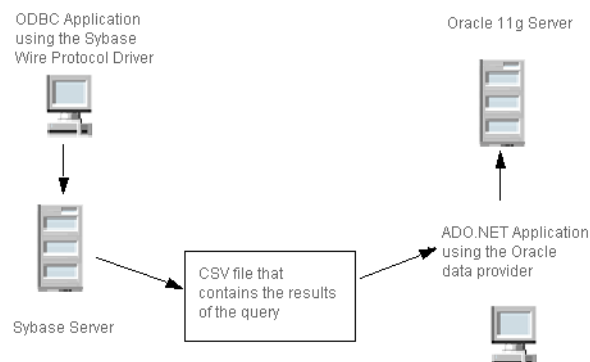
- Upgrade to a new PSQL version and copy data from the old PSQL data source to the new one, as shown in Figure 3.

Figure 3 Using PSQL Bulk Load Between Two Data Sources



- Export data from a database and migrate the results to a PSQL database. Figure 4 shows an ODBC environment copying data to an ADO.NET database server.

Figure 4 Copying Data from ODBC to ADO.NET



In this figure, the ODBC application includes code to export data to the CSV file, and the ADO.NET application includes code to specify and open the CSV file. Because the PSQL ADO.NET Data Provider and the DataDirect ODBC drivers use a consistent format, interoperability is supported via these standard interfaces.

## **PSQL Common Assembly**

The PSQL Bulk Load implementation for ADO.NET uses the de facto standard defined by the Microsoft `SqlBulkCopy` classes, and adds powerful built-in features to enhance interoperability as well as the flexibility to make bulk operations more reliable.

The data provider includes provider-specific classes to support PSQL Bulk Load. See [Data Provider-Specific Classes](#) for more information. If you use the Common Programming Model, you can use the classes in the PSQL Common Assembly (see [PSQL Common Assembly Classes](#)).

The `Pervasive.Data.Common` assembly includes classes that support PSQL Bulk Load, such as the `CsvDataReader` and `CsvDataWriter` classes that provide functionality between bulk data formats.

The Common assembly also extends support for bulk load classes that use the Common Programming Model. This means that the `SqlBulkCopy` patterns can now be used in a new `DbBulkCopy` hierarchy.

Future versions of the data provider will include other features that enhance the Common Programming Model experience. See [PSQL Common Assembly Classes](#) for more information on the classes supported by the `Pervasive.Data.Common` assembly.

## **Bulk Load Data File**

The results of queries between dissimilar data stores are persisted in a comma-separated value (CSV) format file, a bulk load data file. The file name, which is defined by the `BulkFile` property, is used for writing and reading the bulk data. If the file name does not contain an extension, the ".csv" extension is assumed.

### **Example**

The PSQL source table `GBMAXTABLE` contains four columns. The following C# code fragment writes the `GBMAXTABLE.csv` and `GBMAXTABLE.xml` files that will be created by the `CsvDataWriter`. Note that this example uses the `DbDataReader` class.

```
cmd.CommandText = "SELECT * FROM GBMAXTABLE ORDER BY INTEGERCOL";
DbDataReader reader = cmd.ExecuteReader();
CsvDataWriter csvWriter = new CsvDataWriter();
csvWriter.WriteToFile(@"\Nc1\net\PSQL\GBMAXTABLE\GBMAXTABLE.csv", reader);
```

The bulk load data file `GBMAXTABLE.csv` contains the results of the query:

```
1,0x6263,"bc","bc"
2,0x636465,"cde","cde"
3,0x64656667,"defg","defg"
4,0x6566676869,"efghi","efghi"
5,0x666768696a6b,"fghijk","fghijk"
6,0x6768696a6b6c6d,"ghijklm","ghijklm"
7,0x68696a6b6c6d6e6f,"hijklmno","hijklmno"
8,0x696a6b6c6d6e6f7071,"ijklmnopq","ijklmnopq"
9,0x6a6b6c6d6e6f70717273,"jklmnopqrs","jklmnopqrs"
10,0x6b,"k","k"
```



The GBMAXTABLE.xml file, which is the bulk load configuration file that provides the format of this bulk load data file, is described in the following section.

### **Bulk Load Configuration File**

A bulk load configuration file is produced when the CsvDataWriter.WriteToFile method is called (see [CsvDataWriter](#) for more information).

The bulk load configuration file defines the names and data types of the columns in the bulk load data file. These names and data types are defined the same way as the table or result set from which the data was exported.

If the bulk data file cannot be created or does not comply with the schema described in the XML configuration file, an exception is thrown. See [XML Schema Definition for a Bulk Data Configuration File](#) for more information about using an XML schema definition.

If a bulk load data file that does not have a configuration file is read, the following defaults are assumed:

- All data is read in as character data. Each value between commas is read as character data.
- The default character set is the character set of the platform on which the Bulk Load CSV file is being read. See [Character Set Conversions](#) for more information.

The bulk load configuration file describes the bulk data file and is supported by an underlying XML Schema defined at:

<http://www.datadirect.com/ns/bulk/BulkData.xsd>.

### **Example**

The format of the bulk load data file shown in the previous section is defined by the bulk load configuration file, GBMAXTABLE.xml. The file describes the data type and other information about each of the four columns in the table.

```
<?xml version="1.0" encoding="utf-8"?>
<table codepage="UTF-16LE"
  xsi:noNamespaceSchemaLocation="http://www.datadirect.com/ns/bulk/BulkData.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <column datatype="DECIMAL" precision="38" scale="0" nullable=
      "false">INTEGERCOL</column>
    <column datatype="VARBINARY" length="10" nullable=
      "true">VARBINCOL</column>
    <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252" nullable="true">VCHARCOL</column>
    <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252" nullable="true">UNIVCHARCOL</column>
  </row>
</table>
```

### **Determining the Bulk Load Protocol**

Bulk operations can be performed using dedicated bulk protocol, that is, the data provider uses the protocol of the underlying database. In some cases, the dedicated bulk protocol is not available, for example, when the data to be loaded is in a data type not supported by the dedicated bulk protocol. Then, the data provider automatically uses a non-bulk method such as array binding to perform the bulk operation, maintaining optimal application uptime.

## Character Set Conversions

At times, you might need to bulk load data between databases that use different character sets.

For the PSQL ADO.NET Data Provider, the default source character data, that is, the output from the `CsvDataReader` and the input to the `CsvDataWriter`, is in Unicode (UTF-16) format. The source character data is always transliterated to the code page of the CSV file. If the threshold is exceeded and data is written to the external overflow file, the source character data is transliterated to the code page specified by the `externalfilecodepage` attribute defined in the bulk configuration XML schema (see [XML Schema Definition for a Bulk Data Configuration File](#)). If the configuration file does not define a value for `externalfilecodepage`, the CSV file code page is used.

To avoid unnecessary transliteration, it's best for the CSV and external file character data to be stored in Unicode (UTF-16). You might want your applications to store the data in another code page in one of the following scenarios:

- The data will be written by ADO.NET and read by ODBC. In this case, the read (and associated transliteration) is done by ODBC. If the character data is already in the correct code page, no transliteration is necessary.
- Space is a consideration. Depending on the code page, the character data could be represented more compactly. For example, ASCII data is a single byte per character, UTF-16 is 2 bytes per character).

The configuration file may optionally define a second code page for each character column. When character data exceeds the value defined by the `CharacterThreshold` property and is stored in a separate file (see [External Overflow File](#)), the value defines the code page for that file.

If the value is omitted or if the code page defined by the source column is unknown, the code page defined for the CSV file will be used.

## External Overflow File

If the value of the `BinaryThreshold` or `CharacterThreshold` property of the `CsvDataWriter` object is exceeded, separate files are generated to store the binary or character data. These overflow files are located in the same directory as the bulk data file.

If the overflow file contains character data, the character set of the file is governed by the character set specified in the CSV bulk configuration file.

The filename contains the CSV filename and a ".lob" extension (for example, `CSV_filename_nnnnnn.lob`). These files exist in the same location as the CSV file. Increments start at `_000001.lob`.

## Bulk Copy Operations and Transactions

By default, bulk copy operations are performed as isolated operations and are not part of a transaction. This means there is no opportunity for rolling the operation back if an error occurs.

PSQL allows bulk copy operations to take place within an existing transaction. You can define the bulk copy operation to be part of a transaction that occurs in multiple steps. Using this approach enables you to perform more than one bulk copy operation within the same transaction, and commit or roll back the entire transaction.

Refer to the Microsoft online help topic "Transaction and Bulk Copy Operations (ADO.NET)" for information about rolling back all or part of the bulk copy operation when an error occurs.

## Using Diagnostic Features

The .NET Framework provides a Trace class that can help end users identify the problem without the program having to be recompiled.

The PSQL ADO.NET Data Provider delivers additional diagnostic capability:

- Ability to trace method calls
- Performance Monitor hooks that let you monitor connection information for your application

### Tracing Method Calls

Tracing capability can be enabled either through environment variables or the PsqlTrace class. The data provider traces the input arguments to all of its public method calls, as well as the outputs and returns from those methods (anything that a user could potentially call). Each call contains trace entries for entering and exiting the method.

During debugging, sensitive data can be read, even if it is stored as a private or internal variable and access is limited to the same assembly. To maintain security, trace logs show passwords as five asterisks (\*\*\*\*\*).

### Using Environment Variables

Using environment variables to enable tracing means that you do not have to modify your application. If you change the value of an environment variable, you must restart the application for the new value to take effect.

Table 3 describes the environment variables that enable and control tracing.

Table 3 Environment Variables

Environment Variable	Description
PVSW_NET_Enable_Trace	If set to 1 or higher, enables tracing. If set to 0 (the default), tracing is disabled.
PVSW_NET_Recreate_Trace	If set to 1, recreates the trace file each time the application restarts. If set to 0 (the default), the trace file is appended.
PVSW_NET_Trace_File	Specifies the path and name of the trace file.

### Notes

- Setting PVSW\_NET\_Enable\_Trace = 1 starts the tracing process. Therefore, you must define the property values for the trace file before enabling the trace. Once the trace processing starts, the values of the other environment variables cannot be changed.
- If tracing is enabled and no trace file is specified by either the connection string option or the environment variable, the data provider saves the results to a file named PVSW\_NETTrace.txt.

### Using Static Methods

Some users may find that using static methods on the data provider's Trace class to be a more convenient way to enable tracing. The following C# code fragment uses static methods on the .NET Trace object to create a PsqlTrace class with a trace file named MyTrace.txt. The values set override the values set in the environmental variables. All subsequent calls to the data provider will be traced to MyTrace.txt.

```

PsqlTrace.TraceFile="C:\\MyTrace.txt";
PsqlTrace.RecreateTrace = 1;
PsqlTrace.EnableTrace = 1;

```

The trace output has the following format:

```

<Correlation#> <Timestamp> <CurrentThreadName>
  <Object Address> <ObjectName.MethodName> ENTER (or EXIT)
    Argument #1 : <Argument#1 Value>
    Argument #2 : <Argument#2 Value>
    ...
    RETURN: <Method ReturnValue> // This line only exists for EXIT

```

where:

*Correlation#* is a unique number that can be used to match up ENTER and EXIT entries for the same method call in an application.

*Value* is the hash code of an object appropriate to the individual function calls.

During debugging, sensitive data can be read, even if it is stored as private or internal variable and access is limited to the same assembly. To maintain security, trace logs show passwords as five asterisks (\*\*\*\*\*).

## PerfMon Support

The Performance Monitor (PerfMon) and VS Performance Monitor (VSPerfMon) utilities allow you to record application parameters and review the results as a report or graph. You can also use Performance Monitor to identify the number and frequency of CLR exceptions in your applications. In addition, you can fine-tune network load by analyzing the number of connections and connection pools being used.

The data provider installs a set of PerfMon counters that let you tune and debug applications that use the data provider. The counters are located in the Performance Monitor under the category name PSQLEXT.ADO.NET data provider.

Table 4 describes the PerfMon counters that you can use to tune connections for your application.

Table 4 PerfMon Counters

Counter	Description
Current # of Connection Pools	Current number of pools associated with the process.
Current # of Pooled and Non-Pooled Connections	Current number of pooled and non-pooled connections.
Current # of Pooled Connections	Current number of connections in all pools associated with the process.
Peak # of Pooled Connections	The highest number of connections in all connection pools since the process started.
Total # of Failed Commands	The total number of command executions that have failed for any reason since the process started.
Total # of Failed Connects	The total number of attempts to open a connection that have failed for any reason since the process started.

For information on using PerfMon and performance counters, refer to the Microsoft documentation library.

## Analyzing Performance With Connection Statistics

The .NET Framework 2.0 and higher supports run-time statistics, which are gathered on a per-connection basis. The PSQL ADO.NET Data Provider supports a wide variety of run-time statistical items. These statistical items provide information that can help you to:

- Automate analysis of application performance
- Identify trends in application performance
- Detect connectivity incidents and send notifications
- Determine priorities for fixing data connectivity problems

Measuring the statistics items affects performance slightly. For best results, consider enabling statistics gathering only when you are analyzing network or performance behavior in your application.

Statistics gathering can be enabled on any Connection object, for as long as it is useful. For example, you can define your application to enable statistics before beginning a complex set of transactions related to performing a business analysis, and disable statistics when the task is complete. You can retrieve the length of time the data provider had to wait for the server and the number of rows that were returned as soon as the task is complete, or wait until a later time. Because the application disables statistics at the end of the task, the statistical items are measured only during the period in which you are interested.

Functionally, the statistical items can be grouped into four categories:

- Network layer items retrieve values associated with network activities, such as the number of bytes and packets that are sent and received and the length of time the data provider waited for replies from the server.
- Aggregate items return a calculated value, such as the number of bytes sent or received per round trip to the server.
- Row disposition statistical items provide information about the time and resources required to dispose of rows not read by the application.
- Statement cache statistical items return values that describe the activity of statements in a statement cache (see [Using Statement Caching](#) for more information on using the statement cache).

## Enabling and Retrieving Statistical Items

When you create a Connection object, you can enable statistics gathering using the `StatisticsEnabled` property. The data provider begins the counts for the statistical items after you open a connection, and continues until the `ResetStatistics` method is called. If the connection is closed and reopened without calling `ResetStatistics`, the count on the statistical items continues from the point when the connection was closed.

Calling the `RetrieveStatistics` method retrieves the count of one or more statistical items. The values returned form a "snapshot in time" at the moment when the `RetrieveStatistics` method was called.

You can define the scope for the statistics gathering and retrieval. In the following C# code fragment, the statistical items measure only the Task A work; they are retrieved after processing the Task B work:

```
connection.StatisticsEnabled = true;
    // do Task A work
connection.StatisticsEnabled = false;
    // do Task B work
IDictionary currentStatistics = connection.RetrieveStatistics();
```

To view all the statistical items, you can use code like the following C# code fragment:

```
foreach (DictionaryEntry entry in currentStatistics) {  
    Console.WriteLine(entry.Key.ToString() + ": " + entry.Value.ToString());  
}  
Console.WriteLine();
```

To view only the SocketReads and SocketWrites statistical items, you can use code like the following C# code fragment:

```
foreach (DictionaryEntry entry in currentStatistics) {  
    Console.WriteLine("SocketReads = {0}",  
        currentStatistics["SocketReads"]);  
    Console.WriteLine("SocketWrites = {0}",  
        currentStatistics["SocketWrites"]);  
}  
Console.WriteLine();
```

**Note for ADO.NET Entity Framework Users:** The PsqlConnection methods and properties for statistics are not available at the ADO.NET Entity Framework layer. Instead, the data provider exposes the same functionality through "pseudo" stored procedures. See [Using Stored Procedures with the ADO.NET Entity Framework](#) for more information.

# *The ADO.NET Data Provider*

---

The PSQL ADO.NET data provider (the ADO.NET data provider) provides data access to any .NET-enabled application or application server. The ADO.NET data provider delivers high-performance point-to-point and  $n$ -tier access to industry-leading data stores across the Internet and intranets. Because it is optimized for the .NET environment, the ADO.NET data provider allows you to incorporate .NET technology and extend the functionality and performance of your existing system.

This chapter describes features that pertain to the PSQL ADO.NET data provider. It contains the following sections:

- [About the PSQL ADO.NET Data Provider](#)
- [Using Connection Strings with the ADO.NET Data Provider](#)
- [Performance Considerations](#)
- [Data Types](#)
- [Parameter Arrays](#)

**Note:** See [The ADO.NET Entity Framework Data Provider](#) for information about using the data provider with the ADO.NET Entity Framework.

## About the PSQL ADO.NET Data Provider

The PSQL ADO.NET data provider is compliant with the Microsoft .NET Framework 2.0, 3.0, 3.5, 3.5 SP1, 4.0, 4.5, 4.5.1, 4.5.2, 4.6.1, and 4.6.2 Preview. The data provider is built with 100% managed code; it can run and connect to the database entirely within the common language runtime (CLR).

Code that runs in the native operating system, such as client libraries and COM components, is called unmanaged code. You can mix managed and unmanaged code within a single application. However, unmanaged code reaches outside the CLR, which means that it effectively raises complexity, reduces performance, and opens possible security risks.

### Namespace

The namespace for the ADO.NET data provider is `Pervasive.Data.SqlClient`. When connecting to the PSQL database, you use the `PsqlConnection` and `PsqlCommand` objects in the `Pervasive.Data.SqlClient` namespace.

The following code fragment shows how to include the ADO.NET data provider's namespace in your applications:

#### C#

```
// Access PSQL
using System.Data;
using System.Data.Common;
using Pervasive.Data.SqlClient;
```

#### Visual Basic

```
' Access PSQL
Imports System.Data
Imports System.Data.Common
Imports Pervasive.Data.SqlClient
```

### Assembly Name

The strongly named assembly for the ADO.NET data provider is placed in the Global Assembly Cache (GAC) during installation. The assembly name is `Pervasive.Data.SqlClient.dll`.

The `Pervasive.Data.Common` assembly includes features such as support for bulk load.



## Using Connection Strings with the ADO.NET Data Provider

You can define the behavior of a connection using a connection string or the properties of the `PsqlConnection` object. However, values set in the connection string cannot be changed by the connection properties.

The basic format of a connection string includes a series of keyword/value pairs separated by semicolons. The following example shows the keywords and values for a simple connection string for the PSQL ADO.NET Data Provider:

```
"Server DSN=SERVERDEMO;Host=localhost"
```

See [Using Connection Strings](#) for guidelines on specifying connection strings.

See Table 27 for a description of the supported connection string options.

### ***Constructing a Connection String***

`PsqlConnectionStringBuilder` property names are the same as the connection string option names. However, the connection string option name can consist of multiple words, with required spaces between the words. For example, the Min Pool Size connection string option is equivalent to the `MinPoolSize` property. Table 27 lists the connection string properties, and describes each property.

The connection string options have the form:

```
option name=value
```

Each connection string option value pair is separated by a semicolon. The following example shows the keywords and values for a simple connection string for the ADO.NET data provider:

```
"Server DSN=SERVERDEMO;Host=localhost"
```

## Performance Considerations

The performance of your application can be affected by the values you set for connection string options and the properties of some data provider objects.

### **Connection String Options that Affect Performance**

**Encrypt:** Data encryption may adversely affect performance because of the additional overhead, mainly CPU usage, required to encrypt and decrypt data.

**Max Statement Cache Size:** Caching all of the prepared statements that an application uses might appear to offer the best performance. However, this approach may come at a cost of database server memory if you implement statement caching with connection pooling. In this case, each pooled connection has its own statement cache, which may contain all of the prepared statements used by the application. All of the cached prepared statements are also maintained in database server memory.

**Pooling:** If you enable the data provider to use connection pooling, you can define additional options that affect performance:

- **Load Balance Timeout:** You can define how long to keep connections in the pool. The pool manager checks a connection's creation time when it is returned to the pool. The creation time is compared to the current time, and if the timespan exceeds the value of the Load Balance Timeout option, the connection is destroyed. The Min Pool Size option can cause some connections to ignore this value.
- **Connection Reset:** Resetting a re-used connection to the initial configuration settings impacts performance negatively because the connection must issue additional commands to the server.
- **Max Pool Size:** Setting the maximum number of connections that the pool can contain too low might cause delays while waiting for a connection to become available. Setting the number too high wastes resources.
- **Min Pool Size:** A connection pool is created when the first connection with a unique connection string connects to the database. The pool is populated with connections up to the minimum pool size, if one has been specified. The connection pool retains this number of connections, even when some connections exceed their Load Balance Timeout value.

**Schema Options:** Returning some types of database metadata can affect performance. To optimize application performance, the data provider prevents the return of performance-expensive database metadata such as procedure definitions or view definitions. If your application needs this database metadata, you can specifically request its return.

To return more than one type of the omitted metadata, specify either a comma-separated list of the names, or the sum of the hexadecimal values of the column collections that you want to return. For example, to return procedure definitions and view definitions, specify one of the following:

- `Schema Option=ShowProcedureDefinitions, ShowViewDefinitions`
- `Schema Options=0x60`

**Statement Cache Mode:** In most cases, enabling statement caching results in improved performance. To enable the caching of prepared statements (Command instances), set this option to Auto. Use this setting if your application has marked prepared statements for implicit inclusion in the statement cache, or has marked some statements for implicit inclusion and others for explicit inclusion. If you want the statement cache to include only prepared statements that are marked Cache, 1) set the `StatementCacheBehavior` property of the Command object to Cache and 2) set this option to ExplicitOnly.

## ***Properties that Affect Performance***

**StatementCacheBehavior:** If your application reuses prepared statements multiple times over an application's lifetime, you can influence performance by using a statement cache. This property identifies how a prepared statement (a Command object instance) is handled during statement caching.

When set to Cache, the prepared statement is included in the statement cache.

When set to Implicit and the Statement Cache Mode connection string option is set to Auto, the prepared statement is included in the statement cache.

When set to DoNotCache, the prepared statement is excluded from the statement cache.

You can use connection statistics to determine the effect that caching specific statements has on performance (see [Analyzing Performance With Connection Statistics](#)).

## Data Types

Table 5 through Table 8 list the data types supported by the PSQL ADO.NET Data Provider.

- Table 5 maps the PSQL data types to the .NET Framework types.
- Table 6 maps the data types the data provider uses if only the System.Data.DbType is specified.
- Table 7 maps the data types the data provider uses to infer a data type if neither the provider-specific type nor the System.Data.DbType are provided.
- Table 8 maps the data types the data provider uses when streams are used as inputs to Long data parameters.

### Mapping PSQL Data Types to .NET Framework Data Types

Table 5 lists the data types supported by the ADO.NET data provider and how they are mapped to the .NET Framework types. You can use the table to infer the data types that will be used when a DataSet is filled using a DataAdapter.

This table also identifies the proper accessors for accessing the data when a DataReader object is used directly.

- The PSQL Data Type column refers to the native type name.
- The PsqlDbType column refers to the PSQL ADO.NET Data Provider's type enumeration. Generally, there is a one to one mapping between the native type and the PsqlDbType. The PSQL NUMBER data type, which can be either a decimal or a double, is an exception to this rule.
- The .NET Framework Type column refers to the base data types available in the Framework.
- The .NET Framework Typed Accessor column refers to the method that must be used to access a column of this type when using a DataReader.

Table 5 Mapping of PSQL Data Types

PSQL Data Type	PsqlDbType	.NET Framework Type	.NET Framework Typed Accessor
BFLOAT4	BFloat4	Single	GetSingle()
BFLOAT8	BFloat8	Double	GetDouble()
BIGIDENTITY	BigInt	Int64	GetDecimal()
BIGINT	BigInt	Int64	GetDecimal()
BINARY	Binary	Byte[]	GetBytes( )
BIT	Bit	Byte[]	GetBytes( )
CHAR	Char	String Char[]	GetString( ) GetChars( )
CURRENCY	Currency	Decimal	GetDecimal( )
DATE	Date	DateTime	GetDateTime()
DATETIME	DateTime1	DateTime	GetDateTime()
DECIMAL	Decimal	Decimal	GetDecimal()

Table 5 Mapping of PSQL Data Types *continued*

PSQL Data Type	PsqldbType	.NET Framework Type	.NET Framework Typed Accessor
DOUBLE	Double	Double	GetDouble()
FLOAT	Float	Double	GetDouble()
IDENTITY	Identity	Int32	GetInt32()
INTEGER	Integer	Int32	GetInt32( )
LONGVARBINARY	LongVarBinary	Byte[]	GetBytes( )
LONGVARCHAR	LongVarChar	Byte[]	GetBytes( )
MONEY	Money	Decimal	GetDecimal( )
NCHAR	NChar	String Char[]	GetString() GetChars()
NLONGVARCHAR	NLongVarChar	String Char[]	GetString() GetChars()
NUMERIC	Decimal	Decimal	GetDecimal( )
NUMERICSA	DecimalSA	Decimal	GetDecimal( )
NUMERICSTS	DecimalSTS	Decimal	GetDecimal( )
NVARCHAR	NVarChar	String Char[]	GetString() GetChars()
REAL	Real	Single	GetSingle( )
SMALLIDENTITY	SmallIdentity	Int16	GetInt16( )
SMALLINT	SmallInt	Int16	GetInt16( )
TIME	Time	Timespan <sup>2</sup>	GetValue()
TIMESTAMP	Timestamp	DateTime	GetDateTime()
TINYINT	TinyInt	SByte	GetByte( )
UBIGINT	UBigInt	UInt64	GetUInt64()
UNIQUE_IDENTIFIER	UniqueIdentifier <sup>1</sup>	String	GetString( )
UINTeger	UInteger	UInt32	GetUInt32( )
USMALLINT	USmallInt	UInt16	GetUInt16( )
UTINYINT	UTinyInt	Byte	GetByte( )
VARCHAR	VarChar	String Char[]	GetString( ) GetChars( )
1 Supported in PSQL 9.5 and higher			
2 Depends on the setting of the timetype connect option.			

## Mapping Parameter Data Types

The type of the parameter is specific to each data provider. The ADO.NET data provider must convert the parameter value to a native format before sending it to the server. The best way for an application to describe a parameter is to use the data provider-specific type enumeration.

In generic programming circumstances, the data provider-specific type may not be available. When no provider-specific DB type has been specified, the data type will be inferred from either the `System.Data.DbType` or from the .NET Framework type of the parameter's value.

The ADO.NET data provider uses the following order when inferring the data type of a parameter:

- The data provider uses the provider-specific data type if it has been specified.
- The data provider infers the data type from the `System.Data.DbType` if it has been specified, but the provider-specific data type has not been specified.
- The data provider infers the data type from the .NET Framework type if neither the provider-specific data type nor the `System.Data.DbType` have been specified.

Table 6 shows how the data provider infers its types if only the `System.Data.DbType` is specified.

Table 6 Mapping `System.Data.DbType` to `PsqlDbType`

<b>System.Data.DbType</b>	<b>PsqlDbType</b>
AnsiString	VarChar
AnsiStringFixedLength	Char
Binary	Binary
Boolean	Integer
Byte	Integer
Currency	Currency
Date	Date
DateTime	DateTime
Decimal	Decimal or Money
Double	Double
Float	Float
GUID	UniquelIdentifier
Int16	SmallInt
Int32	Integer
Int64	BigInt
Sbyte	Integer
Single	BFloat4
String	NVarChar

Table 6 Mapping System.Data.DbTypes to PsqldbTypes continued

System.Data.DbType	PsqldbType
StringFixedLength	NChar
Time	Time
UInt16	USmallInt
UInt32	UInteger
UInt64	UBigInt
VarNumeric	Decimal
1 Supported in PSQL 9.5 and higher.	

Table 7 shows the mapping that the data provider uses to infer a data type if neither the provider-specific data type nor the System.Data.DbType are provided.

Table 7 Mapping .NET Framework Types to PsqldbType

.NET Framework Type	PsqldbType
Boolean	Integer
Byte	Integer
Byte[]	Binary
DateTime	Timestamp
Decimal	Decimal
Double	Double
Int16	SmallInt
Int32	Integer
Int64	BigInt
Single	BFloat4
String	NVarChar VarChar (if PvTranslate=Nothing)
UInt16	USmallInt
UInt32	UInteger
UInt64	UBigInt

### Data Types Supported with Stream Objects

The ADO.NET data provider supports the use of streams as inputs to long data parameters with the data types listed in Table 8.

Table 8 Supported Stream Objects

Provider Data Type	Stream Type Supported
LONGVARBINARY	Stream
LONGVARCHAR	TextReader

See [Using Streams as Input to Long Data Parameters](#) for a discussion of using streams.



---

## Using Streams as Input to Long Data Parameters

Allowing the use of noncontiguous memory to represent a very large binary or text value, such as a video clip or a large document, improves performance, functionality, and scalability.

Stream objects used to read binary data are derived from the `System.IO.Stream` object and use the Framework data type of `byte[]`:

- `System.IO.BufferedStream`
- `System.IO.FileStream`
- `System.IO.MemoryStream`
- `System.Net.Sockets.NetworkStream`
- `System.Security.Cryptography.CryptoStream`

Stream objects used to read text data are derived from the `System.IO.TextReader` object and use the Framework data type of `string`:

- `System.IO.StreamReader`
- `System.IO.StringReader`

To enable the use of streams, you set the `Value` property of the `PsqlParameter` object to a specific instance of the stream (see [PsqlParameter Object](#)). When the command is executed, the data provider reads from the stream to extract the value.

The examples shipped with the data provider include a code example on inserting data into `LONGVARCHAR` and `LONGVARBINARY` columns using randomly generated data. The example also shows how to use streaming objects as inputs to `LONGVARCHAR` and `LONGVARBINARY` columns.

## **Parameter Markers**

Parameter markers, including parameter markers for stored procedures, are specified in the ADO.NET data provider by using the "?" symbol in SQL statements.

```
UPDATE emp SET job = ?, sal = ? WHERE empno = ?
```

Because parameters are not named, the bindings must occur in the order of the parameters in the statement. This means that the calls to the Add() method on the PsqlParameterCollection object (adding the Parameter objects to the collection) must occur in the order of the "?"s in the command text.

## **Parameter Arrays**

Parameter array binding is typically used with INSERT statements to speed up the time needed to fill a table. An application can specify rows of parameter values with a single execution of a command. The values can then be sent to the database server in a single round trip (depending on the native capabilities of the backend database).

The ADO.NET data provider supports input parameter arrays for INSERT and UPDATE statements.



# *The ADO.NET Entity Framework Data Provider*

---

The ADO.NET Entity Framework is an object-relational mapping (ORM) framework for the .NET Framework. Developers can use it to create data access applications by programming against a conceptual application model instead of directly against a relational storage schema. This model allows developers to decrease the amount of code to be written and maintained in data-centric applications.

The PSQL ADO.NET Entity Framework data provider (formerly Pervasive ADO.NET Entity Framework data provider) can be used with applications that use the ADO.NET Entity Framework.

The PSQL ADO.NET Entity Framework data provider is compatible with versions 5.0, 6.0, 6.1., 6.1.1, and 6.1.2 of the Microsoft ADO.NET Entity Framework. It supports the following programming features:

- Applications targeting the following .NET Framework versions:
  - 4.5.x (.NET Framework 4.5 and its point releases, 4.5.1, and 4.5.2)
  - 4.6.x (.NET Framework 4.6 and its point releases, 4.6.1 and 4.6.2)
- Database First, Code First, and Model First workflows
- Enumerated type support in all workflows
- Code First migrations
- "Plain-old" CLR objects (POCO) entities
- DbContext class

The PSQL ADO.NET Entity Framework data provider also supports the following features specific to Microsoft ADO.NET Entity Framework 6.1, 6.1.1, and 6.1.2:

- Multiple DbContext classes
- Code First mapping to Insert, Update, and Delete stored procedures
- Configurable migration history
- Connection resiliency
- Index Attribute for Code First Migrations
- Disable Transactions for Function Imports
- Enum.HasFlag Support
- Allow Migrations commands to use context from reference instead of project
- Interceptors in web/app.config and DatabaseLogger
- Support for identifiers starting with '\_'
- Select concatenated string and numeric property

The ADO.NET Entity Framework data provider uses the ADO.NET data provider to communicate with the ADO.NET database server. This means that the functionality defined by the ADO.NET data provider applies to the ADO.NET Entity Framework data provider unless otherwise noted here. Similarly, any performance configurations made to the ADO.NET data provider are realized by the ADO.NET Entity Framework data provider.

Visual Studio 2012 or later is required when developing applications for the PSQL ADO.NET Entity Framework. If you have configured Microsoft ADO.NET Entity Framework 6.1 (EF 6.1) to use Visual Studio 2012, you must install Entity Framework Tools 6.1.3 for Visual Studio 2012. However, once you install it, all your Visual Studio 2012 applications that previously used Microsoft ADO.NET Entity Framework 5.0 (EF5) must be upgraded to EF 6.1, after which you cannot revert to EF5.

## About the ADO.NET Entity Framework Data Provider

The ADO.NET Entity Framework data provider is built with 100% managed code; it can run and connect to the database entirely within the common language runtime (CLR).

Code that runs in the native operating system, such as client libraries and COM components, is called unmanaged code. You can mix managed and unmanaged code within a single application. However, unmanaged code reaches outside the CLR, which means that it effectively raises complexity, reduces performance, and opens possible security risks.

### Namespace

The namespace for the ADO.NET Entity Framework data provider is `Pervasive.Data.SqlClient.Entity`.

**Note:** The `Pervasive.Data.SqlClient.Entity` namespace is common for Microsoft ADO.NET Entity Framework Versions 5.0 (EF 5) and 6.1 (EF 6.1).

### Assembly Names

The PSQL ADO.NET Entity Framework data provider uses two different assembly versions:

- `Pervasive.Data.SqlClient.Entity.dll` v4.3.0.0 while referring to EF5
- `Pervasive.Data.SqlClient.Entity.dll` v4.3.0.6 while referring to EF6

However, the assembly name is common: `Pervasive.Data.SqlClient.Entity.dll`.

During installation, the assemblies are placed in the Global Assembly Cache (GAC) in two separate subfolders: v4.3.0.0 and v4.3.0.6.

To refer to EF 5, select:

```
%windir%\Microsoft.NET\assembly\GAC_MSIL\Pervasive.Data.SqlClient.Entity\v4.0_4.3.0.0__c84cd5c63851e072
```

To refer to EF 6.1, select:

```
%windir%\Microsoft.NET\assembly\GAC_MSIL\Pervasive.Data.SqlClient.Entity\v4.0_4.3.0.6__c84cd5c63851e072
```

## Configuring Entity Framework 6.1

The PSQL ADO.NET Entity Framework data provider supports the Microsoft ADO.NET Entity Framework versions 5.0 (EF5) and 6.1 (EF 6.1).

To use EF 6.1, you must first register it using one of the following methods:

- [Configuration File Registration](#)
- [Code-Based Registration](#)

**Note:** To register EF 6.1 while testing your applications locally, you can perform a code-based registration during development. However, when you deploy your project, you must perform a configuration file registration.

### Configuration File Registration

#### ►► To configure EF 6.1 by updating the configuration file

- 1 Install the **EntityFramework 6.1.2** NuGet package.

An app.config file is created.

- 2 Remove the **defaultConnectionFactory** registration section from the app.config file and replace it with the following code:

```
<providers>
<provider invariantName="Pervasive.Data.SqlClient"
type="Pervasive.Data.SqlClient.Entity.PsqlProviderServices,
Pervasive.Data.SqlClient.Entity, Version=4.3.0.6, Culture=neutral,
PublicKeyToken=c84cd5c63851e072" />
</providers>
```

The EF 6.1 provider registration is added to Entity Framework section of the app.config file.

### Code-Based Registration

#### ►► To configure EF 6.1 through a code-based registration

- 1 Add the following new *DbConfiguration* class to your test application:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetProviderServices("PsqlProviderServices.ProviderInvariantName, new
PsqlProviderServices());
    }
}
```

- 2 Add the following annotation on top of the DbContext class:

```
[DbConfigurationType(typeof(MyConfiguration))]
```



## ***Using Multiple Entity Framework Versions Against the Same Database***

A single database can use multiple versions of the Microsoft ADO.NET Entity Framework: 5.0 (EF5) and 6.1 (EF 6.1). However, when you switch between EF5 and EF6 applications against the same database, you will receive an error when you try saving to the database.

The error occurs due to the difference between the structure of the "\_\_MigrationHistory" table used by EF5 and EF6.

To use EF5 and EF 6.1 applications against the same database without any errors, run the following command in the database:

```
drop table "__MigrationHistory"
```

## Using Connection Strings with the PSQL ADO.NET Entity Framework Data Provider

The PSQL ADO.NET Entity Framework uses information contained in connection strings to connect to the underlying ADO.NET data provider that supports the Entity Framework. The connection strings also contain information about the required model and mapping files.

The data provider uses the connection string when accessing a model and mapping metadata and connecting to the data source.

You can specify an existing connection in the Entity Framework Wizard, or can define a new connection. Connection string options can be defined directly in a connection string, or set in the Advanced Properties dialog box in Visual Studio (see [Adding Connections in Server Explorer](#)).

### Defining Connection String Values in Server Explorer

See [Adding Connections in Server Explorer](#) for detailed information about using Visual Studio to add and modify connections.

See Table 27 for a description of the supported connection string options.

### Changes in Default Values for Connection String Options

Most default values of the connection string options used by the ADO.NET Entity Framework data provider are the same as those used by the PSQL ADO.NET data provider (see Table 27 for more information). Table 9 lists the connection string options that have a different default value when used with an ADO.NET Entity Framework application.

Table 9 Default Values of Connection String Options Used in an Application

Connection String Option	Default Value in ADO.NET Entity Framework Application
Parameter Mode	Not supported.
Statement Cache Mode	ExplicitOnly is the only supported value.

---

## Code First and Model First Support

Entity Framework 4.1 and later provide support for the Model First and Code First features. Implementing support for these features requires changes to the data provider, such as the way that long identifier names are handled. However, these changes do not require changes to your application.

Code First and Model First implementations require type mapping changes. See [Mapping Data Types and Functions](#) for more information.

### ***Handling Long Identifier Names***

Most PSQl identifiers have a maximum length of 20 bytes. The identifier name can exceed this size because the names of the objects to be created on the server are taken from the class and property names. In addition, constraint names are often created by concatenating several object names. In these cases, the chances of exceeding the maximum identifier length are even greater.

The data provider shortens identifiers to database-allowed maximum identifier length, replacing the end of the identifier with an integer hash-code, for example, the string `ColumnMoreThanTwentyCharacters` is shortened to `ColumnMor_2873286151`. If you access or view the DB object using a DB tool, the names of the created tables may differ from what you might expect based on the Plain Old CLR Object (POCO) class names and property names (Code First), or the entity names and entity property names (Model First).

Note that when two identifiers that have the same leading characters are shortened, the difference between the identifiers is less obvious to a visual inspection. For example, assume that a table has two supporting sequences, `ColumnMoreThanTwentyCharacters` and `ColumnMoreThanTwenty1Characters`. When these sequences are shortened, they are renamed `ColumnMor_2873286151` and `ColumnMor_672399971`.

## Using Code First Migrations with the ADO.NET Entity Framework

Entity Framework 4.3 and later support Code First Migrations, which enables you to update your database schema to reflect POCO classes without having to drop and recreate them.

Migrations enable you to incrementally evolve your database schema as your model changes. Each set of changes to the database is expressed in a code file, known as a migration. The migrations are ordered, typically using a timestamp, and a table in the database keeps track of which migrations are applied to the database.

Code First Migrations implementation requires type mapping changes. See [Mapping Data Types and Functions](#) for more information,

To implement Code First Migrations using Progress DataDirect Connect for PSQL ADO.NET Data Provider, you must perform the following additional settings:

- 1 Add references to the Pervasive.Data.SqlClient.Entity assembly in the project.
- 2 Inherit the Configuration Class changes and register the SQL Generator in the constructor of the Configuration Class. Do the following:
  - Inherit the Configuration Class from PervasiveDbMigrationsConfiguration <TContext>. For example:

```
internal sealed class Configuration:
    PervasiveDbMigrationsConfiguration<%Context Name%>
```
  - Register the Class Generator.

After you enable migrations using Package Manager Console, specify the Connection String either in the app.config or configuration.cs file along with additional settings in the configuration.cs file. However, if Connection String is specified in the app.config file, then ensure that the Connection String and the context have the same name.

If the Connection String is specified in the app.config file, use the following syntax to register SQL Generator in the app.config file:

```
<providers>
  <provider invariantName="Pervasive.Data.SqlClient" type=
"Pervasive.Data.SqlClient.Entity.PsqlProviderServices,
Pervasive.Data.SqlClient.Entity, Version=4.3.0.6, Culture=neutral,
PublicKeyToken=c84cd5c63851e072" />
</providers>
```

To register SQL Generator in configuration.cs, use the following syntax:

```
SetSqlGenerator(PervasiveConnectionInfo.InvariantName, new
    PervasiveEntityMigrationSqlGenerator());
```

---

## Using Enumerations with the ADO.NET Entity Framework

The `enum` keyword is used to declare an enumeration, a distinct type consisting of a set of named constants called the enumerator list. Every enumeration type has an underlying type. By default, every underlying type of the enumeration element is mapped to type `int32`. By default, the first enumerator has the value 0, and the value of each consecutive enumerator is incremented by 1. For example, you would specify a days-of-the-week enum type as:

```
enum Days {MON, TUE, WED, THU, FRI, SAT, SUN};
```

In this enumeration, `MON` would be 0, `TUE` 1, `WED` 2, and so forth. Enumerators can have initializers to override the default values. For example:

```
enum Days {MON=1, TUE, WED, THU, FRI, SAT, SUN};
```

In this enumeration, the sequence is forced to start at 1 instead of 0. The names of an enum type's fields are in uppercase letters., by convention, because they are constants.

Microsoft ADO.NET Entity Framework 5.0 and later support Enumerations. To use the enumeration feature, you must target .NET Framework 4.5 or later. Visual Studio 2012 targets .NET Framework 4.5 by default. Enumerations are supported in all three workflows, namely, Model First, Code First, and Database First.

In Entity Framework, an enumeration can have the following underlying types:

- `Byte`
- `Int16`
- `Int32`
- `Int64`
- `SByte`

By default, the enumeration is of type `Int32`. Another integral numeric type can be specified using a colon.

```
enum Days : byte{MON=1, TUE, WED, THU, FRI, SAT, SUN};
```

The underlying type specifies how much storage is allocated for each enumerator. However, an explicit cast is needed to convert from enum type to an integral type. Enum implementations also support type mapping changes. See [Mapping Data Types and Functions](#) for more information.

As part of Entity Framework, Entity Developer fully supports enum types by providing a new Enum node in its Model Explorer window. You can use the Enum property just like any other scalar property, such as in LINQ queries and updates.

## Mapping Data Types and Functions

Developers can use the ADO.NET Entity Framework to create data access applications by programming against a conceptual application model instead of programming directly against a relational storage schema.

### *Type Mapping for Database First*

In a Database First model, the data provider uses a store-centric type mapping scheme, in which the PSQL (store) type influences the EDM type used when the model is generated.

[Mapping PSQL Types to EDM Types](#) shows PSQL types are mapped to primitive types used in a Database First model. Some PSQL data types can map to several different EDM types; the default values are shown in *italics*.

The columns are defined as follows:

- The PSQL Type column refers to the native type name.
- The Store (SSDL) Type column refers to data types used by the store schema definition language (SSDL) file. The storage metadata schema is a formal description of the database that persists data for an application built on the EDM.
- The PrimitiveTypeKind column refers to the common data primitives used to specify the valid content of properties of entities used in defining EDM applications.

*Table 10 Mapping PSQL Types to EDM Types*

PSQL Type	Store (SSDL) Type	PrimitiveTypeKind
BFLOAT4	BFloat4	Single
BFLOAT8	BFloat8	Double
BIGIDENTITY	Bigint	Int64
BIGINT	Bigint	Int64
BINARY	binary	Byte[]
BIT	Bit	Boolean
CHAR	Char	String
CURRENCY	Currency	Decimal
DATE	Date	DateTime
DECIMAL	Decimal	Decimal
DOUBLE	Double	Double
FLOAT	Float	Float
IDENTITY	Identity	Int32
INTEGER	Integer	Int32
LONGVARBINARY	LongVarBinary	Byte[]

Table 10 Mapping PSQL Types to EDM Types *continued*

PSQL Type	Store (SSDL) Type	PrimitiveTypeKind
LONGVARCHAR	LongVarChar	String
MONEY	Money	Decimal
NCHAR	NChar	String
NLONGVARCHAR	NLongVarChar	String
NUMERIC	Decimal	Decimal
NUMERICSA	DecimalSA	Decimal
NUMERICSTS	DecimalSTS	Decimal
NVARCHAR	NVarChar	String
REAL	Real	Single
SMALLIDENTITY	SmallIdentity	Int16
ROWID	Rowid	Binary
SMALLINT	Smallint	Int16
TIME	Time	Time
DATETIME	DateTime	DateTime
TINYINT	TinyInt	SByte
UBIGINT	UBigInt	UInt64
UNIQUE_IDENTIFIER	Guid	Guid
INTEGER	Integer	UInt32
USMALLINT	USmallInt	UInt16
UTINYINT	UTinyInt	Byte
VARCHAR	Varchar	String

### **Type Mapping for Model First**

[Mapping EDM Types to PSQL Types](#) shows the model-centric type mapping, where the EDM Simple Types influences the PSQL (store) type used when the database is created. The columns are defined as follows:

- The PrimitiveTypeKind column refers to the common data primitives used to specify the valid content of properties of entities used in defining EDM applications.
- The Property Values Affecting Type Mapping identifies any property values that can affect type mapping.
- The Store (SSDL) column refers to data types used by the store schema definition language (SSDL) file. The storage metadata schema is a formal description of the database that persists data for an application built on the EDM.

- The PSQL Type column refers to the native type name.

Table 11 Mapping EDM Types to PSQL Types

PrimitiveTypeKind	Property Values That Affect Type Mapping	Store (SSDL) Type	PSQL Type
Binary	Fixed Length: TRUE	Binary	Binary(n)
	Fixed Length: FALSE	LongVarBinary	LongVarBinary
Boolean		Boolean	Bit
Byte		TinyInt_as_byte	TinyInt
DateTime		DateTime	DateTime
Decimal		Decimal	Decimal
Double		Double	Double
Guid		Guid	Guid
Single		Float	Float
SByte		SmallInt_as_Sbyte	SmallInt
Int16		SmallInt	SmallInt
Int32		Integer	Integer
Int64		BigInt	BigInt



Table 11 Mapping EDM Types to PSQL Types *continued*

PrimitiveTypeKind	Property Values That Affect Type Mapping	Store (SSDL) Type	PSQL Type
String	MaxLength= (1<=n<=8000) Fixed Length=True Unicode=False	Char	Char(n)
	MaxLength= (1<=n<=8000) Fixed Length=False Unicode=False	Varchar	Varchar(n)
	MaxLength= (>8000) Fixed Length=False Unicode=False	LongVarChar	LongVarchar
	MaxLength= (1<=n<=4000) Fixed Length=True Unicode=True	NChar	NChar(n)
	MaxLength= (1<=n<=4000) Fixed Length=False Unicode=True	NVarChar	NVarChar(n)
	MaxLength= (>4000) Fixed Length=False Unicode=True	NLongVarChar	NLongVarChar
Time		Time	Time
DateTimeOffset		DateTime	DateTime

### Type Mapping for Code First

[Mapping CLR Types to PSQL Data Types in a Code First Model](#) shows the model-centric type mapping, where the CLR type influences the PSQL (store) type used when the database is created. Some CLR types can map to several different PSQL types; the default values are shown in italics. ).

The columns are defined as follows:

- The CLR Type column refers to the common language runtime type name.
- The PSQL Type column refers to the native type name.

Table 12 Mapping CLR Types to PSQL Data Types in a Code First Model

CLR Type	PSQL Data Type
Byte	BINARY
Boolean	BIT

Table 12 Mapping CLR Types to PSQL Data Types in a Code First Model *continued*

CLR Type	PSQL Data Type
Byte	TINYINT
DateTime	DATETIME
Decimal	DECIMAL
Double	DOUBLE
Guid	UNIQUEIDENTIFIER
	BINARY
Single	FLOAT
Sbyte	SMALLINT
Int16	SMALLINT
Int32	INTEGER
Int64	BIGINT
String <sup>1</sup>	NCHAR
	NVARCHAR
	NLONGVARCHAR
TimeSpan	TIME
DateTimeOffset	DateTime
<sup>1</sup> In the Code First workflow, if the length of the string field in an entity is not specified, the data provider sets the default length to 2048 and 4096 bytes for unicode and non-unicode types respectively. However, if the length of the string field is set to a maximum allowed limit, that is 4000 bytes for unicode types and 8000 bytes for non-unicode types, the data provider resets it to 2048 bytes and 4096 bytes respectively. For all the other scenarios where the length of the string field is specified, the data provider uses the specified length.	

### Mapping EDM Canonical Functions to PSQL Functions

The ADO.NET Entity Framework translates the Entity Data Model (EDM) canonical functions to the corresponding data source functionality for the ADO.NET Entity Framework Data Provider for PSQL. The function invocations are expressed in a common form across data sources.

Because these canonical functions are independent of data sources, argument and return types of canonical functions are defined in terms of types in the EDM. When an Entity SQL query uses canonical functions, the appropriate function is called at the data source.

Both null-input behavior and error conditions are explicitly specified for all canonical functions. However, the ADO.NET Entity Framework does not enforce this behavior. Further details are available at: <http://msdn.microsoft.com/en-us/library/bb738626.aspx>

### Aggregate Canonical Functions

Table 13 describes the mapping of EDM aggregate canonical functions to PSQL functions.

Table 13 Mapping Aggregate Canonical Functions

Aggregate Canonical Function	PSQL functions
Avg( <i>expression</i> )	avg( <i>expression</i> )
BigCount( <i>expression</i> )	count( <i>expression</i> )
Count( <i>expression</i> )	count( <i>expression</i> )
Max( <i>expression</i> )	max( <i>expression</i> )
Min( <i>expression</i> )	min( <i>expression</i> )
StDev( <i>expression</i> )	stdev( <i>expression</i> )
StDevP( <i>expression</i> )	stdevp( <i>expression</i> )
Sum( <i>expression</i> )	sum( <i>expression</i> )
Var( <i>expression</i> )	var( <i>expression</i> )
VarP( <i>expression</i> )	varp( <i>expression</i> )

## Math Canonical Functions

Table 14 describes the mapping of EDM math canonical functions to PSQL functions used to process columns that contain only decimal and integer values.

For more information, refer to the [Numeric Functions](#).

Table 14 Mapping Math Canonical Functions

Math Canonical Function	PSQL Function
Abs( <i>value</i> )	abs( <i>value</i> )
Ceiling( <i>value</i> )	ceiling( <i>value</i> )
Floor( <i>value</i> )	floor( <i>value</i> )
Power( <i>value</i> , <i>exponent</i> )	power( <i>value</i> , <i>exponent</i> )
Round( <i>value</i> )	round( <i>numeric_expression1</i> , <i>integer_expression2</i> )
Round( <i>value</i> , <i>digits</i> )	round( <i>value</i> , <i>digits</i> )
Truncate( <i>value</i> , <i>digits</i> )	truncate( <i>value</i> , <i>digits</i> )

## Date and Time Canonical Functions

Table 15 describes the mapping of EDM date and time canonical functions to PSQL functions that generate, process, and manipulate data that consists of data types such as DATE and TIME.

Table 15 Mapping Date and Time Canonical Functions

Date and Time Canonical Function	PSQL Functions
AddNanoseconds(expression,number)	dateadd(millisecond,number/1000000)
AddMicroseconds(expression,number)	dateadd(millisecond,number/1000)
AddMilliseconds(expression,number)	dateadd(millisecond,number)
AddSeconds(expression,number)	dateadd(second,number)
AddMinutes(expression,number)	dateadd(minute,number)
AddHours(expression,number)	dateadd(hour,number)
AddDays(expression,number)	dateadd(day,number)
AddMonths(expression,number)	dateadd(month,number)
AddYears(expression, number)	dateadd(year,number)
CreateDateTime(year,month,day,hour,minute,second)	datetimefromparts(year,month,day,hour,minute,second,0)
CreateDateTimeOffset(year,month,day,hour,minute,second,tzoffset)	datetimeoffsetfromparts(year,month,day,hour,minute,second,tzoffset)
CreateTime(hour,minute,second)	timefromparts(hour,minute,second,0,0)
CurrentDateTime()	now()
CurrentDateTimeOffset()	sysdatetimeoffset()
CurrentUtcDateTime()	current_timestamp()
Day(expression)	datepart(day,expression)
DayOfYear(startexpression,endexpression)	dayofyear(expression)
DiffNanoSeconds(startexpression,endexpression)	datediff(millisecond,startexpression,endexpression)*1000000
DiffMilliSeconds(startexpression,endexpression)	datediff(millisecond,startexpression,endexpression)
DiffMicroSeconds(startexpression,endexpression)	datediff(millisecond,startexpression,endexpression)*1000
DiffSeconds(startexpression,endexpression)	datediff(second,startexpression,endexpression)
DiffMinutes(startexpression,endexpression)	datediff(minute,startexpression,endexpression)
DiffHours(startexpression,endexpression)	datediff(hour, startexpression,endexpression)
DiffDays(startexpression,endexpression)	datediff(day, startexpression, endexpression)
DiffMonths(startexpression,endexpression)	datediff(month,startexpression,endexpression)
DiffYears(startexpression,endexpression)	datediff(year,startexpression,endexpression)
GetTotalOffsetMinutes(DateTime Offset)	datepart(tzoffset,expression)

Table 15 Mapping Date and Time Canonical Functions continued

Date and Time Canonical Function	PSQL Functions
Year(expression)	datepart(year,expression)
Month(expression)	datepart(month,expression)
Day(expression)	datepart(day,expression)
Hour(expression)	datepart(hour,expression)
Minute(expression)	datepart(minute,expression)
Second(expression)	datepart(second,expression)
Millisecond(expression)	datepart(millisecond,expression)
TruncateTime(expression)	convert(expression, SQL_DATE)
Requires PSQL v11.30 Update 4 (May 2013)	

## Bitwise Canonical Functions

Table 16 describes the mapping of EDM bitwise canonical functions to PSQL functions.

Table 16 Mapping Bitwise Canonical Functions

Bitwise Canonical Function	PSQL Functions
BitWiseAnd (value1, value2)	bit_and (value1, value2)
BitWiseNot (value)	bit_compliment
BitWiseOr (value1, value2)	bit_or
BitWiseXor (value1, value2)	bit_xor

## String Canonical Functions

Table 17 describes the mapping of EDM string canonical functions to PSQL functions.

Table 17 Mapping String Canonical Functions

String Canonical Function	PSQL Function
Concat(string1, string2)	concat(string1, string2)
Contains(string, target)	contains(string, target)
EndsWith(string, target)	endswith(string, target)
IndexOf(target, string2)	instr(target, string2)
Left(string1, length)	left(string1, length)
Length(string)	length(string)
LTrim(string)	ltrim(string)
Trim(string)	trim (BOTH FROM string)

Table 17 Mapping String Canonical Functions continued

String Canonical Function	PSQL Function
Replace( <i>string1</i> , <i>string2</i> , <i>string3</i> )	replace( <i>string1</i> , <i>string2</i> , <i>string3</i> )
Reverse( <i>string</i> )	reverse( <i>string</i> )
RTrim( <i>string</i> )	rtrim( <i>string</i> )
StartsWith( <i>string</i> , <i>target</i> )	startswith( <i>string</i> , <i>target</i> )
Substring( <i>string</i> , start, length)	INCOMPLETE regexp_substr(...)
ToLower( <i>string</i> )	lower( <i>string</i> )
ToUpper( <i>string</i> )	upper( <i>string</i> )

## Other Canonical Functions

Table 18 describes the mapping of other canonical functions to PSQL functions.

Table 18 Mapping Other Canonical Functions

Other Canonical Function	PSQL Function
NewGuid()	newid()

## **Extending Entity Framework Functionality**

The ADO.NET Entity Framework offers powerful productivity gains by masking many ADO.NET features, simplifying application development. The PSQl ADO.NET Data Provider includes functionality designed to optimize performance.

Applications that use the standard Logging Application Block (LAB) from the Microsoft Enterprise Library 6.0 and the related design patterns can quickly display the SQL generated as part of the ADO.NET Entity Framework data providers.

See [Logging Application Blocks](#) for more information.

## Enhancing Entity Framework Performance

Although the Entity Framework offers powerful productivity gains, some developers believe that the Entity Framework takes too much control of the features they need to optimize performance in their applications.

### ***Limiting the Size of XML Schema Files***

Building large models with the Entity Data Model (EDM) can be very inefficient. For optimal results, consider breaking up a model when it has reached 50 to 100 entities.

The size of the XML schema files is to some extent proportional to the number of tables, views, or stored procedures in the database from which you generated the model. As the size of the schema files increase, additional time is needed to parse and create an in-memory model for the metadata. This is a one-time performance cost that is incurred for eachObjectContext instance.

This metadata is cached per application domain, based on the EntityConnection String. This means that if you use the same EntityConnection string in multipleObjectContext instances in a single application domain, the application incurs the cost of loading metadata only once. However, the performance cost could still be significant if the size of the model becomes large and the application is not a long-running one.



## Using Stored Procedures with the ADO.NET Entity Framework

Using stored procedures with the ADO.NET Entity Framework requires mapping functions. Calling these stored procedures is complex and requires some coding.

### Providing Functionality

The Connection object includes properties and methods that provide enhanced statistics functionality that are standard in the ADO.NET data provider, but are not available at the ADO.NET Entity Framework layer. Instead, you expose the same functionality through "pseudo" stored procedures.

This approach uses the Entity Data Model (EDM) to achieve results that correspond to the ADO.NET results. This in effect provides entities and functions backed by pseudo stored procedures.

Table 19 lists the mapping of the data provider's Connection properties to the corresponding pseudo stored procedure.

Table 19 Mapping to Pseudo Stored Procedure

Connection Property	Pseudo Stored Procedure
StatisticsEnabled	Psql_Connection_EnableStatistics Psql_Connection_DisableStatistics
Connection Method	Pseudo Stored Procedure
ResetStatistics	Psql_Connection_ResetStatistics
RetrieveStatistics	Psql_Connection_RetrieveStatistics

Applications must use theObjectContext to create a stored procedure command as shown in the following C# code fragment:

```
using (MyContext context = new MyContext())
{
    EntityConnection entityConnection = (EntityConnection)context.Connection;

    // The EntityConnection exposes the underlying store connection
    DbConnection storeConnection = entityConnection.StoreConnection;
    DbCommand command = storeConnection.CreateCommand();
    command.CommandText = "Psql_Connection_EnableStatistics";
    command.CommandType = CommandType.StoredProcedure;
    command.Parameters.Add(new SqlParameter("cid", 1));
}

//

bool openingConnection = command.Connection.State == ConnectionState.Closed;
if (openingConnection) { command.Connection.Open(); }
int result;
try
{
    result = command.ExecuteNonQuery();
}
finally
```

```
{  
    if (openingConnection && command.Connection.State == ConnectionState.Open) {  
        command.Connection.Close(); }  
}
```

### ***Using Overloaded Stored Procedures***

If you have multiple overloaded stored procedures, the PSQL Entity Framework data provider appends an identifier to each stored procedure name so you can distinguish between them in the SSDL. The data provider removes the appended identifier before calling the stored procedure for your application.

## Using .NET Objects

The ADO.NET Entity Framework data provider supports the .NET public objects, exposing them as sealed objects.

For more information, see [.NET Objects Supported](#).

The ADO.NET Entity Framework programming contexts inherently eliminate the need to use some ADO.NET methods and properties. These properties and methods remain useful for standard ADO.NET applications. The online help, which is integrated into Visual Studio, describes the public methods and properties of each class.

Table 20 lists the properties and methods that are not required or are implemented differently when using the data provider with an ADO.NET Entity application.

*Table 20 Properties and Methods Differences with the ADO.NET Entity Data Provider*

Property or Method	Behavior
<b>PsqlCommand</b>	
AddRowID	Not supported. The ADO.NET Entity Framework does not process the additional data that is returned.
ArrayBindCount	Not supported. The application cannot influence this bind count on top of the ADO.NET Entity Framework.
ArrayBindStatus	Not supported. The application cannot influence this bind count on top of the ADO.NET Entity Framework.
BindByName	Not supported. Instead, the data provider uses the ADO.NET Entity Framework programming contexts.
CommandTimeout	Not supported. Instead, the data provider uses the ADO.NET Entity Framework programming contexts.
UpdatedRowSource	Not supported. Instead, the data provider uses the ADO.NET Entity Framework programming contexts.
<b>PsqlCommandBuilder</b>	
DeriveParameters	Not supported. Instead, the data provider uses the ADO.NET Entity Framework programming contexts.
<b>PsqlConnection</b>	
ConnectionTimeout	Supported only in a connection string.
StatisticsEnabled	Use the StatisticsEnabled or StatisticsDisabled stored procedure. See <a href="#">Using Stored Procedures with the ADO.NET Entity Framework</a> for information on using this functionality in an ADO.NET Entity Framework application.
<b>DataAdapter</b>	
UpdateBatchSize	Not supported. Instead, the data provider uses the ADO.NET Entity Framework programming contexts.
<b>Error</b>	
ErrorPosition	Not supported. Instead, the data provider uses the ADO.NET Entity Framework programming contexts.
SQLState	Not supported. Instead, the data provider uses the ADO.NET Entity Framework programming contexts.

## Creating a Model

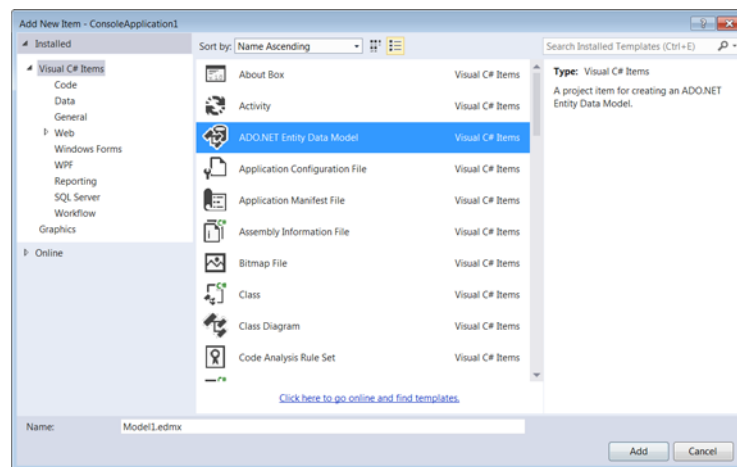
The Entity Framework creates a model of your data in Visual Studio.

**Note:** Developing with the Microsoft ADO.NET Entity Framework data provider requires that you use Microsoft .NET Framework Version 4.5.x or 4.6.x and Visual Studio 2012 or later with version 4.3 of the PSQL ADO.NET Entity Framework data provider.

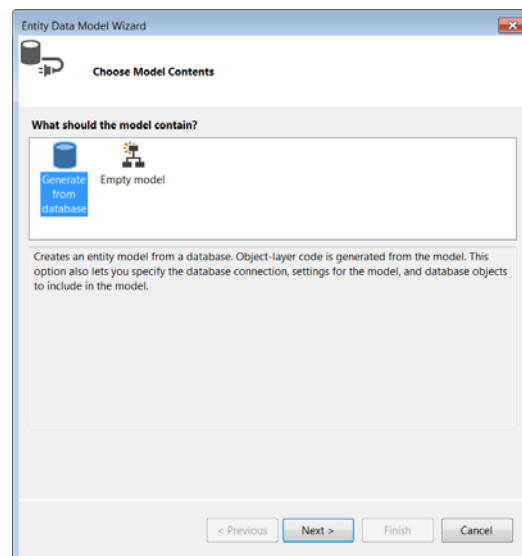
To create a model of your data in Visual Studio using the Entity Framework, you must first ensure that you already have the database schema available.

### ► To use the Entity Framework for creating a model of your data in Visual Studio

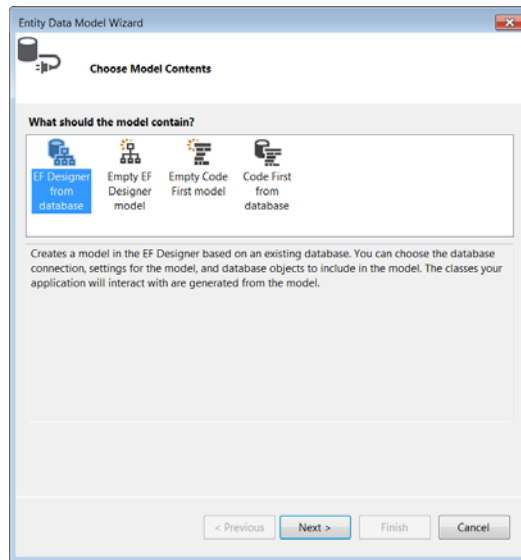
- 1 Create a new .NET application, such as Windows Console, Windows Forms, in Visual Studio.
- 2 In the Solution Explorer, right-click the project and select **Add > New Item**.
- 3 Select **ADO.NET Entity Data Model**, then click **Add**.



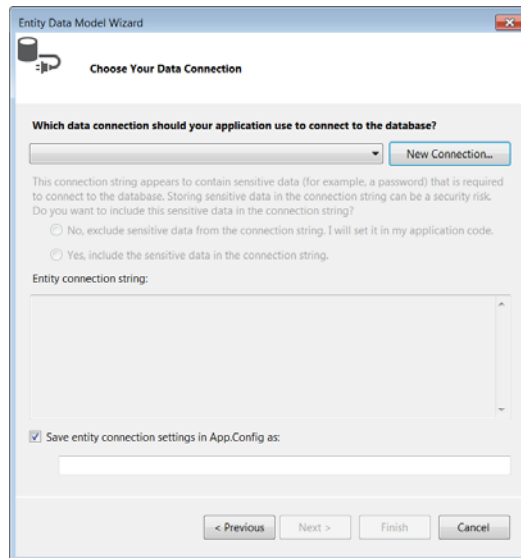
- 4 The Entity Data Model Wizard appears. Based on whether you have configured Microsoft ADO.NET Entity Framework 6.1 (EF 6.1), do one of the following:
  - If you **have not** configured EF 6.1, select **Generate from database** and click **Next**.



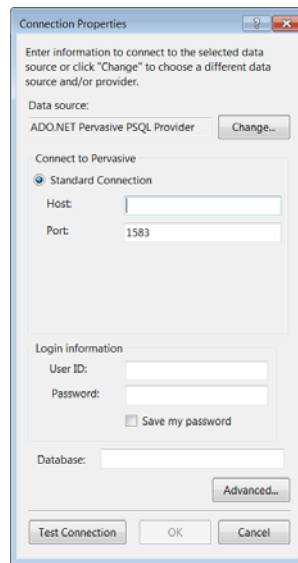
- If you **have** configured EF 6.1, select **EF Designer from database** and click **Next**.



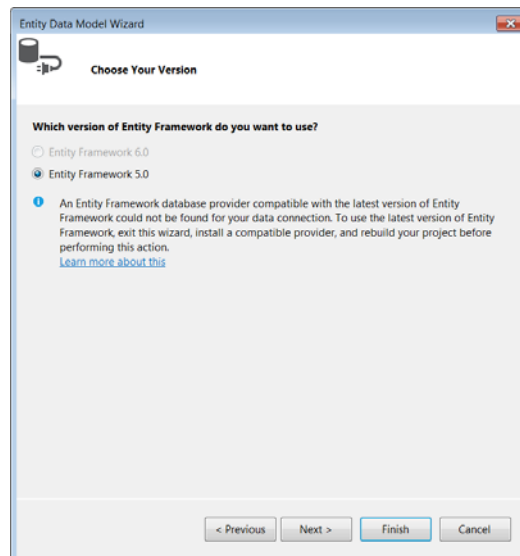
- 5 On the Choose your Data Connection page, click **New Connection** to create a new connection. If you have an established connection, you can select it from the drop-down list.



- 6 The Connection Properties window appears. Provide the necessary connection information and click **OK**.



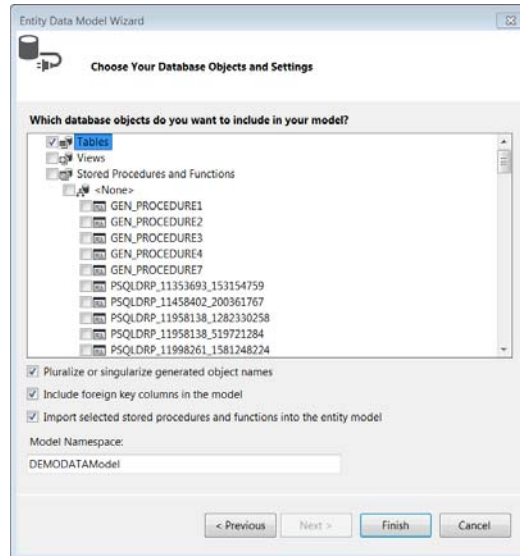
- 7 The Wizard creates an Entity connection string.
  - a. If the radio buttons are selectable, select **Yes, include the sensitive data in the connection string** to include the sensitive data in the connection string.
  - b. In the **Save entity connection settings** field, enter a name for the name of the main data access class or accept the default.
  - c. Click **Next**.
- 8 Based on the configured Entity Framework version, do one of the following:
  - If you have configured EF5 for the current project, on the Choose Your Version page, proceed with the default **Entity Framework 5.0** by clicking **Next**.



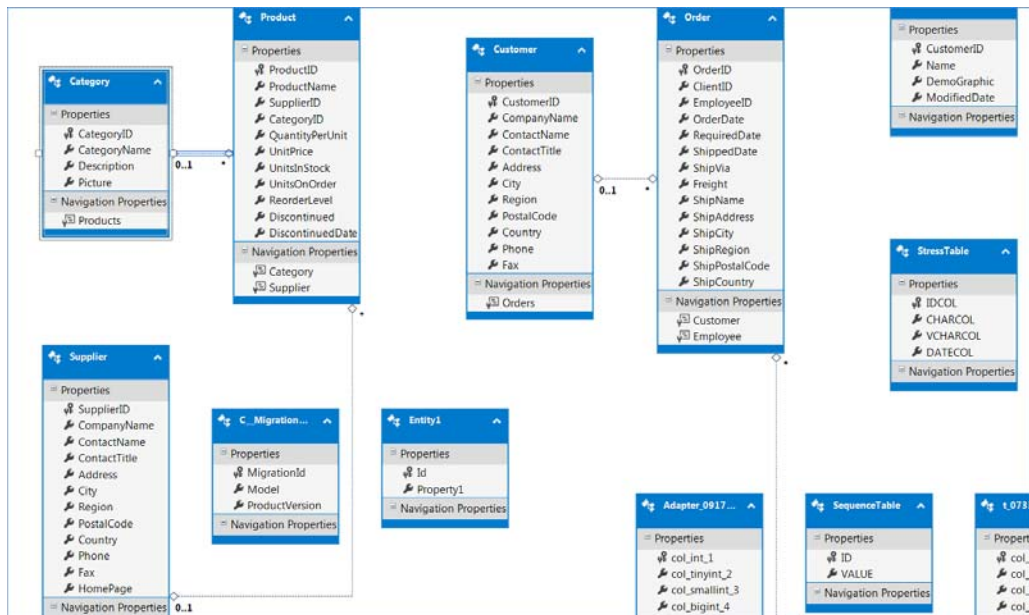
**Note:** To use the EF 6.1 with your current project, exit the wizard, configure EF 6.1, and then rebuild the project. When you rebuild the project after configuring EF 6.1, the wizard does not display the Choose Your Version page and you can directly proceed to the next step.

- If you have configured EF 6.1 for the current project, proceed to the next step.

**9** Select the database objects that will be used in the model.



**10** Click **Finish**. The model is generated and opened in the Model Browser.



## For More Information

Refer to the following sources for additional information about the ADO.NET and the Entity Framework:

- [“Programming Entity Framework”](#) by Julie Lerman provides a comprehensive discussion of using the ADO.NET Entity Framework.
- [ADO.NET Entity Framework](#) introduces the Entity Framework and provides links to numerous detailed articles.
- [Connection Strings \(Entity Framework\)](#) describes how connection strings are used by the Entity Framework. The connection strings contain information used to connect to the underlying ADO.NET data provider as well as information about the required Entity Data Model mapping and metadata.
- [Entity Data Model Tools](#) describes the tools that help you to build applications graphically with the EDM: the Entity Data Model Wizard, the ADO.NET Entity Data Model Designer (Entity Designer), and the Update Model Wizard. These tools work together to help you generate, edit, and update an Entity Data Model.
- [LINQ to Entities](#) enables developers to write queries against the database from the same language used to build the business logic.



# *Using the PSQL Data Providers in Visual Studio*

---

The PSQL data provider supports integration into Visual Studio. This means that developers can use the graphical user interface of Microsoft Visual Studio to perform a variety of tasks.

The following topics describe how the features of the PSQL data providers are integrated into Visual Studio:

- [Adding Connections](#)
- [Using the PSQL Performance Tuning Wizard](#)
- [Using Provider-Specific Templates](#)
- [Using the PSQL Visual Studio Wizards](#)
- [Adding Components from the Toolbox](#)
- [Data Provider Integration Scenario](#)

## Adding Connections

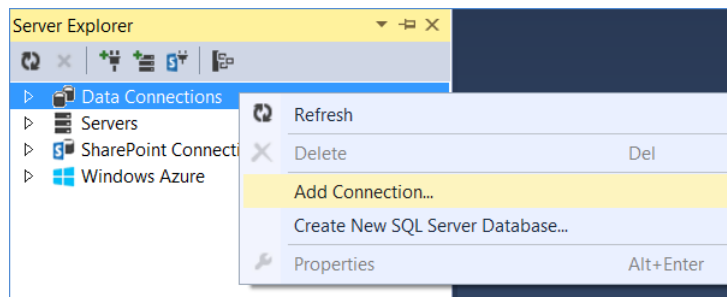
You can add connections in several ways in Visual Studio:

- [Adding Connections in Server Explorer](#)
- [Adding Connections with the Data Source Configuration Wizard](#)

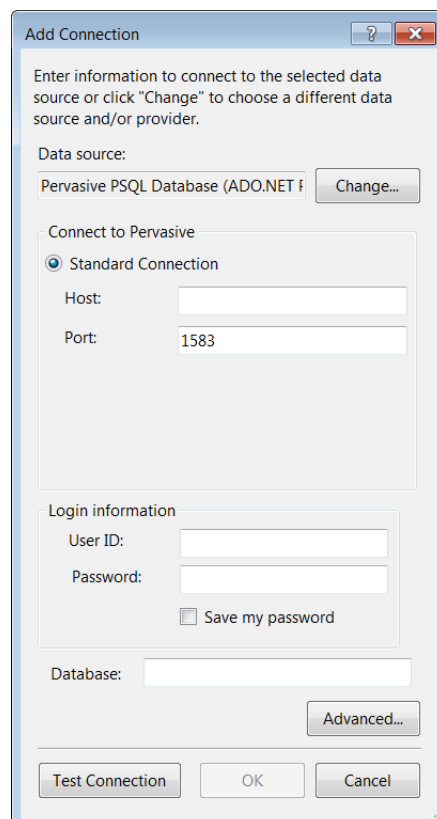
### *Adding Connections in Server Explorer*

#### ►► To add a connection

- 1 Right-click the **Data Connections** node in the Server Explorer and select **Add Connection**.

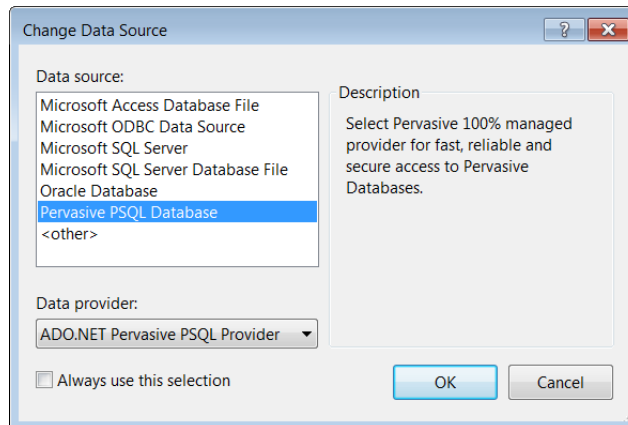


The Add Connection window appears.

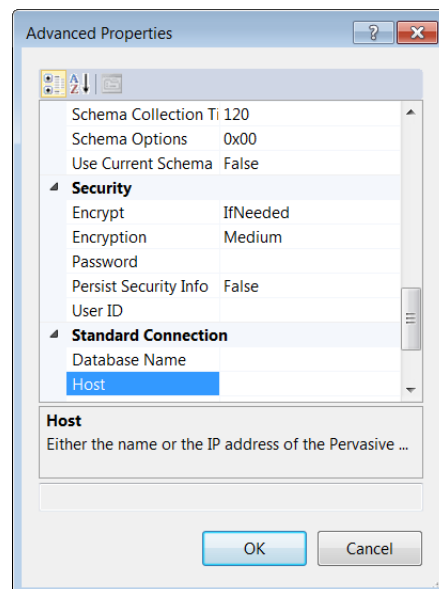


- 2 If the PSQL data provider is displayed in the **Data source** field, skip to Step 4. Otherwise, click **Change**.

3 The Change Data Source window appears.



- a. In the Data source list box, select **Pervasive PSQL Database**.
  - b. In the Data provider list, select **ADO.NET Pervasive PSQL Provider**.
  - c. If you want to use these selections for other connections, select the **Always use this selection** check box.
  - d. Click **OK** to return to the Add Connection window.
- 4 In the Add Connection window, do the following steps:
- a. Enter the Host name.
  - b. Enter the User ID and password. These values are required for authentication.
  - c. (Optional) If you want to save the password for the lifetime of connection instance defined in Server Explorer, select the **Save my password** check box.
  - d. (Optional) In the Database entry field, enter the name of the database to which you want to connect.
- 5 Click the **Advanced** button to specify additional provider-specific property values.



To change a value in the Advanced Properties dialog box, select or type the new value into the field and press ENTER. The value is added to the connection string that appears in the field below the description of the property. If you accept the default values, the connection string field remains unchanged. When you have made the necessary changes, click **OK** to return to the Add Connection window.

## Advanced

**EnableIPv6:** Provides backward compatibility for connecting to the PSQL Server using an IPv4 address.

If set to True, a client with IPv6 protocol installed can connect to the server using either an IPv4 address or an IPv6 address.

If set to False, the clients run in the backward compatibility mode. The client always connects to the server using an IPv4 address.

The default value for 4.0 is set to True.

For more information about IPv6 formats, see [IPv6](#) in *Getting Started with PSQL*.

**Encoding:** Type the ANSI name or Windows code page to be used for translating string data stored in the database. By default, the Windows code page is used.

**Initial Command Timeout:** Specifies the default wait time (timeout in seconds) before the data provider terminates the attempt to execute the command and generates an error. This option provides the same functionality as the PsqlCommand object's CommandTimeout property without the need to make changes to the application code. Subsequently, an application can use the CommandTimeout property to override the Initial Command Timeout connection string option.

The initial default value is 30 seconds.

**Note:** Set the Initial Command Timeout option to a value that is greater than the largest default deadlock detection and timeout value on the server. This ensures that the application receives a more meaningful reply in case of a timeout.

**Initialization String:** Type one statement that will be issued immediately after connecting to the database to manage session settings.

Example: To handle CHAR columns that are padded with NULLs, set the value to:

```
Initialization String=SET ANSI PADDING ON
```

**Note:** If the statement fails to execute for any reason, the connection to the server fails. The data provider throws an exception that contains the errors returned from the server.

**Parameter Mode:** Select the behavior of native parameter markers and binding. This allows applications to reuse provider-specific SQL code and simplifies migration to the PSQL data provider. Note that this option does not apply to the ADO.NET Entity Framework data provider.

If set to ANSI (the default), the ? character is processed as a parameter marker and bound as ordinal.

If set to BindByOrdinal, native parameter markers are used and are bound as ordinal.

If set to BindByName, native parameter markers are used and are bound by name.

**PVTranslate:** Select whether the client should negotiate a compatible encoding with the server.

If set to Auto (the default for the 4.2 provider), the data provider will set the Encoding connection property to the database code page. In addition, SQL query text will be sent to the engine using UTF-8 encoding instead of the data encoding. This preserves NCHAR string literals in the query text.

If set to Nothing (the default for the 4.0 provider), the setting for the Encoding connection property is used.

**Timestamp:** Select whether PSQL timestamps are stored and retrieved as strings.

If set to DateTime (the initial default), the data provider maps timestamps to DateTime. This setting may be appropriate when native precision is required, for example, when using the CommandBuilder with timestamp.

If set to String, the data provider maps PSQL timestamps as strings.

**TimeType:** Select whether PSQL times are retrieved as Timespan or DateTime in the ADO.NET data provider.

If set to As DateTime, the data provider maps the SQL type TIME to the .NET type System.DateTime.

If set to As TimeSpan, the data provider maps the SQL type TIME to the .NET type System.DateTimespan.

## Connection Pooling

**Connection Reset:** Select whether a connection that is removed from the connection pool for reuse by an application will have its state reset to the initial configuration settings of the connection.

If set to False (the initial default), the data provider does not reset the state of the connection.

**Connection Pool Behavior:** Select the order in which a connection is removed from the connection pool for reuse, based on how frequently or how recently the connection has been used.

If set to MostRecentlyUsed, the data provider uses a Last In First Out (LIFO) approach to return the connection that was returned to the pool most recently.

If set to LeastRecentlyUsed, the data provider uses a First In First Out (FIFO) approach to return the connection with the lowest use count. This value ensures a balanced use of connections in the pool.

If set to MostFrequentlyUsed, the data provider returns the connection with the highest use count. This value enables applications to give preference to the most seasoned connection.

If set to LeastFrequentlyUsed, the data provider returns the connection with the lowest use count. This value ensures a balanced use of connections in the pool.

**Connection Timeout:** Type the number of seconds after which the attempted connection to the server will fail if not yet connected. If connection failover is enabled, this option applies to each connection attempt.

If set to 0, the data provider never times out on a connection attempt.

The initial default is 15 seconds.

**Load Balance Timeout:** Type the number of seconds to keep connections in a connection pool. The pool manager periodically checks all pools, and closes and removes any connection that exceeds this value. The Min Pool Size option can cause some connections to ignore the value specified for the Load Balance Timeout option.

The value can be any integer from 0 to 65535.

If set to 0 (the initial default), the connections have the maximum timeout.

See [Removing Connections from a Pool](#) for a discussion of connection lifetimes.

**Max Pool Size:** Type the maximum number of connections within a single pool. When the maximum number is reached, no additional connections can be added to the connection pool.

The value can be any integer from 1 to 65535.

The initial default is 100.

**Max Pool Size Behavior:** Select whether the data provider can exceed the number of connections specified by the Max Pool Size option when all connections in the connection pool are in use.

If set to SoftCap, when all connections are in use and another connection is requested, a new connection is created, even when the connection pool exceeds the number set by the MaxPoolSize option. If a connection is returned and the pool is full of idle connections, the pooling mechanism selects a connection to be discarded so the connection pool never exceeds the Max Pool Size.

If set to HardCap, when the maximum number of connections allowed in the pool are in use, any new connection requests wait for an available connection until the Connection Timeout is reached.

**Min Pool Size:** Type the minimum number of connections that are opened and placed in a connection pool when it is created. The connection pool retains this number of connections, even when some connections exceed their Load Balance Timeout value.

The value can be any integer from 0 to 65535.

If set to 0 (the initial default), no additional connections are placed in the connection pool when it is created.

**Pooling:** Select True (the initial default) to enable connection pooling.

## Failover

**Alternate Servers:** Type a list of alternate database servers to which the data provider will try to connect if the primary database server is unavailable. Specifying a value for this property enables connection failover for the data provider.

For example, the following Alternate Servers value defines two alternate servers for connection failover:

```
Alternate Servers="Host=AcctServer;Port=1584,  
Host=123.456.78.90;Port=1584"
```

**Connection Retry Count:** Type the number of times the data provider tries to connect to the primary server, and, if specified, the alternate servers after the initial unsuccessful attempt.

The value can be any integer from 0 to 65535.

If set to 0 (the initial default), there is no limit to the number of attempts to reconnect.

**Connection Retry Delay:** Type the number of seconds the data provider waits after the initial unsuccessful connection attempt before retrying a connection to the primary server, and, if specified, the alternate servers.

The initial default is 3.

This property has no effect unless the Connection Retry Count property is set to an integer value greater than 0.

**Load Balancing:** Select True or False to determine whether the data provider uses client load balancing in its attempts to connect to primary and alternate database servers.

If set to False (the initial default), the data provider does not use client load balancing.

## Performance

**Enlist:** Select True or False to determine whether the data provider automatically attempts to enlist the connection in creating the thread's current transaction context.

**Note:** Because PSQL does not support distributed transactions, any attempt to enlist the connection in the thread's current transaction context will fail.

If set to False (the initial default), the data provider does not automatically attempt to enlist the connection.

If set to True, the data provider returns an error on the connection if a current transaction context exists. If a current transaction context does not exist, the data provider raises a warning.

**Max Statement Cache Size:** Type the maximum number of statements generated by the application that can be held in the statement cache for this connection.

The value can be 0, or any integer greater than 1.

If set to 0, statement caching is disabled.

If set to an integer greater than 1, the value determines the number of statements that can be held in the statement cache.

The initial default is 10.

**Statement Cache Mode:** Select the statement caching mode for the lifetime of the connection. See [Using Statement Caching](#) for more information.

If set to Auto, statement caching is enabled. Statements marked as Implicit by the Command property StatementCacheBehavior are cached. These commands have a lower priority than that of explicitly marked commands, that is, if the statement pool reaches its maximum number of statements, the statements marked implicit are removed from the statement pool first to make room for statements marked Cache.

If set to ExplicitOnly (the initial default), only commands that are marked Cache by the StatementCacheBehavior property are cached. Note that this is the only valid value for the ADO.NET Entity Framework data provider.

## Schema Information

**Schema Collection Timeout:** Type the number of seconds after which an attempted schema collection operation fails if it is not yet completed.

The initial default is 120.

**Schema Options:** Specifies additional database metadata that can be returned. By default, the data provider prevents the return of some performance-expensive database metadata to optimize performance. If your application needs this database metadata, specify the name or hexadecimal value of the metadata.

This option can affect performance.

If set to ShowColumnDefaults or 0x04, column defaults are returned.

If set to ShowParameterDefaults or 0x08, column defaults are returned.

If set to FixProcedureParamDirection or 0x10, procedure definitions are returned.

If set to ShowProcedureDefinitions or 0x20, procedure definitions are returned.

If set to ShowViewDefinitions or 0x40, view definitions are returned.

If set to ShowAll or 0xFFFFFFFF, all database metadata is returned.

For example, to return descriptions of procedure definitions, specify Schema Options=ShowProcedureDefinitions or Schema Options=0x20.

To show more than one piece of the omitted database metadata, specify either a comma-separated list of the names, or the sum of the hexadecimal values of the column collections that you want to restrict.

See Table 28 for the name and hexadecimal value of the database metadata that the data provider can add.

**Use Current Schema:** This connection string option is not supported. Setting it will cause the data provider to throw an exception.

## Security

**Encrypt:** Select whether the data provider uses Encrypted Network Communications, also known as wire encryption.

If set to IfNeeded (the initial default), the data provider reflects the server's setting.

If set to Always, the data provider uses encryption, or, if the server does not allow wire encryption, returns an error.

If set to Never, the data provider does not use encryption and returns an error if wire encryption is required by the server.

**Encryption:** Select the minimum level of encryption allowed by the data provider. The meaning of these values depends on the encryption module used. With the default encryption module, the values Low, Medium, and High correspond to 40-, 56-, and 128-bit encryption, respectively.

The initial default is Medium.

**Password:** Type a case-insensitive password used to connect to your PSQL database. A password is required only if security is enabled on your database. If so, contact your system administrator to get your password.

**Persist Security Info:** Select whether to display secure information in clear text in the ConnectionString property.

If set to True, the value of the password connection string option is displayed in clear text.

If set to False (the initial default), the data provider does not display secure information in clear text.

**User ID:** Type the default PSQL user name used to connect to your PSQL database.



## Standard Connection

**Database Name:** Type a string that identifies the internal name of the database to which you want to connect.

If you enter a value for this field, the Server DSN field is not available.

**Host:** Type the name or the IP address of the PSQL server to which you want to connect. For example, you can specify a server name such as `accountingserver`. Or, you can specify an IPv4 address such as `199.262.22.34` or an IPv6 address such as `2001:DB8:0000:0000:8:800:200C:417A`.

**Port:** Type the TCP port number of the listener running on the PSQL database.

The default port number is 1583.

**Server DSN:** The name of the data source on the server, such as `DEMODATA`.

If you enter a value for this field, the Database Name field is not available.

## Tracing

**Enable Trace:** Type a value of 1 or higher to enable tracing. If set to 0 (the default), tracing is not enabled.

**Trace File:** Type the path and name of the trace file. If the specified trace file does not exist, the data provider creates it. The default is an empty string.

- 6 Click **Test Connection**. At any point during the configuration process, you can click **Test Connection** to attempt to connect to the data source using the connection properties specified in the Add Connection window.
  - If the data provider can connect, it releases the connection and displays a `Connection Established` message. Click **OK**.
  - If the data provider cannot connect because of an incorrect environment or incorrect connection value, it displays an appropriate error message.

Click **OK**.

**Note:** If you are configuring alternate servers for use with the connection failover feature, be aware that the Test Connection button tests only the primary server, not the alternate servers.

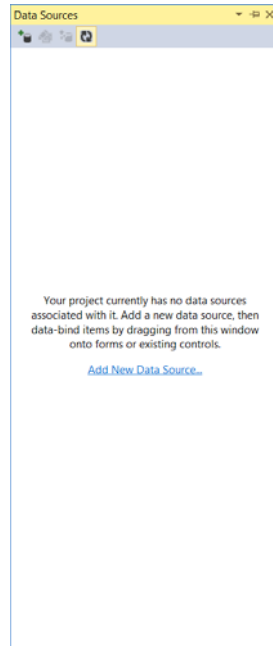
- 7 Click **OK** or **Cancel**. If you click **OK**, the values you have specified become the defaults when you connect to the data source. You can change these defaults by using this procedure to reconfigure your data source. You can override these defaults by connecting to the data source using a connection string with alternate values.

## Adding Connections with the Data Source Configuration Wizard

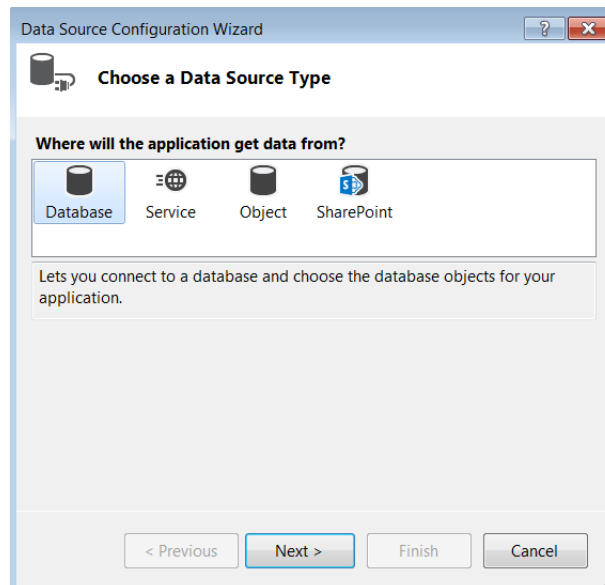
You can add a new connection to your application using the Data Configuration Wizard.

### ►► To add a connection

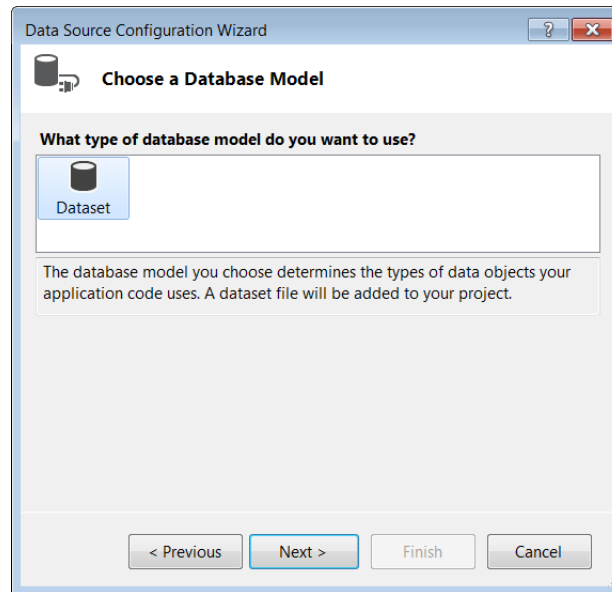
- 1 In the Data Sources window in Visual Studio, select **Add New Data Source**. To open the Data Sources window, select **View** from the main menu and then select **Other Windows > Data Sources**.



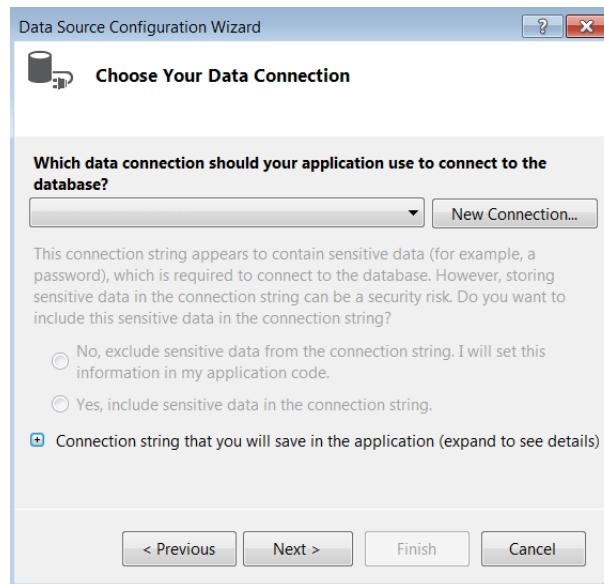
The Data Source Configuration Wizard appears.



- 2 Select **Database** and then click **Next**. The Choose Your Data Connection window appears.



- 3 Click **New Connection**. The Add Connection window is displayed. Continue from Step in [Adding Connections in Server Explorer](#).



## Using the PSQL Performance Tuning Wizard

The PSQL Performance Tuning Wizard leads you step-by-step through a series of questions about your application. Based on your answers, the Wizard provides the optimal settings for performance-related connection string options for your PSQL data provider.

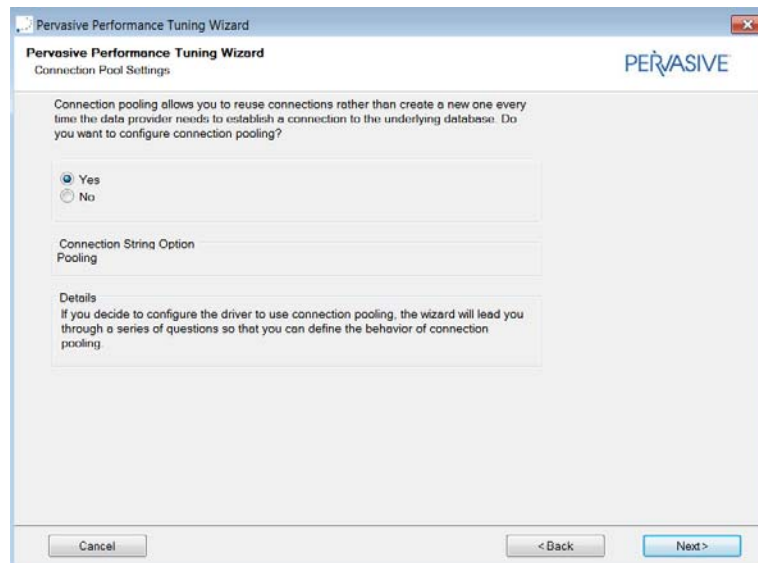
When you launch the PSQL Performance Tuning Wizard from Visual Studio, you can:

- Generate the values for connection string options that are related to performance. These values can be copied into a connection string.
- Modify an existing connection.
- Generate a new application preconfigured with a connection string optimized for your environment. The Performance Tuning Wizard provides options to select the type of application and the version of ADO.NET code that you want to use.

### ► To use the PSQL Performance Tuning Wizard in Visual Studio

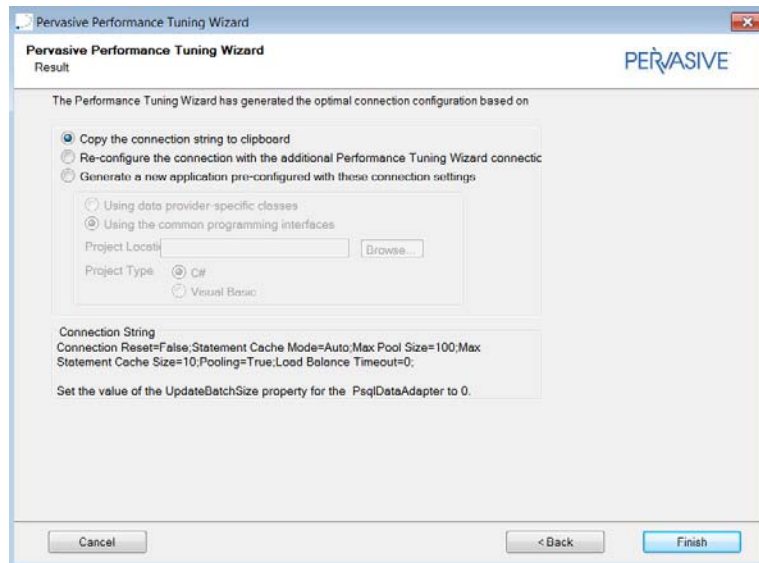
- 1 Do one of the following steps to start the Performance Tuning Wizard:
  - To create a new connection, select **Tools > Pervasive > Run Pervasive Performance Tuning Wizard**. When the Performance Tuning Wizard Welcome dialog appears, click **Next**. Continue at Step 2.
  - To modify an existing connection, in Server Explorer, right-click a data connection, and then select **Run Pervasive Performance Tuning Wizard**. When the Performance Tuning Wizard Welcome dialog appears, click **Next**. Continue at Step 2.
- 2 The Wizard presents a series of questions about your environment. Accept the default or change the answers as required and then, click **Next** to proceed, until you reach the Result page.

The following screen shot shows an example of one of the questions you may be asked.



- 3 When you have answered all questions for a data provider, the Result dialog appears, and a connection string is displayed.

The following screen shot shows the connection string options related to performance that the PSQL Performance Tuning Wizard generated.



**4** Select one of the following:

- To make the connection string available to other applications (the initial default), select **Copy the connection string to clipboard**. You can use the connection string in other applications.
- Based on whether you have used a new connection or an existing connection to launch the wizard, select one of the following options:
  - **Create a new connection with the Performance Tuning Wizard connection string options**

When you select this option and click **Finish**, the Modify Connection dialog box appears, where you must specify the connection information, such as a host, password, user ID, and other information.

- **Reconfigure the connection with the additional Performance Tuning Wizard connection string options.**
- To create a new application, select **Generate a new application preconfigured with these connection settings.**

When you select this option and click **Finish**, a Pervasive application is generated using the Pervasive Project template. See [Creating a New Project](#) for more information about the provider-specific templates.

**5** Define additional information for the new application:

- Select **Using data provider-specific interfaces** to create an application compatible with the ADO.NET 2.0 specification.
- Select **Using common programming interfaces** to create an application that uses the ADO.NET common programming model.
- Type the location for the project, or click **Browse** to select the location.
- Select the project type. By default, the Wizard creates a C# application.

**6** Click **Finish** to exit the PSQL Performance Tuning Wizard.

## Using Provider-Specific Templates

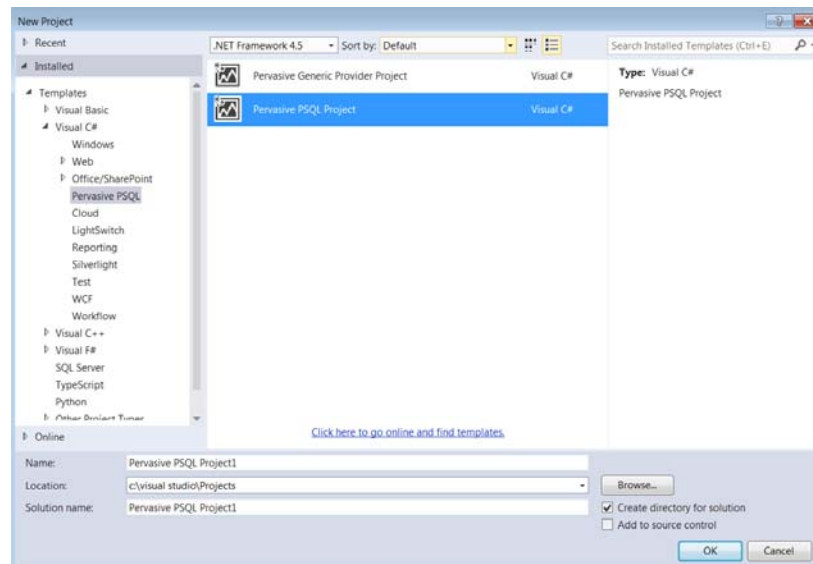
Visual Studio offers a set of templates to help you build applications that automatically include features such as SQL leveling.

### Creating a New Project

When you create a new project in Visual Studio, you can use a template specific to the PSQL data provider, or a template that creates an application with generic code.

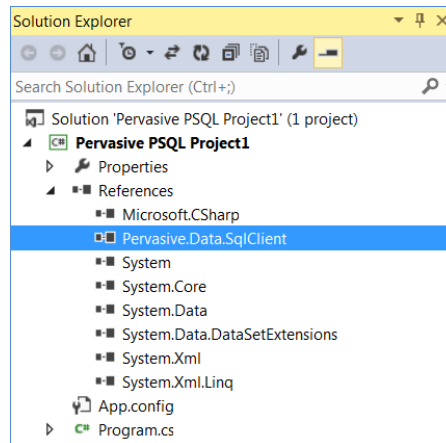
In the following example, we create a new project in Visual Studio using the template for the PSQL data provider.

- 1 Select **File > New > Project** The New Project dialog appears.
- 2 In the Installed List, select **Visual C# > PSQL**.
- 3 Select **Pervasive PSQL Project** in the middle pane.



- 4 Make changes to the other fields if required, and click **OK**.

- 5 The new project appears in the Solution Explorer. The namespace for the PSQL ADO.NET data provider is automatically added to the project.

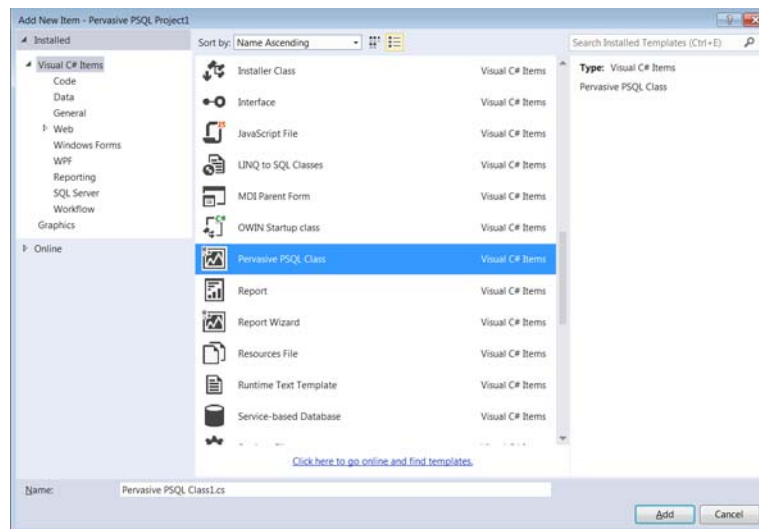


**Note:** If you are using the ADO.NET 2.0 common programming model, select the PSQL Generic Provider Project template. In this case, the project does not require a specific reference to an assembly.

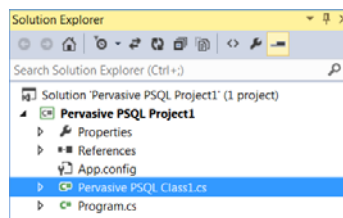
### ***Adding a Template to an Existing Project***

#### **» To add a PSQL template to an existing project**

- 1 In Solution Explorer, right-click the project and select **Add > New Item**.
- 2 In the **Add New Item** dialog, select the PSQL class.



- 3 Click **Add**. The class for the PSQL data provider is added to the project.



## Using the PSQL Visual Studio Wizards

Wizards simplify typical tasks that you perform when you create an application:

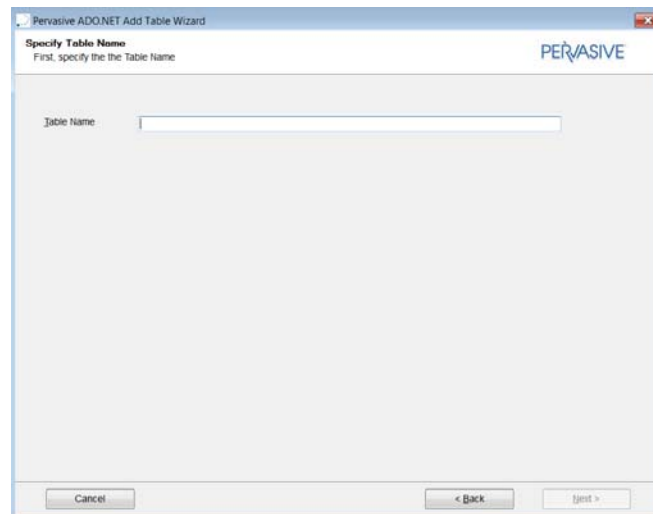
- [Creating Tables With the Add Table Wizard](#)
- [Creating Views With the Add View Wizard](#)

Before beginning this procedure, create a project using a PSQL template, as described in [Creating a New Project](#), and add a data connection.

### ***Creating Tables With the Add Table Wizard***

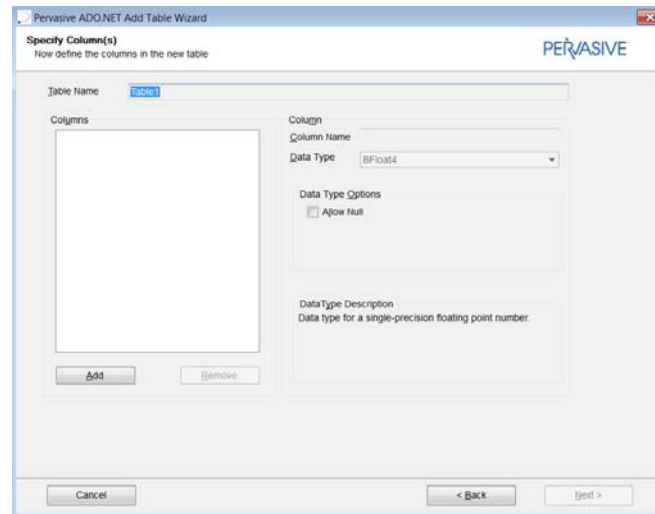
You can quickly and easily define new tables in Visual Studio using the PSQL ADO.NET Add Table Wizard.

- 1 Select **View > Server Explorer**.
- 2 Double-click a data source connection to expose the nodes below it.
- 3 Right-click the **Tables** node, and select **Add New Table**. The PSQL ADO.NET Add Table Wizard appears.
- 4 Click **Next**. The Specify Table Name dialog appears.
- 5 In the **Table Name** field, type a name for the table.



- 6 Click **Next**. The Specify Column(s) dialog appears.

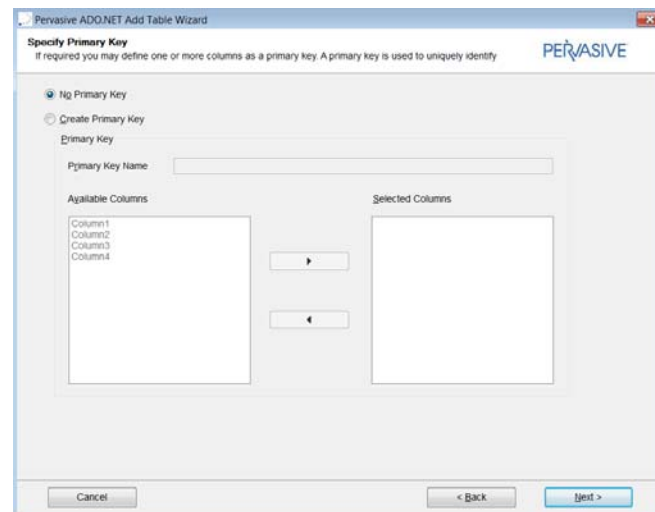




**7** Define the columns for the new table. Your choices may cause additional fields to appear in the Data Type Options pane.

- Click **Add** to add a column to the table. The Column Name and Data Type fields become editable.
- Type a name in the Column Name field.
- Select the data type for the column, and, if required, supply any additional information:
  - If you select a character data type, the Length field appears in the Data Type Options pane. Type the maximum size of the column (in bytes).
  - If you select Number, the Precision and Scale fields appear in the Data Type Options pane.
- If the column can have a Null value, select the **Allow Null** check box.
- To remove a column from the table, select the column name and then click **Remove**.

**8** Click **Next**. The Specify Primary Key dialog appears.



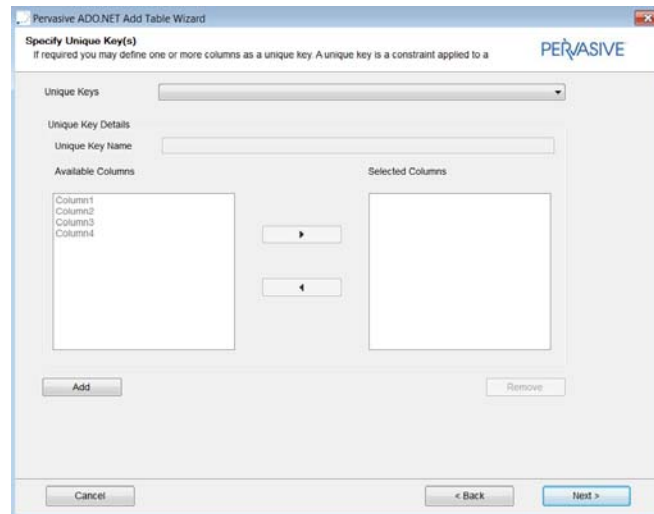
**9** Do one of the following:

- If you do not want to specify a primary key for the table, select **No Primary Key**, and then click **Next**. The Specify Unique Key(s) dialog appears. Continue at Step 12.
- If you want to specify a primary key for the table, select **Create Primary Key**, and then continue at Step 10.

**10** Complete the fields on the Specify Primary Key dialog:

- In the Primary Key Name field, type the name for the primary key, or accept the default name.
- Select a column from the Available Columns field and move it to the Selected Columns field.

**11** Click **Next**. The Specify Unique Key(s) dialog appears.



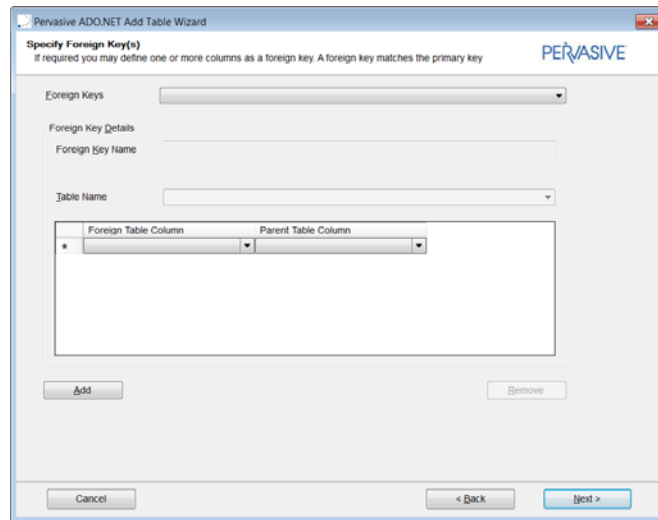
**12** Do one of the following:

- If you do not want to specify unique keys for the table, click **Next**. The Specify Foreign Key(s) dialog appears. Continue at Step 15.
- If you want to specify one or more unique keys for the table, continue at Step 15.

**13** Click **Add**. The fields on the dialog become selectable:

- In the **Unique Keys** drop-down list, select a unique key.
- In the **Unique Key Name** field, edit the name or accept the default name.
- In the **Available Columns** list box, select one or more columns to be used to specify the unique key, and move them to the Selected Columns list box.

**14** Click **Next**. The Specify Foreign Key(s) dialog appears.



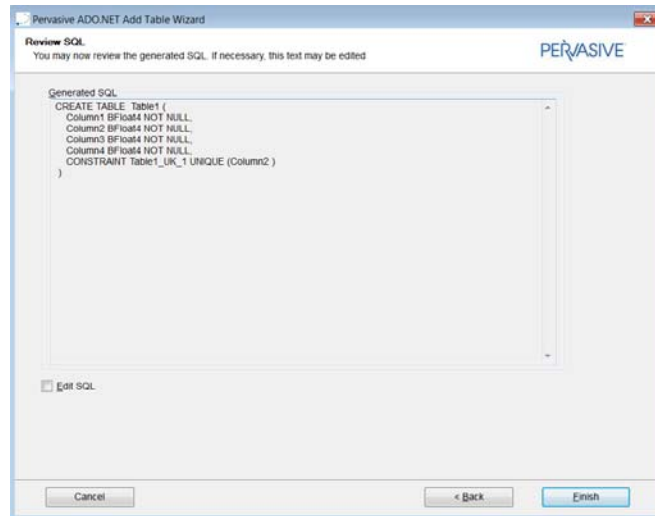
**15** Do one of the following:

- If you do not want to specify foreign keys for the table, click **Next**. The Review SQL dialog appears. Continue at Step 18.
- If you want to specify one or more foreign keys for the table, continue at Step 16.

**16** Click **Add**. The fields on the dialog become selectable:

- In the **Foreign Keys** drop-down list, select a foreign key.
- In the **Foreign Key Name** field, edit the name or accept the default name.
- In the **Table Schema** list, select a table schema.
- In the **Table Name** list, select a table schema.
- In the **Foreign Key Column** list, select one or more columns to be used to specify the foreign key.
- In the **Parent Table Column** list, select the corresponding column from the parent table.

**17** Click **Next**. The Review SQL dialog appears.



- 18 Review the SQL statement that has been generated by your choices.
  - If you are satisfied with the SQL statement, click **Finish**. The table that you created appears in Server Explorer under the Tables node for the connection.
  - If you want to supplement the SQL statement, for example, add a view or specific keywords, continue at Step 19.
- 19 Select the **Edit SQL** check box. The text in the Generated SQL field becomes editable.

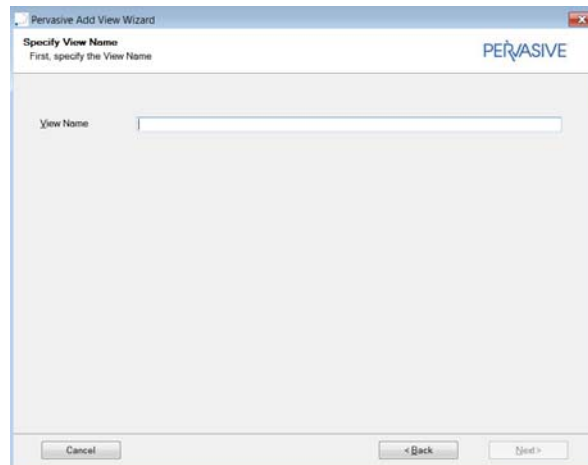
**Note:** When you select the Edit SQL check box, the Back button is disabled.
- 20 When you are satisfied with your changes to the SQL statement, click **Finish**. The table that you created appears in Server Explorer under the Tables node for the connection.

### ***Creating Views With the Add View Wizard***

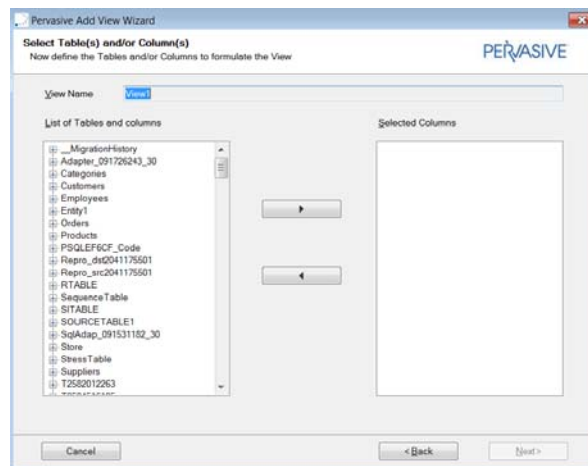
You can quickly and easily define new views in Visual Studio using the PSQL Add View Wizard.

- 1 Select **View > Server Explorer** if it is not already open.
- 2 Double-click a data source connection to expose the nodes under it.
- 3 Right-click the **Views** node, and select **Add New View**. The PSQL Add View Wizard welcome dialog appears.
- 4 Click **Next**. The Specify View Name dialog appears.

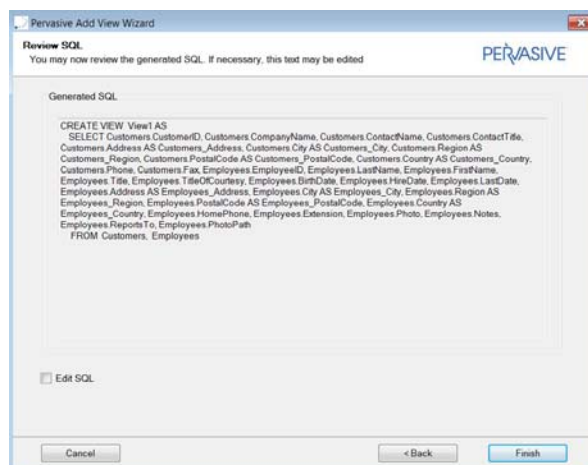
- 5 Type a name for the view in the View Name field.



- 6 Click **Next**. The Select Table(s) and/or Column(s) dialog appears.



- 7 In the **List of Tables and columns** list box, select the tables or columns that will make up the view, and move them to the Selected Columns column.
- 8 Click **Next**. The Review SQL dialog appears.



- 9** Review the SQL statement that has been generated by your choices.
  - If you are satisfied with the SQL statement, click **Finish**. The view that you created appears in Server Explorer under the Views node for the connection.
  - If you want to supplement the SQL statement, for example, add a view or specific keywords, continue at Step 11.
- 10** Select the **Edit SQL** check box. The text in the Generated SQL field becomes editable.

**Note:** When you select the Edit SQL check box, the Back button is disabled.
- 11** When you are satisfied with your changes to the SQL statement, click **Finish**. The view that you created appears in Server Explorer under the Views node for the connection.

## **Adding Components from the Toolbox**

You can add components from the Visual Studio Toolbox to a Windows Forms application. For information about creating Windows Forms applications, refer to the Visual Studio online Help.

Before beginning this procedure, create a Windows Forms application and add a data connection.

### **►► To add PSQL data provider components to a Windows Forms application**

- 1** Select **View > Toolbox**. Scroll down the Toolbox until the PSQL ADO.NET Provider section appears.
- 2** Select the **PsqlCommand** widget and drag it onto the Windows Forms application.
- 3** Continue adding widgets to the application as needed.

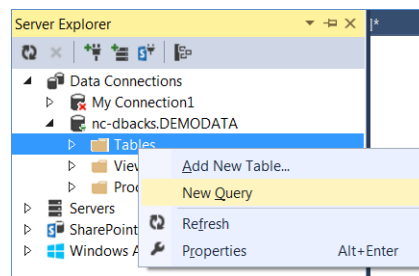
## Data Provider Integration Scenario

Because the PSQL data provider is integrated into Visual Studio, many typical data access tasks can be simplified. For example, after making the connection to the database, you can create queries using Query Builder.

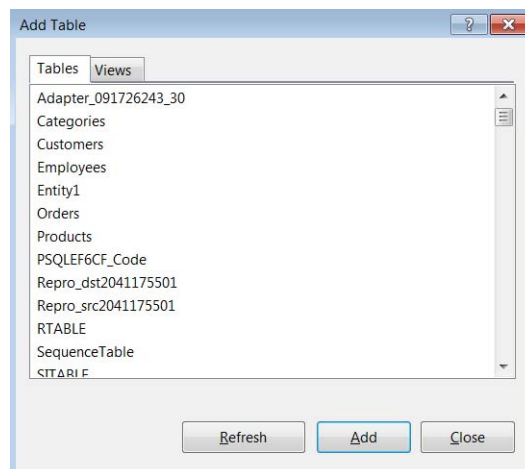
The Query Builder can help you graphically design database queries.

### ►► To create a simple query

- 1 Establish a data source connection (see [Using the PSQL Visual Studio Wizards](#)).
- 2 Select the data source in Server Explorer.
- 3 Right-click **Tables** and select **New Query**.



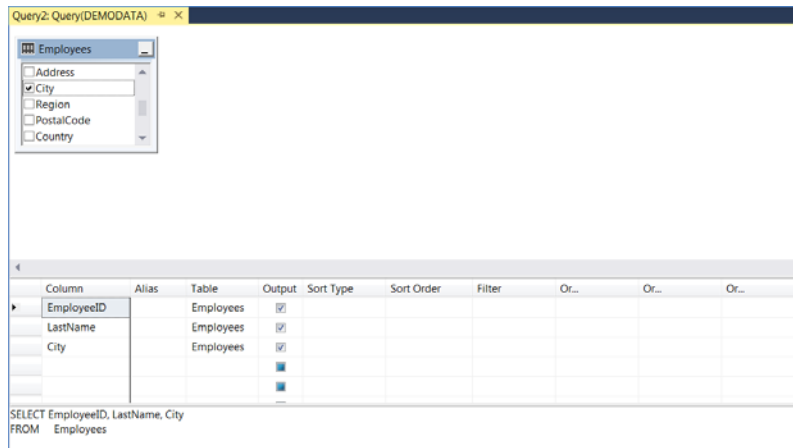
- 4 The Add Table window appears. Select the table that contains the data that you want to use; then, click **Add**.



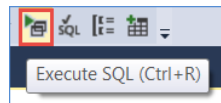
- 5 Click **Close** to close the Add Table window.



- 6 Select the columns that you want returned. In this example, we select the id, name, and salary columns of the employee table.



- 7 Click the **Execute SQL** button on the toolbar.



- 8 Examine the results displayed.



# Using the Microsoft Enterprise Libraries

---

## *Using the Data Access Application Blocks in Applications*

Using the Microsoft Enterprise Libraries can simplify application development by wrapping common tasks, such as data access, into portable code that makes it easier to move your application from one DBMS to another.

The PSQL data providers support the Data Access Application Blocks (DAAB) and Logging Application Blocks (LAB). The classes in the DAABs provide access to the most frequently used features of ADO.NET. Applications can use the DAABs for tasks such as passing data through application layers and returning changed data back to the database. Using DAABs eliminates the need to keep writing the same data access tasks for each new or revised application, so you can spend your time more productively

Applications that use the standard Logging Application Block and design patterns can quickly display the SQL that is generated as part of the ADO.NET Entity Framework data provider.

To use features of the Enterprise Library with your data provider, download Microsoft Enterprise Library 6.0 from <http://msdn.microsoft.com/en-us/library/dn169621.aspx>. You can also download the Enterprise Library documentation, which contains detailed information about using the application blocks.

## Data Access Application Blocks

The Data Access Application Block (DAAB) is designed to allow developers to replace ADO.NET boilerplate code with standardized code for everyday database tasks. The overloaded methods in the Database class can:

- Return Scalar values or XmlReaders
- Determine which parameters it needs and create them
- Involve commands in a transaction

If your application needs to address specific DBMS functionality, you can use the PSQL ADO.NET Data Provider.

### ***When Should You Use the DAAB?***

The DAAB includes a small number of methods that simplify the most common methods of accessing a database. Each method encapsulates the logic required to retrieve the data and manage the connection to the database. You should consider using the application block if your application uses standard data access techniques.

The DAAB is used with ADO.NET, increasing efficiency and productivity when creating applications for ADO.NET. The abstract Database class provides a number of methods, such as `ExecuteNonQuery`, `ExecuteReader`, and `ExecuteScalar`, that are the same as the methods used by the `DbCommand` class, or, if you are using database-specific code, a data provider-specific class such as `PsqlCommand`.

Although using the default DAAB during development is convenient, the resulting application lacks portability. When you use the provider-specific PSQL DAAB implementation, the application includes the PSQL ADO.NET Data Provider's SQL leveling capabilities. You have more flexibility, whether your application needs to access multiple databases, or whether you anticipate a change in your target data source.

### ***Should You Use Generic or Database-specific Classes?***

The application block supplements the code in ADO.NET that allows you to use the same code with different database types. You have two choices when using the DAAB with the ADO.NET data provider:

- The `GenericDatabase` class
- The provider-specific PSQL DAAB implementation

The `GenericDatabase` class option is less suited to applications that need specific control of database behaviors. For portability, the provider-specific solution is the optimal approach.

If your application needs to retrieve data in specialized way, or if your code needs customization to take advantage of features specific to PSQL, using the PSQL ADO.NET Data Provider might suit your needs better.

### ***Configuring the DAAB***

Before you can configure the DAAB for use with your application, you must set up the environment:

- 1 Download and install Microsoft Enterprise Library 6.0 from <http://msdn.microsoft.com/en-us/library/dn169621.aspx>.

- 2 Open the PSQL DAAB project for your data provider, located in *install\_dir*\Enterprise Library\Src\CS\Pervasive.
- 3 Then, compile the project and note the output directory.

Configuring the Data Access Application Block consists of two procedures:

- [Adding a New DAAB Entry](#)
- [Adding the Data Access Application Block to Your Application](#)

### Adding a New DAAB Entry

Now, use the Enterprise Library Configuration Tool to add a new DAAB entry:

- 1 Right-click **Enterprise Library Configuration**, and select **New Application**.
- 2 Right-click **Application Configuration**, then select **New > Data Access Application Block**. The Enterprise Library Configuration window appears.
- 3 In the Name field, type a name for the DAAB connection string.
- 4 In the ConnectionString field, enter a connection string.
- 5 Right-click the field, and select the data provider. For example, select `Pervasive.Data.SqlClient`.
- 6 Right-click **Custom Provider Mappings** and select **New > Provider Mappings**.
- 7 In the Name field, select the data provider name you specified in Step 5.
- 8 Select the **TypeName** field, and then choose the **Browse** button to navigate to the Debug output directory of the PSQL DAAB that you want to build. Then select the **TypeName**. For example, the PSQL **TypeName** is `Pervasive.EnterpriseLibrary.Data.Pervasive.dll`.
- 9 Leave the Enterprise Library Configuration window open for now and do not save this configuration until you complete the following section.

### Adding the Data Access Application Block to Your Application

To add the DAAB to a new or existing application, perform these steps:

- 1 Add two additional References to your Visual Studio solution:
  - Enterprise Library Shared Library
  - Enterprise Library Data Access Application Block
- 2 Add the following directive to your C# source code:
 

```
using Microsoft.Practices.EnterpriseLibrary.Data;
using System.Data;
```
- 3 Rebuild the solution to ensure that the new dependencies are functional.
- 4 Determine the output Debug or Release path location of your current solution, and switch back to the Enterprise Library Configuration window (see [Adding a New DAAB Entry](#)).
- 5 Right-click the connection string under the **Application Configuration** node and select **Save Application**.
- 6 Navigate to the Debug or Release output directories of your current solution, and locate the .exe file of the current solution.

- 7 Click the file name once, and add .config to the name, for example, MyPSQL.config.
- 8 Ensure that Save as type 'All Files' is selected, and select **Save**.
- 9 Using File Explorer, copy the Pervasive.EnterpriseLibrary.Data.Pervasive.dll from the PSQL DAAB directory.
- 10 Place the copy of this DLL into either the Debug or Release output directory of your current solution.

### ***Using the DAAB in Application Code***

Now that you have configured the DAAB, you can build applications on top of this DAAB.

In the example below, we use the DAAB MyPSQL and the DatabaseFactory to generate an instance of a Database object backed by a PSQL data source.

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Practices.EnterpriseLibrary.Data;
using System.Data;

namespace DAAB_Test_App_1
{
    class Program
    {
        static void Main(string[] args)
        {
            Database database = DatabaseFactory.CreateDatabase("MyPSQL");
            DataSet ds = database.ExecuteDataSet(CommandType.TableDirect,
                "SQLCOMMANDTEST_NC_2003SERVER_1");
        }
    }
}
```

The Microsoft Enterprise Library DAAB coding patterns are now at your disposal.

## Logging Application Blocks

Using the Enterprise Library Logging Application Block (LAB) makes it easier to implement common logging functions. The ADO.NET Entity Framework data provider uses the standard Logging Application Block and design patterns, and offer LAB customizations for additional functionality.

### When Should You Use the LAB?

If your applications have a requirement to log information to a database, a message queue, the Windows Event Log, or Windows Management Instrumentation (WMI), the LAB provides this functionality. In particular, the LAB is useful if you need to filter logging messages based on category or priority, if you need to format the messages, or if you need to change the destination of the message without changing the application code.

### Configuring the LAB

A logging capability can be added to an application by adding an entry to an applications configuration file (either app.config or web.config) using the Enterprise Library configuration tool. This tool contains specific instructions in order to enable the Logging Application Block .config file. The tool also contains the necessary AppSetting to enable the LAB.

To enable the Logging Application Block output, set the environment property `Psql_Enable_Logging_Application_Block_Trace` to true. Alternatively, in the app.config file, set the AppSetting property `Psql.EnableLoggingApplicationBlock` to true.

A following C# code snippet shows the loggingConfiguration property of the app.config file.

```
<loggingConfiguration name="Logging Application Block" tracingEnabled="true"
defaultCategory="General" logWarningsWhenNoCategoriesMatch="true">
```

Setting either of these properties to false disables the logging block.

If enabled, the data provider must establish a new LogEntry entry instance for each SQL statement generated by the ADO.NET Entity Framework canonical query tree.

The SQL logged to the Logging Block must be the SQL that is ultimately transmitted to over the wire.

### ►► To configure the Logging Application Block

- 1 Select **Start > Programs > Microsoft patterns and practices > Enterprise Library 6.0 > Enterprise Library Configuration**. The Enterprise Library Configuration window appears.
- 2 Select **File > New Application**.
- 3 Right-click the **Application Configuration** node and select **New > Logging Application Block**.
- 4 Right-click **Category Sources**, and select **New > Category**.
- 5 In the Name pane, select **Name**. Type the name of the new category, and then press ENTER. The following example creates the category name **PSQL Error**.
- 6 From the **SourceLevels** drop-down list, set the logging level for the new category. By default, all logging levels are enabled.

- 7 Right-click the new category and select **New > TraceListener Reference**. A Formatted EventLog TraceListener node is added. From the ReferencedTraceListener drop-down list, select **Formatted EventLog TraceListener**.
- 8 Repeat Step 4 through Step 7 to create the following categories:
  - PSQL Information: Information not related to errors
  - PSQL Command: Enables SQL, Parameter, and DbCommandTree logging
- 9 Select **File > Save Application**. In the Save As dialog, enter a name for your configuration file. By default, the file is saved to C:\Program Files\Microsoft Enterprise Library\Bin\filename.exe.config, where *filename* is the name that you entered for Save As.

### ***Adding a New Logging Application Block Entry***

Now, use the Enterprise Library Configuration Tool to add a new Logging Application Block entry:

- 1 Start **Enterprise Library Configuration**, and select **File > New Application**.
- 2 Right-click **Application Configuration**, then select **New > Logging Application Block**. The Configuration section appears in the right pane.
- 3 In the **TracingEnabled** field, enter `True`.
- 4 Save the Logging application block.

### ***Using the LAB in Application Code***

The LAB that you configured must be added to the app.config or web.config file for your application.

The following settings can be used to enable and configure the data provider's interaction with the LAB.

- `EnableLoggingApplicationBlock`: Enables the Logging Application Block.
- `LABAssemblyName`: Specifies the assembly name to which the Logging Application Block applies.

**Note:** If you use any version of the LAB other than the Microsoft Enterprise Library 4.1 binary release, you must set the `LABAssemblyName`. For example, if you use an older or newer version of the LAB, or a version that you have customized, you must specify a value for `LABAssemblyName`.

- `LABLoggerTypeName`: Specifies the type name for the Logging Application Block.
- `LABLogEntryTypeName`: Specifies the type name for the LogEntry object.

The following code fragment provides an example of a Logging Application Block that could be added to a PSQL data access application.

```
<loggingConfiguration name="Logging Application Block"
tracingEnabled="true"
  defaultCategory=" " logWarningsWhenNoCategoriesMatch="true">
  <listeners>
    <add fileName="rolling.log"
      footer="-----"
      header="-----"
      rollFileExistsBehavior="Overwrite"
      rollInterval="None" rollSizeKB="0"
      timeStampPattern="yyyy-MM-dd"

      listenerDataType="Microsoft.Practices.EnterpriseLibrary.Logging.Configuration.RollingFlatFileTraceListenerData, Microsoft.Practices.EnterpriseLibrary.Logging,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
```



```

        traceOutputOptions="None" filter="All"
type="Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners.RollingFlatFileTraceListener, Microsoft.Practices.EnterpriseLibrary.Logging, Version=6.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
        name="Rolling Flat File Trace Listener" />
    </listeners>
    <formatters>
        <add template="Message: {message}&#xD;&#xA;Category: {category}&#xD;&#xA;Priority: {priority}&#xD;&#xA;EventId: {eventid}&#xD;&#xA;Severity: {severity}&#xD;&#xA;Title: {title}&#xD;&#xA;&#xD;&#xA;"
type="Microsoft.Practices.EnterpriseLibrary.Logging.Formatters.TextFormatter, Microsoft.Practices.EnterpriseLibrary.Logging, Version=6.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
        name="Text Formatter" />
    </formatters>
    <categorySources>
        <add switchValue="All" name="Psql">
            <listeners>
                <add name="Rolling Flat File Trace Listener" />
            </listeners>
        </add>
    </categorySources>
    <specialSources>
        <allEvents switchValue="All" name="All Events" />
        <notProcessed switchValue="All" name="Unprocessed Category" />
        <errors switchValue="All" name="Logging Errors & Warnings">
            <listeners>
                <add name="Rolling Flat File Trace Listener" />
            </listeners>
        </errors>
    </specialSources>
</loggingConfiguration>

```

## Additional Resources

The following sources provide additional information about using Application Blocks:

- The Microsoft patterns & practices Developer Center includes a section on [Application Blocks](#).
- See "Introduction to the Data Access Application Block" for an overview of tasks and applications using Data Access Application Blocks: <http://msdn.microsoft.com/en-us/library/cc309168.aspx>.
- Microsoft patterns & practices Enterprise Library includes an FAQ section on using Logging Application Blocks: <http://entlib.codeplex.com/Wiki/View.aspx?title=EntLib%20FAQ>.
- The Microsoft Channel 9 Enterprise Library FAQs includes some conceptual information about using Application Blocks:  
<http://channel9.msdn.com/wiki/default.aspx/Channel9.EnterpriseLibraryFAQ>.

# *.NET Objects Supported*

## A

---

### *Using the .NET Objects*

ADO.NET 2.0 introduced a new set of classes that provided an additional, more generic interface between applications and data sources.

Predecessors of ADO.NET 2.0 opted for a tighter factoring of data providers into each specific instance of the data provider used by an application. In contrast, ADO.NET 2.0 and higher deliver a set of base classes that permit applications to handle a heterogeneous set of data sources with a single API, much like is done with ODBC and JDBC today. This means that in ADO.NET 2.0 and higher, all data classes derive from base classes, and exist in a specific dedicated namespace, `System.Data.Common`.

The data provider supports:

- [.NET Base Classes](#)
- [Data Provider-Specific Classes](#)
- [PSQL Common Assembly Classes](#)

## **.NET Base Classes**

The interfaces on which ADO.NET 1.0 and ADO.NET 1.1 data providers were built were retained for application compatibility. The base classes of ADO.NET 2.0 and higher provide additional functionality:

- DbCommand
- DbCommandBuilder
- DbConnection
- DbDataAdapter
- DbDataReader
- DBDataPermission
- DbParameter
- DbParameterCollection
- DbConnectionStringBuilder
- DbTransaction

From a day-to-day programming perspective, these classes are provided as abstract implementation. This means they cannot be instantiated directly, but must be used with Provider factories. Each data provider must supply a Factory class, such as PsqlFactory, that derives from the DbFactory class, which contains a set of static methods. Each of these static methods is a factory method for producing an instance of the base classes.

When a data provider is installed, it is registered with the .NET Framework. This allows the common .NET Framework DbFactory to locate any registered data provider that an application requires and provide a common mechanism to establish a connection to a data source. Ultimately, the .NET Framework provides a fully fledged common programming API for ADO.NET data sources.

---

## Data Provider-Specific Classes

The PSQL ADO.NET Data Provider supports all of the .NET public objects. The PSQL ADO.NET Data Provider attaches the provider-specific prefix *Psql* to the public .NET objects, for example, *PsqlCommand*.

The following objects are described:

- [PsqlBulkCopy](#)
- [PsqlBulkCopyColumnMapping](#)
- [PsqlBulkCopyColumnMappingCollection](#)
- [PsqlCommand Object](#)
- [PsqlCommandBuilder Object](#)
- [PsqlConnection Object](#)
- [PsqlConnectionStringBuilder Object](#)
- [PsqlCredential Object](#)
- [PsqlDataAdapter Object](#)
- [PsqlDataReader Object](#)
- [PsqlError Object](#)
- [PsqlErrorCollection Object](#)
- [PsqlException Object](#)
- [PsqlFactory Object](#)
- [PsqlInfoMessageEventArgs Object](#)
- [PsqlParameter Object](#)
- [PsqlParameterCollection Object](#)
- [PsqlTrace Object](#)
- [PsqlTransaction Object](#)

For more information on public objects, refer to the Microsoft .NET Framework Version 2.0 SDK documentation.

### ***PsqlBulkCopy***

The *PsqlBulkCopy* object uses an API pattern similar to the ADO.NET Bulk API patterns, and has no provider-specific properties or methods. For information about the properties and methods supported, refer to the data provider's online help and the Microsoft .NET Framework SDK documentation.

### ***PsqlBulkCopyColumnMapping***

The *PsqlBulkCopyColumnMapping* object uses an API pattern similar to the ADO.NET Bulk API patterns, and has no provider-specific properties or methods. For information about the properties and methods supported, refer to the data provider's online help and the Microsoft .NET Framework SDK documentation.

## **PsqlBulkCopyColumnMappingCollection**

The PsqlBulkCopyColumnMappingCollection object follows an API pattern similar to the Microsoft SqlBulkCopyColumnMappingCollection class, and has no provider-specific properties or methods. For information about the properties and methods supported, refer to the data provider's online help and the Microsoft .NET Framework SDK documentation.

## **PsqlCommand Object**

Table 21 describes the public properties of the PsqlCommand object.

Table 21 Public Properties of the PsqlCommand Object

Property	Description
AddRowID	<p>Adds the ROWID as part of the Select list of a SQL statement.</p> <p>If set to true, the values returned in the ROWID column are used to generate more efficient Insert, Delete, and Update commands when using the PsqlCommandBuilder.</p> <p>If set to false (the initial default), the data provider does not add the ROWID column to the Select list.</p>
ArrayBindCount	<p>Specifies the number of rows of parameters that will be used. The application must set this property before executing a command that uses parameter array binding. The count must equal the length of each of the arrays that is set for each parameter value.</p> <p>The initial default value is 0. The application does not use parameter array binding.</p>
ArrayBindStatus	<p>Returns an array of row status values. This property enables the application to inspect the per row status after executing a command that uses parameter array binding. The property's type is an array of PsqlRowStatus.</p> <p>Parameter array binding is performed as a single atomic operation. This means that if the operation succeeds, every entry will be set to OK; if the operation fails, none of the entries will be set to OK.</p> <p>The PsqlRowStatus enumeration has the following possible values:</p> <ul style="list-style-type: none"><li>• OK. The operation succeeded. All entries are marked as OK.</li><li>• Failed. The operation failed. The data provider assigns this value to all status entries except for the row that caused the failure.</li><li>• SchemaViolation. When an operation fails, the data provider assigns this value to the row that caused the failure.</li></ul>

Table 21 Public Properties of the PsqlCommand Object *continued*

Property	Description
BindByName	<p>Specifies how the data provider processes named parameters when executing a stored procedure. The application can use named parameters or use default values for parameters to the stored procedure.</p> <p>If set to true, the data provider uses the names of parameters supplied in the PsqlParameter objects for the parameter bindings to the PSQL server. See example for <a href="#">CommandText</a>.</p> <p>Alternatively, the user can specify a default value for a named parameter using either of the following methods:</p> <ul style="list-style-type: none"> <li>The application binds the parameters using named parameters, but does not add a PsqlParameter object to the PsqlParameterCollection for the parameters for which the application wants to use the default value.</li> <li>The application sets the Value property of the PsqlParameter object to null. The data provider does not send this parameter to the server and uses the parameter's default value when executing the stored procedure.</li> </ul> <p>When BindByName is set to true and the Parameter Mode connection string option is defined as BindByName or BindByOrdinal, those values defined in the connection string are overridden for the lifetime of the Command object.</p> <p>If set to false (the initial default), the data provider ignores the names for the parameters supplied in the PsqlParameter objects, and assumes that the parameters are in the same order as they were specified in the Create Procedure statement.</p>
CommandText	<p>Gets or sets the text command to run against the data source.</p> <p>When using stored procedures, set CommandText to the name of the stored procedure, for example:</p> <pre>cmd.CommandType = System.Data.CommandType.StoredProcedure; cmd.CommandText = "call EnrollStudent(!!Stud_id!!,!!Class_Id!!, !!GPA!!)"; cmd.BindByName = true; PsqlParameter Class_Id = new PsqlParameter(); Class_Id.Value = 999; Class_Id.ParameterName = "!!Class_Id!!"; PsqlParameter Stud_id = new PsqlParameter(); Stud_id.Value = 1234567890; Stud_id.ParameterName = "!!Stud_id!!"; PsqlParameter GPA = new PsqlParameter(); GPA.Value = 3.2; GPA.ParameterName = "!!GPA!!"; cmd.Parameters.Add(Class_Id); cmd.Parameters.Add(Stud_id); cmd.Parameters.Add(GPA);</pre>

Table 21 Public Properties of the PsqlCommand Object *continued*

Property	Description
CommandTimeout	Gets or sets the wait time before terminating the attempt to execute a command and generating an error.  The initial default is 30 seconds.  We recommend that the application sets the CommandTimeout property to a value that is greater than the largest default timeout value on the server. This ensures that the application gets a more meaningful reply in case of a timeout.
CommandType	Indicates or specifies how the CommandText property is interpreted.  To use stored procedures, set CommandType to StoredProcedure.
Connection	Gets or sets the IDbConnection used by this instance of the IDbCommand.
Parameters	Gets the PsqlParameterCollection.
RowsetSize	Limits the number of rows returned by any query executed on this Command object to the value specified at execute time. The data type for the Read-Write property is signed integer.  Valid values are 0 to 2147483647.  If set to 0 (the initial default), the data provider does not limit the number of rows returned.
StatementCacheBehavior	Retrieves the statement cache behavior or sets the statement cache behavior to one of the values in the PsqlStatementCacheBehavior enumeration. See <a href="#">Enabling Statement Caching</a> for more information.  If set to Implicit (the default) and the Statement Cache Mode connection string option is set to Auto, statement caching occurs implicitly.  If set to Cache and the Statement Cache Mode connection string option is set to ExplicitOnly, statements identified as Cache are cached.  If set to DoNotCache, statement caching does not occur.
Transaction	Gets or sets the transaction in which the PsqlCommand object executes.
UpdatedRowSource	Gets or sets how command results are applied to the DataRow, when used by the Update method of a DataAdapter.  When the UpdateBatchSize property is set to a value other than 1, the UpdatedRowSource property for UpdateCommand, DeleteCommand, and InsertCommand must be set to None or OutputParameters.  If set to None, any returned parameters or rows are ignored.  If set to OutputParameters, output parameters are mapped to the changed row in the DataSet.

Table 22 describes the public methods supported by the PsqlCommand object.

Table 22 Public Methods of the PsqlCommand Object

Method	Description
Cancel	Attempts to cancel the execution of an IDbCommand.
CreateParameter	Creates a new instance of an IDbDataParameter object.
Dispose	Releases the resources used by the component. Overloaded.



Table 22 Public Methods of the PsqlCommand Object *continued*

Method	Description
ExecuteNonQuery	Executes a SQL statement against the PsqlConnection object, and returns the number of rows affected. This method is intended for commands that do not return results.
ExecuteReader	Executes the CommandText against the connection and builds an IDataReader.
ExecuteScalar	Executes the query, and returns the first row of the resultset that the query returns. Any additional rows or columns are ignored.
Prepare	Creates a prepared version of the command on an instance of PSQL.  <b>Note:</b> The Prepare method has no effect in this release of the data provider.

### **PsqlCommandBuilder Object**

Using a PsqlCommandBuilder object can have a negative effect on performance. Because of concurrency restrictions, the PsqlCommandBuilder can generate highly inefficient SQL statements. The end user can often write more efficient update and delete statements than those that the PsqlCommandBuilder generates.

Table 23 describes the public properties supported by the PsqlCommandBuilder object.

Table 23 Public Properties of the PsqlCommandBuilder Object

Property	Description
DataAdapter	Gets or sets the PsqlDataAdapter object associated with this PsqlCommandBuilder.

Table 24 provides the public methods supported for the PsqlCommandBuilder object.

Table 24 Public Methods of the PsqlCommandBuilder Object

Method	Description
DeriveParameters	Populates the specified PsqlCommand object's Parameters collection with parameter information for a stored procedure specified in the PsqlCommand.
GetDeleteCommand	Gets the automatically generated PsqlCommand object required to perform deletions on the database when an application calls Delete on the PsqlDataAdapter.
GetInsertCommand	Gets the automatically generated PsqlCommand object required to perform inserts on the database when an application calls Insert on the PsqlDataAdapter.
GetUpdateCommand	Gets the automatically generated PsqlCommand object required to perform updates on the database when an application calls Update on the PsqlDataAdapter.
QuoteIdentifier	Given an unquoted identifier in the correct catalog case, returns the correct quoted form of that identifier, including properly escaping any embedded quotes in the identifier.
UnquoteIdentifier	Given a quoted identifier, returns the correct unquoted form of that identifier, including properly un-escaping any embedded quotes in the identifier.

## PsqlConnection Object

The PsqlConnection object supports the public properties described in Table 25. Some properties return the values specified for the corresponding connection string option. Unlike the connection string options, the PsqlConnection property names do not include a space.

Table 25 Public Properties of the PsqlConnection Object

Property	Description
ConnectionString	Gets or sets the string used to open a database. See Table 27 for a description of the values you can set.
ConnectionTimeout	<p>Gets the time to wait while trying to establish a connection before the data provider terminates the attempt and generates an error.</p> <p>You can set the amount of time a connection waits to time out by using the ConnectTimeout property or the Connection Timeout connection string option.</p> <p>If connection failover is enabled (the AlternateServers property defines one or more alternate database servers), this property applies to each attempt to connect to an alternate server. If connection retry is also enabled (the Connection Retry Count connection string option is set to an integer greater than 0), the ConnectionTimeout property applies to each retry attempt.</p>
Database	Gets the name of the current database or the database to be used when a connection is open.
Host	Returns the value specified for the Host connection string option. Read-only.
Port	Returns the value specified for the Port connection string option. Read-only.
ServerDSN	Returns the value specified for the Server DSN connection string option. Read-only.
ServerName	Returns the value specified for the Server Name connection string option. Read-only.
ServerVersion	<p>Returns a string containing the version of the PSQL server to which this object is currently connected.</p> <p>If the PsqlConnection object is not currently connected, the data provider generates an InvalidOperationException exception.</p>
State	Gets the current state of the connection.
StatisticsEnabled	<p>Enables statistics gathering.</p> <p>If set to True, enables statistics gathering for the current connection.</p>

Table 26 describes the public methods of PsqlConnection.

Table 26 Public Methods of the PsqlConnection Object

Method	Description
BeginTransaction	<p>Begins a database transaction.</p> <p>When using the overloaded BeginTransaction(IsolationLevel) method, the data provider supports isolation levels ReadCommitted and Serializable. See <a href="#">Isolation Levels</a> for more information.</p>
ChangeDatabase	Changes the current database for an open Connection object.
ClearAllPools	Empties the connection pools for the data provider.

Table 26 Public Methods of the *PsqlConnection* Object *continued*

Method	Description
ClearPool	Clears the connection pool that is associated with connection.  If additional connections associated with the connection pool are in use at the time of the call, they are marked appropriately and are discarded when Close is called on them.
Close	Closes the connection to the database.
CreateCommand	Creates and returns a <i>PsqlCommand</i> object associated with the <i>PsqlConnection</i> .
Dispose	Releases the resources used by the <i>PsqlConnection</i> object.
Open	Opens a database connection with the settings specified by the <i>ConnectionString</i> property of the <i>PsqlConnection</i> object.
ResetStatistics	Resets all values to zero in the current statistics gathering session on the connection.  When the connection is closed and returned to the connection pool, statistics gathering is switched off and the counts are reset.
RetrieveStatistics	Retrieves a set of statistics for a connection that is enabled for statistics gathering (see the <i>StatisticsEnabled</i> property). The set of name=value pairs returned forms a "snapshot in time" of the state of the connection when the method is called.

You can use the *InfoMessage* event of the *PsqlConnection* object to retrieve warnings and informational messages from the database. If the database returns an error, an exception is thrown. Clients that want to process warnings and informational messages sent by the database server should create a *PsqlInfoMessageEventHandler* delegate to register for this event.

The *InfoMessage* event receives an argument of type *PsqlInfoMessageEventArgs* containing data relevant to this event.

### ***PsqlConnectionStringBuilder* Object**

*PsqlConnectionStringBuilder* property names are the same as the connection string option names of the *PsqlConnection.ConnectionString* property. However, the connection string option name can have spaces between the words. For example, the connection string option name Min Pool Size is equivalent to the property name *MinPoolSize*.

The basic format of a connection string includes a series of keyword/value pairs separated by semicolons. The following example shows the keywords and values for a simple connection string for the PSQl ADO.NET Data Provider:

```
"Server DSN=SERVERDEMO;Host=localhost"
```

Table 27 lists the properties that correspond to the connection string options supported by the PSQl data providers, and describes each property.

Table 27 Connection String Properties

Property	Description
AlternateServers	<p>Specifies a list of alternate database servers to which the data provider will try to connect if the primary database server is unavailable. Specifying a value for this connection string option enables connection failover for the data provider.</p> <p>The value you specify must be in the form of a string that defines connection information for each alternate server. You must specify the name or the IP address of each alternate server and the port number, if you are not using the default port value of 1583. The string has the format:</p> <pre>"Host=hostvalue;Port=portvalue[ , ... ]"</pre> <p>For example, the following Alternate Servers value defines two alternate servers for connection failover:</p> <pre>Alternate Servers="Host=AcctServer;Port=1584 , Host=123.456.78.90;Port=1584"</pre> <p>See <a href="#">Using Connection Failover</a> for a discussion of connection failover and information about other connection string options that you can set for this feature.</p>
ConnectionPoolBehavior	<p>{LeastRecentlyUsed   MostRecentlyUsed   LeastFrequentlyUsed   MostFrequentlyUsed}. Specifies the order in which a connection is removed from the connection pool for reuse, based on how frequently or how recently the connection has been used.</p> <p>If set to MostRecentlyUsed, the data provider uses a Last In First Out (LIFO) approach to return the connection that was returned to the pool most recently.</p> <p>If set to LeastRecentlyUsed (the initial default), the data provider uses a First In First Out (FIFO) approach to return the connection that has been in the pool for the longest time. This value ensures a balanced use of connections in the pool.</p> <p>If set to MostFrequentlyUsed, the data provider returns the connection with the highest use count. This value enables applications to give preference to the most seasoned connection.</p> <p>If set to LeastFrequentlyUsed, the data provider returns the connection with the lowest use count. This value ensures a balanced use of connections in the pool.</p>
ConnectionReset	<p>{True   False}. Specifies whether a connection that is removed from the connection pool for reuse by an application will have its state reset to the initial configuration settings of the connection. Resetting the state impacts performance because the new connection must issue additional commands to the server, for example, resetting the current database to the value specified at connect time.</p> <p>If set to False (the initial default), the data provider does not reset the state of the connection.</p>
ConnectionRetryCount	<p>Specifies the number of times the data provider tries to connect to the primary server, and, if specified, the alternate servers after the initial unsuccessful attempt.</p> <p>The value can be any integer from 0 to 65535.</p> <p>If set to 0 (the initial default), the data provider does not try to reconnect after the initial unsuccessful attempt.</p> <p>If a connection is not established during the retry attempts, the data provider returns an error that is generated by the last server to which it attempted to connect.</p> <p>This option and Connection Retry Delay, which specifies the wait interval between attempts, can be used in conjunction with connection failover. See <a href="#">Using Connection Failover</a> for a discussion of connection failover and for information about other connection string options that you can set for this feature.</p>

Table 27 Connection String Properties *continued*

Property	Description
ConnectionRetryDelay	<p>Specifies the number of seconds the data provider waits after the initial unsuccessful connection attempt before retrying a connection to the primary server, and, if specified, the alternate servers.</p> <p>The value can be any integer from 0 to 65535.</p> <p>The initial default is 3 (seconds). If set to 0, there is no delay between retrying the connection.</p> <p><b>Note:</b> This option has no effect unless the Connection Retry Count connection string option is set to an integer value greater than 0.</p> <p>This option and the Connection Retry Count connection string option, which specifies the number of times the data provider attempts to connect after the initial attempt, can be used in conjunction with connection failover. See <a href="#">Using Connection Failover</a> for a discussion of connection failover and for information about other connection string options that you can set for this feature.</p>
ConnectionTimeout	<p>Specifies the number of seconds after which the attempted connection to the server will fail if not yet connected. If connection failover is enabled, this option applies to each connection attempt.</p> <p>If set to 0, the data provider never times out on a connection attempt.</p> <p>The initial default is 15 seconds.</p>
DatabaseName	<p>Specifies the internal name of the database to which you want to connect. Use this option when you need to connect to a PSQL data source for which a ServerDSN has not been defined.</p> <p>The default value is an empty string.</p> <p><b>Note:</b> Do not combine the Database Name and Server DSN connection string options in a connection string.</p> <p>Alias: DBQ</p>
EnableIPv6	<p>Provides backward compatibility for connecting to the PSQL Server using an IPv4 address.</p> <p>If set to True, a client with IPv6 protocol installed can identify itself to the server using either an IPv4 address or an IPv6 address.</p> <p>If set to False, the clients runs in the backward compatibility mode. The client always identifies itself to the server using an IPv4 address.</p> <p>The default value for 4.0 is set to True.</p> <p>For more information about IPv6 formats, see <a href="#">IPv6</a> in <i>Getting Started With PSQL</i>.</p>
EnableTrace	<p>{0   1}. Specifies whether tracing is enabled.</p> <p>If set to 0 (the initial default), tracing is not enabled.</p>
Encoding	<p>Specifies an IANA name or Windows code page number to be used for translating the string data stored in the database.</p> <p>The default value is an empty string; the current Windows Active Code Page (ACP) is used.</p>

Table 27 Connection String Properties continued

Property	Description
Encrypt	<p>{If Needed   Always   Never}. Determines whether the data provider uses Encrypted Network Communications, also known as wire encryption.</p> <p>If set to Always, the data provider uses encryption, or, if the server does not allow wire encryption, returns an error.</p> <p>If set to Never, the data provider does not use encryption and returns an error if wire encryption is required by the server.</p> <p>If set to IfNeeded (the default), the data provider uses the default setting on the server.</p> <p><b>Note:</b> This option may adversely affect performance because of the additional overhead, mainly CPU usage, required to encrypt and decrypt data.</p>
Encryption	<p>{Low   Medium   High}. Determines the minimum level of encryption allowed by the data provider.</p> <p>The initial default is Medium.</p> <p>The meaning of these values depends on the encryption module used. With the default encryption module, these values correspond to 40-, 56-, and 128-bit encryption, respectively.</p>
Enlist	<p>{True   False}. Specifies whether the data provider automatically attempts to enlist the connection in creating the thread's current transaction context.</p> <p><b>Note:</b> Because PSQL does not support distributed transactions, any attempt to enlist the connection in the thread's current transaction context will fail.</p> <p>If set to False, the data provider does not automatically attempt to enlist the connection.</p> <p>If set to True (the initial default), the data provider returns an error on the connection if a current transaction context exists. If a current transaction context does not exist, the data provider raises a warning.</p>
Host	<p>Specifies the name or the IP address of the PSQL database server to which you want to connect. For example, you can specify a server name such as <code>Accountingserver</code>. Or, you can specify an IP address such as <code>199.226.22.34</code> (IPv4) or <code>1234:5678:0000:0000:0000:0000:9abc:def0</code> (IPv6).</p> <p>The initial default value is an empty string.</p> <p>Alias: Server, Server Name</p>

Table 27 Connection String Properties continued

Property	Description
InitialCommandTimeout	<p>Specifies the default wait time (timeout in seconds) before the data provider terminates the attempt to execute the command and generates an error. This option provides the same functionality as the PsqlCommand object's CommandTimeout property without the need to make changes to the application code. Subsequently, an application can use the CommandTimeout property to override the Initial Command Timeout connection string option.</p> <p>The initial default value is 30. If set to 0, the query never times out.</p> <p>For example, in the following C# code fragment, the connection string instructs the application to wait 60 seconds before terminating the attempt to execute the command. The application then specifies a CommandTimeout of 45 seconds, which overrides the value set in the connection string:</p> <pre>PsqlCommand command = new PsqlCommand(); PsqlConnection conn = new PsqlConnection("...; Initial Command     Timeout=60; ..."); conn.Open(); command.Connection = connection; // command.CommandTimeout returns 60; command.CommandTimeout = 45; // command.CommandTimeout returns 45</pre> <pre>command = new PsqlCommand(); command.CommandTimeout = 45; command.Connection = conn; // command.CommandTimeout still returns 45</pre> <p><b>Note:</b> Set the Initial Command Timeout option to a value that is greater than the largest default deadlock detection and timeout value on the server. This ensures that the application receives a more meaningful reply in case of a timeout.</p>
InitializationString	<p>Specifies one statement that will be issued immediately after connecting to the database to manage session settings.</p> <p>The initial default is an empty string.</p> <p>Example: To handle CHAR columns that are padded with NULLs, set the value to: Initialization String=SET ANSI PADDING ON</p> <p><b>Note:</b> If the statement fails to execute for any reason, the connection to the PSQL server fails. The data provider throws an exception that contains the errors returned from the server.</p>
LoadBalanceTimeout	<p>Specifies the number of seconds to keep connections in a connection pool. The pool manager periodically checks all pools, and closes and removes any connection that exceeds its lifetime. The <b>MinPoolSize</b> option can cause some connections to ignore this value. See <a href="#">Removing Connections from a Pool</a> for a discussion of connection lifetimes.</p> <p>The value can be any integer from 0 to 65335.</p> <p>If set to 0, (the initial default), the connections have the maximum timeout.</p> <p>Alias: Connection Lifetime</p>

Table 27 Connection String Properties continued

Property	Description
LoadBalancing	<p>{True   False}. Determines whether the data provider uses client load balancing in its attempts to connect to primary and alternate database servers. The list of alternate servers is specified by the Alternate Servers connection option.</p> <p>If set to True, the data provider attempts to connect to the database servers in random order. See <a href="#">Using Client Load Balancing</a> for more information about load balancing.</p> <p>If set to False (the initial default), client load balancing is not used and the data provider connects to each server based on its sequential order (primary server first, then, alternate servers in the order they are specified).</p> <p><b>Note:</b> This option has no effect unless alternate servers are defined for the Alternate Servers connection string option.</p> <p>The Load Balancing connection string option is an optional setting that you can use in conjunction with connection failover. See <a href="#">Using Connection Failover</a> for more information for a discussion of connection failover and for information about other connection options that you can set for this feature.</p>
MaxPoolSize	<p>Specifies the maximum number of connections within a single pool. When the maximum number is reached, no additional connections can be added to the connection pool. The Max Pool Size Behavior connection string option can cause some connections to ignore this value temporarily.</p> <p>The value can be any integer from 1 to 65335.</p> <p>The initial default is 100.</p>
MaxPoolSizeBehavior	<p>{SoftCap   HardCap}. Specifies whether the data provider can exceed the number of connections specified by the Max Pool Size option when all connections in the connection pool are in use.</p> <p>If set to SoftCap, the number of connections created can exceed the value set for Max Pool Size, but the number of connections pooled does not. When the maximum connections for the pool are in use and a connection request is received, the data provider creates a new connection. If a connection is returned to a pool that is full and contains idle connections, the pooling mechanism selects a connection to be discarded so the connection pool never exceeds the Max Pool Size.</p> <p>If set to HardCap (the initial default), when the maximum number of connections allowed in the pool are in use, any new connection requests wait for an available connection until the Connection Timeout is reached.</p>
MaxStatementCacheSize	<p>Specifies the maximum number of statements that can be held in the statement cache. The value can be 0, or any integer greater than 1.</p> <p>Setting the cache size to 0 disables statement caching.</p> <p>The initial default is 10.</p> <p>In most cases, using statement caching results in improved performance. See the "Performance Considerations" section in the chapter for your data provider for information on how this option can affect performance.</p>



Table 27 Connection String Properties *continued*

Property	Description
MinPoolSize	<p>Specifies the number of connections created when a connection pool is initialized and the minimum number of connections that will be kept in the pool. The connection pool retains this number of connections even when some connections have exceeded their <b>LoadBalanceTimeout</b> value.</p> <p>The value can be any integer from 0 to 65335.</p> <p>If set to 0 (the initial default), when the connection is closed and sent to the connection pool, the pool retains only the original connection used to create the pool.</p> <p>If set to an integer from 1 to 65535, the specified number of duplicates of the connection are placed in the pool.</p> <p>See the "Performance Considerations" section in the chapter for your data provider for information on how pooling can affect performance.</p>
ParameterMode	<p>Specifies the behavior of native parameter markers and binding. This allows applications to reuse provider-specific SQL code and simplifies migration to the ADO.NET data provider.</p> <p>If set to ANSI (the initial default), the ? character is processed as a parameter marker and bound as ordinal. Applications can toggle the behavior of the BindByName property on a per-command basis.</p> <p>If set to BindByOrdinal, native parameter markers are used and are bound as ordinal for stored procedures and standard commands.</p> <p>If set to BindByName, native parameter markers are used and are bound by name for stored procedures and standard commands.</p> <p><b>Note:</b> This option is not supported for the ADO.NET Entity Framework data provider.</p>
Password	<p>Specifies a case-insensitive password used to connect to your PSQL database. A password is required only if security is enabled on your database. If so, contact your system administrator to get your password.</p> <p>Alias: PWD</p>
PersistSecurityInfo	<p>{True   False}. Specifies whether to display security information in clear text in the ConnectionString property.</p> <p>If set to True, the value of the Password connection string option is displayed in clear text.</p> <p>If set to False (the initial default), the data provider does not display the password in the connection string.</p>
Pooling	<p>{True   False}. Specifies whether connections are pooled. See <a href="#">Using Connection Pooling</a> for more information about connection pooling.</p> <p>If set to True (the initial default), connection pooling is enabled.</p> <p>See the "Performance Considerations" section in the chapter for your data provider for information on how pooling can affect performance.</p>
Port	<p>Specifies the TCP port of the listener running on the PSQL database.</p> <p>The default port number is 1583.</p>

Table 27 Connection String Properties continued

Property	Description
PVTranslate	<p>{Auto   Nothing}. Specifies whether the client should negotiate a compatible encoding with the server.</p> <p>If set to Auto (the default for the 4.2 provider), the data provider will set the Encoding connection property to the database code page. In addition, SQL query text will be sent to the engine using UTF-8 encoding instead of the data encoding. This preserves NCHAR string literals in the query text.</p> <p>If set to Nothing (the default for the 4.0 provider), the setting for the Encoding connection property is used.</p>
SchemaCollectionTimeout	<p>Specifies the number of seconds after which an attempted schema collection operation fails if it is not yet completed.</p> <p>If set to 0, the data provider never times out on a schema collection operation attempt.</p> <p>The initial default is 120.</p>
SchemaOptions	<p>Specifies additional database metadata that can be returned. By default, the data provider prevents the return of some available performance-expensive database metadata to optimize performance. If your application needs this database metadata, specify the name or hexadecimal value of the metadata.</p> <p>This option can affect performance.</p> <p>See Table 28 for the name and hexadecimal value of the database metadata that the data provider can add.</p> <p>If set to ShowColumnDefaults or 0x04, column defaults are returned.</p> <p>If set to ShowParameterDefaults or 0x08, column defaults are returned.</p> <p>If set to FixProcedureParamDirection or 0x10, procedure definitions are returned.</p> <p>If set to ShowProcedureDefinitions or 0x20, procedure definitions are returned.</p> <p>If set to ShowViewDefinitions or 0x40, view definitions are returned.</p> <p>If set to ShowAll or 0xFFFFFFFF (the initial default), all database metadata is returned.</p> <p>For example, to return descriptions of procedure definitions, specify Schema Options=ShowProcedureDefinitions or Schema Options=0x20.</p> <p>To show more than one piece of the omitted database metadata, specify either a comma-separated list of the names, or the sum of the hexadecimal values of the column collections that you want to restrict. For example, to return descriptions of procedure definitions and view definitions (hexadecimal values 0x20 and 0x40, respectively), specify Schema Options=ShowProcedureDefinitions, ShowViewDefinitions or Schema Options=0x60.</p> <p><b>Note:</b> This connection string option may adversely affect performance. See the "Performance Considerations" section in the chapter for your data provider for more information.</p>
ServerDSN	<p>Specifies the name of the data source on the server, such as Server DSN=SERVERDEMO.</p> <p>The default value is DEMODATA.</p> <p><b>Note:</b> Do not combine the Database Name and Server DSN connection string options in a connection string.</p>

Table 27 Connection String Properties continued

Property	Description
StatementCacheMode	<p>Specifies the statement cache mode. The statement cache mode controls the behavior of the statement cache. Statements can be cached automatically or only cached when a command is explicitly marked.</p> <p>If set to Auto, statement caching is enabled for statements marked as Implicit by the PsqlCommand property StatementCacheBehavior. These commands have a lower priority than that of explicitly marked commands, that is, if the statement pool reaches its maximum number of statements, the statements marked Implicit are removed from the statement pool first to make room for statements marked Cache.</p> <p>If set to ExplicitOnly (the initial default), only statements that are marked Cache by the StatementCacheBehavior property are cached.</p> <p>In most cases, enabling statement caching results in improved performance. See <a href="#">Performance Considerations</a> for information on how this option can affect performance of the ADO.NET data provider.</p> <p><b>Note:</b> This option is not supported for the ADO.NET Entity Framework data provider.</p>
Timestamp	<p>{DateTime   String}. Specifies whether PSQL timestamps are stored and retrieved as strings in the data provider.</p> <p>If set to DateTime or not defined (the default), the data provider maps timestamps to the .NET DateTime type. This setting may be appropriate when native precision is required, for example, when using the PsqlCommandBuilder with TimeStamp.</p> <p>If set to String, the timestamps are returned as strings. The data provider maps PSQL timestamps to the .NET String type.</p>
TimeType	<p>{DateTime   TimeSpan}. Specifies whether PSQL Times are retrieved as Timespan or DateTime in the ADO.NET data provider.</p> <p>If set to As DateTime, the data provider maps the SQL type TIME to the .NET type System.DateTime.</p> <p>If set to As TimeSpan, the data provider maps the SQL type TIME to the .NET type System.DateTimespan.</p>
TraceFile	<p>Specifies the path and file name of the trace file.</p> <p>The initial default is an empty string. If the specified file does not exist, the data provider creates it.</p>
UseCurrentSchema	This connection string option is not supported. Setting it will cause the data provider to throw an exception.
UserID	<p>Specifies the default PSQL user name used to connect to your PSQL database.</p> <p>Alias: UID</p>

Table 28 lists the name and the hexadecimal value of the column collection that the data provider will omit from the returned data. To specify multiple values, specify a comma-separated list of the names, or the sum of the hexadecimal values of the column collections that you want to return.

Table 28 PSQL Column Collections

Name	Hex Value	Collection/Column
ShowColumnDefaults	0x04	Columns/COLUMN_DEFAULT
ShowParameterDefaults	0x08	ProcedureColumns//PARAMETER_DEFAULT

Table 28 PSQL Column Collections continued

Name	Hex Value	Collection/Column
FixProcedureParamDirection	0x10	ProcedureColumns/PARAMETER_TYPE
ShowProcedureDefinitions	0x20	Procedures/PROCEDURE_DEFINITION
ShowViewDefinitions	0x40	Views/VIEW_DEFINITION
ShowAll	0x7F	All
1 COLUMN_HAS_DEFAULT is always reported with a value of null.		

The PsqlConnectionStringBuilder object has no provider-specific methods. For information about the methods supported, refer to the data provider's online help and the Microsoft .NET Framework SDK documentation.

### PsqlCredential Object

The PsqlCredential object provides a secure way to log in using PSQL Server authentication. PsqlCredential is comprised of a user ID and a password recognized by PSQL Server.

The password in a PsqlCredential object is of type SecureString, unlike Connection String where the password is unsecure until the provider reads it and converts it to SecureString. The password is handled in a secure way without writing it to memory. The string that stores the password is cleaned after use.



**Note** Use PsqlCredential only when the authentication method requires the user ID and password. Also, if you are using Kerberos or Client, you should not use PsqlCredential. Finally, the Connection String should not include the user ID and password when the Credential object is being used.

The following code snippet shows how the PsqlCredential class can be used. The method used to convert a string into a SecureString in this example is one of many possible methods.

```
PsqlConnection con = null;
PsqlCredential lobjCredential = null;
string userId = "ABCD";
SecureString password = ConvertToSecureString("XYXYX");
private static SecureString ConvertToSecureString(string value)
{
    var securePassword = new SecureString();
    foreach (char c in value.ToCharArray())
        securePassword.AppendChar(c);
    securePassword.MakeReadOnly();
    return securePassword;
}
try
{
    lobjCredential = new PsqlCredential(userId, password);
    con = new PsqlConnection("Host=nc-xxx;Port=xxxx;Database Name=xxxx" ,
        lobjCredential);
    con.Open();
    Console.WriteLine("Connection Successfully Opened...");
    con.Close();
}
catch (Exception e)
```

```

{
    Console.WriteLine(e.Message)
}
finally
{
    if (null != con)
    {
        con.Close();
        con = null;
    }
    if (null != lobjCredential)
    {
        lobjCredential = null;
    }
}

```

The following table lists the provider-specific implementation of the public properties of the PsqlCredential object.

*Table 29 Public Properties of the PsqlCredential Object*

Property	Description
User ID	Returns the user ID component of the PsqlCredential object. Uses String data type. NULL and empty are invalid values.
Password	Returns the password component of the PsqlCredential object. Uses SecureString data type. NULL is an invalid value.

If you use the PsqlCredential object while opening the connection and want to use the same pooled connection, you need to reference the same PsqlCredential object so that the same connection is fetched from the available connection pool.

If you create a new credential object for each connection, the driver treats them separately and puts them into different connection pools, even if the same user ID and password are used.

### ***PsqlDataAdapter Object***

The PsqlDataAdapter object uses PsqlCommand objects to execute SQL commands on the PSQL database, to load the DataSet with data, and to reconcile the changed data in the DataSet to the database.

Table 30 describes the public properties of PsqlDataAdapter.

Table 30 Public Properties of the *PsqlDataAdapter* Object

Property	Description
UpdateBatchSize	<p>Gets or sets a value that specifies the number of commands that can be executed in a batch.</p> <p>If your application uses disconnected DataSets and updates those DataSets, you can positively influence performance by setting this property to a value greater than 1. By default, the data provider attempts to use the largest batch size possible. However, this may not equate to optimal performance for your application. Set the value based on the number of rows you typically update in the DataSet. For example, if you are updating less than 50 rows, a suggested setting for this property is 25.</p> <p>If set to 0, the PsqlDataAdapter uses the largest batch size the data source can support. The UpdatedRowSource property for the InsertCommand, UpdateCommand, and DeleteCommand must be set to None or OutputParameters.</p> <p>If set to 1, batch updating is disabled.</p> <p>If set to a value greater than 1, the specified number of commands are executed in a batch. The UpdatedRowSource property for the InsertCommand, UpdateCommand, and DeleteCommand must be set to None or OutputParameters.</p>
DeleteCommand	Gets or sets a SQL statement for deleting records from the PSQL data source.
InsertCommand	Gets or sets a SQL statement used to insert new records into the PSQL database.
SelectCommand	Gets or sets a SQL statement used to select records in the PSQL database.
UpdateCommand	Gets or sets a SQL statement used to update records in the data source.

## ***PsqlDataReader* Object**

The PsqlDataReader object is a forward-only cursor that retrieves read-only records from a database. Performance is better than using PsqlDataAdapter, but the result set cannot be modified.

Table 31 describes the public properties of PsqlDataReader.

Table 31 Public Properties of the *PsqlDataReader* Object

Property	Description
Depth	Gets a value indicating the depth of nesting for the current row.
HasRows	Gets a value indicating whether the PsqlDataReader contains one or more rows.
IsClosed	Gets a value indicating whether the data reader is closed.
RecordsAffected	Gets the number of rows that were changed, inserted, or deleted by execution of the SQL statement.

Table 32 describes some of the public methods of the PsqlDataReader.

Table 32 Supported Methods of the *PsqlDataReader* Object

Method	Description
Close	Closes the DataReader. Always call the Close method when you finish using the DataReader object.
GetSchemaTable	Returns a DataTable that describes the column metadata of the PsqlDataReader. See Table 50 for more information.

Table 32 Supported Methods of the *PsqlDataReader* Object

Method	Description
NextResult	Advances the data reader to the next result when reading the results of batch SQL statements.
Read	Advances the IDataReader to the next result.

### ***PsqlError* Object**

The *PsqlError* object collects information relevant to errors and warnings generated by the PSQL server.

Table 33 describes the public properties supported by *PsqlError*.

Table 33 Public Properties of the *PsqlError* Object

Property	Description
Message	Gets the error message text returned from the PSQL server.
Number	Gets the error number returned from the PSQL server.
SQLState	Gets the string representation of the SQLState when an exception is thrown by the PSQL data provider, or 0 if the exception is not applicable to the error. This property is read-only. Note: For all of the ADO.NET client error messages which do not have any SQLstate information, S1000 is used as the default SQLState.

### ***PsqlErrorCollection* Object**

The *PsqlErrorCollection* object is created by a *PsqlException* to contain all the errors generated by the PSQL server.

Table 34 provides the public provider-specific properties supported for the *PsqlErrorCollection* object. For information about other properties and methods supported, refer to the data provider's online help and the Microsoft .NET Framework SDK documentation.

Table 34 Public Properties of the *PsqlErrorCollection* Object

Property	Description
Count	Gets the number of <i>PsqlError</i> objects generated by the PSQL server.

The *PsqlErrorCollection* object supports the public methods described in Table 35.

Table 35 Public Methods of the *PsqlErrorCollection* Object

Method	Description
CopyTo	Copies the <i>PsqlError</i> objects from the <i>ErrorCollection</i> to the specified array.
GetEnumerator	Returns the <i>IEnumerator</i> interface for a given array.

## PsqlException Object

Provider-specific exceptions are derived directly from the System.Data interface. Only the public properties and methods, for example, the Message property, are directly available on the System.Exception object in a generic sense. The SQLState and Number properties are only accessible through provider-specific code or by using reflection.

ADO.NET 2.0 introduced a new property on the DbException class, Data. This property returns a collection of key-value pair tuples that provide additional user-defined information about an exception. The PSQL ADO.NET Data Provider gets a collection of key/value pairs such as SQLState, Number, and ErrorPosition.

The Psql.Data.SqlClient prefix is applied to each key, for example:

```
Psql.Data.SqlClient.Data["SQLState"] = 28000;
```

The properties described in Table 36 apply to the last error generated, if multiple errors exist. The application should check the Count property of the PsqlErrorCollection returned in the Errors property of this object to determine whether multiple errors occurred. See [PsqlErrorCollection Object](#) for more information.

Table 36 Public Properties of the PsqlException Object

Property	Description
Errors	Gets or sets a PsqlErrorCollection of one or more PsqlError objects.
Message	Specifies the error message text that is returned from the PSQL server.
Number	Gets or sets the number returned from the PSQL server.
SQLState	Returns the string representation of the SQLState when an exception is thrown by the PSQL data provider, or 0 if the exception is not applicable to the error. This property is read-only.

## PsqlFactory Object

Provider Factory classes allow users to program to generic objects. Once instantiated from DbProviderFactory, the factory generates the proper type of concrete class.

Table 37 lists the static methods used to accommodate choosing the PSQL ADO.NET Data Provider and instantiating its DbProviderFactory.

Table 37 Methods of the PsqlFactory Object

Method	Description
CreateCommand	Returns a strongly typed DbCommand instance.
CreateCommandBuilder	Returns a strongly typed DbCommandBuilder instance.
CreateConnection	Returns a strongly typed DbConnection instance.
CreateConnectionStringBuilder	Returns a strongly typed DbConnectionString instance.
CreateDataAdapter	Returns a strongly typed DbDataAdapter instance.



Table 37 *Methods of the PsqlFactory Object continued*

Method	Description
CreateDataSourceEnumerator	Returns a strongly typed PsqlDataSourceEnumerator instance.
CreateParameter	Returns a strongly typed DbParameter instance.

### ***PsqlInfoMessageEventArgs Object***

The PsqlInfoMessageEventArgs object is passed as an input to the PsqlInfoMessageEventHandler and contains information relevant to a warning generated by the PSQL server.

Table 38 describes the public properties for PsqlInfoMessageEventArgs.

Table 38 *Public Properties of PsqlInfoMessageEventArgs*

Property	Description
Errors	Specifies a PsqlErrorCollection that contains a collection of warnings sent from the PSQL server. See <a href="#">PsqlErrorCollection Object</a> for more information.
Message	Returns the text of the last message returned from the PSQL server. The application should check the Count property of the PsqlErrorCollection returned in the Errors property of this object to determine whether multiple warnings occurred.

### ***PsqlParameter Object***

The PsqlParameter object represents a parameter to a PsqlCommand object.

Table 39 describes the public properties for PsqlParameter.

Table 39 *Public Properties of the PsqlParameter Object*

Property	Description
ArrayBindStatus	Determines whether any values in the array of PsqlParameterStatus entries should be bound as null. The PsqlParameterStatus enumeration contains the entry NullValue.  When this property is not set, then no values are null. The length of the array should match the amount specified by the PsqlCommand object's ArrayBindCount property (see <a href="#">PsqlCommand Object</a> ).  The initial default is null.
DbType	Gets or sets the DbType of the parameter.
Direction	Gets or sets a value that indicates whether the parameter is input-only, output-only, bidirectional, or the return value parameter of a stored procedure.
IsNullable	Gets or sets a value that indicates whether the parameter accepts null values.
ParameterName	Gets or sets the name of the PsqlParameter object.
Precision	Gets or sets the maximum number of digits used to represent the Value property.
Scale	Gets or sets the number of decimal places to which the Value property is resolved.
Size	Gets or sets the maximum size, in bytes, of the data within the column.

Table 39 Public Properties of the *PsqlParameter* Object *continued*

Property	Description
SourceColumn	Gets or sets the name of the source column that is mapped to the DataSet and used for loading or returning the Value property.
SourceColumnNullMapping	Sets or gets a value that indicates whether the source column is nullable.
SourceVersion	Gets or sets the DataRowVersion to use when loading the Value property.
Value	<p>Gets or sets the value of the parameter.</p> <p>The initial default value is null.</p> <p><b>Note:</b> When array binding is enabled (see the ArrayBindCount property of the <a href="#">PsqlCommand Object</a>), this property is specified as an array of values. Each array's length must match the value of the ArrayBindCount property. When specifying the array's values for binary type columns, the data will actually be specified as byte[]. This is an array of arrays of bytes. The data provider anticipates a "jagged" array as such when using parameter array binding with parameters.</p> <p>If set to null for a stored procedure parameter, the data provider does not send the parameter to the server. Instead, the default value for the parameter is used when executing the stored procedure.</p>

## ***PsqlParameterCollection* Object**

The PsqlParameterCollection object represents a collection of parameters relevant to a PsqlCommand, and includes their mappings to columns in a DataSet.

Table 40 describes the public properties for PsqlParameterCollection.

Table 40 Public Properties of *PsqlParameterCollection*

Property	Description
Count	Gets the number of PsqlParameter objects in the collection.
IsFixedSize	Gets a value that indicates whether the PsqlParameterCollection has a fixed size.
IsReadOnly	Gets a value that indicates whether the PsqlParameterCollection is read-only.
IsSynchronized	Gets a value that indicates whether the PsqlParameterCollection is thread-safe.
Item	Gets the parameter at the specified index. In C#, this property is the indexer for the IDataParameterCollection class.
SynchRoot	Gets the object used to synchronize access to the PsqlParameterCollection.

Table 41 provides the public methods for PsqlParameterCollection.

Table 41 Public Methods of the *PsqlParameterCollection* Object

Method	Description
Contains	Gets a value that indicates whether a parameter in the collection has the specified source table name.

Table 41 Public Methods of the PsqlParameterCollection Object

Method	Description
IndexOf	Gets the location of the IDataParameter within the collection.
RemoveAt	Removes the IDataParameter from the collection.

## PsqlTrace Object

The PsqlTrace object is created by the application to debug problems during development. Setting the properties in the PsqlTrace object overrides the settings of the environment variables. For your final application, be sure to remove references to the PsqlTrace object.

The following code fragment creates a Trace object named MyTrace.txt. All subsequent calls to the data provider will be traced to that file.

```
PsqlTrace MyTraceObject = new PsqlTrace();
MyTraceObject.TraceFile="C:\\MyTrace.txt";
MyTraceObject.RecreateTrace = 1;
MyTraceObject.EnableTrace = 1;
```

Table 42 describes the public properties for the PsqlTrace object.

Table 42 Public Properties of PsqlTrace

Property	Description
EnableTrace	If set to 1 or higher, enables tracing. The initial default value is 0. Tracing is disabled.
RecreateTrace	If set to 1, recreates the trace file each time the application restarts. If set to 0 (the initial default), the trace file is appended
TraceFile	Specifies the path and name of the trace file. The initial default is an empty string. If the specified file does not exist, the data provider creates it.
<b>Note:</b> Setting EnableTrace starts the tracing process. Therefore, you must define the property values for the trace file before setting EnableTrace. Once the trace processing starts, the values of TraceFile and RecreateTrace cannot be changed.	

Table 43 describes the public methods for PsqlTrace.

Table 43 Public Methods of the PsqlTrace Object

Method	Description
DumpFootprints	Displays the footprint of all source files in a data provider.

## PsqlTransaction Object

Table 44 describes the public properties of PsqlTransaction.

Table 44 Public Properties of the PsqlTransaction Object

Property	Description
Connection	Specifies the PsqlConnection object associated with the transaction. See <a href="#">PsqlConnection Object</a> for more information.
IsolationLevel	Defines the isolation level for the entire transaction. If the value is changed, the new value is used at execution time.

Table 45 describes the public methods of the PsqlTransaction object.

Table 45 Public Methods of the PsqlTransaction Object

Method	Description
Commit	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
Rollback	Cancels modifications made in a transaction before the transaction is committed.

## PSQL Common Assembly Classes

The PSQL ADO.NET Data Provider supports additional classes that provide enhanced functionality, such as bulk load. All classes are created with 100% managed code. The following classes are provided in the Pervasive.Data.Common.dll assembly:

- [CsvDataReader](#)
- [CsvDataWriter](#)
- [DbBulkCopy](#)
- [DbBulkCopyColumnMapping](#)
- [DbBulkCopyColumnMapping](#)

The classes used for bulk loading implement the generic programming model. They can be used with any DataDirect Technologies ADO.NET data provider or ODBC driver that supports PSQL Bulk Load and any supported database.

### **CsvDataReader**

The CsvDataReader class provides the DataReader semantics for the CSV file format defined by PSQL Bulk Load.

Table 46 lists the public properties for the CsvDataWriter object.

*Table 46 Public Properties of the CsvDataReader Object*

Property	Description
BulkConfigFile	<p>Specifies the CSV bulk configuration file that is produced when the WriteToFile method is called. A bulk load configuration file defines the names and data types of the columns in the bulk load data file in the same way as the table or result set from which the data was exported. A bulk load configuration file is supported by an underlying XML schema.</p> <p>The path may be fully qualified. Otherwise, the file is considered to exist in the current working directory.</p> <p><b>Note:</b> This property can only be set prior to the Open() call and after the Close() call; otherwise, an InvalidOperationException is thrown.</p>
BulkFile	<p>Specifies the bulk load data file that contains the CSV-formatted bulk data. The file name is used for writing (exporting) and reading (importing) the bulk data. If the file name provided does not contain an extension, the ".csv" extension is assumed.</p> <p>The path may be fully qualified. Otherwise, the file is considered by default to exist in the current working directory. An InvalidOperationException is thrown if this value is not set.</p> <p><b>Note:</b> This property can only be set prior to the Open() call and after the Close() call; otherwise, an InvalidOperationException is thrown.</p>
ReadBufferSize	<p>Specifies the size of the read buffer when using bulk load to import data from a data source.</p> <p>The initial default is 2048 KB.</p> <p>Values equal to or less than zero cause a System.ArgumentOutOfRangeException to be thrown.</p>

Table 46 Public Properties of the CsvDataReader Object continued

Property	Description
RowOffset	<p>Specifies the row from which to start the bulk load read. The RowOffset is relative to the first (1) row.</p> <p>The initial default is 1.</p> <p>Values equal to or less than zero cause a System.ArgumentOutOfRangeException to be thrown.</p> <p><b>Note:</b> This property can only be set prior to the Open() call and after the Close() call; otherwise, an InvalidOperationException is thrown.</p>
SequentialAccess	<p>Determines whether columns are accessed in order for improved performance.</p> <p>The initial default is False.</p> <p><b>Note:</b> This property can only be set prior to the Open() call and after the Close() call; otherwise, an InvalidOperationException is thrown.</p>

Table 47 lists the public methods for the CsvDataReader object.

Table 47 Public Methods of the CsvDataReader Object

Property	Description
Open	Opens the bulk file instance and associated metadatafile for processing.

## CsvDataWriter

The CsvDataWriter class provides the DataWriter semantics of the CSV file format that is written by PSQL Bulk Load.

For more information, refer to the data provider's online help.

Table 48 lists the public properties for the CsvDataWriter object.

Table 48 Public Properties of the CsvDataWriter Object

Property	Description
BinaryThreshold	<p>Specifies the threshold (in KB) at which separate files are generated to store binary data during a bulk unload.</p> <p>The Initial default is 32.</p> <p>Values less than zero throw a System.ArgumentOutOfRangeException to be thrown.</p>

Table 48 Public Properties of the CsvDataWriter Object

Property	Description
CharacterThreshold	<p>Specifies the threshold (in KB) at which separate files are generated to store character data during a bulk unload.</p> <p>The initial default is 64.</p> <p>Values less than zero cause a System.ArgumentOutOfRangeException to be thrown.</p>
CsvCharacterSetName	<p>Specifies any of the supported IANA code page names that may be used as values. See <a href="#">IANA Code Page Mappings</a> for the supported values.</p> <p>Applications can obtain the database character that was set using the PsqIConnection.DatabaseCharacterSetName property.</p> <p>If an unrecognized CharacterSetName is used, an exception is thrown, declaring that invalid character set has been used.</p> <p>The initial default value is UTF-16.</p> <p>Note this property enforces the character set used in the CSV data file and overflow files added.</p>

Table 49 lists the public methods for the CsvDataWriter object.

Table 49 Public Methods of the CsvDataWriter Object

Property	Description
Open	Opens the bulk file instance and associated metadatafile for processing.
WriteToFile	Writes the contents of the IDataReader to the bulk data file.

## DbBulkCopy

The DbBulkCopy class facilitates copying rows from one data source to another.

The DbBulkCopy object follows the de facto standard defined by the Microsoft SqlBulkCopy class, and has no provider-specific properties or methods. For information about the properties and methods supported, refer to the data provider's online help and the Microsoft .NET Framework SDK documentation.

## DbBulkCopyColumnMapping

The DbBulkCopyColumnMapping class represents a column mapping from the data sources table to a destination table.

The DbBulkCopyColumnMapping object follows the de facto standard defined by the Microsoft SqlBulkCopyColumnMapping class, and has no provider-specific properties or methods. For information about the properties and methods supported, refer to the data provider's online help and the Microsoft .NET Framework SDK documentation.

## DbBulkCopyColumnMappingCollection

The DbBulkCopyColumnMappingCollection class is a collection of DbBulkCopyColumnMapping objects.

The `DbBulkCopyColumnMappingCollection` object follows the de facto standard defined by the `Microsoft.SqlBulkCopyColumnMappingCollection` class, and has no provider-specific properties or methods. For information about the properties and methods supported, refer to the data provider's online help and the Microsoft .NET Framework SDK documentation.



# Getting Schema Information

---

## *Finding and Returning Metadata for a Database*

Applications can request that data providers find and return metadata for a database. Schema collections specific to each data provider expose database schema elements such as tables and columns. The data provider uses the `GetSchema` method of the `Connection` class. You can also retrieve schema information from a result set, as described in [Columns Returned by the `GetSchemaTable` Method](#).

The data provider also includes provider-specific schema collections. Using the schema collection name `MetaDataCollections`, you can return a list of the supported schema collections, and the number of restrictions that they support.

## Columns Returned by the GetSchemaTable Method

While a `PsqlDataReader` is open, you can retrieve schema information from the result set. The result set produced for `PsqlDataReader.GetSchemaTable()` returns the columns described in Table 50, in the order shown.

Table 50 Columns Returned by `GetSchemaTable` on `PsqlDataReader`

Column	Description
ColumnName	Specifies the name of the column, which might not be unique. If the name cannot be determined, a null value is returned. This name reflects the most recent renaming of the column in the current view or command text.
ColumnOrdinal	Specifies the ordinal of the column, which cannot be null. The bookmark column of the row, if any, is 0. Other columns are numbered starting with 1.
ColumnSize	Specifies the maximum possible length of a value in the column. For columns that use a fixed-length data type, this is the size of the data type.
NumericPrecision	Specifies the precision of the column, which depends on how the column is defined in <code>ProviderType</code> . If <code>ProviderType</code> is a numeric data type, this is the maximum precision of the column. If <code>ProviderType</code> is not a numeric data type, the value is null.
NumericScale	Specifies the number of digits to the right of the decimal point if <code>ProviderType</code> is <code>DBTYPE_DECIMAL</code> or <code>DBTYPE_NUMERIC</code> . Otherwise, this is a null value. The value depends on how the column is defined in <code>ProviderType</code> .
DataType	Maps to the .NET Framework type of the column.
ProviderType	Specifies the indicator of the column's data type. This column cannot contain a null value. If the data type of the column varies from row to row, this must be <code>Object</code> .
IsLong	Set if the column contains a BLOB that contains very long data. The setting of this flag corresponds to the value of the <code>IS_LONG</code> column in the <code>PROVIDER_TYPES</code> rowset for the data type. The definition of very long data is provider-specific.
AllowDBNull	Set if the consumer can set the column to a null value, or if the data provider cannot determine whether the consumer can set the column to a null value. Otherwise, no value is set. A column can contain null values, even if it cannot be set to a null value.
IsReadOnly	Determines whether a column can be changed. The value is true if the column can be modified; otherwise, the value is false.
IsRowVersion	Is set if the column contains a persistent row identifier that cannot be written to, and has no meaningful value except to identify the row.
IsUnique	Specifies whether the column constitutes a key by itself or if there is a constraint of type <code>UNIQUE</code> that applies only to this column. If set to true, no two rows in the base table (the table returned in <code>BaseTableName</code> ) can have the same value in this column. If set to false (the initial default), the column can contain duplicate values in the base table.

Table 50 Columns Returned by GetSchemaTable on PsqlDataReader continued

Column	Description
IsKey	<p>Specifies whether a set of columns uniquely identifies a row in the rowset. This set of columns may be generated from a base table primary key, a unique constraint, or a unique index.</p> <p>The value is true if the column is one of a set of columns in the rowset that, taken together, uniquely identify the row. The value is false if the column is not required to uniquely identify the row.</p>
IsAutoIncrement	<p>Specifies whether the column assigns values to new rows in fixed increments.</p> <p>If set to VARIANT_TRUE, the column assigns values to new rows in fixed increments.</p> <p>If set to VARIANT_FALSE (the initial default), the column does not assign values to new rows in fixed increments.</p>
BaseSchemaName	<p>Specifies the name of the schema in the database that contains the column. The value is null if the base schema name cannot be determined.</p> <p>The initial default is null.</p>
BaseCatalogName	<p>Specifies the name of the catalog in the data store that contains the column. A null value is used if the base catalog name cannot be determined.</p> <p>The initial default is null.</p>
BaseTableName	<p>Specifies the name of the table or view in the data store that contains the column. A null value is used if the base table name cannot be determined.</p> <p>The initial default is null.</p>
BaseColumnName	<p>Specifies the name of the column in the data store. This might be different than the column name returned in the ColumnName column if an alias was used. A null value is used if the base column name cannot be determined or if the rowset column is derived from, but is not identical to, a column in the database.</p> <p>The initial default is null.</p>
IsAliased	<p>Specifies whether the name of the column is an alias. The value true is returned if the column name is an alias; otherwise, false is returned.</p>
IsExpression	<p>Specifies whether the name of the column is an expression. The value true is returned if the column is an expression; otherwise, false is returned.</p>
IsIdentity	<p>Specifies whether the name of the column is an identity column. The value true is returned if the column is an identity column; otherwise, false is returned.</p>
IsHidden	<p>Specifies whether the name of the column is hidden. The value true is returned if the column is hidden; otherwise, false is returned.</p>

## Retrieving Schema Metadata with the GetSchema Method

Applications use the GetSchema method of the Connection object to retrieve Schema Metadata about a data provider and/or data source. Each provider implements a number of Schema collections, including the five standard metadata collections.

- [MetaDataCollections Schema Collections](#)
- [DataSourceInformation Schema Collection](#)
- [DataTypes Collection](#)
- [ReservedWords Collection](#)
- [Restrictions Collection](#)

Additional collections are specified and must be supported to return Schema information from the data provider.

See [Additional Schema Collections](#) for details about the other collections supported by the data providers.

**Note:** Refer to the .NET Framework documentation for additional background functional requirements, including the required data type for each ColumnName.

### **MetaDataCollections Schema Collections**

The MetaDataCollections schema collection is a list of the schema collections available to the logged in user. The MetaDataCollection can return the supported columns described in Table 51 in any order.

Table 51 Columns Returned by the MetaDataCollections Schema Collection

ColumnName	Description
CollectionName	The name of the collection to pass to the GetSchema method to return the collection.
NumberOfRestrictions	The number of restrictions that may be specified for the collection.
NumberOfIdentifierParts	The number of parts in the composite identifier/data base object name.

### **DataSourceInformation Schema Collection**

The DataSourceInformation schema collection can return the supported columns, described in Table 52, in any order. Note that only one row is returned.

Table 52 ColumnNames Returned by the DataSourceInformation Collection

ColumnName	Description
CompositelIdentifierSeparatorPattern	The regular expression to match the composite separators in a composite identifier.
DataSourceProductName	The name of the product accessed by the data provider.
DataSourceProductVersion	Indicates the version of the product accessed by the data provider, in the data source's native format.
DataSourceProductVersionNormalized	A normalized version for the data source. This allows the version to be compared with String.Compare().

Table 52 ColumnNames Returned by the DataSourceInformation Collection *continued*

ColumnName	Description
GroupByBehavior	Specifies the relationship between the columns in a GROUP BY clause and the non-aggregated columns in the select list.
Host	The host to which the data provider is connected.
IdentifierCase	Indicates whether non-quoted identifiers are treated as case sensitive.
IdentifierPattern	A regular expression that matches an identifier and has a match value of the identifier.
OrderByColumnsInSelect	Specifies whether columns in an ORDER BY clause must be in the select list. A value of true indicates that they are required to be in the Select list, a value of false indicates that they are not required to be in the Select list.
ParameterMarkerFormat	A format string that represents how to format a parameter.
ParameterMarkerPattern	A regular expression that matches a parameter marker. It will have a match value of the parameter name, if any.
ParameterNameMaxLength	The maximum length of a parameter name in characters.
ParameterNamePattern	A regular expression that matches the valid parameter names.
QuotedIdentifierCase	Indicates whether quoted identifiers are treated as case sensitive.
QuotedIdentifierPattern	A regular expression that matches a quoted identifier and has a match value of the identifier itself without the quotation marks.
StatementSeparatorPattern	A regular expression that matches the statement separator.
StringLiteralPattern	A regular expression that matches a string literal and has a match value of the literal itself.
SupportedJoinOperators	Specifies the types of SQL join statements that are supported by the data source.

## DataTypes Collection

Table 53 describes the supported columns of the DataTypes schema collection. The columns can be returned in any order.

Table 53 ColumnNames Returned by the DataTypes Collection

ColumnName	Description
ColumnSize	The length of a non-numeric column or parameter refers to either the maximum or the length defined for this type by the data provider.
CreateFormat	Format string that represents how to add this column to a data definition statement, such as CREATE TABLE.

Table 53 ColumnNames Returned by the DataTypes Collection continued

ColumnName	Description
CreateParameters	<p>The creation parameters that must be specified when creating a column of this data type. Each creation parameter is listed in the string, separated by a comma in the order they are to be supplied.</p> <p>For example, the SQL data type DECIMAL needs a precision and a scale. In this case, the creation parameters should contain the string "precision, scale".</p> <p>In a text command to create a DECIMAL column with a precision of 10 and a scale of 2, the value of the CreateFormat column might be <code>DECIMAL ( { 0 } , { 1 } )</code> and the complete type specification would be <code>DECIMAL ( 10 , 2 )</code>.</p>
DataType	The name of the .NET Framework type of the data type.
IsAutoIncrementable	<p>Specifies whether values of a data type are auto-incremented.</p> <p>true: Values of this data type may be auto-incremented.</p> <p>false: Values of this data type may not be auto-incremented.</p>
IsBestMatch	<p>Specifies whether the data type is the best match between all data types in the data store and the .NET Framework data type indicated by the value in the DataType column.</p> <p>true: The data type is the best match.</p> <p>false: The data type is not the best match.</p>
IsCaseSensitive	<p>Specifies whether the data type is both a character type and case-sensitive.</p> <p>true: The data type is a character type and is case-sensitive.</p> <p>false: The data type is not a character type or is not case-sensitive.</p>
IsConcurrencyType	<p>true: The data type is updated by the database every time the row is changed and the value of the column is different from all previous values.</p> <p>false: The data type is not updated by the database every time the row is changed.</p>
IsFixedLength	<p>true: Columns of this data type created by the data definition language (DDL) will be of fixed length.</p> <p>false: Columns of this data type created by the DDL will be of variable length.</p>
IsFixedPrecisionScale	<p>true: The data type has a fixed precision and scale.</p> <p>false: The data type does not have a fixed precision and scale.</p>
IsLiteralsSupported	<p>true: The data type can be expressed as a literal.</p> <p>false: The data type cannot be expressed as a literal.</p>
IsLong	<p>true: The data type contains very long data. The definition of very long data is provider-specific.</p> <p>false: The data type does not contain very long data.</p>
IsNullable	<p>true: The data type is nullable.</p> <p>false: The data type is not nullable.</p>
IsSearchable	<p>true: The data type contains very long data. The definition of very long data is provider-specific.</p> <p>false: The data type does not contain very long data.</p>
IsSearchableWithLike	<p>true: The data type can be used with the LIKE predicate.</p> <p>false: The data type cannot be used with the LIKE predicate.</p>

Table 53 ColumnNames Returned by the DataTypes Collection continued

ColumnName	Description
IsUnsigned	true: The data type is unsigned. false: The data type is signed.
LiteralPrefix	The prefix applied to a given literal.
LiteralSuffix	The suffix applied to a given literal.
MaximumScale	If the type indicator is a numeric type, this is the maximum number of digits allowed to the right of the decimal point. Otherwise, this is DBNull.Value.
MinimumScale	If the type indicator is a numeric type, this is the minimum number of digits allowed to the right of the decimal point. Otherwise, this is DBNull.Value.
ProviderDbType	The provider-specific type value that should be used when specifying a parameter's type.
TypeName	The provider-specific data type name.

### ReservedWords Collection

This schema collection exposes information about the words that are reserved by the database to which the data provider is connected. Table 54 describes the columns that the data provider supports.

Table 54 ReservedWords Schema Collection

ColumnName	Description
Reserved Word	Provider-specific reserved words.

### Restrictions Collection

The Restrictions schema collection exposes information about the restrictions supported by the data provider that is currently connected to the database. Table 55 describes the columns returned by the data providers. The columns can be returned in any order.

The PSQL ADO.NET Data Provider uses standardized names for restrictions. If the data provider supports a restriction for a Schema method, it always uses the same name for the restriction.

The case sensitivity of any restriction value is determined by the underlying database, and can be determined by the IdentifierCase and QuotedIdentifierCase values in the DataSourceInformation collection (see [DataSourceInformation Schema Collection](#)).

Table 55 ColumnNames Returned by the Restrictions Collection

ColumnName	Description
CollectionName	The name of the collection to which the specified restrictions apply.
RestrictionName	The name of the restriction in the collection.
RestrictionDefault	Ignored.

Table 55 *ColumnNames Returned by the Restrictions Collection continued*

ColumnName	Description
RestrictionNumber	The actual location in the collection restrictions for this restriction.
IsRequired	Specifies whether the restriction is required.

See [Additional Schema Collections](#) for the restrictions that apply to the each of the additional supported schema collections.



## Additional Schema Collections

The PSQL ADO.NET Data Provider supports the following additional schema collections:

- [Columns Schema Collection](#)
- [ForeignKeys Schema Collection](#)
- [Indexes Schema Collection](#)
- [PrimaryKeys Schema Collection](#)
- [ProcedureParameters Schema Collection](#)
- [Procedures Schema Collection](#)
- [TablePrivileges Schema Collection](#)
- [Tables Schema Collection](#)
- [Views Schema Collection](#)

### Columns Schema Collection

**Description:** The Columns schema collection identifies the columns of tables (including views) defined in the catalog that are accessible to a given user. Table 56 identifies the columns of tables defined in the catalog that are accessible to a given user.

**Number of restrictions:** 3

**Restrictions available:** TABLE\_CATALOG, TABLE\_NAME, COLUMN\_NAME

**Sort order:** TABLE\_CATALOG, TABLE\_NAME, ORDINAL\_POSITION

Table 56 Columns Schema Collection

Column Name	.NET Framework DataType	Description
CHARACTER_MAXIMUM_LENGTH	Int32	The maximum possible length of a value in the column. For character, binary, or bit columns, this is one of the following: <ul style="list-style-type: none"> <li>■ The maximum length of the column in characters, bytes, or bits, respectively, if one is defined.</li> <li>■ The maximum length of the data type in characters, bytes, or bits, respectively, if the column does not have a defined length.</li> <li>■ Zero (0) if neither the column or the data type has a defined maximum length, or if the column is not a character, binary, or bit column.</li> </ul>
CHARACTER_OCTET_LENGTH	Int32	The maximum length in octets (bytes) of the column, if the type of the column is character or binary.  A value of zero (0) means the column has no maximum length or that the column is not a character or binary column.
COLUMN_DEFAULT	String	The default value of the column.
COLUMN_HASDEFAULT	Boolean	TRUE: The column has a default value.  FALSE: The column does not have a default value, or it is unknown whether the column has a default value.

Table 56 Columns Schema Collection continued

Column Name	.NET Framework DataType	Description
COLUMN_NAME	String	The name of the column; this might not be unique.
DATA_TYPE	Object	The indicator of the column's data type. This value cannot be null.
IS_NULLABLE	Boolean	TRUE: The column might be nullable. FALSE: The column is known not to be nullable.
NATIVE_DATA_TYPE	String	The data source description of the type. This value cannot be null.
NUMERIC_PRECISION	Int32	If the column's data type is of numeric data, this is the maximum precision of the column.
NUMERIC_PRECISION_RADIX	Int32	The radix indicates in which base the values in NUMERIC_PRECISION and NUMERIC_SCALE are expressed. It is only useful to return either 2 or 10.
NUMERIC_SCALE	Int16	If the column's type is a numeric type that has a scale, this is the number of digits to the right of the decimal point.
ORDINAL_POSITION	Int32	The ordinal of the column. Columns are numbered starting from one.
PROVIDER_DEFINED_TYPE	Int32	The data source defined type of the column is mapped to the type enumeration of the data provider, for example, the PsqldbType enumeration. This value cannot be null.
PROVIDER_GENERIC_TYPE	Int32	The provider-defined type of the column is mapped to the System.Data.DbType enumeration. This value cannot be null.
TABLE_CATALOG	String	The database name.
TABLE_NAME	String	The table name.

\* All classes are System.XXX. For example, System.String.

## ForeignKeys Schema Collection

**Description:** The ForeignKeys schema collection identifies the foreign key columns defined in the catalog by a given user.

**Number of restrictions:** 2

**Restrictions available:** FK\_TABLE\_CATALOG, PK\_TABLE\_NAME

**Sort order:** FK\_TABLE\_CATALOG, FK\_TABLE\_NAME

Table 57 ForeignKeys Schema Collection

Column Name	.NET Framework Datatype	Description
DEFERRABILITY	String	The deferrability of the foreign key. The value is one of the following: <ul style="list-style-type: none"> <li>■ INITIALLY DEFERRED</li> <li>■ INITIALLY IMMEDIATE</li> <li>■ NOT DEFERRABLE</li> </ul>
DELETE_RULE	String	If a delete rule was specified, the value is one of the following: CASCADE: A referential action of CASCADE was specified. SET NULL: A referential action of SET NULL was specified. SET DEFAULT: A referential action of SET DEFAULT was specified. NO ACTION: A referential action of NO ACTION was specified.
FK_COLUMN_NAME	String	The foreign key column name.
FK_NAME	String	The foreign key name. This is a required restriction.
FK_TABLE_CATALOG	String	The catalog name in which the foreign key table is defined.
FK_TABLE_NAME	String	The foreign key table name. This is a required restriction.
ORDINAL	Int32	The order of the column names in the key. For example, a table might contain several foreign key references to another table. The ordinal starts over for each reference; for example, two references to a three-column key would return 1, 2, 3, 1, 2, 3.
PK_COLUMN_NAME	String	The primary key column name.
PK_NAME	String	The primary key name.
PK_TABLE_CATALOG	String	The catalog name in which the primary key table is defined.
PK_TABLE_NAME	String	The primary key table name.
UPDATE_RULE	String	If an update rule was specified, one of the following: CASCADE: A referential action of CASCADE was specified. SET NULL: A referential action of SET NULL was specified. SET DEFAULT: A referential action of SET DEFAULT was specified. NO ACTION: A referential action of NO ACTION was specified.

\* All classes are System.XXX. For example, System.String

## Indexes Schema Collection

**Description:** The Indexes schema collection identifies the indexes defined in the catalog that are owned by a given user.

**Number of restrictions:** 2

**Restrictions available:** TABLE\_CATALOG, TABLE\_NAME

**Sort order:** UNIQUE, TYPE, INDEX\_CATALOG, INDEX\_NAME, ORDINAL\_POSITION

Table 58 Indexes Schema Collection

Column Name	.NET Framework Data Type	Description
CARDINALITY	Int32	The number of unique values in the index.
COLLATION	String	This is one of the following: ASC: The sort sequence for the column is ascending. DESC: The sort sequence for the column is descending.
COLUMN_NAME	String	The column name.
FILTER_CONDITION	String	The WHERE clause that identifies the filtering restriction.
INDEX_CATALOG	String	The catalog name.
INDEX_NAME	String	The index name.
ORDINAL_POSITION	Int32	The ordinal position of the column in the index, starting with 1.
PAGES	Int32	The number of pages used to store the index.
TABLE_CATALOG	String	The catalog name.
TABLE_NAME	String	The table name.
TYPE	String	The type of the index. This is one of the following values: BTREE: The index is a B+-tree. HASH: The index is a hash file using, for example, linear or extensible hashing. CONTENT: The index is a content index. OTHER: The index is some other type of index.
UNIQUE	Boolean	

\* All classes are System.XXX. For example, System.String.

### PrimaryKeys Schema Collection

**Description:** The PrimaryKeys schema collection identifies the primary key columns defined in the catalog by a given user.

**Number of restrictions:** 2

**Restrictions available:** TABLE\_CATALOG, TABLE\_NAME

**Sort order:** TABLE\_CATALOG, TABLE\_NAME

Table 59 PrimaryKeys Schema Collection

Column Name	.NET Framework DataType	Description
COLUMN_NAME	String	The primary key column name.
ORDINAL	Int32	The order of the column names in the key.
PK_NAME	String	The primary key name.
TABLE_CATALOG	String	The database name in which the table is defined.
TABLE_NAME	String	The table name.

\* All classes are System.XXX. For example, System.String.

### ProcedureParameters Schema Collection

**Description:** The ProcedureParameters schema collection returns information about the parameters and return codes of procedures that are part of the Procedures collection.

**Number of restrictions:** 3

**Restrictions available:** PROCEDURE\_CATALOG, PROCEDURE\_NAME, PARAMETER\_NAME

**Sort order:** PROCEDURE\_CATALOG, PROCEDURE\_NAME, ORDINAL\_POSITION

Table 60 ProcedureParameters Schema Collection

Column Name	.NET Framework DataType	Description
CHARACTER_OCTET_LENGTH	Int32	The maximum length in octets (bytes) of the parameter, if the type of the parameter is character or binary.  If the parameter has no maximum length, the value is zero (0).  For all other types of parameters, the value is -1.
DATA_TYPE	Object	The indicator of the column's data type.  This value cannot be null.
DESCRIPTION	String	The description of the parameter. For example, the description of the Name parameter in a procedure that adds a new employee might be Employee name.
IS_NULLABLE	Boolean	TRUE: The parameter might be nullable.  FALSE: The parameter is not nullable.
NATIVE_DATA_TYPE	String	The data source description of the type.  This value cannot be null.
NUMERIC_PRECISION	Int32	If the column's data type is numeric, this is the maximum precision of the column.  If the column's data type is not numeric, this is DbNull.

Table 60 ProcedureParameters Schema Collection *continued*

Column Name	.NET Framework DataType	Description
NUMERIC_SCALE	Int16	If the column's type is a numeric type that has a scale, this is the number of digits to the right of the decimal point.  Otherwise, this is DbNull.
ORDINAL_POSITION	Int32	If the parameter is an input, input/output, or output parameter, this is the one-based ordinal position of the parameter in the procedure call.  If the parameter is the return value, this is DbNull.
PARAMETER_DEFAULT	String	The default value of parameter.  If the default value is a NULL, then the PARAMETER_HASDEFAULT column returns TRUE and the PARAMETER_DEFAULT column will not exist.  If PARAMETER_HASDEFAULT is set to FALSE, then the PARAMETER_DEFAULT column will not exist.
PARAMETER_HASDEFAULT	Boolean	TRUE: The parameter has a default value.  FALSE: The parameter does not have a default value, or it is unknown whether the parameter has a default value.
PARAMETER_NAME	String	The parameter name. If the parameter is not named, this is DbNull.
PARAMETER_TYPE	String	This is one of the following:  INPUT: The parameter is an input parameter.  INPUTOUTPUT: The parameter is an input/output parameter.  OUTPUT: The parameter is an output parameter.  RETURNVALUE: The parameter is a procedure return value.  UNKNOWN: The parameter type is unknown to the data provider.
PROCEDURE_CATALOG	String	The catalog name.
PROCEDURE_NAME	String	The procedure name.
PROVIDER_DEFINED_TYPE	Int32	The data source defined type of the column as mapped to the type enumeration of the data provider, for example, the PSQDbType enumeration.  This value cannot be null.
PROVIDER_GENERIC_TYPE	Int32	The data source defined type of the column as mapped to the System.Data.DbType enumeration.  This value cannot be null.

\* All classes are System.XXX. For example, System.String.

## Procedures Schema Collection

**Description:** The Procedures schema collection identifies the procedures defined in the catalog. When possible, only procedures for which the connected user has execute permission should be returned.

**Number of restrictions:** 2

**Restrictions available:** PROCEDURE\_CATALOG, PROCEDURE\_NAME, PROCEDURE\_TYPE

**Sort order:** PROCEDURE\_CATALOG, PROCEDURE\_NAME

Table 61 Procedures Schema Collection

Column Name	.NET Framework DataType	Description
PROCEDURE_CATALOG	String	The database name.
PROCEDURE_NAME	String	The procedure name.
PROCEDURE_TYPE	String	This is one of the following:  UNKNOWN: It is not known whether a value is returned.  PROCEDURE: Procedure; no value is returned.  FUNCTION: Function; a value is returned.

\* All classes are System.XXX. For example, System.String.

### TablePrivileges Schema Collection

**Description:** The TablePrivileges schema collection identifies the privileges on tables defined in the catalog that are available to or granted by a given user.

**Number of restrictions:** 3

**Restrictions available:** TABLE\_CATALOG, TABLE\_NAME, GRANTEE

**Sort order:** TABLE\_CATALOG, TABLE\_NAME, PRIVILEGE\_TYPE

Table 62 TablePrivileges Schema Collection

Column Name	Type Indicator *	Description
GRANTEE	String	The user name (or PUBLIC) to whom the privilege has been granted.
PRIVILEGE_TYPE	String	The privilege type. This is one of the following types:  ■ DELETE  ■ INSERT  ■ REFERENCES  ■ SELECT  ■ UPDATE
TABLE_CATALOG	String	The name of the database in which the table is defined.
TABLE_NAME	String	The table name.

\* All classes are System.XXX. For example, System.String.

## Tables Schema Collection

**Description:** The Tables schema collection identifies the tables (including views) defined in the catalog that are accessible to a given user.

**Number of Restrictions:** 3

**Restrictions Available:** TABLE\_CATALOG, TABLE\_NAME, TABLE\_TYPE

**Sort order:** TABLE\_TYPE, TABLE\_CATALOG, TABLE\_NAME

Table 63 Tables Schema Collection

Column Name	.NET Framework DataType	Description
DESCRIPTION	String	A description of the table. If no description is associated with the column, the data provider returns DbNull.
TABLE_CATALOG	String	The name of the database in which the table is defined.
TABLE_NAME	String	The table name.
TABLE_TYPE	String	The table type. One of the following: <ul style="list-style-type: none"> <li>■ ALIAS</li> <li>■ GLOBAL TEMPORARY</li> <li>■ LOCAL TEMPORARY</li> <li>■ SYNONYM</li> <li>■ SYSTEM TABLE</li> <li>■ SYSTEM VIEW</li> <li>■ TABLE</li> <li>■ VIEW</li> </ul> This column cannot contain an empty string.

\* All classes are System.XXX. For example, System.String.

## Views Schema Collection

**Description:** The Views schema collection identifies the views defined in the catalog that are accessible to a given user.

**Number of restrictions:** 2

**Restrictions available:** TABLE\_CATALOG, TABLE\_NAME

**Sort order:** TABLE\_CATALOG, TABLE\_NAME



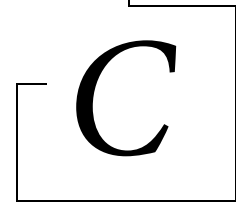
Table 64 Views Schema Collection

Column Name	Type * Indicator	Description
TABLE_CATALOG	String	The name of the database in which the table is defined.
TABLE_NAME	String	The table name.
VIEW_DEFINITION	String	The view definition. This is a query expression.

\* All classes are System.XXX. For example, System.String.



# SQL Escape Sequences for .NET



---

A number of language features, such as outer joins and scalar function calls, are commonly implemented by DBMSs. The syntax for these features is often DBMS-specific, even when a standard syntax has been defined. .NET supports escape sequences that contain standard syntaxes for the following language features:

- Date, time, and timestamp literals
- Scalar functions such as numeric, string, and data type conversion functions
- Outer joins

The escape sequence used by .NET is:

```
{extension}
```

The escape sequence is recognized and parsed by the PSQL ADO.NET Data Provider, which replaces the escape sequences with data store-specific grammar.

---

## Date, Time, and Timestamp Escape Sequences

The escape sequence for date, time, and timestamp literals is:

```
{literal-type 'value'}
```

where *literal-type* is one of the following:

literal-type	Description	Value Format
d	Date	yyyy-mm-dd
t	Time	hh:mm:ss [.]
ts	Timestamp	yyyy-mm-dd hh:mm:ss[.f...]

**Note:** If you receive an error while running a query in Visual Studio to insert data into the Date field of a table, ensure that your system's Date format is set to yyyy-mm-dd. If not, change it to yyyy-mm-dd.

### Example

```
UPDATE Orders SET OpenDate={d '1997-01-29'}  
WHERE OrderID=1023
```

## Scalar Functions

You can use scalar functions in SQL statements with the following syntax:

```
{fn scalar-function}
```

where *scalar-function* is a scalar function supported by the PSQL ADO.NET Data Provider.

### Example

```
SELECT {fn UCASE(NAME)} FROM EMP
```

Table 65 lists the scalar functions supported.

Table 65 Scalar Functions Supported

String Functions	Numeric Functions	Timedate Functions	System Functions
ASCII BIT_LENGTH CHAR CHAR_LENGTH CHARACTER_LENGTH CONCAT LCASE or LOWER LEFT LENGTH LOCATE LTRIM OCTET_LENGTH POSITION REPLACE REPLICATE RIGHT RTRIM SPACE STUFF SUBSTRING UCASE or UPPER	ABS ACOS ASIN ATAN ATAN2 CEILING COS COT DEGREES EXP FLOOR LOG LOG10 MOD PI POWER RADIANS RAND ROUND SIGN SIN SQRT TAN TRUNCATE	CURDATE CURRENT_DATE CURTIME CURRENT_TIME CURRENT_TIMESTAMP DAYNAME DAYOFMONTH DAYOFYEAR EXTRACT HOUR MINUTE MONTH MONTHNAME NOW QUARTER SECOND TIMESTAMPADD TIMESTAMPDIFF WEEK YEAR	DATABASE USER

---

## Outer Join Escape Sequences

.NET supports the SQL92 left, right, and full outer join syntax. The escape sequence for outer joins is:

`{oj outer-join}`

where *outer-join* is:

```
table-reference {LEFT | RIGHT | FULL} OUTER JOIN  
{table-reference | outer-join} ON search-condition
```

where:

*table-reference* is a table name and *search-condition* is the join condition you want to use for the tables.

### Example

```
SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status  
FROM {oj Customers LEFT OUTER JOIN  
      Orders ON Customers.CustID=Orders.CustID}  
WHERE Orders.Status='OPEN'
```

The PSQL ADO.NET Data Provider supports the following outer join escape sequences as supported by PSQL 9.x and higher:

- Left outer joins
- Right outer joins
- Full outer joins

# *Locking and Isolation Levels*

## *D*

---

Different database systems support various locking and isolation levels. The following topics cover locking and isolation levels and how their settings affect the data you retrieve:

- [Locking](#)
- [Isolation Levels](#)
- [Locking Modes and Levels](#)

## **Locking**

Locking is a database operation that restricts a user from accessing a table or record. Locking is used in situations where more than one user might try to use the same table or record at the same time. By locking the table or record, the system ensures that only one user at a time can affect the data.

Locking is critical in multiuser databases, where different users can try to access or modify the same records concurrently. Although such concurrent database activity is desirable, it can create problems. Without locking, for example, if two users try to modify the same record at the same time, they might encounter problems ranging from retrieving bad data to deleting data that the other user needs. If, however, the first user to access a record can lock that record to temporarily prevent other users from modifying it, such problems can be avoided. Locking provides a way to manage concurrent database access while minimizing the various problems it can cause.



## Isolation Levels

An isolation level represents a particular locking strategy employed in the database system to improve data consistency. The higher the isolation level, the more complex the locking strategy behind it. The isolation level provided by the database determines whether a transaction will encounter the following behaviors in data consistency:

Dirty reads	User 1 modifies a row. User 2 reads the same row before User 1 commits. User 1 performs a rollback. User 2 has read a row that has never really existed in the database. User 2 may base decisions on false data.
Non-repeatable reads	User 1 reads a row but does not commit. User 2 modifies or deletes the same row and then commits. User 1 rereads the row and finds it has changed (or has been deleted).
Phantom reads	User 1 uses a search condition to read a set of rows but does not commit. User 2 inserts one or more rows that satisfy this search condition, then commits. User 1 rereads the rows using the search condition and discovers rows that were not present before.

Isolation levels represent the database system's ability to prevent these behaviors. The American National Standards Institute (ANSI) defines four isolation levels:

- Read uncommitted (0)
- Read committed (1)
- Repeatable read (2)
- Serializable (3)

In ascending order (0–3), these isolation levels provide an increasing amount of data consistency to the transaction. At the lowest level, all three behaviors can occur. At the highest level, none can occur. The success of each level in preventing these behaviors is due to the locking strategies that they employ, which are as follows:

Read uncommitted (0)	Locks are obtained on modifications to the database and held until end of transaction (EOT). Reading from the database does not involve any locking.
Read committed (1)	Locks are acquired for reading and modifying the database. Locks are released after reading, but locks on modified objects are held until EOT.
Repeatable read (2)	Locks are obtained for reading and modifying the database. Locks on all modified objects are held until EOT. Locks obtained for reading data are held until EOT. Locks on non-modified access structures (such as indexes and hashing structures) are released after reading.
Serializable (3)	A lock is placed on the affected rows of the DataSet until EOT. All access structures that are modified, and those used by the query, are locked until EOT.

Table 66 shows what data consistency behaviors can occur at each isolation level.

*Table 66 Isolation Levels and Data Consistency*

Level	Dirty Read	Nonrepeatable Read	Phantom Read
0, Read uncommitted	Yes	Yes	Yes

Table 66 Isolation Levels and Data Consistency continued

1, Read committed	No	Yes	Yes
2, Repeatable read	No	No	Yes
3, Serializable	No	No	No

Although higher isolation levels provide better data consistency, this consistency can be costly in terms of the *concurrency* provided to individual users. Concurrency is the ability of multiple users to access and modify data simultaneously. As isolation levels increase, so does the chance that the locking strategy used will create problems in concurrency.

*Put another way:* The higher the isolation level, the more locking involved, and the more time users may spend waiting for data to be freed by another user. Because of this inverse relationship between isolation levels and concurrency, you must consider how people use the database before choosing an isolation level. You must weigh the trade-offs between data consistency and concurrency, and decide which is more important.

---

## **Locking Modes and Levels**

Different database systems employ various locking modes, but they have two basic modes in common: shared and exclusive. Shared locks can be held on a single object by multiple users. If one user has a shared lock on a record, then a second user can also get a shared lock on that same record; however, the second user cannot get an exclusive lock on that record. Exclusive locks are exclusive to the user that obtains them. If one user has an exclusive lock on a record, then a second user cannot get either type of lock on the same record.

Performance and concurrency can also be affected by the locking level used in the database system. The locking level determines the size of an object that is locked in a database. For example, many database systems let you lock an entire table, as well as individual records. An intermediate level of locking, page-level locking, is also common. A page contains one or more records and is typically the amount of data read from the disk in a single disk access. The major disadvantage of page-level locking is that if one user locks a record, a second user may not be able to lock other records because they are stored on the same page as the locked record.



# *Designing .NET Applications for Performance Optimization*

---

## *E*

Developing performance-oriented .NET applications is not easy. The ADO.NET data providers do not throw exceptions to say that your code is running too slowly.

## Retrieving Data

To retrieve data efficiently, return only the data that you need, and choose the most efficient method of doing so. The guidelines in this section will help you to optimize system performance when retrieving data with .NET applications.

### ***Understanding the Architecture***

ADO.NET uses ADO.NET data providers to access data. All .NET runtime data access is done through ADO.NET data providers.

ADO.NET development wizards use OLE DB to populate the grids of the wizards. Although OLE DB is accessed for these wizards at design time, the runtime components generated use ADO.NET data providers for accessing data. Optimizing data access via OLE DB in a .NET application is not a useful exercise. Instead, optimize the ADO.NET data provider as the component that will do all of the runtime work.

### ***Retrieving Long Data***

Unless it is necessary, applications should not request long data because retrieving long data across a network is slow and resource-intensive.

Most users don't want to see long data. If the user wants to see these result items, then the application can query the database again, specifying only the long columns in the select list. This method allows the average user to retrieve the result set without having to pay a high performance penalty for network traffic.

Although the best method is to exclude long data from the select list, some applications do not formulate the select list before sending the query to the ADO.NET data providers (that is, some applications use syntax such as `select * from <table name> . . .`). If the select list contains long data, then some data providers must retrieve that data at fetch time even if the application does not bind the long data in the result set. When possible, try to implement a method that does not retrieve all columns of the table.

Sometimes long data must be retrieved. When this is the case, remember that most users do not want to see 100 KB, or more, of text on the screen.

### ***Reducing the Size of Data Retrieved***

To reduce network traffic and improve performance, you can reduce the size of any data being retrieved to some manageable limit by calling `set max rows` or `set max field size`, or some other database-specific command to limit row size or field size. Another method of reducing the size of the data being retrieved is to decrease the column size. If the data provider allows you to define the packet size, use the smallest packet size that will meet your needs.

In addition, be careful to return only the rows you need. If you return five columns when you only need two columns, performance is decreased, especially if the unnecessary rows include long data.

### ***Using CommandBuilder Objects***

It is tempting to use `CommandBuilder` objects because they generate SQL statements. However, this shortcut can have a negative effect on performance. Because of concurrency restrictions, the `CommandBuilder` does not generate efficient SQL statements. For example, the following SQL statement was created by the `Command Builder`:

```

CommandText: UPDATE TEST01.EMP SET EMPNO = ?, ENAME = ?, JOB = ?, MGR = ?, HIREDATE
= ?, SAL = ?, COMM = ?, DEPT = ?
WHERE
    ( (EMPNO = ?) AND ((ENAME IS NULL AND ? IS NULL)
    OR (ENAME = ?)) AND ((JOB IS NULL AND ? IS NULL)
    OR (JOB = ?)) AND ((MGR IS NULL AND ? IS NULL)
    OR (MGR = ?)) AND ((HIREDATE IS NULL AND ? IS NULL)
    OR (HIREDATE = ?)) AND ((SAL IS NULL AND ? IS NULL)
    OR (SAL = ?)) AND ((COMM IS NULL AND ? IS NULL)
    OR (COMM = ?)) AND ((DEPT IS NULL AND ? IS NULL)
    OR (DEPT = ?)) )

```

The end user can often write more efficient update and delete statements than those that the CommandBuilder generates.

Another drawback is also implicit in the design of the CommandBuilder object. The CommandBuilder object is always associated with a DataAdapter object and registers itself as a listener for RowUpdating and RowUpdated events that the DataAdapter object generates. This means that two events must be processed for every row that is updated.

### ***Choosing the Right Data Type***

Retrieving and sending certain data types can be expensive. When you design a schema, select the data type that can be processed most efficiently. For example, integer data is processed faster than floating-point data. Floating-point data is defined according to internal database-specific formats, usually in a compressed format. The data must be decompressed and converted into a different format so that it can be processed by the wire protocol.

Processing time is shortest for character strings, followed by integers, which usually require some conversion or byte ordering. Processing floating-point data and timestamps is at least twice as slow as integers.

## Selecting .NET Objects and Methods

The guidelines in this section will help you to optimize system performance when selecting and using .NET objects and methods.

### ***Using Parameter Markers as Arguments to Stored Procedures***

When calling stored procedures, always use parameter markers for the argument markers instead of using literal arguments.

When you set the CommandText property in the Command object to the stored procedure name, do not physically code the literal arguments into the CommandText. For example, do not use literal arguments such as:

```
{call expense (3567, 'John', 987.32)}
```

ADO.NET data providers can call stored procedures on the database server by executing the procedure as any other SQL query. Executing the stored procedure as a SQL query results in the database server parsing the statement, validating the argument types, and converting the arguments into the correct data types.

In the following example, the application programmer might assume that the only argument to getCustName is the integer 12345:

```
{call getCustName (12345)}
```

However, SQL is always sent to the database server as a character string. When the database server parses the SQL query and isolates the argument value, the result is still a string. The database server must then convert the string '12345' into the integer 12345. Using a parameter marker eliminates the need to convert the string and reduces the amount of processing by the server:

```
{call getCustName (?)}
```



---

## Designing .NET Applications

The guidelines in this section will help you to optimize system performance when designing .NET applications.

### ***Managing Connections***

Connection management is important to application performance. Optimize your application by connecting once and using multiple statement objects, instead of performing multiple connections. Avoid connecting to a data source after establishing an initial connection.

You can improve performance significantly with connection pooling, especially for applications that connect over a network or through the World Wide Web. Connection pooling lets you reuse connections. Closing connections does not close the physical connection to the database. When an application requests a connection, an active connection is reused, thus avoiding the network I/O needed to create a new connection.

Pre-allocate connections. Decide what connection strings you will need to meet your needs. Remember that each unique connection string creates a new connection pool.

Once created, connection pools are not destroyed until the active process ends or the connection lifetime is exceeded. Maintenance of inactive or empty pools involves minimal system overhead.

Connection and statement handling should be addressed before implementation. Spending time and thoughtfully handling connection management improves application performance and maintainability.

### ***Opening and Closing Connections***

Open connections just before they are needed. Opening them earlier than necessary decreases the number of connections available to other users and can increase the demand for resources.

To keep resources available, explicitly Close the connection as soon as it is no longer needed. If you wait for the garbage collector to implicitly clean up connections that go out of scope, the connections will not be returned to the connection pool immediately, tying up resources that are not actually being used.

Close connections inside a finally block. Code in the finally block always runs, even if an exception occurs. This guarantees explicit closing of connections. For example:

```
try
{
    DBConn.Open();
    ... // Do some other interesting work
}
catch (Exception ex)
{
    // Handle exceptions
}
finally
{
    // Close the connection
    if (DBConn != null)
        DBConn.Close();
}
```

If you are using connection pooling, opening and closing connections is not an expensive operation. Using the Close() method of the data provider's Connection object adds or returns the connection to the

connection pool. Remember, however, that closing a connection automatically closes all `DataReader` objects that are associated with the connection.

### ***Using Statement Caching***

A statement cache is a group of prepared statements or instances of `Command` objects that can be reused by an application. Using statement caching can improve application performance because the actions on the prepared statement are performed once even though the statement is reused multiple times over an application's lifetime.

A statement cache is owned by a physical connection. After being executed, a prepared statement is placed in the statement cache and remains there until the connection is closed.

Caching all of the prepared statements that an application uses might appear to offer increased performance. However, this approach may come at a cost of database memory if you implement statement caching with connection pooling. In this case, each pooled connection has its own statement cache that may contain all of the prepared statements that are used by the application. All of these pooled prepared statements are also maintained in the database's memory.

### ***Using Commands Multiple Times***

Choosing whether to use the `Command.Prepare` method can have a significant positive (or negative) effect on query execution performance. The `Command.Prepare` method tells the underlying data provider to optimize for multiple executions of statements that use parameter markers. Note that it is possible to `Prepare` any command regardless of which execution method is used (`ExecuteReader`, `ExecuteNonQuery`, or `ExecuteScalar`).

Consider the case where an ADO.NET data provider implements `Command.Prepare` by creating a stored procedure on the server that contains the prepared statement. Creating stored procedures involves substantial overhead, but the statement can be executed multiple times. Although creating stored procedures is performance-expensive, execution of that statement is minimized because the query is parsed and optimization paths are stored at create procedure time. Applications that execute the same statement multiples times can benefit greatly from calling `Command.Prepare` and then executing that command multiple times.

However, using `Command.Prepare` for a statement that is executed only once results in unnecessary overhead. Furthermore, applications that use `Command.Prepare` for large single execution query batches exhibit poor performance. Similarly, applications that either always use `Command.Prepare` or never use `Command.Prepare` do not perform as well as those that use a logical combination of prepared and unprepared statements.

### ***Using Native Managed Providers***

Bridges into unmanaged code, that is, code outside the .NET environment, adversely affect performance. Calling unmanaged code from managed code causes the data provider to be significantly slower than data providers that are completely managed code. Why take that kind of performance hit?

If you use a bridge, your code will be written for this bridge. Later, when a database-specific ADO.NET data provider becomes available, the code must be rewritten; you will have to rewrite object names, schema information, error handling, and parameters. You'll save valuable time and resources by coding to managed data providers instead of coding to the bridges.

---

## Updating Data

This section provides general guidelines to help you to optimize system performance when updating data in databases.

### ***Using the Disconnected DataSet***

Keep result sets small. The full result set must be retrieved from the server before the DataSet is populated. The full result set is stored in memory on the client.

### ***Synchronizing Changes Back to the Data Source***

You must build the logic into the PsqlDataAdapter for synchronizing the changes back to the data source using the primary key, as shown in the following example:

```
string updateSQL As String = "UPDATE emp SET sal = ?, job = ?" +  
    " = WHERE empno = ?";
```



## Using an .edmx File

---

An .edmx file is an XML file that defines an Entity Data Model (EDM), describes the target database schema, and defines the mapping between the EDM and the database. An .edmx file also contains information that is used by the ADO.NET Entity Data Model Designer (Entity Designer) to render a model graphically.

The following code examples illustrate the necessary changes to the .edmx file in order to provide Extended Entity Framework functionality to the EDM layer.

The Entity Framework includes a set of methods similar to those of ADO.NET. These methods have been tailored to be useful for the new Entity Framework consumers – LINQ, EntitySQL, and ObjectServices.

The ADO.NET Entity Framework data provider models this functionality in the EDM by surfacing the PsqlStatus and PsqlConnectionStatistics entities, allowing you to model this functionality using standard tools in Visual Studio.

## Code Examples

The following code fragment is an example of the SSDL model:

```
<?xml version="1.0" encoding="utf-8"?>
<Schema Namespace="EntityModel.Store" Alias="Self"
  xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl"
    Provider=Pervasive.Data.SqlClient
    ProviderManifestToken="PSQL">
  <EntityContainer Name="SampleStoreContainer">
    <EntitySet Name="Connection_Statistics"
      EntityType="EntityModel.Store.Connection_Statistics" />
    <EntitySet Name="Status" EntityType="EntityModel.Store.Status" />
  </EntityContainer>
  <Function Name="RetrieveStatistics" Aggregate="false" BuiltIn="false"
    NiladicFunction="false" IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    StoreFunctionName=""Psql_Connection_RetrieveStatistics"" />
  <Function Name="EnableStatistics" Aggregate="false" BuiltIn="false"
    NiladicFunction="false" IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    StoreFunctionName=""Psql_Connection_EnableStatistics"" />
  <Function Name="DisableStatistics" Aggregate="false" BuiltIn="false"
    NiladicFunction="false" IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    StoreFunctionName=""Psql_Connection_DisableStatistics"" />
  <Function Name="ResetStatistics" Aggregate="false" BuiltIn="false"
    NiladicFunction="false" IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    StoreFunctionName=""Psql_Connection_ResetStatistics"" />
  </Function>
  <EntityType Name="Connection_Statistics">
    <Key>
      <PropertyRef Name="Id" />
    </Key>
    <Property Name="SocketReadTime" Type="double" Nullable="false" />
    <Property Name="MaxSocketReadTime" Type="double" Nullable="false" />
    <Property Name="SocketReads" Type="bigint" Nullable="false" />
    <Property Name="BytesReceived" Type="bigint" Nullable="false" />
    <Property Name="MaxBytesPerSocketRead" Type="bigint" Nullable="false" />
    <Property Name="SocketWriteTime" Type="double" Nullable="false" />
    <Property Name="MaxSocketWriteTime" Type="double" Nullable="false" />
    <Property Name="SocketWrites" Type="bigint" Nullable="false" />
    <Property Name="BytesSent" Type="bigint" Nullable="false" />
    <Property Name="MaxBytesPerSocketWrite" Type="bigint" Nullable="false" />
    <Property Name="TimeToDisposeOfUnreadRows" Type="double" Nullable="false" />
    <Property Name="SocketReadsToDisposeUnreadRows" Type="bigint" Nullable="false" />
  </EntityType>
  <Property Name="BytesRecvToDisposeUnreadRows" Type="bigint" Nullable="false" />
  <Property Name="IDUCount" Type="bigint" Nullable="false" />
  <Property Name="SelectCount" Type="bigint" Nullable="false" />
  <Property Name="StoredProcedureCount" Type="bigint" Nullable="false" />
  <Property Name="DDLCount" Type="bigint" Nullable="false" />
  <Property Name="PacketsReceived" Type="bigint" Nullable="false" />
  <Property Name="PacketsSent" Type="bigint" Nullable="false" />
  <Property Name="ServerRoundTrips" Type="bigint" Nullable="false" />
  <Property Name="SelectRowsRead" Type="bigint" Nullable="false" />
</Schema>
```

```

    <Property Name="StatementCacheHits" Type="bigint" Nullable="false" />
    <Property Name="StatementCacheMisses" Type="bigint" Nullable="false" />
    <Property Name="StatementCacheReplaces" Type="bigint" Nullable="false" />
    <Property Name="StatementCacheTopHit1" Type="bigint" Nullable="false" />
    <Property Name="StatementCacheTopHit2" Type="bigint" Nullable="false" />
    <Property Name="StatementCacheTopHit3" Type="bigint" Nullable="false" />
    <Property Name="PacketsReceivedPerSocketRead" Type="double" Nullable="false" />
    <Property Name="BytesReceivedPerSocketRead" Type="double" Nullable="false" />
    <Property Name="PacketsSentPerSocketWrite" Type="double" Nullable="false" />
    <Property Name="BytesSentPerSocketWrite" Type="double" Nullable="false" />
    <Property Name="PacketsSentPerRoundTrip" Type="double" Nullable="false" />
    <Property Name="PacketsReceivedPerRoundTrip" Type="double" Nullable="false" />
    <Property Name="BytesSentPerRoundTrip" Type="double" Nullable="false" />
    <Property Name="BytesReceivedPerRoundTrip" Type="double" Nullable="false" />
    <Property Name="Id" Type="integer" Nullable="false" />
</EntityType>
<EntityType Name="Status">
    <Key>
        <PropertyRef Name="Id" />
    </Key>
    <Property Name="ServerVersion" Type="varchar" MaxLength="127" Nullable="false" />
</>
    <Property Name="Host" Type="varchar" MaxLength="127" Nullable="false" />
    <Property Name="Port" Type="integer" Nullable="false" />
    <Property Name="DatabaseName" Type="varchar" MaxLength="127" Nullable="false" />
    <Property Name="SessionId" Type="integer" Nullable="false" />
    <Property Name="StatisticsEnabled" Type="smallint_as_boolean" Nullable="false" />
</>
    <Property Name="Id" Type="integer" Nullable="false" />
</EntityType>
</Schema>

```

The following code fragment is an example of the MSL model:

```

<?xml version="1.0" encoding="utf-8"?>
<Mapping Space="C-S" xmlns="urn:schemas-microsoft-com:windows:storage:mapping:CS">
    <EntityContainerMapping
        StorageEntityContainer="SampleStoreContainer"
        CdmEntityContainer="SampleContainer">
        <EntitySetMapping Name="PsqlConnectionStatistics">
            <EntityTypeMapping TypeName="EntityModel.PsqlConnectionStatistics">
                <MappingFragment StoreEntitySet="Connection_Statistics">
                    <ScalarProperty Name="SocketReadTime" ColumnName="SocketReadTime" />
                    <ScalarProperty Name="MaxSocketReadTime" ColumnName="MaxSocketReadTime" />
                </MappingFragment>
                <ScalarProperty Name="SocketReads" ColumnName="SocketReads" />
                <ScalarProperty Name="BytesReceived" ColumnName="BytesReceived" />
                <ScalarProperty Name="MaxBytesPerSocketRead" ColumnName="MaxBytesPerSocketRead" />
                <ScalarProperty Name="SocketWriteTime" ColumnName="SocketWriteTime" />
                <ScalarProperty Name="MaxSocketWriteTime" ColumnName="MaxSocketWriteTime" />
                <ScalarProperty Name="SocketWrites" ColumnName="SocketWrites" />
                <ScalarProperty Name="BytesSent" ColumnName="BytesSent" />
                <ScalarProperty Name="MaxBytesPerSocketWrite" ColumnName="MaxBytesPerSocketWrite" />
                <ScalarProperty Name="TimeToDisposeOfUnreadRows" ColumnName="TimeToDisposeOfUnreadRows" />
            </EntityTypeMapping>
        </EntitySetMapping>
    </EntityContainerMapping>
</Mapping>

```

```

        <ScalarProperty Name="SocketReadsToDisposeUnreadRows"
ColumnName="SocketReadsToDisposeUnreadRows" />
        <ScalarProperty Name="BytesRecvToDisposeUnreadRows"
ColumnName="BytesRecvToDisposeUnreadRows" />
        <ScalarProperty Name="IDUCount" ColumnName="IDUCount" />
        <ScalarProperty Name="SelectCount" ColumnName="SelectCount" />
        <ScalarProperty Name="StoredProcedureCount"
ColumnName="StoredProcedureCount" />
        <ScalarProperty Name="DDLCount" ColumnName="DDLCount" />
        <ScalarProperty Name="PacketsReceived" ColumnName="PacketsReceived" />
        <ScalarProperty Name="PacketsSent" ColumnName="PacketsSent" />
        <ScalarProperty Name="ServerRoundTrips" ColumnName="ServerRoundTrips" />
        <ScalarProperty Name="SelectRowsRead" ColumnName="SelectRowsRead" />
        <ScalarProperty Name="StatementCacheHits"
ColumnName="StatementCacheHits" />
        <ScalarProperty Name="StatementCacheMisses"
ColumnName="StatementCacheMisses" />
        <ScalarProperty Name="StatementCacheReplaces"
ColumnName="StatementCacheReplaces" />
        <ScalarProperty Name="StatementCacheTopHit1"
ColumnName="StatementCacheTopHit1" />
        <ScalarProperty Name="StatementCacheTopHit2"
ColumnName="StatementCacheTopHit2" />
        <ScalarProperty Name="StatementCacheTopHit3"
ColumnName="StatementCacheTopHit3" />
        <ScalarProperty Name="PacketsReceivedPerSocketRead"
ColumnName="PacketsReceivedPerSocketRead" />
        <ScalarProperty Name="BytesReceivedPerSocketRead"
ColumnName="BytesReceivedPerSocketRead" />
        <ScalarProperty Name="PacketsSentPerSocketWrite"
ColumnName="PacketsSentPerSocketWrite" />
        <ScalarProperty Name="BytesSentPerSocketWrite"
ColumnName="BytesSentPerSocketWrite" />
        <ScalarProperty Name="PacketsSentPerRoundTrip"
ColumnName="PacketsSentPerRoundTrip" />
        <ScalarProperty Name="PacketsReceivedPerRoundTrip"
ColumnName="PacketsReceivedPerRoundTrip" />
        <ScalarProperty Name="BytesSentPerRoundTrip"
ColumnName="BytesSentPerRoundTrip" />
        <ScalarProperty Name="BytesReceivedPerRoundTrip"
ColumnName="BytesReceivedPerRoundTrip" />
        <ScalarProperty Name="Id" ColumnName="Id" />
    </MappingFragment>
</EntityTypeMapping>
</EntitySetMapping>
<EntitySetMapping Name="PsqlStatus">
    <EntityTypeMapping TypeName="EntityModel.PsqlStatus">
        <MappingFragment StoreEntitySet="Status">
            <ScalarProperty Name="ServerVersion" ColumnName="ServerVersion" />
            <ScalarProperty Name="Host" ColumnName="Host" />
            <ScalarProperty Name="Port" ColumnName="Port" />
            <ScalarProperty Name="DatabaseName" ColumnName="DatabaseName" />
            <ScalarProperty Name="SessionId" ColumnName="SessionId" />
            <ScalarProperty Name="StatisticsEnabled" ColumnName="StatisticsEnabled"
/>
            <ScalarProperty Name="Id" ColumnName="Id" />
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```



```

    <FunctionImportMapping FunctionImportName="RetrieveStatistics"
    FunctionName="EntityModel.Store.RetrieveStatistics" />
    <FunctionImportMapping FunctionImportName="EnableStatistics"
    FunctionName="EntityModel.Store.EnableStatistics" />
    <FunctionImportMapping FunctionImportName="DisableStatistics"
    FunctionName="EntityModel.Store.DisableStatistics" />
    <FunctionImportMapping FunctionImportName="ResetStatistics"
    FunctionName="EntityModel.Store.ResetStatistics" />
    <FunctionImportMapping FunctionImportName="Reauthenticate"
    FunctionName="EntityModel.Store.Reauthenticate" />
  </EntityContainerMapping>
</Mapping>

```

Breaking the model down further, a CSDL model is established at the conceptual layer – this is what is exposed to the EDM.

```

<?xml version="1.0" encoding="utf-8"?>
<Schema Namespace="EntityModel" Alias="Self"
  xmlns="http://schemas.microsoft.com/ado/2006/04/edm">
  <EntityContainer Name="SampleContainer">
    <EntitySet Name="PsqlConnectionStatistics"
    EntityType="EntityModel.PsqlConnectionStatistics" />
    <EntitySet Name="PsqlStatus" EntityType="EntityModel.PsqlStatus" />
    <FunctionImport Name="RetrieveStatistics" EntitySet="PsqlConnectionStatistics"
    ReturnType="Collection(EntityModel.PsqlConnectionStatistics)" />
    <FunctionImport Name="EnableStatistics" EntitySet="PsqlStatus"
    ReturnType="Collection(EntityModel.PsqlStatus)" />
    <FunctionImport Name="DisableStatistics" EntitySet="PsqlStatus"
    ReturnType="Collection(EntityModel.PsqlStatus)" />
    <FunctionImport Name="ResetStatistics" EntitySet="PsqlStatus"
    ReturnType="Collection(EntityModel.PsqlStatus)" />
    <FunctionImport Name="Reauthenticate" EntitySet="PsqlStatus"
    ReturnType="Collection(EntityModel.PsqlStatus)">
      <Parameter Name="CurrentUser" Type="String" />
      <Parameter Name="CurrentPassword" Type="String" />
      <Parameter Name="CurrentUserAffinityTimeout" Type="Int32" />
    </FunctionImport>
  </EntityContainer>
  <EntityType Name="PsqlConnectionStatistics">
    <Key>
      <PropertyRef Name="Id" />
    </Key>
    <Property Name="SocketReadTime" Type="Double" Nullable="false" />
    <Property Name="MaxSocketReadTime" Type="Double" Nullable="false" />
    <Property Name="SocketReads" Type="Int64" Nullable="false" />
    <Property Name="BytesReceived" Type="Int64" Nullable="false" />
    <Property Name="MaxBytesPerSocketRead" Type="Int64" Nullable="false" />
    <Property Name="SocketWriteTime" Type="Double" Nullable="false" />
    <Property Name="MaxSocketWriteTime" Type="Double" Nullable="false" />
    <Property Name="SocketWrites" Type="Int64" Nullable="false" />
    <Property Name="BytesSent" Type="Int64" Nullable="false" />
    <Property Name="MaxBytesPerSocketWrite" Type="Int64" Nullable="false" />
    <Property Name="TimeToDisposeOfUnreadRows" Type="Double" Nullable="false" />
    <Property Name="SocketReadsToDisposeUnreadRows" Type="Int64" Nullable="false" />
  </EntityType>
  <EntityType Name="PsqlStatus">
    <Property Name="BytesRecvToDisposeUnreadRows" Type="Int64" Nullable="false" />
    <Property Name="IDUCount" Type="Int64" Nullable="false" />
    <Property Name="SelectCount" Type="Int64" Nullable="false" />
    <Property Name="StoredProcedureCount" Type="Int64" Nullable="false" />
  </EntityType>

```

```
<Property Name="DDLCount" Type="Int64" Nullable="false" />
<Property Name="PacketsReceived" Type="Int64" Nullable="false" />
<Property Name="PacketsSent" Type="Int64" Nullable="false" />
<Property Name="ServerRoundTrips" Type="Int64" Nullable="false" />
<Property Name="SelectRowsRead" Type="Int64" Nullable="false" />
<Property Name="StatementCacheHits" Type="Int64" Nullable="false" />
<Property Name="StatementCacheMisses" Type="Int64" Nullable="false" />
<Property Name="StatementCacheReplaces" Type="Int64" Nullable="false" />
<Property Name="StatementCacheTopHit1" Type="Int64" Nullable="false" />
<Property Name="StatementCacheTopHit2" Type="Int64" Nullable="false" />
<Property Name="StatementCacheTopHit3" Type="Int64" Nullable="false" />
<Property Name="PacketsReceivedPerSocketRead" Type="Double" Nullable="false" />
<Property Name="BytesReceivedPerSocketRead" Type="Double" Nullable="false" />
<Property Name="PacketsSentPerSocketWrite" Type="Double" Nullable="false" />
<Property Name="BytesSentPerSocketWrite" Type="Double" Nullable="false" />
<Property Name="PacketsSentPerRoundTrip" Type="Double" Nullable="false" />
<Property Name="PacketsReceivedPerRoundTrip" Type="Double" Nullable="false" />
<Property Name="BytesSentPerRoundTrip" Type="Double" Nullable="false" />
<Property Name="BytesReceivedPerRoundTrip" Type="Double" Nullable="false" />
<Property Name="Id" Type="Int32" Nullable="false" />
</EntityType>
<EntityType Name="PsqlStatus">
  <Key>
    <PropertyRef Name="Id" />
  </Key>
  <Property Name="ServerVersion" Type="String" Nullable="false" />
  <Property Name="Host" Type="String" Nullable="false" />
  <Property Name="Port" Type="Int32" Nullable="false" />
  <Property Name="DatabaseName" Type="String" Nullable="false" />
  <Property Name="SessionId" Type="Int32" Nullable="false" />
  <Property Name="StatisticsEnabled" Type="Boolean" Nullable="false" />
  <Property Name="Id" Type="Int32" Nullable="false" />
</EntityType>
</Schema>
```

# *Bulk Load Configuration Files*

---

The following topics describe the configuration files used by PSQL Bulk Load.

- [Sample Bulk Data Configuration File](#)
- [XML Schema Definition for a Bulk Data Configuration File](#)

See [Using PSQL Bulk Load](#) for more information about this feature.

## Sample Bulk Data Configuration File

The bulk format configuration file is produced when either a table or a DataReader is exported (unloaded) using the BulkCopy and BulkLoad operation.

```
<?xml version="1.0"?>
<!--
Sample DDL
-----

CREATE_STMT = CREATE TABLE GTABLE (CHARCOL char(10),VCHARCOL varchar2(10), \

DECIMALCOL number(15,5), NUMERICCOL decimal(15,5), SMALLCOL number(38), \

INTEGERCOL integer, REALCOL number, \

FLOATCOL float, DOUBLECOL number, LVCOL clob, \

BITCOL number(1),TINYINTCOL number(19), BIGINTCOL number(38), BINCOL raw(10), \

VARBINCOL raw(10), LVARBINCOL blob, DATECOL date, \

TIMECOL date, TSCOL date) -->

<table codepage="UTF-16" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="file:///c:/Documents and Settings/jbruce/My
  Documents/Current Specs/BulkData.xsd">
  <row>
    <column codepage="UTF-16" datatype="CHAR" length="10"
    nullable="true">CHARCOL</column>
    <column codepage="UTF-16" datatype="LONGVARCHAR" length="10">VCHARCOL</column>
    <column codepage="UTF-16" datatype="DECIMAL" precision="15"
    scale="5">DECIMALCOL</column>
    <column codepage="UTF-16" datatype="DECIMAL" precision="15"
    scale="5">NUMERICCOL</column>
    <column codepage="UTF-16" datatype="DECIMAL" precision="38">SMALLCOL</column>
    <column codepage="UTF-16" datatype="INTEGER">INTEGERCOL</column>
    <column codepage="UTF-16" datatype="SINGLEPRECISION">REALCOL</column>

    <!--More definitions can follow here -->
  </row>
</table>
```

## XML Schema Definition for a Bulk Data Configuration File

The bulk configuration XML schema governs bulk configuration files. The bulk configuration files in turn govern bulk load data files that are processed by PSQL Bulk Load.

This schema, which is published at <http://media.datadirect.com/download/docs/ns/bulk/BulkData.xsd>, provides a foundation for any third party functionality to be built using this standard. Custom applications or tools that manage large amounts of data can employ this schema as a loosely-coupled PSQL Bulk Load across ODBC, JDBC, and ADO.NET APIs and across multiple platforms.

If you want to generate CSV data that can be consumed by PSQL Bulk Load, you must supply an XML Schema for your XML configuration file.

Each bulk operation generates an XML configuration file in UTF-8 format that describes the bulk data file produced. If the bulk data file cannot be created or does not comply with the schema described in the XML configuration file, an exception is returned.



# IANA Code Page Mappings

## H

This following table maps the most widely used IBM code pages to IANA code page names.

IBM Number	IANA Code Page Name
37	IBM037
38	IBM038
290	IBM290
300	IBM300
301	IBM301
500	IBM500
813	ISO_8859-7:1987
819	ISO_8859-1:1987
857	IBM857
860	IBM860
861	IBM861
897	IBM897
932	IBM-942_P120-2000
939	IBM-939
943	Windows-932-2000 (for Windows clients)
943	IBM-943_P14A-2000 (for UNIX clients)
950	Big5
1200	UTF-16
1208	UTF-8
1251	Windows-1251
1252	Windows-1252
4396	IBM-930
5025	IBM5025
5035	IBM5035
5297	UTF-16

IBM Number	IANA Code Page Name
5304	UTF-8
13488	UTF-16BE



# Glossary

## I

---

**.NET  
Architecture**

Microsoft defines Microsoft .NET as a set of software technologies for connecting information, people, systems, and devices. To optimize software integration, the .NET Framework uses small, discrete building-block applications called Web services that connect to each other as well as to other, larger applications over the Internet.

The .NET Framework has two key parts:

- ASP.NET is an environment for building smart client applications (Windows Forms), and a loosely coupled data access subsystem (ADO.NET).
- The common language runtime (CLR) is the core runtime engine for executing applications in the .NET Framework. You can think of the CLR as a safe area – a "sandbox" – inside of which your .NET code runs. Code that runs in the CLR is called managed code.

**ADO.NET**

The data access component for the .NET Framework. ADO.NET is made of a set of classes that are used for connecting to a database, providing access to relational data, XML, and application data, and retrieving results.

**ADO.NET Entity  
Framework**

An object-relational mapping (ORM) framework for the .NET Framework. Developers can use it to create data access applications by programming against a conceptual application model instead of programming directly against a relational storage schema. This model allows developers to decrease the amount of code that must be written and maintained in data-centric applications

**assembly**

A compiled representation of one or more classes. Each assembly is self-contained, that is, the assembly includes the metadata about the assembly as a whole. Assemblies can be private or shared.

- Private assemblies, which are used by a limited number of applications, are placed in the application folder or one of its subfolders. For example, even if the client has two different applications that call a private assembly named formulas, each client application loads the correct assembly.

- Shared assemblies, which are available to multiple client applications, are placed in the Global Assembly Cache (GAC). Each shared assembly is assigned a strong name to handle name and version conflicts.

**bulk load**

Rows from the database client are sent to the database server in a continuous stream instead of in numerous smaller packets of data. Performance improves because far fewer network round trips are required.

**client load balancing**

A mechanism that distributes new connections in a computing environment so that no one server is overwhelmed with connection requests.

**code access security (CAS)**

A mechanism provided by the common language runtime through which managed code is granted permissions by security policy; permissions are enforced, limiting the operations that the code will be allowed to perform.

**common language runtime (CLR)**

The common language runtime (CLR) is the core runtime engine in the Microsoft .NET Framework. The CLR supplies services such as cross-language integration, code access security, object lifetime management, and debugging support. Applications that run in the CLR are sometimes said to be running "in the sandbox."

**connection failover**

A mechanism that allows an application to connect to an alternate, or backup, database server if the primary database server is unavailable, for example, because of a hardware failure or traffic overload.

**connection pooling**

The process by which connections can be reused rather than creating a new one every time the data provider needs to establish a connection to the underlying database.

**connection retry**

Connection retry defines the number of times the data provider attempts to connect to the primary and, if configured, alternate database servers after the initial unsuccessful connection attempt. Connection retry can be an important strategy for system recovery.

**Data Access Application Block (DAAB)**

A pre-defined code block that provides access to the most often used ADO.NET data access features. Applications can use the application block to pass data through application layers, and submit changed data back to the database.

**destination table**

In a PSQL Bulk Load operation, the table on the database server into which the data is copied.

**entity**

An entity is an instance of an EntityType. It has a unique identity, independent existence, and forms the operational unit of consistency. An EntityType defines the principal data objects about which information has to be managed such as person, places, things or activities relevant to the application.

**global assembly cache (GAC)**

The part of the assembly cache that stores assemblies specifically installed to be shared by many applications on the computer. Applications deployed in the

global assembly cache must have a strong name to handle name and version conflicts.

<b><i>isolation level</i></b>	<p>A particular locking strategy employed in the database system to improve data consistency. The higher the isolation level number, the more complex the locking strategy behind it. The isolation level provided by the database determines how a transaction handles data consistency.</p> <p>The American National Standards Institute (ANSI) defines four isolation levels:</p> <ul style="list-style-type: none"><li>■ Read uncommitted (0)</li><li>■ Read committed (1)</li><li>■ Repeatable read (2)</li><li>■ Serializable (3)</li></ul>
<b><i>load balancing</i></b>	See <a href="#">client load balancing</a> .
<b><i>locking level</i></b>	<p>A database operation that restricts a user from accessing a table or record. Locking is used in situations more than one user might try to use the same table or the same time. By locking the table or record, the system that only one user at a time can affect the data.</p>
<b><i>Logging Application Block (LAB)</i></b>	<p>A component of the Microsoft Enterprise Libraries that simplifies the implementation of common logging functions. Developers can use the Logging Block to write information to a variety of locations, such as the event log, an e-mail message, or a database.</p>
<b><i>managed code</i></b>	<p>Code executed and managed by the .NET Framework, specifically by the CLR. Managed code must supply the information necessary for the CLR to provide services such as memory management and code access security.</p>
<b><i>namespace</i></b>	<p>A logical naming scheme for grouping related types. The .NET Framework uses a hierarchical naming scheme for grouping types into logical categories of related functionality, such as the ASP.NET technology or remoting functionality. Design tools can use namespaces to make it easier for developers to browse and reference types in their code. A single assembly can contain types whose hierarchical names have different namespace roots, and a logical namespace root can span multiple assemblies. In the .NET Framework, a namespace is a logical design-time naming convenience, whereas an assembly establishes the name scope for types at run time.</p>
<b><i>Performance Monitor</i></b>	<p>A tool in the Windows SDK that identifies areas in which performance problems exist. You can use Performance Monitor to identify the number and frequency of CLR exceptions in your applications. The Performance Monitor (PerfMon) and VS Performance Monitor (VSPerfMon) utilities also allow you to record application parameters and review the results as a report or graph.</p>
<b><i>stream</i></b>	<p>An abstraction of a sequence of binary or text data. The Stream class and its derived classes provide a generic view of these different types of input and output.</p>

<b><i>schema collection</i></b>	Closely related schemas that can be handled more efficiently when grouped together. Database schema elements such as tables and columns are exposed through schema collections.
<b><i>strong name</i></b>	A name that consists of an assembly's text name, version number, and culture information (if provided), with a public key and a digital signature generated over the assembly. Assemblies with the same strong name should be identical.
<b><i>unmanaged code</i></b>	Code that is executed directly by the operating system, outside of the CLR. Unmanaged code includes all code written before the .NET Framework was introduced. Because it outside the .NET environment, unmanaged code cannot make use of any .NET managed facilities.