Progress®



Use ProDataSets

OpenEdge®

Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

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

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

February 2019

Last updated with new content: Release 12.0

Updated: 2019/03/01

Table of Contents

Preface....11
Purpose....11
Audience....11
Organization....12
Typographical conventions....12
Examples of syntax descriptions....13
Long syntax descriptions split across lines....15
Complex syntax descriptions with both required and optional elements....15
OpenEdge messages....16
Obtaining more information about OpenEdge messages....16

Chapter 1: Introducing the OpenEdge DataSet....19
Overview....19
Capabilities....20
ProDataSets and Microsoft ADO.NET DataSets....20
Standard ABL components....21
ProDataSet goals....21
Architecture....21
Using ProDataSets....22
ProDataSets and temp-tables....22
ProDataSet relations....24
Data sources....24
FILL operations....25
Integrating business logic into ProDataSets using events....26
Updating a ProDataSet....26
Typical use cases for ProDataSets....26
Defining a static ProDataSet....29
Static ProDataSet and its Data-Relations....29
Using the REPOSITION Data-Relation....32
Defining an inactive Data-Relation....34
Getting the handle to a static ProDataSet....34
Data-Source object....35
Defining a static Data-Source....35
Attaching a Data-Source to a ProDataSet buffer....38
Populating a ProDataSet....41
Filling the entire ProDataSet....42
Partially filling a ProDataSet....42
Defining a query on a child table....43

Recursively filling a ProDataSet....43
Summary of a FILL....44
Controlling the filling of each table....44
Testing the Order ProDataSet....45
Object life cycles with ProDataSets....49
Summary....50

Chapter 2: ProDataSet Parameters....51

Passing a ProDataSet as a parameter....51
Passing a ProDataSet with BY-REFERENCE or BIND....53
Passing a ProDataSet parameter by reference....53
Passing a ProDataSet parameter by binding....60
ProDataSet parameter table....62
Local parameter passing example....65
Deleting a dynamic ProDataSet passed as a parameter....65
Reducing the data to be passed in a parameter....66
Specifying the session startup parameter for all temp-table parameters....66
Setting attributes for individual temp-table parameters....67
Passing a ProDataSet with APPEND....68
Extending the sample procedure to pass a parameter....69

Chapter 3: ProDataSets Events....79

Event procedures for ProDataSets....79
Defining FILL events....81
Using event procedures in the sample procedure....84
Summary....90

Chapter 4: Dynamic ProDataSet Basics....91

Creating a dynamic ProDataSet....91
Passing ProDataSets....92
Deleting a dynamic ProDataSet....93
Specifying member buffers....93
Creating Data-Relation objects....94
Creating a dynamic Data-Source....95
Duplicating ProDataSets with the CREATE-LIKE method....97
Sample procedure: creating a dynamic ProDataSet....99
Summary....103

Chapter 5: ProDataSet Attributes and Methods....105

Accessing the handle of a ProDataSet....106
Accessing a member buffer of a ProDataSet....106
Sample procedures: using attributes and methods....107

Accessing Data-Relations....111
Using Data-Source attributes and methods....116
Enhanced query support for ProDataSet buffers....120
QUERY attribute for Data-Sources, Data-Relations, and buffers....121
KEYS attribute for buffers....121
TOP-NAV-QUERY attribute....122
Session attributes for ProDataSets, Data-Sources, and queries....122
Other ProDataSet methods....123
Building a dynamic user interface from a ProDataSet....123
Using the SYNCHRONIZE method....129
Sample procedure: adding REPOSITION and SYNCHRONIZE....130
Summary....133

Chapter 6: Updating Data with ProDataSets....135

Tracking changes in the temp-tables of a ProDataSet....135
ROW-STATE attribute....137
Creating or defining the before-tables....138
Locating rows in the before- and after-tables....140
Extending the sample procedures to track changes....141
Comparison with change tracking in .NET....143
Processing changes....144
GET-CHANGES method....144
MERGE-CHANGES and MERGE-ROW-CHANGES methods....146
ACCEPT-CHANGES and ACCEPT-ROW-CHANGES methods....148
REJECT-CHANGES and REJECT-ROW-CHANGES methods....148
SAVE-ROW-CHANGES method....149
DATA-SOURCE-ROWID attribute....151
MARK-ROW-STATE method....151
MARK-NEW method....152
Special support for change conflicts....153
Using the SAVE-WHERE-STRING attribute....155
Extending the samples to GET, SAVE, MERGE, and ACCEPT changes....156
Using the SAVE-ROW-CHANGES method in the update procedure....166
Using the MERGE-CHANGES method in the window procedure....166
Setting and using ERROR, ERROR-STRING, and REJECTED....167
Using the error attributes in the sample procedures....168
ProDataSet change events....171
Applying callback procedures programmatically....173

Chapter 7: Advanced Events and Attributes....175

Query OFF-END Event....176
Buffer BATCH-SIZE and LAST-BATCH attributes....178
NEXT-ROWID attribute....179

RESTART-ROWID attribute....179
RESTART-ROW attribute....180
ProDataSet buffer FIND-FAILED event....181
SYNCHRONIZE event for a ProDataSet buffer....182
Successive loading of ProDataSet data....182
Doing a partial ProDataSet FILL to return Order headers....185
Forcing the ProDataSet to be passed BY-VALUE....187
Filtering the top-level query based on the user selection....190
Returning the partial ProDataSet to the client....190
Retrieving detail for the ProDataSet....191
COPY-DATASET and COPY-TEMP-TABLE methods....193
Using COPY-DATASET with a dynamic target ProDataSet....197
Using the COPY-DATASET method for successive FILLs....199
Summary....202

Chapter 8: Batching Data with ProDataSets....205
Overview....205
Using the include-field list to limit the fields copied into the table....207
Setting up an event handler for the OFF-END query event....208
Setting up an event handler for the FIND-FAILED buffer event....210
Summary....212

Chapter 9: Advanced Read Operations....215
ProDataSets as a data access layer....216
Defining the right internal representation....216
Defining the right granularity for your ProDataSets....216
Defining the right top-level table for a ProDataSet....218
ProDataSets with more than one top-level table....218
Dynamic versus static ProDataSets....218
Sharing ProDataSet and temp-table definitions between procedures....218
Building a data access support procedure....219
Data access procedure example....219
Caching complex derived data in a ProDataSet 220
Summary of the data access procedure....223
Caching data using a ProDataSet....223
Using a subset of the tables in a ProDataSet....223
Summary of caching data using a ProDataSet....228
Creating views with ProDataSets as Data-Sources....228
Sample procedure: creating a view....228
Summary....232

Chapter 10: Advanced Update Operations 233
Creating a data access procedure for the Order ProDataSet....234

Building a business entity procedure to support the ProDataSet....238
Building general update procedures for client and server....241
Building the client side change handler....242
Building the server side change handler....243
Changing the window procedure to use the new procedures....246
Running standard validation procedures on update....248
Summary....251

Chapter 11: Data Access and Business Entity Objects....253
ProDataSets and the OpenEdge Reference Architecture....254
Data Access object....255
Isolating the data source from the internal view of data....255
Elements of a Data Access object....256
Data Access object template....262
Business Entity object....263
Elements of a Business Entity....263
Business logic options....270
Standard validation procedures....271
Accessing other entities in your business logic....271
Trigger procedure logic....272
Including context information in Business Entities....272
Summary....273

Index....275

Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages

Purpose

The Progress® ProDataSet™ represents a significant OpenEdge® technology for creating efficient "in-memory databases" from Progress® and non-Progress data sources. ProDataSets use familiar and new Advanced Business Language (ABL) elements, thereby making ProDataSets easy to absorb for experienced ABL programmers. This manual provides basic information about ProDataSets, expert-level insight into their mechanics, as well as forward-looking application design suggestions.

Audience

This Expert Series manual is full of the technical detail and design advice that experienced ABL programmers and application architects will need to fully exploit ProDataSet technology. For less experienced ABL programmers, there are many exercises that carefully walk through and develop ProDataSet sample code.

Organization

Introducing the OpenEdge DataSet on page 19

Provides a thorough introduction to the fundamental elements of the ProDataSet.

ProDataSet Parameters on page 51

Describes the basic parameters used with most ProDataSet applications.

ProDataSets Events on page 79

Describes the basic events used with most ProDataSet applications.

Dynamic ProDataSet Basics on page 91

Using dynamic ProDataSets presents expanded possibilities. This chapter covers their use in detail.

ProDataSet Attributes and Methods on page 105

Introduces more attributes and methods to further develop ProDataSet programming techniques.

Updating Data with ProDataSets on page 135

Covers two critical programming skills: tracking changes and processing changes. Together these techniques allow you to update your data sources with data from ProDataSets.

Advanced Events and Attributes on page 175

Provides a thorough introduction to the fundamental elements of the ProDataSet.

Batching Data with ProDataSets on page 205

Provides a thorough discussion of batching data with ProDataSets.

Advanced Read Operations on page 215

Provides some interesting use cases for ProDataSet reads.

Advanced Update Operations on page 233

Discusses several kinds of reusable procedures you may want to develop to execute specific ProDataSet tasks in a distributed environment.

Data Access and Business Entity Objects on page 253

Discusses a model architecture for the design of enterprise applications that exploit ProDataSets.

Typographical conventions

This documentation uses the following typographical and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.

Convention	Description
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax, code examples, system output, and file names.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax.
`Fixed-width bold`	Fixed-width bold italic indicates variables in syntax with special emphasis.
`UPPERCASE fixed width`	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
Period (.) or colon (:)	All statements except `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` end with a period. `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

Examples of syntax descriptions

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

Syntax

```
ACCUM aggregate expression
```

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

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

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

Syntax

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

In this example, you must specify MESSAGE and at least one *expression* or SKIP [(*n*)], and any number of additional *expression* or SKIP [(*n*)] is allowed:

Syntax

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

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

Syntax

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

Long syntax descriptions split across lines

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

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

Syntax

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

Complex syntax descriptions with both required and optional elements

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

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

Syntax

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

OpenEdge messages

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

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

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

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

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

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

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

Obtaining more information about OpenEdge messages

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

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

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

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

1. Start the Procedure Editor:

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

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

1

Introducing the OpenEdge DataSet

This chapter presents a thorough overview of the OpenEdge® ProDataSet™.

For details, see the following topics:

- Overview
- Using ProDataSets
- Defining a static ProDataSet
- Data-Source object
- Populating a ProDataSet
- Object life cycles with ProDataSets
- Summary

Overview

Advanced Business Language (ABL) is a powerful and flexible tool used to build most parts of an enterprise business application. Enabling application developers to capture domain expertise, produce intelligent designs, and quickly code the necessary business logic are special strengths of ABL. The result is full-featured and intelligent applications.

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

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time:

"ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an `ERROR`."

The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature:

"The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

Recent ABL versions increased the developers' ability to build applications that are more open to distributed access across many platforms. Your applications can access both OpenEdge and non-OpenEdge platforms using the newer OpenEdge AppServer, Open Client, and other technologies.

Capabilities

To extend both these capabilities in the latest major release, OpenEdge 10 includes a major new object called the Progress DataSet (commonly referred to as a *ProDataSet*). The ProDataSet:

- Extends your ability to define complex business objects with many levels of related data.
- Lets you define the relationships between those levels of data.
- Lets you associate each level with a distinct data source when you need to fill the ProDataSet with data or pass updates back to the database.
- Allows you to define a mapping between the database tables that are the source of your data and its representation within the ProDataSet. The internal view of the data can be significantly different from how it is physically stored.
- Defines hooks that let you associate your own custom procedures with many events in the life cycle of a ProDataSet.
- Logs changes to ProDataSet records so that you can pass the ProDataSet to a separate procedure to write those changes back to the database.
- Allows you to pass the ProDataSet as a single parameter with a single handle from one procedure to another, within a single ABL session or between sessions.

ProDataSets and Microsoft ADO.NET DataSets

ProDataSets map very closely to the ADO.NET DataSets defined by Microsoft as its new standard for data access. This mapping allows you to pass a ProDataSet from OpenEdge to .NET (by using the support for .NET in the OpenEdge 10 Open Client) and have it received as an ADO.NET DataSet with no loss of either data or definitional information. The same DataSet can then be passed back to business logic on an OpenEdge AppServer, which can process the changes and write them to the database.

In this way, the ProDataSet extends the power of expressing business logic in ABL and the requirements for open applications that can be partitioned between OpenEdge and other technologies, such as .NET.

Standard ABL components

At the same time, the ProDataSet is made up of mostly familiar components, in particular temp-tables, which you use to define the data at a particular level of the business object the ProDataSet represents. In this way, this powerful new tool can build on the strength of existing ABL components and existing procedural code while extending it in new ways.

The ProDataSet can be thought of as an "in-memory database" in the sense that it is filled with a set of related records in potentially multiple temp-tables that can then be traversed in a predictable way, and also passed as a single parameter from procedure to procedure or session to session with all the data being passed together.

ProDataSet goals

Overall, the fundamental **goal** is to define a representation of a set of related records that:

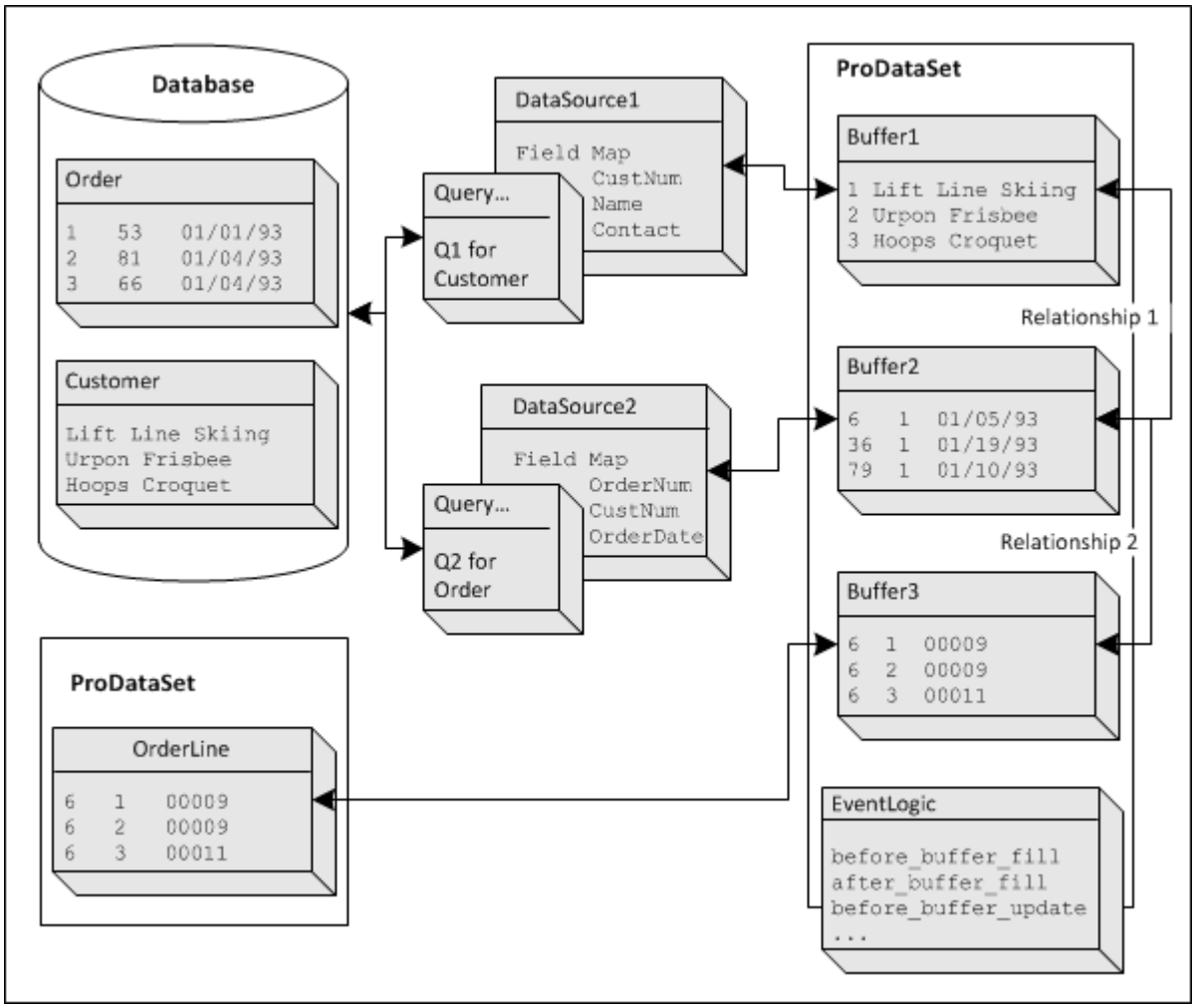
- Constitute the data for a single business object
- Provide a convenient, powerful, and consistent way to separate application data access from the specifics of the underlying data definition or data access mechanism
- Define not only the data but the relationships between tables that together make up the object
- Can be marshaled between procedures or between sessions as a single object, using a single parameter, extending the way temp-tables and their data are marshaled today
- Can be passed as parameters to .NET and other open technologies, and mapped to .NET data objects (and potentially other third-party objects)
- Can be represented by a single handle, with all the elements of the ProDataSet accessible through that handle
- Can be updated on either client or server in such a way that changes, adds, and deletions are captured, marshaled efficiently to the server, and written to the database or other data source under flexible transaction control
- Reuse and combine to the greatest extent possible existing objects in ABL, including queries, buffers, and temp-tables, in order to provide a programming experience that is familiar to today's developers and a convenient extension of existing procedures and programming techniques
- Can be represented as either a static or dynamic object, in a way consistent with other objects such as temp-tables
- Support events that allow developers to define business logic for the ProDataSet and its elements that is executed consistently and more or less transparently whenever that ProDataSet object is populated or updated
- Provide a migration strategy for the SDOs and SBOs of the ADM2 and Progress Dynamics®, improving performance while reusing business logic defined for those objects with minimal change

Architecture

The ProDataSet object is basically a collection of one or more member temp-tables. It also optionally contains a collection of Data-Relations among the member tables. Each ProDataSet member table can attach to a Data-Source object that allows filling the ProDataSet table from the source, or updating the source from the table. In addition, you can attach ABL procedures to numerous events that occur during the life cycle of a ProDataSet so that you can customize its behavior.

The following figure shows an example of the overall architecture.

Figure 1: ProDataSet architecture



Using ProDataSets

This section introduces the key concepts you need to successfully understand and use ProDataSets.

ProDataSets and temp-tables

The ABL temp-table already provides a large measure of support for managing sets of data independent of the database. Because the ProDataSet is in many ways an extension of that support, it is worth briefly reviewing some of the temp-table characteristics that are also part of the ProDataSet.

Temp-tables compared to database tables

A temp-table allows you to define a table that is not part of any persistent database. You can use temp-tables within a session or pass them between sessions. Temp-tables provide in-memory buffering of data and transparent overflow to a temporary database on disk when necessary. You can define a temp-table as being `LIKE` a database table, and you can add fields of any type to this definition, or you can define the table as a set of fields independent of any particular database table. You can create, read, update, and delete records in the temp-table much as you can with records in a database table. You can define indexes to manage large sets of data efficiently. You can access the records in the table with `FIND` and `FOR EACH` logic or with queries.

Static or dynamic temp-tables

You can define a temp-table in a static definition, specifying all its fields and indexes in a single statement. You can also create a temp-table as a dynamic object with a handle and then add fields, buffers, and indexes to it in individual successive statements, finalizing the temp-table structure with a `TEMP-TABLE-PREPARE` method. You can also retrieve the handle to a static temp-table and then access its attributes, the same as you can a dynamic temp-table, in order to see what fields and indexes it contains, what its default buffer is, and so forth.

Passing temp-tables

You can pass a temp-table between procedures within a session or between sessions, such as between a server session that loads database data into the table and a client session that views the data and perhaps allows changes. You can pass a temp-table as a static parameter and receive it as a static or dynamic object. You can also pass a dynamic temp-table and receive it as either a static or dynamic object.

In all these cases, the names of the tables and fields do not need to match on the two sides of the call. Only the essentials of the field definitions—their number, sequence, and data types—need to match. Field names along with the format and other attributes can be different, as can index definitions, because indexes are rebuilt as the temp-table is received.

In addition, you can pass a temp-table by `HANDLE`, in which case only the handle as a pointer to either a static or dynamic temp-table is passed. Nothing is copied from caller to callee. Because handles are not valid between sessions, and because there is no way to share either the definition or data across sessions without copying it, the `HANDLE` form can be used only between procedures in the same session.

ProDataSet comparison

A ProDataSet is constructed from multiple temp-tables and shares and extends all of these temp-table characteristics:

- The temp-table definitions that make up the ProDataSet allow developers to define, in a standard way, a level of indirection between the actual data configuration in a database, or other data source, and the data definition that the application uses. These table definitions can mask joins between tables in order to simplify data representation. They can rename fields to match names used in business logic or to standardize naming. They can combine data from one or more databases with data from other data sources. They can mask changes to the underlying data structure, for example, as database design work is done to gradually clean up a complex and inconsistent older design. Temp-tables allow all these things today, but the ProDataSet makes it more straightforward for developers to standardize their design on this view of data.
- Developers can define ProDataSets statically or dynamically. The ProDataSet, whether static or dynamic, has a single handle that you can use to access its methods and attributes. You can pass the entire ProDataSet as a single parameter to a procedure within the same session, in another OpenEdge session, or to an application running on an entirely different platform such as Microsoft .NET.

- The ProDataSet supports the parameter forms `HANDLE`, `DATASET`, and `DATASET-HANDLE`. `Like` the `TABLE` parameter form for an individual temp-table, the `DATASET` form passes the ProDataSet as a static parameter. Similarly, the `DATASET-HANDLE` form is like `TABLE-HANDLE` in that it passes a dynamic reference to the ProDataSet. When the procedure call is within a session, the ProDataSet is normally passed by reference for maximum efficiency. When the call is remote, the entire definition of the ProDataSet, with its temp-table buffers, relationships between buffers, and so on, is marshaled along with the data just as is done for temp-tables today.

ProDataSet relations

A single temp-table lets you define any two-dimensional set of data. A business application typically must be able to treat a number of different sets of data as a single object, representing header and detail records, various parent-child relationships, useful lists of codes or other lookup values that are part of the validation logic, and so forth. The fundamental feature of a ProDataSet beyond what you can do with a single temp-table is that it allows you to define these relations as part of the ProDataSet definition. It also uses relations when it loads data into the ProDataSet and also to filter data in a useful way when application code is navigating a ProDataSet after it has been populated.

Part of the ProDataSet definition, whether static or dynamic, is a set of relation objects, called *Data-Relations*, each of which defines a relationship between a parent buffer and a child buffer in the ProDataSet. These relations name the fields in parent and child that form a primary-foreign key relationship between the buffers. The AVM uses this information when the ProDataSet is populated, or filled, with a set of related records to retrieve child records related to a parent automatically.

The Data-Relation is also used after the ProDataSet has been filled, in what we can term "navigation" mode, when the application, either through a user interface or in business logic, needs to traverse the ProDataSet to display or examine the data, make changes, add records, and so on. In this mode, the AVM uses the Data-Relation to provide an automatically generated dynamic query on each child buffer of a relation that filters the records at that level to include only those related to the currently selected parent. This is helpful when the records in one of the temp-tables are displayed in a browse, or when the application code needs to traverse records related to the current parent.

The ProDataSet supports any number of levels of relation, so that you can represent parent-child-grandchild relationships in a complex business object using a single ProDataSet.

You can deactivate Data-Relations and then later reactivate them. You can do this to populate portions of a ProDataSet at a time. Or you might deactivate a relation because it is more efficient to populate all the records for the child table at once, rather than selecting them individually for each parent record.

Data sources

Each temp-table in a ProDataSet can be loaded from a different data source. Updates made to a ProDataSet also need to be applied back to the data source. In the simplest case, the data source is an OpenEdge database or another database type that you can access through an OpenEdge Data Server. To support this, there is a *Data-Source* object that you define independent of a particular ProDataSet, and then attach to the ProDataSet when you need to fill data from the Data-Source or apply updates back to it. This allows you to pass a ProDataSet from one session to another without any database-specific definitions or dependencies going along with it. Also, in some cases you might need the same ProDataSet to attach to different Data-Sources at run time.

When a ProDataSet buffer has a Data-Source, the ProDataSet can perform a fill operation for the buffer's temp-table in one of two ways. If the buffer is for a top-level temp-table in the ProDataSet, meaning that it is not the child in some Data-Relation, then you can define a query that is referred to in the Data-Source definition to tell the AVM what records to load. This might identify, for example, a single header record representing something like a Purchase Order, which has many related child records in other tables. Or it could be a set of records satisfying some condition. If there is no query, then the top-level table is loaded with all the records from the underlying database table. This might be appropriate when the table represents a list of coded values such as states or order status codes that are to be used as a lookup list within the ProDataSet.

If a buffer is for a child of a Data-Relation, then the AVM can fill the child temp-table without any explicit query definition simply by examining the Data-Relation between it and the parent, and retrieving all child records for each parent. This is the standard case for a child of a Data-Relation. You can still define a query for the child buffer if the default selection based on the relation is not sufficient to identify the right records to retrieve, or if the query involves a join to one or more other tables that contribute fields to the temp-table. Note that if you define your own fill queries, or add rows to the ProDataSet's temp-tables in any other way, the AVM does not enforce the field relationships defined by the Data-Relation for you. You will not get an error if you add rows to a child table that do not match a row in the parent table.

One of the important values of the ProDataSet is that the contents of its tables, and therefore the business logic that uses its records and the user interface logic that displays them, can be defined independent of the nature or structure of the actual data sources. In some cases this simply means that the ProDataSet's temp-tables allow you to define internal tables whose fields come from different database tables, or have different names, or represent expressions, or are different in some other way from the underlying database.

In addition, you can fill a ProDataSet without having a Data-Source for a member buffer at all. If this case, you must supply the code that fills that buffer's table. Whether there is a Data-Source or not, an event is triggered that can execute application-specific code to handle the fill (if there is no Data-Source) or to extend the default behavior (when there is a Data-Source). This is just one example of the usefulness of events that allow custom code to either augment or replace default behavior, or to provide behavior where there would be no default.

In addition, because each buffer in the ProDataSet has its own Data-Source, a single ProDataSet can be filled, in a single operation, from a variety of different data sources that represent different databases or in some cases non-database customized data sources. And data can be loaded from the disparate sources in a coordinated, interleaved fashion so that you can be sure that the integrity of the Data-Relation is maintained during the fill. These are capabilities that .NET does not provide for its DataSets.

FILL operations

A ProDataSet is exactly what its name implies—a set of related data. It is typically populated in a single operation, where one or more top-level records and all their related records are retrieved at once. This is called a `FILL` operation, or just a `FILL`. You can define a query for the top-level buffer (or for each top-level buffer if there is more than one) and execute a `FILL` on the ProDataSet itself, in which case all of its tables are loaded. The AVM executes the queries for top-level buffers and also for other buffers that have specific queries. The ProDataSet follows the Data-Relations to identify and load the children for each parent. A simple example is a purchase order header (where the top-level query identifies a single `Order` by its key), plus all of the `OrderLines` for the `Order`, plus `Items` for the `OrderLines`, plus `Inventory` for the `OrderLines`, and so on.

In other cases, you might want to populate only a part of a ProDataSet at one time. To do this, you can run the `FILL` method on a buffer rather than on the ProDataSet as a whole, and it is filled from there on down. When you do this, you can also deactivate Data-Relations in order to limit the scope of the `FILL`. You can also execute a `FILL` multiple times to add more data to a ProDataSet that already has data. Each temp-table has a `FILL-MODE` that allows you to specify that a table should be emptied before it is filled again, or that it should be skipped altogether, or that new records are simply to be appended to existing ones, or merged with existing ones by checking for duplicates.

Integrating business logic into ProDataSets using events

One of the most basic advantages of OpenEdge as a development environment is the tight integration between the data access layer of an application and the business logic that operates on that data. This is something that set-oriented data access languages like SQL, combined with programming languages that were generally not designed specifically for data navigation, have never been able to match.

The ProDataSet extends this integration by allowing developers to associate business logic procedures with ProDataSets, to be executed when specific events occur, such as beginning a ProDataSet `FILL` operation, loading a record into one of its member buffers, making a change to a row, and so forth. This event logic can be written in separate procedures from the procedure where the ProDataSet is defined and used, so that it can be coded independent of the ProDataSet and attached to the ProDataSet in any procedure where it is used. Generally speaking, these event procedures receive the ProDataSet as an `INPUT` parameter so that they can examine and modify any records in the ProDataSet, depending on the nature of the logic and the event. This provides developers with a standard way to associate business logic with a ProDataSet to extend or replace the default behavior the ProDataSet provides, or to provide behavior where no default support is possible. This logic is executed consistently wherever the ProDataSet is used so that the developer can write business logic in one place and know that it will be applied uniformly.

Updating a ProDataSet

There is also support for updating data through a ProDataSet. This can be more complicated than performing individual database record updates because a ProDataSet, by its very nature, represents a set of related records, possibly in many tables, which are likely to be changed in one session (in a client session, typically) and then returned to a server session to be applied to the database or other data source. The ProDataSet therefore needs a standard way to keep track of multiple changes, to identify which records have been modified, added, or deleted. It also needs to keep track of the original or before-image versions of modified or deleted records, so that the server can verify whether records have been changed by other users since they were read, and then apply all changes to the database either in a single transaction or in multiple transactions, as appropriate.

The server must also be able to return errors to the client and pass back the final versions of records that might have been further changed on the server. All this is done in a consistent way with a substantial level of default support from the ProDataSet methods, plus events to allow the process to be extended through custom ABL code at all the appropriate places. As changes are made to the ProDataSet's tables, they are logged within the ProDataSet so that, when the ProDataSet is returned to the server, processing the changes and applying them to the database is straightforward.

Typical use cases for ProDataSets

Fundamentally, you can think of a ProDataSet as an in-memory data store that holds a set of related records and is aware of their relationships. You now know that you can pass a ProDataSet as a single object from one session to another. Typically, this would be from an AppServer session where the necessary database connections or other data sources are available, to a client session where there is a user interface or other code that uses and possibly updates the data. The client session could be an ABL session (for example one running WebClient), or a non-ABL session (for example a Microsoft .NET application).

In a typical interaction, a client-based session requests a ProDataSet from the server as a result of a UI event, such as the user requesting the data for a particular purchase order. The client runs a procedure on the server that has the PO number as an `INPUT` parameter and the PO ProDataSet as an `OUTPUT` parameter.

On the server side, the application procedure typically provides a static definition for the ProDataSet, attaches Data-Sources to the ProDataSet's buffers, and executes a `FILL` operation to load a set of related data for the PO into the ProDataSet. It then returns the ProDataSet to the caller on the client.

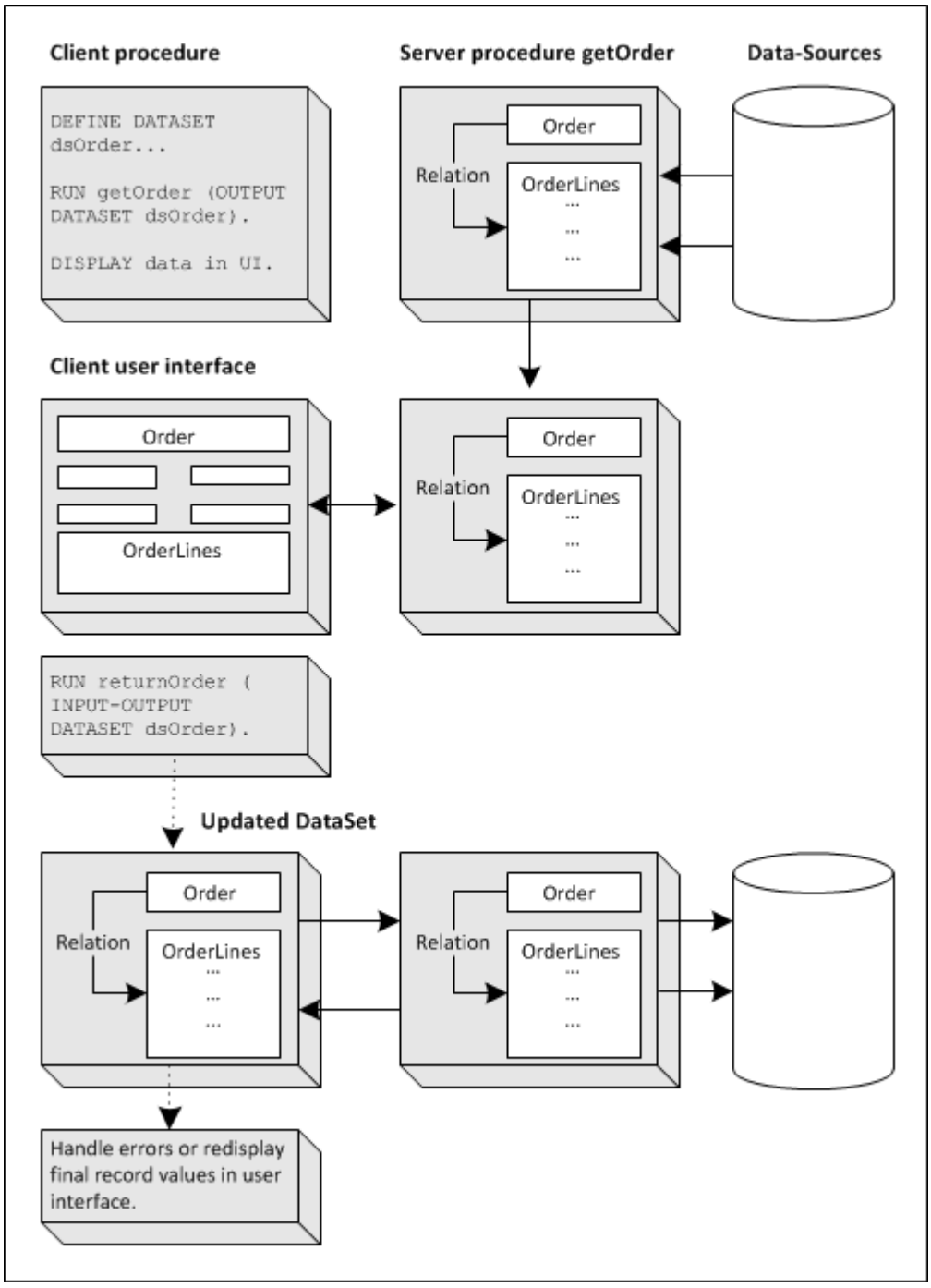
The client session receives the ProDataSet as an `OUTPUT` parameter into its own ProDataSet definition. It then makes the ProDataSet available to the user interface. Individual temp-tables might be displayed in different browses and individual records from a table displayed in a Viewer or single-record frame where record detail is displayed and updated. The client can update different records in any of the ProDataSet's tables, add records to some tables (such as new `OrderLines` for the PO), and delete other records. Lists of lookup codes that are part of the ProDataSet can be displayed, in browses of their own or in drop-down lists, and cached on the client for future use within that session. Some of the updates and other user interactions could require additional calls to the AppServer for validation or for additional information. This is all application-dependent.

When the user is finished making a set of changes, a client procedure runs another procedure on the server that accepts the changes made to the ProDataSet as an `INPUT-OUTPUT` parameter. If the ProDataSet contains a large number of records that have not been changed, then the client can return only changed records and their before-images to reduce the amount of data sent across the network. Typically, the original ProDataSet is no longer available on the server when the changes are passed back. This is especially true when the environment uses a stateless AppServer connection, which is the norm for most modern applications. The server-side procedure reads the contents of the ProDataSet, and possibly other parameters that accompany it, and applies changes to the data sources based on the before-image records in the ProDataSet that indicate what the changes were.

The server procedure then performs any required validation of the data. This normally is encapsulated in a server-side logic procedure. The server procedure can pass errors and other server-side changes made to the records back to the client.

This is the basic client-to-server round-trip scenario. This works more or less exactly the same whether the client is an OpenEdge session or a .NET session. The following figure illustrates this scenario.

Figure 2: Typical client/server round trip with a ProDataSet



The second basic use case is for strictly server-side business logic. In this scenario, a business logic procedure on the server uses one or more ProDataSets to access the application data it needs to look at. The ProDataSets encapsulate the data in a standard way, insulating the logic from the actual structure of the data sources. They also apply update validation logic and other ProDataSet behavior in a standard, consistent way. The business logic therefore fills a ProDataSet with data it needs to see or change, and then applies updates back to the database. This is somewhat different from the individual `FIND`, `FOR EACH`, and `ASSIGN` statements an application would typically use today to access various tables it needs to look at, but this change in programming standards reflects the degree of encapsulation we are trying to achieve. In this second type of use case, there is no need for client/server interaction or passing ProDataSets between sessions. The ProDataSet is strictly being used to encapsulate data definition, data access, and update logic in a standard way. This is an essential part of a stable, future-proof application architecture.

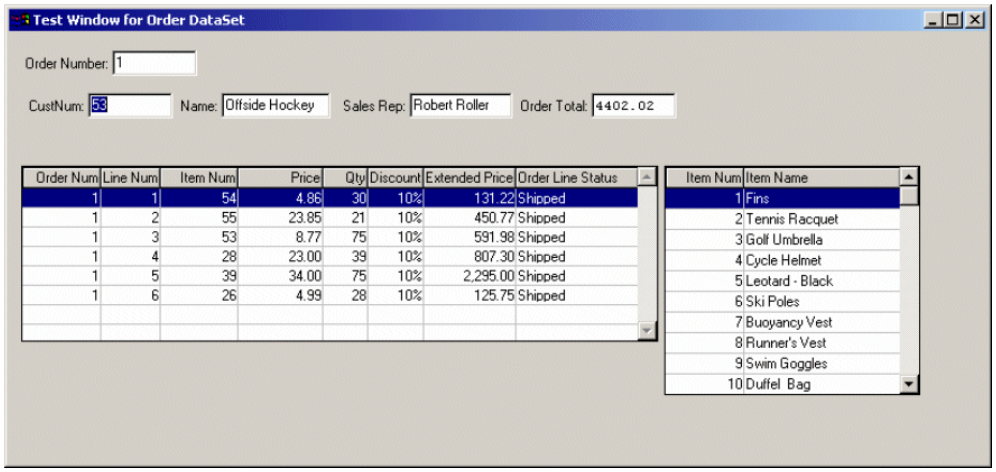
There are many other types of use cases as well, including:

- Using update events and custom business logic procedures to provide validation logic on both the client and server, where the client-side logic is restricted to what can be evaluated without direct database access
- Building a ProDataSet from a set of complex calculated data, such as a price sheet, that is expensive to derive from the original data sources but needed for frequent reference once it is loaded
- Using a ProDataSet as a mechanism to pass a number of different but possibly unrelated tables of data, such as lookup values, to the client session to use in building lookup lists or validating client field values

Defining a static ProDataSet

This section covers all the aspects of defining a ProDataSet and its Data-Sources as static objects. The section builds an example ProDataSet definition along with its event procedures and a sample window to display its data, as shown in the following figure.

Figure 3: Test window for order ProDataSet



Static ProDataSet and its Data-Relations

As with other ABL objects, there is a `DEFINE` statement for a static ProDataSet, which allows you to name the ProDataSet, identify the temp-table buffers it incorporates, and define the Data-Relations between those buffers in a single statement. The Data-Relation is not defined as a separate object because the relations have no significance outside the scope of a particular ProDataSet.

This is the syntax for the static `DEFINE DATASET` statement:

Syntax

```
DEFINE [NEW [SHARED] ] DATASET dataset-name FOR buffer-name …
  [, buffer-name ] [ DATA-RELATION [ data-rel-name ] FOR data-rel-spec ] …
```

Where:

- *dataset-name* is a standard ABL object name.
- *buffer-name* is a static buffer name for a previously defined temp-table whose scope includes the procedure where the ProDataSet definition is.
- *data-rel-name* is the optional name given to an optional Data-Relation, which can be used to obtain the handle of the Data-Relation at run time. The default name is `Relation`*n* (with *n* starting at 1 for each ProDataSet). The presence of this option requires the `FOR` keyword.
- This is the syntax for a *data-rel-spec*:

```
parent-buffer-name, child-buffer-namefield-mapping-phrase
  [REPOSITION] [NESTED] [RECURSIVE] [NOT-ACTIVE].
```

- *parent-buffer-name* and *child-buffer-name* are member *buffer-names* from the `FOR` phrase of the definition, identifying the parent and child of a relation.
- This is the syntax for a *field-mapping-phrase*:

```
RELATION-FIELDS (parent-field1, child-field1 [, parent-fieldn,
  child-fieldn ] …)
```

The first field of each pair of `RELATION-FIELDS` is from the parent buffer, the second field is from the child.

In some cases, there might be a relation between two buffers that cannot be defined in terms of an equality match between fields. This might be if the relation is based on a concatenation of fields or some other expression, or if special application logic is required to identify the related records. It is then the responsibility of the developer to define a query on the child buffer's Data-Source that identifies the correct records, or to supply custom logic in response to the `FILL` events to take over complete responsibility for filling that level of the ProDataSet. Any fields in this query that are not also in the data-relation will not be used by the ProDataSet for navigation after it has been filled.

Because a ProDataSet is made up of ABL temp-tables, you must define those temp-tables before you reference them in the ProDataSet.

For the example window, there are three temp-table definitions. To make it easier to use those definitions in more than one procedure, the definitions are in the include file dsOrderTT.i, as shown:

```
/* dsOrderTT.i -- include file for Temp-Table defintions in dsOrder. */
DEFINE TEMP-TABLE ttOrder NO-UNDO LIKE-SEQUENTIAL Order
  FIELD OrderTotal AS DECIMAL
  FIELD CustName LIKE Customer.Name
  FIELD RepName  LIKE SalesRep.RepName.
DEFINE TEMP-TABLE ttOline NO-UNDO LIKE-SEQUENTIAL OrderLine.
DEFINE TEMP-TABLE ttItem NO-UNDO
  FIELD ItemNum  LIKE ITEM.ItemNum
  FIELD ItemName LIKE ITEM.ItemName
  FIELD Price    LIKE ITEM.Price
  FIELD Weight   LIKE ITEM.Weight
  FIELD OnHand   LIKE ITEM.OnHand
  FIELD OnOrder  LIKE ITEM.OnOrder.
```

The first temp-table, ttOrder, is based on the Order database table but adds three fields to it:

- The first additional field is a calculated field that shows the total of all OrderLines for the Order. You will write event logic later on to calculate this field for each selected Order.
- The second field is the Customer Name from the Customer table. You will define the join for this table in the query for the ttOrder table's Data-Source.
- The third field is the Sales Rep name from the SalesRep table. You will also join this table into your Data-Source.

The second temp-table, ttOline, is exactly like the OrderLine database table. The third temp-table, ttItem, uses a subset of the fields in the Item database table.

Note that the ttOrder and ttOline temp-tables are defined using the LIKE-SEQUENTIAL option. This option is preferable to the LIKE option, especially in a client-server deployment scenario. Unlike the LIKE option, which creates temp-table fields in metaschema _field._field-rpos order (POSITION order in the .df schema definition file) of the source table's fields, LIKE-SEQUENTIAL creates fields in _field._order sequence, as defined in the Data Dictionary.

Note: Using the DEFINE TEMP-TABLE*table*LIKE or LIKE-SEQUENTIAL option requires that the client have a connection to the database, otherwise the compile will fail. You can always compile your application on the server, where the database connection exists, then deploy your client-side components where there is no database connection.

Tip: When you design your ProDataSets, define temp-tables that reflect the proper internal representation of the data, regardless of what the external structure of your database might be. This way you can write your business logic and data display logic to reference properly normalized and properly organized data, in the form best suited to the data and the application. You can use ProDataSet mapping features and your own custom logic to map that internal representation to your existing database design. If your database design requires change or cleanup at a later date, you can simply change the mapping code so that the internal representation stays the same. You can also use a temp-table as a target for complex data calculations far removed from specific fields in your database, and also for data that is derived from a data source other than your database, and keep the internal form the same.

Once you have defined the temp-tables your ProDataSet needs, you define the ProDataSet itself. In this case, the ProDataSet definition is also in an include file, `dsOrder.i`, as shown:

```
/* dsOrder.i -- include file definition of DATASET dsOrder. */
DEFINE DATASET dsOrder FOR ttOrder, ttOline, ttItem
  DATA-RELATION OrderLine FOR ttOrder, ttOline
    RELATION-FIELDS (OrderNum, OrderNum)
  DATA-RELATION LineItem FOR ttOline, ttItem
    RELATION-FIELDS (ItemNum, ItemNum).
```

Note that in this exmaple, the parent and child fields in the field mapping phrases happen to have the same names, `OrderNum` and `ItemNum`, but this is not always the case.

Tip: To have a consistent definition of a static ProDataSet that you only need to maintain in one place, keep your ProDataSet definitions in include files that you include where the ProDataSet is referenced. Generally, keep your temp-table definitions in separate include files so that you can maintain them separately, and so that you can include one without the other when this is needed (for example when you need to define the temp-tables separately in the AppBuilder's temp-table utility).

Each Data-Relation defines an implicit query for the `FILL` operation. When you attach a Data-Source to `ttOline` at run time that uses the `OrderLine` table in the database, then the predicate for this implicit query is:

```
WHERE OrderLine.OrderNum = ttOrder.OrderNum
```

This way the ProDataSet automatically loads all `OrderLines` for the current record in the parent temp-table `ttOrder`.

Likewise, the Data-Relation from `ttOline` to `ttItem` selects the `Item` for each record in `ttOline` as the `ttOline` table is filled. This results in the ProDataSet containing all `Items` for the `OrderLines` that are part of the specified top-level `Orders`. Later, you will modify the ProDataSet so that it also loads all Items into the `ttItem` table. The initial definition provides the flexibility to load various combinations of data into the ProDataSet at run time by changing or deactivating relations.

Using the REPOSITION Data-Relation

You can include the `REPOSITION` option on a Data-Relation definition that is part of a ProDataSet definition. As you will later see, there is also a `REPOSITION` logical attribute on a Data-Relation handle that lets you set this mode on or off at run time.

The purpose of the reposition mode on a relation is to handle the situation where you want to load a limited set of child records into a ProDataSet table that is related to possibly multiple parent rows of another table in the ProDataSet. For example, you might want to include a separate State table with state abbreviation and full state name in your ProDataSet rather than joining that information into each individual `Customer` or `Order` record in the ProDataSet. Or you might want to include an `Item` table with all available `Items` in your ProDataSet and be able to join an `Item` to all of the `OrderLines` that include that `Item`.

If `REPOSITION` mode is true at the time of a `FILL`, either because the option is part of the definition or because you have set the attribute to true, then the AVM ignores the Data-Relation. Otherwise the relation would filter the children to retrieve only those records related to a parent of the relation, and fill the table as a top-level table.

Note: A Data-Relation defined with REPOSITION or SELECTION is overridden if a query is defined for a child relation Data-Source. Normally, if REPOSITION is not specified, the child query selects the children of the parent. But coding a query for the child Data-Source overrides whether or not a SELECTION or REPOSITION relation mode was defined.

Example with REPOSITION not set

For example, consider an Order ProDataSet with a Data-Relation from ttOrderLine to ttItem, with relation fields of ttOrderline.ItemNum and ttItem.ItemNum. If you do not define or set the REPOSITION attribute, then the AVM loads into ttItem the Item information for every Item used in at least one OrderLine that is also loaded into the ProDataSet. There might be multiple OrderLines that use the same Order, but because the default fill processing is to discard records with duplicate keys, the AVM automatically discards all duplicates based on the unique index definition for the ttItem temp-table, and you wind up with exactly one instance of each Item used on the ProDataSet's OrderLines. If there is no unique index on the ttItem temp-table, then you wind up with one instance of ttItem for each OrderLine that uses a given Item, which is almost certainly not what you want.

Example with REPOSITION set: loading

Alternatively, if you define or set the REPOSITION attribute of the relation, when the AVM fills the ProDataSet, it disregards the relation fields of that Data-Relation and fills that table independent of its parent. This means that, by default, the AVM loads all records from the child table's database buffer into the temp-table. Or, if you have prepared a specific query for the child table's Data-Source, then the records that satisfy the query are loaded.

Using the same example for OrderLines and their Items, with REPOSITION set to true, the AVM loads all Items into ttItem regardless of whether or not they are used by OrderLines in this instance of the ProDataSet. This makes the entire list of Items available for display and selection in the user interface, or within business logic that uses the ProDataSet. You can set REPOSITION true or false for this type of relation, depending on your application requirements.

Example with REPOSITION set: navigation

During navigation, REPOSITION mode serves a somewhat different purpose. As you learned in the introduction, the AVM generates a dynamic query for each child table of a Data-Relation. You can use this query to browse the current records at that level, where the meaning of "current" is normally the set of children for the currently selected parent. If REPOSITION is true when your application is navigating the ProDataSet, then the query works differently. The query does not filter children, but selects all rows in the child table regardless of the parent. However, the child query is **repositioned** to the child row for the currently selected parent.

Once again using the same example, if your user interface uses browse objects to show OrderLines of the current Order in one browse and Items in another browse, then with REPOSITION false, the Item browse will show only the one ttItem row for the current ttOrderLine. With REPOSITION true, the ttItem browse displays all Items and repositions to the Item for the current OrderLine. This REPOSITION mode has no other effect when you are navigating the ProDataSet. Again, set REPOSITION to true or false for this kind of many-to-one relationship, depending on which default behavior you want.

Note: The ATTACH-DATA-SOURCE( ) method of the buffer handle provides much of the internal definition for the FILL. Among other things, this method creates the FILL query. If you want to change the REPOSITION attribute, you must do it before the ATTACH-DATA-SOURCE( ) method. If you need to do it afterward, you must do a detach and attach on the child table.

Defining an inactive Data-Relation

The NOT-ACTIVE option on the static Data-Relation definition is used to define a relation as inactive at compile time. This allows you to have two relations between the same two ProDataSet temp-table buffers, but only have one active at a time.

One reason you might want multiple Data-Relations between the same buffers is because it is efficient to fill the ProDataSet using one parent-child hierarchy, but to navigate using a reverse hierarchy. A *data-rel-spec* such as "FOR Customer, Order" might be good for navigation, but if the source is based on order.orderdate, then "FOR Order, Customer" is better for filling the ProDataSet.

For dynamic ProDataSets, an optional parameter of the ADD-RELATION method is used to define a Data-Relation as inactive. For more information about this method, see Creating Data-Relation objects on page 94.

At run time, you selectively enable or disable a Data-Relation using the ACTIVE attribute on the Data-Relation handle.

Getting the handle to a static ProDataSet

Later you will learn how to create a dynamic ProDataSet that you access through its handle. In the meantime, it is important to know that you can access the handle of a static ProDataSet in the same way that you can for other ABL objects. To get an object handle to a static ProDataSet, you simply assign the ProDataSet's HANDLE attribute to a variable of type HANDLE. This is the syntax for the ProDataSet object HANDLE attribute:

Syntax

```
handle-var = DATASET dataset-name:HANDLE.
```

In the case of the sample ProDataSet:

```
DEFINE VARIABLE hDSOrder AS HANDLE NO-UNDO.

hDSOrder = DATASET dsOrder:HANDLE.
```

Through the handle to a ProDataSet, you can access all its attributes and methods, along with its sub-elements (its temp-tables, queries, buffers, relations, and so on.) and their attributes and methods.

As we go along, you will learn the new object attributes defined for the ProDataSet. Standard ABL object attributes that are valid for ProDataSets include:

- ADM-DATA
- DATA-SOURCE-MODIFIED
- DYNAMIC
- ERROR
- HANDLE
- INSTANTIATING-PROCEDURE
- NAME
- NAMESPACE-PREFIX

- NAMESPACE-URI
- NEXT-SIBLING
- NUM-BUFFERS
- NUM-REFERENCES
- NUM-RELATIONS
- NUM-TOP-BUFFERS
- PRIVATE-DATA
- REJECTED
- RELATIONS-ACTIVE
- TOP-NAV-QUERY
- TYPE
- UNIQUE-ID

Data-Source object

There is a new object called a Data-Source to support the filling of a ProDataSet temp-table from one or more database tables, as well as saving ProDataSet updates back to the database. You define a distinct Data-Source for each member buffer, which allows a single ProDataSet, and a single FILL operation on that ProDataSet, to combine data from multiple tables or even databases.

Defining a static Data-Source

Before you can populate a ProDataSet from a database, you must define a Data-Source object for each of its member temp-table buffers. A Data-Source names either a database buffer that supplies fields for a ProDataSet temp-table or a query name, which in turn references one or more buffers as well as defines a specific set of retrieval criteria. If you simply name the buffer, then the AVM determines the query when a FILL occurs by examining the Data-Relation between the ProDataSet member buffer for the Data-Source and its parent. If it has no parent and there is no query definition, then all records from the database table are retrieved. If the Data-Source involves a join between two or more database tables, the user-written query is required to describe the relationship between the database tables.

The Data-Source is defined independently of any ProDataSet. This is the syntax for the DEFINE DATA-SOURCE statement:

Syntax

```
DEFINE DATA-SOURCE dsource-name FOR
  [ QUERY query-name ] [ source-buffer-phrase [, source-buffer-phrase ] … ].
```

In this statement:

- *dsource-name* is the name you are giving to the Data-Source, as you would for any other static object.
- *query-name* is the name of a query you have defined separately, before the Data-Source definition.

This is the syntax for the *source-buffer-phrase*:

Syntax

```
buffer-name [ KEYS ( { field [,field ]… | ROWID } ) ].
```

In this phrase:

- *buffer-name* is a buffer name for a database table.
- *field* is a field name in that table.
- *ROWID* can occur exactly once in place of a field list to use the ROWID of the database record as the key. If you do this, you must define a field in the ProDataSet temp-table to hold the value of the ROWID and map the ROWID to that field when you attach the Data-Source to the ProDataSet temp-table buffer.

Note that the QUERY phrase and the *source-buffer-phrase* are not mutually exclusive. If you specify a query, then the query definition itself names the buffer or buffers it uses. However, you might still want to include a *source-buffer-phrase* in the Data-Source definition in order to define the fields that make up the unique key the AVM should use to eliminate duplicate records when it is filling the table from the Data-Source, or to locate the database records for a temp-table record when you have updated the record.

If you specify a *source-buffer-phrase* without the QUERY phrase and the Data-Source is for a child of a Data-Relation, then the AVM can generate the correct query when you fill the ProDataSet to load children of each of the parent records. Otherwise, the default is to load all records from the database table. If there is no QUERY phrase, then you can have only one *source-buffer-phrase* if you intend to use the Data-Source to fill a ProDataSet. The AVM cannot automatically join multiple tables without a query definition.

Note that if you use the Data-Source solely to write changes back to the database using the Data-Source definition, then there is no need for a query at all.

Attaching Data-Sources

After you have defined a ProDataSet and its Data-Sources, you use the ATTACH-DATA-SOURCE method to associate them. This method, which is described in more detail later in this chapter, lets you specify which fields from the Data-Source go into the ProDataSet, and whether any are renamed in the process. You will learn how to use this method a little later. In the meantime, there are a few rules about how you specify the key fields for a Data-Source.

All the fields in the KEYS phrase must be represented in the ProDataSet buffer that the Data-Source is attached to. That is, they must not be excluded from the table in an EXCEPT list as part of the ATTACH-DATA-SOURCE method that associated the buffer and the Data-Source. The fields might be renamed in the ProDataSet buffer, however, and mapped in the ATTACH-DATA-SOURCE method. Instead of a field list, the KEYS phrase can specify (also within parentheses) the single keyword ROWID, in which case the table ROWID is used as the key for retrieval and updating. In this case, there must be a field in the ProDataSet buffer that is mapped to the table ROWID at the time of the ATTACH.

If you do not specify the KEYS phrase, then the AVM uses one or more primary keys of the database tables to determine the key fields. Therefore, if the primary key is in fact the appropriate key to use to do a unique FIND on a record, then you do not need to specify it in the definition. If the KEYS phrase is not specified and the database tables do not have a unique primary key, then the AVM has no way to locate a record for update or delete or to eliminate duplicate records. This means that if records in the ProDataSet are deleted or updated, the developer must provide an event procedure for that event that handles the operation—there can be no default support provided by the AVM.

Because the KEYS list is associated with a specific buffer, you must include the *source-buffer-phrase* in the definition if you need to specify the KEYS. If you also specify a QUERY name, then the buffer list itself is really redundant, except to identify which keys go with which buffer, since the query definition also specified them. In this case the AVM simply verifies that the list of buffers is the same.

Example

In the example for ProDataSet dsOrder, these are the Data-Source definitions, along with their queries:

```
DEFINE QUERY qOrder FOR Order, Customer, SalesRep.
DEFINE QUERY qItem  FOR ITEM.

DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder
  Order KEYS (OrderNum), Customer KEYS (CustNum), SalesRep KEYS (SalesRep).
DEFINE DATA-SOURCE srcOline FOR OrderLine.
DEFINE DATA-SOURCE srcItem FOR QUERY qItem ITEM KEYS (ItemNum).
```

You need a definition for query qOrder for two reasons. First, it is the top-level query, so unless you want all Orders and all their OrderLines and Items in the ProDataSet at the same time, which is unlikely (and which would be very expensive to load and pass between sessions), you must open that query with specific selection criteria for a single Order or a group of related orders. The second reason it is needed is to join the Customer and SalesRep tables to the Order to pick up the Customer Name and SalesRep Name fields. You cannot do this join without the query because the AVM will not be able to join the tables for you automatically.

There is also a query qItem for the Item table so that under some circumstances you can specify that you want all Items loaded into the ProDataSet.

The Data-Source srcOrder shows you how to define the keys for each table in query qOrder. In this case, each of these is a unique primary key for the table, so the buffer list and the KEYS phrases are not strictly necessary. However, they provide you with explicit confirmation within your procedure of which keys are used to identify records.

Likewise, the phrase ITEM KEYS (ItemNum) is not strictly necessary in the Data-Source srcItem.

Data-Source as a separate object

Defining the Data-Source as a separate object allows you to define the ProDataSet without having to combine the Data-Source definitions with the ProDataSet definition. When a ProDataSet is passed to another session, the Data-Sources are not passed as part of the object, since they have no meaning on another session and their database tables cannot normally be referenced there. Separating the ProDataSet from its Data-Sources also allows a ProDataSet to attach and detach Data-Sources during program execution, and to switch Data-Sources when this is necessary, for example to retrieve data from different databases. You cannot pass a Data-Source as a parameter, but you **can** access the handle of an attached Data-Source through its ProDataSet if the ProDataSet is passed locally. You can also simply include the same Data-Source definition in multiple procedures where this is appropriate.

However, once the Data-Source is instantiated as an actual object instance, it can be attached to only one ProDataSet at a time.

Whether or not to define a query for a Data-Source

There are several ways you can use a Data-Source. How you are using it determines whether it is appropriate or necessary to associate it with an actual query or simply with one or more buffer names. In the following cases it is appropriate not to have a query for the Data-Source:

- If the Data-Source is for a child table in a relation, then it might indicate that the default query generated by the AVM on a `FILL` is sufficient. The AVM retrieves all child records for the parent based on their foreign key relationship, so no query is needed.
- If the Data-Source is for a top-level table (one with no parent) and you want to retrieve all records from the table named in its definition, then no query is needed. For a top-level table, retrieving all records is the default action on a `FILL` operation. This might be appropriate for something like the `State` table, if it is made a part of a ProDataSet in its entirety for lookup purposes.
- If you are using the Data-Source for update purposes only, then no query is used or needed. The query is used only on a `FILL`. In this case the Data-Source is actually used as a "Target" for the updates.
- If you intend to attach a query to the Data-Source dynamically at run time, then you do not need to associate it with a query in its definition. As with other ABL objects, the ProDataSet enables you to combine static object definitions and dynamic actions on those objects very flexibly.

When you would not use a Data-Source at all

If you do not attach a Data-Source to a table before filling or updating the ProDataSet, then you must provide an event procedure to fill or update that table. The AVM provides no default fill or update behavior in this case. If the source for the ProDataSet table is in fact one or more database tables, then you will normally want to define a Data-Source and attach it to the ProDataSet buffer to get the default behavior the AVM provides. You can still extend that behavior in event procedures to further qualify which records go into the ProDataSet or to make other changes.

In some cases, however, your actual data source might not be a OpenEdge or Data Server database at all. It could be a flat file that you read data from, an XML or JSON document that you process yourself, or some other source of data. In this case you simply do not define a Data-Source at all. The AVM depends on the event procedure code you write for the fill events for the table, and your own code must do the work of populating that table. If there is no Data-Source for a temp-table and no supporting event procedure to replace it, then no data is loaded into that table, and no error results.

This gives you the flexibility to attach very different sources of data to the same ProDataSet without the ProDataSet definition or its business logic changing at all. At one time, the data might come from a file you read yourself. At another time, it might be moved into a database table. Or in some future OpenEdge release, the AVM might natively support the nonstandard data source you use, and you can define a Data-Source object to handle it. You can do all this without changing anything at all in the ProDataSet itself.

Attaching a Data-Source to a ProDataSet buffer

A major part of the reason for a distinct Data-Source object is that the ProDataSet itself needs to be defined without dependence on its Data-Sources. When the ProDataSet is passed to another session, for example, the Data-Sources do not and really cannot go along as part of its definition because they have no meaning on another session. Also, it might be necessary to associate different Data-Sources with a ProDataSet at different times. For example, a ProDataSet might want to switch from one Data-Source to another, depending on application logic, to fill from or update tables in different databases.

The AVM provides methods to attach and detach Data-Sources for the Data-Source buffers that use them. There is no static statement equivalent for these methods. For static objects, you can always obtain the handle through the object's `HANDLE` attribute.

The `ATTACH-DATA-SOURCE` method on a ProDataSet buffer handle associates a Data-Source with that ProDataSet buffer. This the syntax for the `ATTACH-DATA-SOURCE` method:

Syntax

```
[logical-var = ] buffer-handle:ATTACH-DATA-SOURCE
 (data-source-hdl [ , field-mapping [, except-fields [, include-fields ]]]).
```

Where:

- *logical-var* is an optional variable of type LOGICAL. The ATTACH-DATA-SOURCE method returns a logical value that is true if it succeeded and false if it failed. The method could fail if any of its other parameters is found to be invalid at run time, for example, if the *field-mapping* names fields in the ProDataSet temp-table or the Data-Source table that do not exist. It could also fail if the *buffer-handle* is not part of a ProDataSet.
- *buffer-handle* is the handle of a temp-table buffer in the ProDataSet.
- *data-source-hdl* is the handle of a Data-Source object.
- *field-mapping* is an optional character expression that evaluates to a comma-separated list of field pairs with different names in the source database buffer and the ProDataSet buffer. The list is in the form *source-field*, *dset-field1*[, *source-field2*, *dset-field2*][,…]. A field can be an array element as well as a simple scalar field reference. Be sure the list does not contain embedded spaces between names, as the AVM does not trim white space from around the elements in the list.
- *except-fields* is an optional character expression that evaluates to a comma-separated list of fields that are in the ProDataSet buffer, but that are excluded from being populated with data from the Data-Source. This is useful to be able to reuse a ProDataSet in situations where not all the fields are needed and it is expensive to load them all and ship the data around.
- *include-fields* is an optional character expression that evaluates to a comma-separated list of fields to copy into the buffer, as an alternative to the *except-fields* when it is easier to specify those to include rather than those to exclude. You can specify *except-fields* or *include-fields*, but not both. If you specify the *except-fields*, you can simply omit the optional *include-fields* argument, which otherwise must have the Unknown value (?). If you specify the *include-fields* argument, then the *except-fields* argument, which precedes it in the argument list, must have the Unknown value (?).

The fields in the Data-Source's buffers are mapped to the fields in the target buffer of the ProDataSet in the same way that fields in a source buffer are mapped to fields in a target buffer during the BUFFER-COPY method. That is, the AVM uses name matching to associate target and source fields. The *except-fields* are skipped if specified, or only the *include-fields* are copied. Otherwise, all fields with matching names are copied. The AVM uses the *field-mapping* to make specific assignments where the field name is different in the ProDataSet buffer. As with a BUFFER-COPY, any fields in the Data-Source whose names do not match any field in the target temp-table and which are not in the *field-mapping* list are simply skipped without error.

Since there is the possibility of a joined set of tables mapping to the target table, the *except-fields*, *include-fields*, and *field-mapping* arguments of the BUFFER-COPY method have been enhanced to take buffer name qualifiers such as Customer.CustNum.

In addition, it is legal to use a ROWID reference in the *field-mapping*, such as ROWID(SalesRep), ttSalesRowid. When used in this context, the ROWID function should have a source query buffer name as its argument. You do this when you want to use the ROWID of the database record as the key for the AVM to uniquely identify the record. In this case, you would also use the phrase KEYS(ROWID) in the Data-Source definition.

Using the ATTACHED-PAIRLIST attribute, you can get a comma-separated list of field name pairs for fields in a ProDataSet temp-table buffer that are mapped to corresponding fields in an attached Data-Source object. This list includes only the field name pairs you specified with the most recently attached Data-Source object.

This list is formatted as a comma-separated list of field name pairs using the following syntax:

Syntax

```
source-field1,dset-field1 [,source-fieldn,dset-fieldn ] ...
```

If the buffer is not part of a ProDataSet object, or if the buffer does not have an attached Data-Source object, or if you did not specify a field name pair list when you attached the Data-Source object, this attribute returns the `Unknown value (?)`.

Use the `DATA-SOURCE-COMPLETE-MAP` attribute to retrieve a list of field name pairs for all fields in a ProDataSet temp-table buffer that are mapped to corresponding fields in an attached Data-Source object. Here is a simple example of using the attribute:

```
DEFINE VARIABLE cFieldMapping      AS CHARACTER NO-UNDO.
DEFINE VARIABLE cDataBaseFieldName AS CHARACTER NO-UNDO.
DEFINE VARIABLE cLocalFieldName    AS CHARACTER NO-UNDO.
DEFINE VARIABLE httMember          AS HANDLE    NO-UNDO.
. . .
ASSIGN
  cLocalFieldName    = "ttMember.ID"
  httMember          = DATASET dset:GET-BUFFER-HANDLE("ttMember")
  cFieldMapping      = httMember:DATA-SOURCE-COMPLETE-MAP
  cDatabaseFieldName = ENTRY(LOOKUP(cLocalFieldName, cFieldMapping) + 1,
                           cFieldMapping).
. . .
```

Here are `ATTACH-DATA-SOURCE` methods you can use to associate database tables with each of the three temp-table buffers in the `dsOrder` ProDataSet:

```
BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE,
                                  "Customer.Name,CustName").
BUFFER ttOline:ATTACH-DATA-SOURCE(DATA-SOURCE srcOline:HANDLE).
BUFFER ttItem:ATTACH-DATA-SOURCE(DATA-SOURCE srcItem:HANDLE).
```

The first of these maps the `Name` field in the `Customer` table to the field `CustName` in the `ttOrder` temp-table.

This syntax detaches the Data-Source from the buffer it is currently attached to:

Syntax

```
[ logical-var = ] buffer-handle:DETACH-DATA-SOURCE().
```

Generally, it is good practice to detach Data-Sources as soon as you are done using them, unless you know that the same ProDataSet instance will again be used for another `FILL` or `UPDATE` operation.

Using BUFFER-COPY and BUFFER-COMPARE with a ProDataSet

The `ATTACH-DATA-SOURCE` method defines some of the same elements, such as a *`pairs-list`* for field mapping, as a `BUFFER-COPY` statement. The `FILL` method then uses these definitions to copy database fields into the ProDataSet temp-tables. Since this behavior has been extended to support ProDataSets more effectively, there are also equivalent extensions to the behavior of the `BUFFER-COPY` and also the `BUFFER-COMPARE` methods on a buffer handle.

In the optional *pairs-list* argument of a BUFFER-COPY or BUFFER-COMPARE method, you can now specify an array element as one or both of the fields to map. This allows you to instruct the AVM to copy a field or array element from one buffer to a field or array element in the other buffer, when the two fields do not have the same name. Previously, you could not specify an array element as one of the fields in the *pairs-list*. This enhancement is universally available to these two methods (though not to their static statement counterparts).

The second enhancement is specific to ProDataSet usage. It is often necessary to write BUFFER-COMPARE or BUFFER-COPY methods in custom code for FILL or update-related event logic. Because the ATTACH-DATA-SOURCE method already allows you to define a field mapping between the Data-Source buffer and the ProDataSet temp-table buffer, as well as to define a list of fields to include or exclude from the operation, it should not be necessary to specify those in a BUFFER-COPY or BUFFER-COMPARE method between the same two buffers. Therefore, the AVM checks whether a BUFFER-COPY or BUFFER-COMPARE method satisfies these two requirements:

- It is between a buffer on a Data-Source table and the corresponding temp-table buffer in a ProDataSet. (Note that this means that the operation can use a different buffer for the Data-Source database table but only the default buffer for the ProDataSet temp-table.)
- There is no *except-list* and no *pairs-list* in the method's arguments. (Note that the two methods do not support an *include-list* in any case—this optional argument is a ProDataSet enhancement for the ATTACH-DATA-SOURCE method only.)

If these two requirements are satisfied, then the method uses the *pairs-list* from the ATTACH-DATA-SOURCE method for the Data-Source, if any, along with either the *except-list* or the *include-list*, if any, to determine what fields to copy or compare. This works in both directions, so in the expression hFromBuf:BUFFER-COPY(hToBuf) or hFromBuf:BUFFER-COMPARE(hToBuf), either hFromBuf or hToBuf can be the Data-Source buffer, and the other the temp-table buffer. This saves you from having to repeat the *field-mapping* from the ATTACH-DATA-SOURCE method in a BUFFER-COPY on the same buffers.

Populating a ProDataSet

To populate the ProDataSet's tables in a uniform way, there is a FILL method that you can apply to either the ProDataSet as a whole or to a member buffer. Before you do a FILL, you must prepare a query for each top-level table you are filling. If you do not do this, then that table is filled with all records from its Data-Source. Even when you are using a static query for a table, you must use the dynamic QUERY-PREPARE method to define the selection criteria for it. The AVM cannot associate a static OPEN QUERY statement with a ProDataSet.

For example, to fill the dsOrder ProDataSet with Order number 1 and all its OrderLines and Items, you prepare this query on the query you have associated with that temp-table:

```
QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.OrderNum = 1, " +
  "FIRST Customer OF Order, FIRST SalesRep OF Order").
```

The AVM opens the query for you when you start the FILL. If the Data-Relation describes the parent-child relationships for children of the top-level table, then you do not need to define or prepare an explicit query for them.

Filling the entire ProDataSet

To fill the ProDataSet as a whole (that is, traverse through each of its member buffers) use the method on the ProDataSet handle, as in this example:

```
hDSOrder:FILL().
```

Or without using a HANDLE variable:

```
DATASET dsOrder:FILL().
```

Note: There is **no** static FILL DATASET statement, but the second form of the FILL method makes it easy to execute on a static ProDataSet without explicitly having to retrieve its handle separately.

A FILL operation does not support any explicit batching or "chunking" of records in order to limit the size of a ProDataSet and the expense of filling it and passing it remotely as a parameter. However, you can define a query at any level of the ProDataSet to limit the number of records filled in one operation.

When applied to the ProDataSet handle, the FILL method finds all the top-level buffers in the ProDataSet, which are those that are not children in any active Data-Relation. This means that a child of a relation that has been deactivated is treated like a top-level buffer for a FILL of a ProDataSet. In this way you assure that a FILL on a ProDataSet touches every buffer in the ProDataSet.

The FILL method then starts a nested filling operation starting at each top-level buffer, paying attention to the Data-Relation for which the top buffer is the parent, and proceeds down through parent-child relationships iteratively. If the buffer is a parent to any other buffer and the relation is active, the method gets each record in the parent, goes to each child of that parent and fills the child temp-table with those records related to the current parent, and cascades as it fills in further children down the hierarchy, before moving on to the next parent record.

If you want to fill a ProDataSet in a non-nested manner, that is, by loading all records at the top buffer level and then loading all records for that buffer's children using a single query, you can do this by deactivating the relations (individually by setting each relation's ACTIVE attribute to false or for the whole ProDataSet by setting its RELATIONS-ACTIVE attribute to false) and defining the appropriate query for the child that retrieves all records for all parents in a single pass. This might be more efficient under some circumstances. You will learn more about using these attributes in ProDataSet Attributes and Methods on page 105.

FILL returns true if successful and false if otherwise.

Partially filling a ProDataSet

You can also fill starting at the level of an individual ProDataSet member table by applying the FILL method to that table's buffer, as in this example:

```
httOrder:FILL().
```

Or, without using a buffer handle:

```
BUFFER ttOrder:FILL().
```

This fills the ProDataSet starting with the buffer's temp-table. If the temp-table is the child of a Data-Relation, then only rows matching the rows already in the parent will be filled. This means that there must be at least one parent row if the FILL is to add any rows to the ProDataSet. If there are active relations to the buffer's child tables, they are filled as well. When you execute the FILL method on a buffer handle rather than on the ProDataSet handle itself, you can deactivate Data-Relations to limit the depth to which the ProDataSet is initially filled.

Note that a FILL on a buffer traverses each branch of the hierarchy below that buffer until it encounters an inactive relation or a buffer marked as NO-FILL, which you will learn about shortly. In either of these cases, the FILL stops on that branch of the hierarchy.

Defining a query on a child table

As the FILL method has been described so far, the AVM fills child tables based on the Data-Relation between the child and its parent. Normally, you define a query for each top-level parent table. But you can also define a query for a child table of a Data-Relation. In this case, the AVM ignores the fields in the Data-Relation and executes the query on the child table. Depending on the query selection criteria, this can result in records in the child table that are not related to a parent. No error results in this case. Such records simply will not be found by the default traversal of parent and child that uses the Data-Relation and must be located in some other way with code in the procedure using the ProDataSet.

In order to facilitate defining a query for a child table that references its parent (but which presumably does additional selection beyond what the AVM would do by default), the query syntax for these queries within ProDataSets is extended to permit the query definition to reference a field in the parent table directly. The child query needs to be prepared only once, and the AVM substitutes the proper value for the current parent record each time the FILL proceeds from parent to child.

Recursively filling a ProDataSet

The RECURSIVE Data-Relation option instructs the ProDataSet FILL to load self-referencing elements. That is, an element can reference a child element that already either directly or indirectly references the parent element in the hierarchy. Examples of this hierarchy are bill-of-materials tables, or organization chart manager-employee relationship tables. A ProDataSet can have no more than one recursive Data-Relation.

Here is an example of a recursive Data-Relation definition for a one-to-many relation:

```
DEFINE TEMP-TABLE ttEmployee NO-UNDO
  FIELD cEmployee AS CHARACTER
  FIELD cManager  AS CHARACTER
  FIELD iAge      AS INTEGER
  INDEX idxEmployee AS UNIQUE cEmployee.

DEFINE DATASET myRecurs FOR ttEmployee
  DATA-RELATION r1 FOR ttEmployee, ttEmployee
    RELATION-FIELDS (cEmployee, cManager) RECURSIVE.
```

Support is provided for a recursive Data-Relation during a FILL of a ProDataSet or temp-table buffer. Navigation is not supported, but can be done normally through .NET, Java, or a recursive ABL procedure.

You can check the RECURSIVE attribute of the Data-Relation handle to determine if a relation is recursive. This is the syntax for the RECURSIVE attribute:

Syntax

```
logical-var = dataset-handle:GET-RELATION(1):RECURSIVE
```

If you are working with dynamic ProDataSets, an optional ADD-RELATION method parameter allows you to specify a recursive Data-Relation at run time. For more information about this method, see Creating Data-Relation objects on page 94.

Summary of a FILL

The AVM effectively goes through the following steps for the buffer on which you execute the FILL method, or for each top-level table in the ProDataSet if you are filling the whole ProDataSet:

1. Opens and performs a GET-FIRST on the top-level database query.
2. Creates a record in the top-level temp-table.
3. Performs a BUFFER-COPY from the database record buffers to the data table buffer, doing field mappings and excluding or including fields as specified in the table and Data-Source definitions and the ATTACH-DATA-SOURCE method.
4. If the data table is a parent in one or more active relations, prepares each child query in turn if this is the first use of it. The FILL then does steps 5-7 for each child.
5. Opens the child query; does a GET-FIRST on it; creates a record in its temp-table; and buffer-copies the database buffers into the temp-table buffer.
6. Repeats Steps 4 and 5 for any children of the child.
7. Does a GET-NEXT on the child for as long as there are more records in its query, and buffer-copies them into new records in its data table.
8. Goes back to the top level, does a GET-NEXT at that level, and continues the process in steps 2 through 7. For each child level, it just plugs key values from the parent into the child query, without having to actually re-prepare it, and reopens the child query for the current parent.

   For more information about events and the custom code that developers can write for them, see ProDataSets Events on page 79.

   This interleaving allows the AVM to maintain integrity throughout the entire FILL process, so that parent and child records are not read in separate loops, which would allow for the possibility that parent records would have changed or been deleted.

Controlling the filling of each table

You can issue a FILL on a ProDataSet or a buffer any number of times. For example, you might do this to load data from multiple Data-Sources into a single ProDataSet, which you could do by successively attaching different Data-Sources to the ProDataSet member buffer. Or you could modify the Data-Source query if you needed multiple successive sets of selection criteria to be used to populate all the data needed. Whether this could result in duplicate rows or other problems is determined by setting an attribute called FILL-MODE.

A ProDataSet buffer has a FILL-MODE attribute that can be set to one of several character values:

- **EMPTY** — If there is any data in the table it is emptied before the fill begins.
- **NO-FILL** — This means that the table should not be filled at all on a FILL method because it has already been filled on a previous operation, or it will be filled separately. This might typically be used when one or more data tables are filled once with static or relatively static data that does not change when other data changes; for example, a list of codes that remains constant even as the rest of the ProDataSet is being reused and refilled for different Orders and their related records. If a NO-FILL data table is encountered in the course of a fill that is initiated at a higher level, then that table is not touched, and any relations to child buffers are not traversed, so the FILL effectively stops on that branch of the relation hierarchy. If you issue a FILL directly on a buffer marked NO-FILL, no error results, and no data is loaded into that buffer's temp-table, beyond what might already be there.
- **APPEND** — A FILL on a buffer whose FILL-MODE is APPEND adds records in addition to anything that is already there, without doing any record comparisons. If this creates duplicate records that violate a unique index constraint on the temp-table, errors will result and the developer must cope with them. This mode is appropriate when you are implementing some form of batching, when a number of rows are added to a table on one FILL and then the following set on another FILL. However, keep in mind that MERGE mode is nearly as efficient as APPEND mode, so generally you should use the APPEND mode only when you want to be notified with a message about duplicates that occur rather than having them skipped without notification.
- **MERGE** — MERGE is the default FILL-MODE. This mode tells the AVM to check for records that are already in the temp-table, based on the table's unique index, and otherwise add new records to the table. This assures that there are no duplicates. Note that MERGE mode requires that there is a unique primary index using the KEYS fields on the buffer's temp-table. The AVM simply allows the standard database support code to attempt to add each new record to the temp-table. If this fails because of a duplicate key violation in a unique index for the temp-table, that error is suppressed. The cost to this is minimal. Also, in MERGE mode, if a record with the same key is found, it is **not** replaced. Thus, MERGE cannot be used to refresh records already in a ProDataSet, but can make it possible to fill a ProDataSet in such a way that the same record might be encountered twice without error or duplication. If you need to refresh records in a ProDataSet table, you can set the table's FILL-MODE to EMPTY, or you can delete all records or selected records you need to refresh before doing the FILL.
- **REPLACE** — If REPLACE is set, the AVM checks each row being filled to see if it is already in the ProDataSet temp-table, based on the temp-table's unique primary index. If a row with the same key values is already present, its field values are replaced by the field values from the database. If it is not already there, it is created in the temp-table. This FILL-MODE can be useful in situations where a row coming from the database may or may not be a duplicate of a row already in the ProDataSet. If the application cannot know which rows may have different values from the current values, it is not feasible to pre-delete the old row(s) reliably before doing a FILL that brings in new values. Keep in mind that this option is significantly more expensive than using APPEND or MERGE modes, and therefore you should use it only in situations where it is needed.

Testing the Order ProDataSet

You can now put these statements together to create a procedure that defines, fills, and displays data from the ProDataSet dsOrder. Procedure fillDSOrder.p follows this process:

1. Includes the temp-table definitions and the ProDataSet definition you have already seen
2. Defines the queries for the ttOrder table and the ttItem table (the second one is used later)
3. Defines Data-Sources for all three temp-tables
4. Prepares the top-level query for the ttOrder table to bring in Order number 1
5. Attaches all three Data-Sources to the ProDataSet buffers

6. Executes the `FILL` method on the ProDataSet handle
7. Detaches the Data-Sources from the ProDataSet buffers, as shown in the following example:

```
/* fillDSOrder.p -- Test procedure for an Order Dataset */
{dsOrderTT.i}
{dsOrder.i}

DEFINE QUERY qOrder FOR Order, Customer, SalesRep.
DEFINE QUERY qItem  FOR Item.

DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder Order KEYS (OrderNum),
  Customer KEYS (CustNum), SalesRep KEYS (SalesRep).
DEFINE DATA-SOURCE srcOline FOR OrderLine KEYS (OrderNum).
DEFINE DATA-SOURCE srcItem FOR Item KEYS (ItemNum).

QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.OrderNum = 1, " +
  "FIRST Customer OF Order, FIRST SalesRep OF Order").

BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE,
                                  "Customer.Name,CustName").
BUFFER ttOline:ATTACH-DATA-SOURCE(DATA-SOURCE srcOline:HANDLE).
BUFFER ttItem:ATTACH-DATA-SOURCE(DATA-SOURCE srcItem:HANDLE).

DATASET dsOrder:FILL().

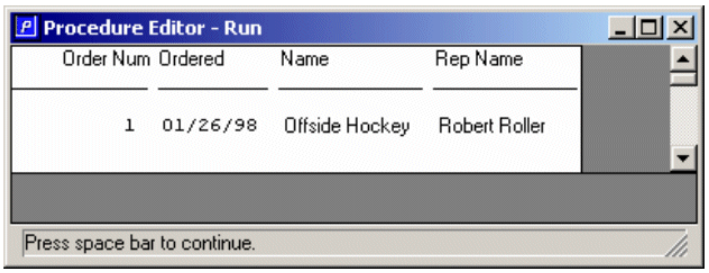
BUFFER ttOrder:DETACH-DATA-SOURCE().
BUFFER ttOline:DETACH-DATA-SOURCE().
BUFFER ttItem:DETACH-DATA-SOURCE().
```

At this point `Order` number `1`, its `OrderLines`, and their `Items`, are all in the ProDataSet temp-tables. You can verify this simply by displaying some of the fields in the temp-tables. For example:

```
FOR EACH ttOrder:
  DISPLAY
    ttOrder.OrderNum
    ttOrder.OrderDate
    ttOrder.CustName FORMAT "X(15)"
    ttOrder.RepName FORMAT "X(15)".
END.
FOR EACH ttOline:
  DISPLAY
    ttOline.OrderNum
    ttOline.LineNum.
END.
FOR EACH ttItem:
  DISPLAY ttItem.ItemNum ttItem.ItemName.
END.
```

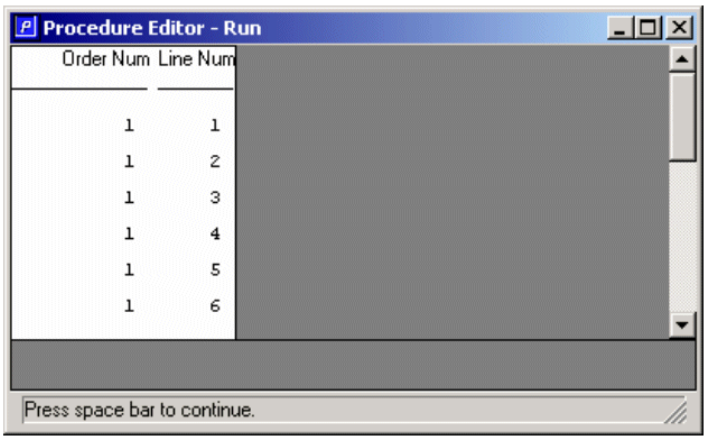
When you run the previous procedure, you will see the results of the `DISPLAY` statements—the `Order`, as shown in the following figure.

Figure 4: Customer orders



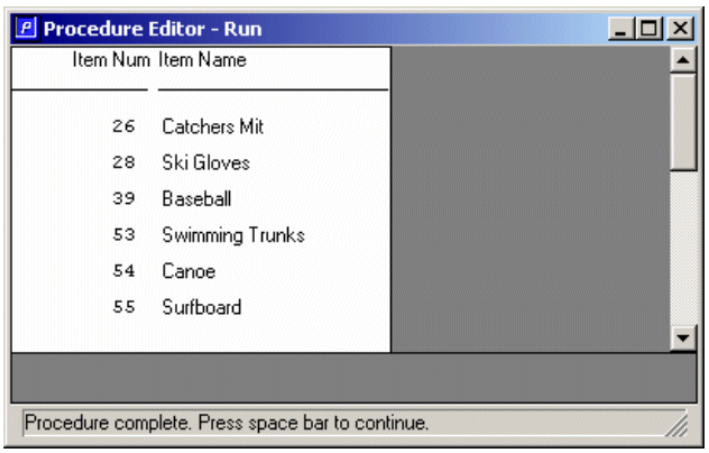
The `ttOline` temp-table contains all of the `OrderLines` for `Order` 1, as shown in the following figure.

Figure 5: Customer order lines



The `ttItem` temp-table contains all of the `Items` used on any of those `OrderLines`, as shown in the following figure.

Figure 6: Customer order line items

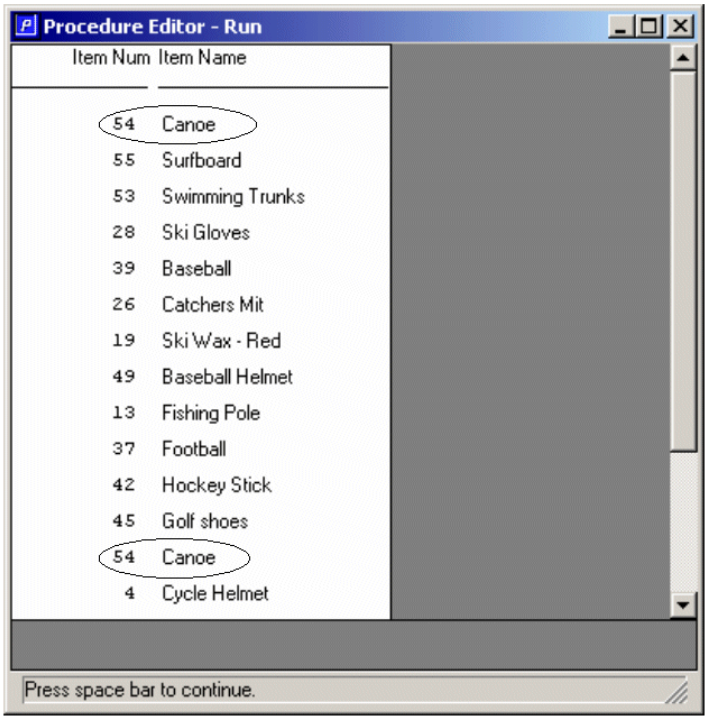


This looks fine, but in fact, you were fortunate you did not get an error when you ran the procedure. Try changing the selection for the `Order` query to fill all `Orders` less than 10, as shown in this code snippet:

```
QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.OrderNum < 10, " +
  "FIRST Customer OF Order, FIRST SalesRep OF Order").
```

This procedure change will also fill all their `OrderLines` and all the `Items` used on any of those `OrderLines`. Run the procedure and display the `Orders`, `OrderLines`, and finally the `Items`, as shown in the following figure.

Figure 7: Duplicate customer order line items



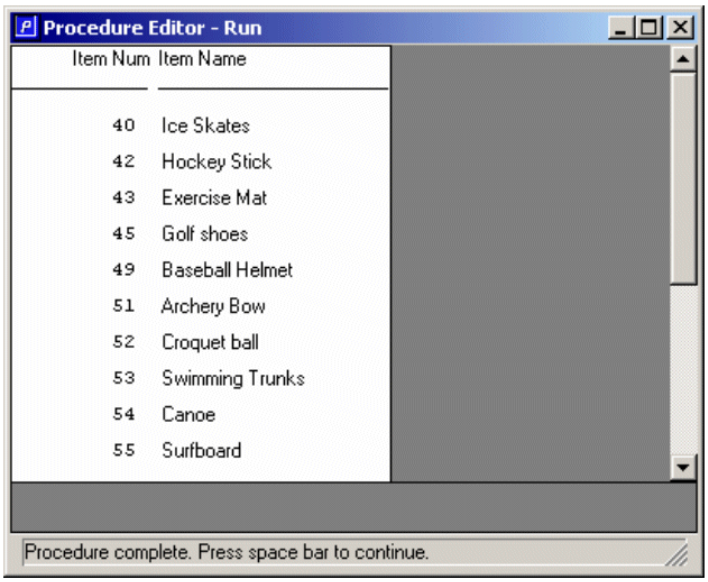
Some of the `Item` records, such as `Item` 54, are represented more than once. This is because they are used in more than one `OrderLine`. You probably do not want them in your `ttItem` table more than once. Since `MERGE` is the default `FILL-MODE`, why did the AVM not eliminate the duplicates?

The answer is that you did not have an index on the `ttItem` table. The AVM eliminates duplicates in a ProDataSet temp-table by relying on the internal indexing mechanism. Because this temp-table is not defined to be `LIKE` a database table, it does not inherit the `Item` table's indexes automatically. For the AVM to eliminate duplicate `Items` from this temp-table should they occur, you need a unique index on the `ItemNum` field. Add an index to the temp-table definition in the include file `dsOrderTT.i`, as shown:

```
DEFINE TEMP-TABLE ttItem NO-UNDO
  FIELD ItemNum  LIKE ITEM.ItemNum
  FIELD ItemName LIKE ITEM.ItemName
  FIELD Price    LIKE ITEM.Price
  FIELD Weight   LIKE ITEM.Weight
  FIELD OnHand   LIKE ITEM.OnHand
  FIELD OnOrder  LIKE ITEM.OnOrder
  INDEX ItemNum IS UNIQUE ItemNum.
```

Run the procedure again and the duplicate records are gone from the `ttItem` table. The `Items` now come out in order as well, because of the index. This is the end of the display, as shown in the following figure.

Figure 8: TEMP-TABLE index enforces uniqueness



Tip: If there is a chance that a `FILL` operation might create duplicate records in a temp-table that you want eliminated, you must either define or inherit from the database schema a unique index for the temp-table, as done here for the `ttItem` table.

Object life cycles with ProDataSets

As you have learned, a ProDataSet is made up of objects that exist in earlier OpenEdge releases, namely buffers, queries, and temp-tables, along with the new objects Data-Relation and Data-Source. This section reviews how these objects are related in terms of creating, scoping, and deleting.

As described earlier, a Data-Relation is an object that exists only within the context of a particular ProDataSet. It cannot be separately defined. It receives a handle when it is created as part of the instantiation of a ProDataSet with Data-Relation definitions, or when the relation is added dynamically to the ProDataSet. It has the same scope as the ProDataSet and is deleted when the ProDataSet is deleted.

You can define or create a Data-Source independent of a ProDataSet. This is important because Data-Source definitions cannot be passed with the ProDataSet to other sessions, where the database tables named in the Data-Source definition likely are not available. A Data-Source, whether static or dynamic, has a life independent of the ProDataSet. It is attached to and detached from a ProDataSet when it is needed. If it is dynamic, it must be independently deleted as an object if this is not done automatically when its procedure is deleted with a widget pool.

A Data-Source cannot be attached to more than one ProDataSet at a time. This is necessary to prevent conflicts between different ProDataSet buffers trying to use the same Data-Source. An attempt to attach a Data-Source to more than one ProDataSet buffer results in a run-time error.

A dynamic buffer must be created before it is used in a ProDataSet, and a static buffer must be defined before it is used in a ProDataSet. Dynamic buffers and temp-tables are deleted by default when the ProDataSet where they are used is deleted, and there is a logical `AUTO-DELETE` attribute for a dynamic buffer, which can be set to false to override this.

A temp-table is a separate object that is defined or created before it is used in a ProDataSet. When it becomes part of a ProDataSet, there are rules on the `FILL` operation, along with a `FILL-MODE` attribute, to control what happens to any data that might be in the temp-table when it is actively used as part of a ProDataSet. A dynamic temp-table is automatically deleted when the ProDataSet it is part of is deleted (unless you set the `NO-AUTO-DELETE` attribute to prevent this). A static temp-table simply goes back to being an ordinary temp-table. There are rules that are enforced in the implementation to prevent unacceptable behavior in ABL. The general intention is not to overly restrict the ability to use these objects flexibly and independently, just because of the possibility that a poorly written procedure might yield results the developer did not intend.

A temp-table can be part of multiple ProDataSets at the same time. If a ProDataSet is built up of tables in other, less complex ProDataSets, which is a very valuable feature, then it makes good sense to allow a temp-table to be simultaneously part of both ProDataSets. For example, a `ttItem` temp-table that is the only table in an `Item` ProDataSet, might be used as part of a more complex `PO` ProDataSet. It would be awkward and unnatural to force these two ProDataSets to use independent temp-table definitions when the whole purpose of building one from the other is to use the same underlying temp-table structure in both. It must be understood that multiple references to a temp-table within the scope of that temp-table are in fact using the same instance of the temp-table. There is only one set of records in the temp-table at any time, and if code manipulating the temp-table through its ProDataSets is at cross-purposes, then the result can be undesired behavior. But this is the developer's responsibility and entirely under the developer's control. There is no way to create multiple instances of a single temp-table definition within a procedure. Each ProDataSet must, however, use its own distinct buffer for the temp-table so that currency can be maintained independently in the ProDataSets.

If you require multiple instances of the same ProDataSet, or multiple instances of some of the components of a ProDataSet such as its temp-tables, you can accomplish this in several ways. If your definition is static, the best general design is to include the ProDataSet definition along with related definitions for objects such as its temp-tables, in a procedure that represents that ProDataSet as a business object. You can then create an API for that object procedure that has the necessary routines to fill the ProDataSet based on some set of parameters, to return it to the caller, to accept updates, to encapsulate business logic, and other requirements. If you need multiple instances of that ProDataSet as a complete business object, you run an instance of its procedure `PERSISTENT` for each one. Each procedure instance then holds the same definition and the same logic, but a distinct set of related data. Using ABL constructs such as super procedures, along with the callback mechanism for ProDataSet events, which is described in Event procedures for ProDataSets on page 79, you can share single business logic procedures among any number of instances of a ProDataSet procedure.

Summary

This chapter has introduced you to the ProDataSet and the components that it uses. The next chapter explores how to pass a ProDataSet as a parameter between procedures or between sessions.

2

ProDataSet Parameters

This chapter shows you how to pass a ProDataSet as a parameter between procedures within a session and between OpenEdge sessions.

For details, see the following topics:

- Passing a ProDataSet as a parameter
- Passing a ProDataSet with BY-REFERENCE or BIND
- ProDataSet parameter table
- Local parameter passing example
- Deleting a dynamic ProDataSet passed as a parameter
- Reducing the data to be passed in a parameter
- Passing a ProDataSet with APPEND
- Extending the sample procedure to pass a parameter

Passing a ProDataSet as a parameter

You can pass a static ProDataSet as a static object using the `PARAMETER DATASET` form, which is similar to the `PARAMETER TABLE` form for a temp-table. Also, you can pass either a static or dynamic ProDataSet through its handle using the `PARAMETER DATASET-HANDLE`, which is similar to the `PARAMETER TABLE-HANDLE` form for temp-tables.

The DATASET parameter form passes a ProDataSet as a static object to another procedure in the same session or another session. This is the syntax for a parameter definition of this form:

Syntax

```
DEFINE [ INPUT | OUTPUT | INPUT-OUTPUT] PARAMETER DATASET FOR dataset-name
  [ APPEND ] [ BIND ].
```

This is the syntax for a parameter passed in a RUN statement:

Syntax

```
RUN procedure ( [ INPUT | OUTPUT | INPUT-OUTPUT ] DATASET dataset-name
  [ APPEND ] [ BY-VALUE | BY-REFERENCE | BIND ] ).
```

The DATASET-HANDLE form passes the ProDataSet as a dynamic object through a handle variable. This is the syntax for a parameter definition of this form:

Syntax

```
DEFINE [ INPUT | OUTPUT | INPUT-OUTPUT ] PARAMETER DATASET-HANDLE handle-var
  [ APPEND ] [ BIND ].
```

This is the syntax for a parameter passed in a RUN statement:

Syntax

```
RUN procedure ( [ INPUT | OUTPUT | INPUT-OUTPUT ] DATASET-HANDLE handle-var
  [ APPEND ] [ BY-VALUE | BY-REFERENCE | BIND ] ).
```

You can pass a ProDataSet statically from one procedure using the DATASET parameter form, and receive it in another procedure in the same session or another session as a dynamic object using the DATASET-HANDLE parameter form, and vice versa.

For example, this technique allows you to take a statically defined ProDataSet on the server and pass it to a general-purpose client procedure that:

- Receives it dynamically through a DATASET-HANDLE
- Analyzes its structure through the handle
- Browses or otherwise uses the ProDataSet and its contents using dynamic client-side objects

A static definition is most useful on the server because it allows static business logic to operate directly on the ProDataSet and its temp-tables.

A ProDataSet can also be passed just as a handle using the HANDLE parameter type. As with other objects, a ProDataSet can be passed by a simple reference to its handle only within the same session, and can be accessed only dynamically, that is, through the handle, in the receiving procedure.

Passing a ProDataSet with BY-REFERENCE or BIND

Similarly to how temp-tables are passed as parameters, the AVM passes the ProDataSet by value, by default. That is, the AVM copies the ProDataSet definition and all the data in its temp-tables from one procedure to the other. This is true whether the call is to another procedure in the same OpenEdge session, or in a separate OpenEdge session on the other side of an AppServer connection. The overhead of doing this can be quite high and should be avoided wherever possible.

When you pass a ProDataSet to a remote session, the AVM has no choice but to copy the data from one session to the other. However, when you pass a ProDataSet as a parameter locally, you can optimize the call by including the BY-REFERENCE or BIND keyword on the parameter in the RUN statement, as shown in the following syntax examples:

Syntax

```
RUN procedure ( [ INPUT | OUTPUT | INPUT-OUTPUT ] DATASET dataset-name
  [ APPEND ] [ BY-VALUE | BY-REFERENCE | BIND ] ).
```

Syntax

```
RUN procedure ( [ INPUT | OUTPUT | INPUT-OUTPUT ] DATASET-HANDLE handle-var
  [ APPEND ] [ BY-VALUE | BY-REFERENCE | BIND ] ).
```

Passing a ProDataSet to a local routine using the BY-REFERENCE or BIND keyword allows the calling routine and the called routine to access the same ProDataSet instance, instead of copying the ProDataSet, and thereby provides a significant performance advantage.

When the call is remote, the AVM ignores the BY-REFERENCE and BIND keywords and passes the ProDataSet by value, as it must in this case.

For more information about using the BY-REFERENCE and BIND keywords, see the parameter definition syntax and parameter passing syntax reference descriptions in *ABL Reference*.

Passing a ProDataSet parameter by reference

You can only specify BY-REFERENCE on an internal procedure or user-defined function call, not on a RUN of an external procedure. When you pass a ProDataSet parameter by reference, the called routine's ProDataSet definition is bound to the calling routine's ProDataSet only for the duration of the call.

To pass a ProDataSet by reference, in the calling routine, use the BY-REFERENCE keyword in the RUN statement that defines the ProDataSet parameter. There is no special syntax required in the called routine. However, since the calling routine's ProDataSet instance is substituted for the called routine's ProDataSet, only the definition of the ProDataSet is required by the called routine. In other words, the same ProDataSet is defined in both routines but only one is actually used. If the called routine's ProDataSet is not directly used to hold its own data anywhere within the routine, you can save the overhead of allocating it by including the REFERENCE-ONLY option on its definition. This keyword tells the AVM to use the definition for compiler references to the table and its fields, but not to instantiate it at run time. Any reference to the ProDataSet, except where it is passed in from another routine BY-REFERENCE, results in a run-time error.

Because of the efficiency of passing the ProDataSet with BY-REFERENCE, you should normally use this keyword in your parameter definitions in RUN statements for any case where the called procedure will always or sometimes be in the same session as the caller. Because the AVM ignores the keyword on a remote call, you get the most efficient behavior in either case.

Why then is BY-REFERENCE not the default behavior?

Passing objects by copying them is the standard for other ABL parameter types. Passing a ProDataSet by reference has certain side effects that you need to be aware of, which could make the behavior confusing if you are not conscious of what the AVM is actually doing in the background to enable your called procedure to point to the same ProDataSet instance as the caller. For this reason, you have to make a specific request to pass by reference, and you should always make sure you have considered the consequences before doing so. Since this is different from how the AVM behaves otherwise, some of the effects can seem counter-intuitive.

In general you must consider that on a call BY-REFERENCE, the AVM substitutes the ProDataSet handle in the calling procedure for the ProDataSet defined in the called procedure, and that is the cause of most of the side effects. Consider the cases described in the following sections.

INPUT BY-REFERENCE can be like INPUT-OUTPUT

In a local call, passing a ProDataSet as an INPUT parameter BY-REFERENCE makes it behave essentially like an INPUT-OUTPUT parameter. The called procedure uses its local definition of the ProDataSet, if any, only to verify that the definition is compatible with the INPUT parameter. It then adjusts any references to the parameter to point to the ProDataSet instance in the caller. As a result, any changes you make to data in the ProDataSet in the called procedure are actually made to the ProDataSet in the calling procedure, and so are visible there after the procedure returns.

Tip: If the called procedure can make changes that should be visible in the caller, then you should make the parameter INPUT-OUTPUT. This reflects what is actually going on, and also provides the correct behavior when the call is remote, as the changes must explicitly be passed back to the caller in that case. If the called procedure will only read the data and not change it, then make the parameter INPUT.

Why not always make the parameter INPUT-OUTPUT in this case?

If you configure your application so that the call is made to a remote session and the called procedure does not make any changes to the ProDataSet, then the AVM will needlessly pass the unchanged ProDataSet back to the caller, creating unnecessary network traffic. This is why the INPUT mode is still useful.

OUTPUT BY-REFERENCE can be like OUTPUT APPEND

If the ProDataSet parameter is OUTPUT BY-REFERENCE and the call is local, then any data added to the ProDataSet in the called procedure is effectively appended to what was already there, again because both procedures are actually pointing at the same ProDataSet instance. If you make the same call remotely or you do not make it BY-REFERENCE, then the target ProDataSet in the calling procedure is emptied automatically by the AVM before the data from the OUTPUT parameter is copied into it. This can give you different behavior between local and remote RUN statements.

Tip: When you pass an OUTPUT parameter BY-REFERENCE, and do not want APPEND behavior, always explicitly empty the ProDataSet in the calling procedure just before the call.

This might seem like extra overhead, but in fact if you execute the statement hDataSet:EMPTY-DATASET(), this executes exactly the same code internally as the AVM uses to empty the ProDataSet for you, so the net cost is the same. If you do want APPEND behavior, then include the APPEND keyword on the parameter as you would for a temp-table.

ProDataSet instance passed BY-REFERENCE must exist in the caller

Consider this situation where a ProDataSet is passed as an OUTPUT parameter. The calling procedure defines only a handle for the ProDataSet and passes it using the DATASET-HANDLE parameter form, without creating a dynamic ProDataSet first. This is the syntax for passing a ProDataSet as an 0 parameter using BY-REFERENCE:

Syntax

```
RUN called-procedure ( OUTPUT DATASET-HANDLE dataset-handle BY-REFERENCE ).
```

The intention is that the *called-procedure's* ProDataSet definition and data are passed back as the OUTPUT parameter to populate the *dataset-handle* in the caller. This cannot work properly when the call is local. When the call is made, the *dataset-handle* has the Unknown value (?) because it has not been used yet. But the BY-REFERENCE keyword instructs the AVM to use the calling procedure's ProDataSet as the basis for the parameter. Since there is no such ProDataSet at the time of the call, this results in an error. A ProDataSet passed BY-REFERENCE, regardless of the parameter mode, must be initialized by specifying its tables and relations before the call.

In this case the calling procedure must do one of the following:

- It can create a ProDataSet using the handle and use dynamic methods to create a structure of temp-tables and relations for it that are compatible with the ProDataSet in the *called-procedure*. In this way the calling procedure's dynamic ProDataSet is used for the call, and the data from the *called-procedure* is effectively appended to it, just as for the static example described earlier.
- It can omit the BY-REFERENCE keyword and pass the ProDataSet by value in all cases, which means that the *called-procedure's* ProDataSet definition and data are copied back into the calling procedure to instantiate a dynamic ProDataSet there.
- It can assign the handle to some valid existing static or dynamic ProDataSet before the call.

If the OUTPUT parameter is a DATASET-HANDLE, it presumably means that the calling procedure is prepared to accept a variety of ProDataSet definitions returned to it. If this is the case, then once a particular ProDataSet has been passed back and has populated the dynamic ProDataSet, any further calls using the same RUN statement form must receive back a ProDataSet of the same type. Even though the AVM empties the target dynamic ProDataSet so that the data from the *called-procedure* replaces any data in the calling procedure's ProDataSet, the AVM does not automatically delete the dynamic ProDataSet structure in the calling procedure. If it is not compatible with the OUTPUT parameter, an error results.

If you want to use the same dynamic ProDataSet handle to receive different ProDataSets during the lifetime of the calling procedure, you must delete the dynamic ProDataSet using the DELETE OBJECT statement. Set the handle variable to the Unknown value (?) before you run a *called-procedure* to get back a different ProDataSet to avoid the error.

Sometimes the calling procedure needs to get back a dynamic ProDataSet (that is, one with a variable definition) **on the first call** and then wants to be able to make further calls to get back new or additional data in the same ProDataSet type, but without copying the ProDataSet definition locally on subsequent calls. If this is the case, then the first RUN statement must be made by value, in order to get back the ProDataSet definition with the data, and then subsequent calls can be made with a separate RUN statement BY-REFERENCE. This avoids copying the ProDataSet definition locally. (Note that in these descriptions the words "by value" are normally not capitalized in order to emphasize that this is the default behavior, so that the keyword BY-VALUE is not required to get this behavior.)

Naturally you could not pass an uninitialized dynamic ProDataSet as an INPUT or INPUT-OUTPUT parameter, as there would be no definition to pass to the *called-procedure*. In such a case you could only pass the handle variable and let the *called-procedure* associate it with a ProDataSet of its own.

Tip: To summarize: if the target ProDataSet in the calling procedure is dynamic, then you must either initialize it before making a call to another local procedure `BY-REFERENCE`, or make the first call by value and subsequent calls `BY-REFERENCE`, so that you obtain the ProDataSet definition the caller needs.

Main block references ignored in internal procedures

When the called procedure is a persistent procedure, its ProDataSet definition will naturally be in the main block of the procedure, that is, outside of any internal procedure. A ProDataSet definition is in fact not even allowed in an internal procedure. However, if its internal procedures are ever passed a ProDataSet parameter `BY-REFERENCE`, it is important that you not reference the ProDataSet handle in any way in the main block if you expect the effect of that reference to be visible in the internal procedures. This case is insidious enough to merit a specific example and diagram.

Here is a simple procedure that defines the `dsOrder` ProDataSet, runs another procedure persistent, and then runs an internal procedure to fill the ProDataSet:

```
/* refCaller.p */
{dsOrderTT.i}
{dsOrder.i}

DEFINE VARIABLE hProc AS HANDLE NO-UNDO.

RUN refCallee.p PERSISTENT SET hProc.
RUN fillProc IN hProc (OUTPUT DATASET dsOrder BY-REFERENCE).
```

Here is the procedure it runs. It defines its own instance of the ProDataSet and then uses its handle to attach three Data-Sources. Inside the internal procedure `fillProc`, it fills the ProDataSet and returns it as an `OUTPUT` parameter, as shown:

```
/* refCallee.p */
{dsOrderTT.i}
{dsOrder.i}

DEFINE VARIABLE hDset AS HANDLE NO-UNDO.

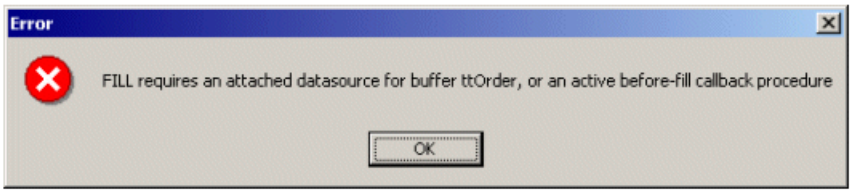
DEFINE DATA-SOURCE srcOrder FOR Customer.
DEFINE DATA-SOURCE srcOline FOR OrderLine.
DEFINE DATA-SOURCE srcItem  FOR Item.

hDset = DATASET dsorder:HANDLE.
hDset:GET-BUFFER-HANDLE(1):ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).
hDset:GET-BUFFER-HANDLE(2):ATTACH-DATA-SOURCE(DATA-SOURCE srcOline:HANDLE).
hDset:GET-BUFFER-HANDLE(3):ATTACH-DATA-SOURCE(DATA-SOURCE srcItem:HANDLE).

PROCEDURE fillProc:
  DEFINE OUTPUT PARAMETER DATASET FOR dsOrder.

  DATASET dsOrder:FILL().
END PROCEDURE.
```

If you run the main procedure `refCaller.p`, you get the following error:



What happened? All three Data-Sources were attached in the main block, so why can the AVM not see them?

The reason is that the instance of `dsOrder` defined in the main block, the one whose handle was used to attach the Data-Sources, is not the one used by the internal procedure. Because the ProDataSet is passed in by reference, `fillProc` is pointing to the instance of `dsOrder` defined in `refCaller.p`, which has no Data-Sources. A few messages confirm this.

Here we display the ProDataSet handle in the calling procedure:

```
RUN refCallee.p PERSISTENT SET hProc.
MESSAGE "In the calling proc, dsOrder is " DATASET dsOrder:HANDLE
  VIEW-AS ALERT-BOX.
RUN fillProc IN hProc (OUTPUT DATASET dsOrder BY-REFERENCE).
```

Also, in the main block of the persistent procedure `refCallee.p`:

```
MESSAGE "In the main block, dsOrder is " DATASET dsOrder:HANDLE
  VIEW-AS ALERT-BOX.
```

And, in the internal procedure `fillProc`:

```
PROCEDURE fillProc:
  DEFINE OUTPUT PARAMETER DATASET FOR dsOrder.

  MESSAGE "In the fillProc, dsOrder is " DATASET dsOrder:HANDLE
    VIEW-AS ALERT-BOX.
  DATASET dsOrder:FILL().
END PROCEDURE.
```

Run `refCaller.p` again and you can see the proof. When `refCallee.p` is first run, it gets a handle for its own ProDataSet. For example:



Next, the calling procedure displays the handle of its copy of the ProDataSet:



Now it runs `fillProc`:



You can see that `fillProc's` ProDataSet has the same handle as the one in the calling procedure `refCaller.p`. In fact, it is the same ProDataSet, the one with no Data-Sources.

If you change the persistent procedure to do all its work in the internal procedure, then everything works, as shown:

```
/* refCallee.p */
{dsOrderTT.i}
{dsOrder.i}

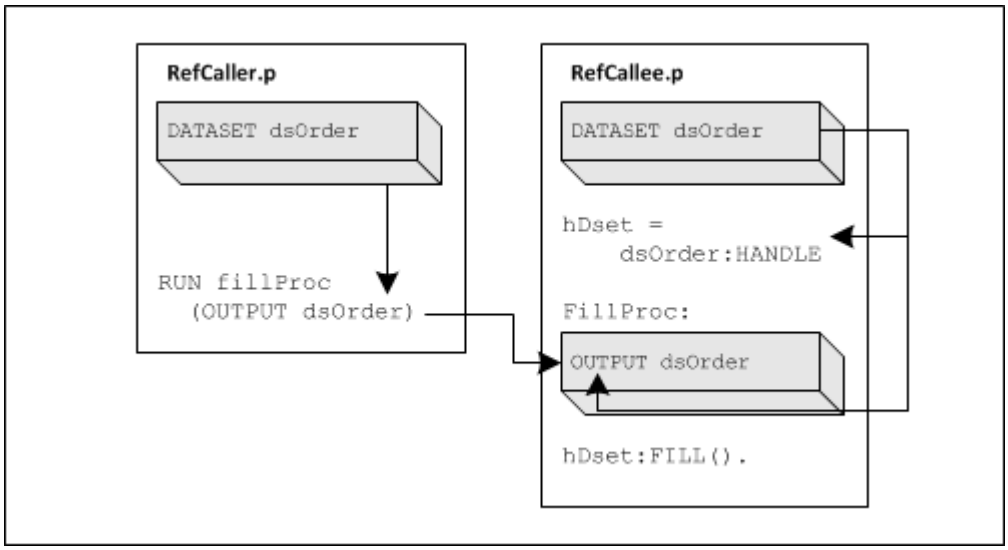
DEFINE DATA-SOURCE srcOrder FOR Customer.
DEFINE DATA-SOURCE srcOline FOR OrderLine.
DEFINE DATA-SOURCE srcItem  FOR Item.

PROCEDURE fillProc:
  DEFINE OUTPUT PARAMETER DATASET FOR dsOrder.
  DEFINE VARIABLE hDset AS HANDLE     NO-UNDO.

  hDset = DATASET dsorder:HANDLE.
  hDset:GET-BUFFER-HANDLE(1):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcOrder:HANDLE).
  hDset:GET-BUFFER-HANDLE(2):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcOline:HANDLE).
  hDset:GET-BUFFER-HANDLE(3):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcItem:HANDLE).
  DATASET dsOrder:FILL().
END PROCEDURE.
```

The following figure shows how the ProDataSet is filled in the called program and passed back as an `OUTPUT` parameter to the calling program.

Figure 9: Passing ProDataSets



Procedure refCallee.p has a definition of dsOrder, but the ProDataSet instance this represents is replaced by the one from refCaller.p when its ProDataSet is passed BY-REFERENCE. All internal references to hDset are therefore invalid because they point to a ProDataSet instance that is not being used. This teaches two important lessons, as described in the following design tips.

Tip: Do not set ProDataSet handles at the main procedure level when they will be accessed in internal procedures. Set the handle variables where they are used to capture a reference to an externally defined ProDataSet.

Tip: It is always good practice to perform operations such as attaching Data-Sources locally to where they are needed. It is essential if the ProDataSet is being passed by reference.

Note: These procedures use the standard include files for the temp-table and ProDataSet definitions. Adding the REFERENCE-ONLY option to these definitions would improve the performance of these procedures by avoiding the instantiation of the called routine's objects. It would also avoid the run-time errors by telling ABL at compile time that the called procedure's ProDataSet is not actually being used.

Specifying BY-VALUE in the called procedure

The parameter list shared by the calling procedure and the called procedure represents a contract between the two procedures that defines how they exchange data. As the cases we explored above illustrate, passing a ProDataSet BY-REFERENCE is a valuable optimization but one with side effects that change the nature of the contract between caller and callee. In some cases, the called procedure might want to force a ProDataSet parameter to be passed by value, regardless of any optimization used by the caller, to enforce the contract of its parameter list, and to avoid some of the side effects that can occur. For example, the called procedure might have some reason why it has to reference the ProDataSet handle in its main block and have that handle retain its validity inside internal procedures. Or, it might need to insist that an INPUT parameter should not result in the caller being able to see changes made to the ProDataSet in the called procedure. In any such case, the called procedure can include the BY-VALUE keyword in its parameter definition to force the ProDataSet to be passed by value, regardless of the caller.

This is the syntax for the DEFINE PARAMETER DATASET statement using BY-VALUE:

Syntax

```
DEFINE [ INPUT | OUTPUT | INPUT-OUTPUT ] PARAMETER
  DATASET FOR dataset-name BY-VALUE.

Or

DEFINE [INPUT | OUTPUT | INPUT-OUTPUT] PARAMETER
  DATASET-HANDLE dataset-handle BY-VALUE.
```

Importance of optimized code with BY-REFERENCE

The information in this section applies to the use of BIND as well, which you will read about in the .

This diversion into a discussion of BY-REFERENCE might seem overly complex, but it has been introduced here for a reason. If you get into the habit of structuring your procedures and their ProDataSet parameters properly from the beginning, you will be well positioned to optimize most calls that are sometimes made locally and sometimes remotely by adding the BY-REFERENCE keyword to your calls without any undesirable consequences. It is always best to design your procedures so that they work properly when run locally—even when in a deployed application they may be distributed on different machines. This makes the initial development and testing of your application more straightforward, and supports the case (even if it is only for demo purposes) where everything is running in a single session. Since the BY-REFERENCE optimization is such a valuable one, it is worth making the effort to prepare for it right from the start, even if you first code and test your application without it and then add the keyword to your calls to improve performance.

You can pass a ProDataSet reference locally as a HANDLE PARAMETER as you can with other objects such as temp-tables. This gives the called procedure access to the ProDataSet instance defined in the caller, but has two essential limitations. First, the called procedure must be in the same OpenEdge session as the calling procedure. You cannot pass an object handle of any kind across an AppServer call and have it maintain its validity. Second, the called procedure can only reference the ProDataSet using dynamic attributes and methods if it receives it as a HANDLE. In other words, the called procedure cannot receive the HANDLE parameter into a static DATASET definition and reference it using static table and field names.

By contrast, when you pass or receive a ProDataSet as a DATASET-HANDLE, this simply means that the procedure using the DATASET-HANDLE parameter form sees the ProDataSet as a dynamic object. The procedure on the other side of the call can see it as a static object. The table below shows the different combinations. Normally, passing a ProDataSet as a DATASET-HANDLE causes its definition and data to be copied to the receiving procedure. Using the BY-REFERENCE option on the parameter makes passing a ProDataSet as a DATASET-HANDLE cost no more within a single session than passing a reference to it as a HANDLE. For this reason, we recommend that you normally use the DATASET-HANDLE form. In this way, if the call is ever moved to a remote procedure, it will still work properly, whereas the HANDLE parameter will fail on a remote call.

Passing a ProDataSet BY-REFERENCE is particularly valuable when the called procedure uses the DATASET parameter form to receive the ProDataSet into a static definition. This will be the norm in most server-side procedures, and especially in most event handling procedures. The business logic in these procedures is much simpler to write if it uses static ABL statements to refer to and manipulate the records in the ProDataSet's temp-tables. All the elements of the ProDataSet definition, along with all the data in its temp-tables, are available to the called procedure. The default buffers that are part of the ProDataSet, which have the same names as their temp-tables, are in fact shared between caller and callee. Therefore, if the called procedure changes the record position in any of these buffers, this will be visible to the caller after the procedure returns. The called procedure can define buffers of its own to avoid this. Any changes to the ProDataSet data made by the called procedure are visible to the caller, as if the object were SHARED.

There are several restrictions in ProDataSet usage that are at least partially related to this parameter support:

- You cannot define a ProDataSet inside an internal procedure or trigger. This is the same restriction that applies to temp-tables.
- The static parameter form (that is, DEFINE INPUT/OUTPUT PARAMETER DATASET FOR*dataset-name*) cannot be used in the main block of a procedure that is run PERSISTENT.

As a final note, it is possible to specify a ProDataSet as a parameter for a dynamic call object, which lets you set up the entire definition of a RUN statement dynamically. You can also specify the BY-REFERENCE qualifier in the arguments to the dynamic CALL. For more information, see the reference documentation or online help for the SET-PARAMETER( ) method on the dynamic call object.

Passing a ProDataSet parameter by binding

You can also save the overhead of copying a ProDataSet's definition and data on a local call by using the BIND keyword. Passing a ProDataSet to a local routine using the BIND keyword allows the calling routine and the called routine to access the same ProDataSet instance by:

- Binding a reference-only static ProDataSet defined in one routine to a ProDataSet instance defined in another routine
- Binding an unknown DATASET-HANDLE parameter defined in one routine to a ProDataSet instance defined in another routine

In the static case, you must define a reference-only ProDataSet in either the calling routine or the called routine by specifying the REFERENCE-ONLY keyword in the DEFINE statement for the ProDataSet.

You use the BIND keyword in both the calling and called routines to tell the AVM to bind both ProDataSet references to the same ProDataSet instance. In the calling routine, you add the keyword to the parameter in the RUN statement. This is the syntax for passing a ProDataSet using the BIND option:

Syntax

```
RUN internal-procedure-name IN procedure-handle
     ( [ INPUT | INPUT-OUTPUT | OUTPUT ] DATASET dataset-name BIND )
```

In the called routine, you specify the keyword as part of the parameter definition. This is the syntax for defining a ProDataSet parameter using the BIND option:

Syntax

```
DEFINE [ INPUT | OUTPUT | INPUT-OUTPUT] PARAMETER
  DATASET FOR dataset-name BIND
```

When you pass a ProDataSet parameter to a local routine using the BIND keyword, you can define a reference-only ProDataSet object in either the calling routine or the called routine. In either case, the reference-only object definition remains bound to the object instance until the routine containing the reference-only object definition is deleted or terminates.

The most basic case where you would want to use the BIND keyword is when you want to use the called routine's ProDataSet instance instead of the calling routine's instance.

ProDataSet parameter table

The following table summarizes the possible combinations of parameter definitions for ProDataSets. Keep in mind that BY-VALUE is the default mode for local calls and the only mode for remote calls.

Caller RUN statement form	Callee parameter form	Parameter mode	Result
DATASET dsXYZ	DATASET dsXYZ	INPUT	Remote or BY-VALUE: Static ProDataSet dsXYZ in the caller is copied to the static definition of dsXYZ in the callee. The ProDataSet definition is passed along with the data for validation only, as it is not actually needed by the client.Local BY-REFERENCE: Callee points to the caller's ProDataSet using the callee's static definition, which must match the ProDataSet passed by the caller.Local BIND: Same as BY-REFERENCE, except the binding persists beyond the end of the called procedure.
DATASET dsXYZ	DATASET-HANDLE hDS	INPUT	Remote or BY-VALUE: Definition and data of static ProDataSet dsXYZ in the caller are copied to the callee, which constructs a dynamic definition using handle hDS and loads the dynamic ProDataSet (and dynamic temp-tables) with the data.Local BY-REFERENCE: Callee points to the same ProDataSet as the caller using its dynamic handle, which can accept any ProDataSet passed in.Local BIND: Same as BY-REFERENCE, except the binding persists beyond the end of the called procedure.
DATASET-HANDLE hDS	DATASET dsXYZ	INPUT	

Caller RUN statement form	Callee parameter form	Parameter mode	Result
			Remote or BY-VALUE: Definition and data of dynamic ProDataSet whose handle is hDS in the caller are copied to the callee, which receives the data into its static definition dsXYZ. The definition passed in must match the local static definition.Local BY-REFERENCE: Callee points to the same ProDataSet as the caller using its static definition, which must match the ProDataSet passed by the caller.Local BIND: Same as BY-REFERENCE, except the binding persists beyond the end of the called procedure.
DATASET-HANDLE hDS	DATASET-HANDLE hDS	INPUT	Remote or BY-VALUE: Definition and data of dynamic ProDataSet whose handle is hDS in the caller are copied to the callee, which receives the definition and uses it to construct a dynamic temp-table using its own local handle hDS, then loads the data into this dynamic table.Local BY-REFERENCE: Callee points to the same ProDataSet as the caller using its dynamic handle, which can accept any ProDataSet passed in.Local BIND: Same as BY-REFERENCE, except the binding persists beyond the end of the called procedure.
DATASET dsXYZ DATASET dsXYZ DATASET-HANDLE hDS DATASET-HANDLE hDS	DATASET dsXYZ DATASET-HANDLE hDS DATASET dsXYZ DATASET-HANDLE hDS	OUTPUT	

Caller RUN statement form	Callee parameter form	Parameter mode	Result
			Remote or BY-VALUE: All the same combinations are supported. Nothing is passed in to the callee. The definition of the ProDataSet and its data are passed back in the same form from callee to caller when callee returns. For the OUTPUT DATASET form, the definition is used to validate compatible ProDataSet definitions; for the OUTPUT DATASET-HANDLE form, it is used to construct the ProDataSet in the caller.Local BY-REFERENCE: Likewise, all the same combinations are supported. Callee uses the caller's ProDataSet until the called procedure ends.Local BIND: If the caller's ProDataSet is an unknown DATASET-HANDLE parameter, or a REFERENCE-ONLY static ProDataSet parameter, the callee's ProDataSet is used during the called procedure, and the caller binds to the callee's ProDataSet after the called procedure returns.
DATASET dsXYZ DATASET dsXYZ DATASET-HANDLE hDS DATASET-HANDLE hDS	DATASET dsXYZ DATASET-HANDLE hDS DATASET dsXYZ DATASET-HANDLE hDS	INPUT-OUTPUT	Remote or BY-VALUE: Once again, the same combinations are supported. The table definition and data are passed in from caller to callee. Callee can make changes to the data in the table, which is returned by being copied back to the caller.Local BY-REFERENCE: Callee points to the same ProDataSet as the caller using its dynamic handle, which can accept any ProDataSet passed in.Local BIND: Same as BY-REFERENCE, except the binding persists beyond the end of the called procedure.
HANDLE	HANDLE	Any	Local only. Cannot be used on a remote call. The handle points to the ProDataSet instance in the caller (for INPUT or INPUT-OUTPUT modes) or the called procedure (for OUTPUT mode). Only dynamic references to the ProDataSet are possible.

Local parameter passing example

Suppose that Procedure A has the ProDataSet definition for `dsOrder` used in the example procedure. After filling the ProDataSet, it passes it to Procedure B, as shown:

```
RUN B.p (INPUT DATASET dsOrder BY-REFERENCE).
```

Procedure B has the same static ProDataSet definition. Procedure B obtains a reference to `DATASET dsOrder` without it being copied. Procedure B can then reference the current records in `ttOrder` and `ttOline` directly. Buffer currency in Procedure B is the same as in Procedure A. For example, if Procedure B is responding to a record fill event for the `ttOrder` table, then the current record in `ttOrder` is the one just filled:

```
IF ttOrder.ShipDate NE ? THEN
  ttOrder.PromiseDate = ttOrder.ShipDate.
```

Procedure B can also navigate through the ProDataSet and its temp-tables:

```
FOR EACH ttOline OF ttOrder:
…
END.
```

On return to Procedure A, any changes made by Procedure B are visible to Procedure A because the ProDataSet is the same object.

If `B.p` is remote from `A.p`, then the ProDataSet is copied into Procedure B because it is an `INPUT` parameter. It is not copied back to Procedure A. If it were defined as an `INPUT-OUTPUT` parameter, then any changes would be passed back across the network to Procedure A and the end result would be the same as if it had been called locally. Therefore the `INPUT-OUTPUT` mode should be used when changes can be made that should be seen by the caller. The `INPUT` case is supported so that when the called procedure does not make any changes, the ProDataSet is not needlessly copied back to Procedure A. The `OUTPUT` case is supported so that if the original data in the ProDataSet is not used by Procedure B and only Procedure B's changes (creates or the result of a `FILL`) are to be used and passed back to Procedure A, the ProDataSet is not needlessly copied into Procedure B.

Deleting a dynamic ProDataSet passed as a parameter

Like any other object, a dynamic ProDataSet should be deleted when you are finished using it. Otherwise it continues to use memory and other resources, and since a ProDataSet can be quite large, this can be significant. When a ProDataSet is passed remotely, and therefore copied, the called procedure must be prepared to delete it if it is received as a dynamic ProDataSet. As we have discussed, however, the same procedure call could be used between two procedures in the same session, and you would not want to inadvertently delete a ProDataSet that is passed by reference in a local call and therefore actually "owned" by another procedure.

In order to simplify this, you can use the `DELETE OBJECT` statement in the called procedure when it uses a `DATASET-HANDLE`. This is the syntax for the `DELETE OBJECT` statement:

Syntax

```
DELETE OBJECT dataset-handle NO-ERROR.
```

In this way, the ProDataSet is deleted when appropriate. The NO-ERROR keyword suppresses any error message if the object has already been deleted (in which case *dataset-handle* is no longer a valid handle). Also, this statement does not delete a ProDataSet object when the ProDataSet is passed locally and by reference, which means that the ProDataSet is not owned by the procedure attempting the delete.

Reducing the data to be passed in a parameter

If you pass temp-tables across the network, the schema descriptor information can be quite substantial, which increases the overhead of the call. If the temp-tables are "wide," with many fields, and only one or a few rows are passed, then the schema information can far exceed the size of the data itself. In cases where the receiving procedure has a static temp-table definition of its own, this schema information is really not needed. Even in a case where the receiving procedure does not have a temp-table definition, schema details such as the field formats and Help strings and so forth might not be needed.

For this reason, there is a Temp-table Schema Marshal (-ttmarshal) startup parameter you can use to specify the amount of schema information marshaled for all temp-table parameters passed during an OpenEdge client session. There are also three temp-table attributes available to specify or reduce the amount of schema information marshaled for individual temp-tables. In any case, the temp-table can be an independent temp-table or a member of a ProDataSet.

Specifying the session startup parameter for all temp-table parameters

Use the Temp-table Schema Marshal (-ttmarshal) startup parameter to specify the amount of schema information marshaled for all temp-table parameters passed during an OpenEdge client session. This is the syntax for the -ttmarshalstartup parameter:

Syntax

```
-ttmarshal n
```

n

A value between 0 and 5 representing the amount of schema information to marshal for temp-table parameters. The default value is 0 (which includes all schema information for all temp-table parameters).

The following table lists the possible schema marshaling startup parameter values.

Value	Description
0	Includes all schema information for all temp-table parameters
1	Minimizes schema information for static temp-table parameters

Value	Description
2	Minimizes schema information for all temp-table parameters
3	Excludes schema information for static temp-table parameters
4	Excludes schema information for all temp-table parameters
5	Excludes schema information for static temp-table parameters and minimizes schema information for dynamic temp-table parameters

Setting attributes for individual temp-table parameters

There are three attributes you can use to specify or reduce the amount of schema information marshaled for individual temp-table parameters.

The first attribute lets you specify the amount of schema information to pass when the temp-table is marshaled. This is the syntax for the SCHEMA-MARSHAL attribute:

Syntax

```
table-handle:SCHEMA-MARSHAL = FULL | MIN | NONE
```

The following table lists the SCHEMA-MARSHAL attribute values.

Attribute value	Description
FULL	Includes all schema information for a temp-table parameter
MIN	Minimizes schema information for a temp-table parameter
NONE	Excludes schema information for a temp-table parameter

If set to the Unknown value (?), the default value is FULL.

The second attribute eliminates all schema description from the parameter, as shown in the following syntax:

Syntax

```
table-handle:NO-SCHEMA-MARSHAL = TRUE
```

If you set this logical attribute to true on a temp-table, then no schema information is passed when the temp-table is marshaled. This includes index descriptions and field information. The receiving side must have a static definition to receive the temp-table into.

Note: The NO-SCHEMA-MARSHAL attribute is supported only for backward compatibility. Use the SCHEMA-MARSHAL attribute instead. The NO-SCHEMA-MARSHAL attribute corresponds to the SCHEMA-MARSHAL attribute with a value of "NONE".

The third attribute reduces the schema information passed to the minimum needed to validate or establish the temp-table dynamically in the receiving procedure, as shown in the following syntax:

Syntax

```
table-handle:MIN-SCHEMA-MARSHAL = TRUE
```

If you set this logical attribute to true on a temp-table, then only minimum schema information is passed along with the data. This includes the field name and data type and extent.

Note: The MIN-SCHEMA-MARSHAL attribute is supported only for backward compatibility. Use the SCHEMA-MARSHAL attribute instead. The MIN-SCHEMA-MARSHAL attribute corresponds to the SCHEMA-MARSHAL attribute with a value of "MIN".

The temp-table ERROR-STRING (a new attribute used by ProDataSets, described in Setting and using ERROR, ERROR-STRING, and REJECTED on page 167) is also passed. Values for other field attributes are passed as the Unknown value (?), so that the protocol for the call remains the same but the amount of data is greatly reduced. No index information is passed. You can use this option even when the receiving procedure has a dynamic parameter (TABLE-HANDLE for a single temp-table or DATASET-HANDLE for a ProDataSet), because the field names, along with the data type and extent, can be used to construct a minimally complete dynamic temp-table on the receiving side without setting all the other field attributes.

If you want to eliminate or minimize temp-table schema information passed as part of a ProDataSet parameter, you must set the SCHEMA-MARSHAL attribute on each temp-table in the ProDataSet. It is possible you might set it differently for different tables, depending on whether you have static definitions of all the temp-tables on the other side of the call.

Note: If you specify both the SCHEMA-MARSHAL attribute and the NO-SCHEMA-MARSHAL or MIN-SCHEMA-MARSHAL attribute for an individual temp-table, the AVM uses the attribute you most recently specified. The NO-SCHEMA-MARSHAL and MIN-SCHEMA-MARSHAL attributes are supported only for backward compatibility. Use the SCHEMA-MARSHAL attribute instead.

Setting any of these temp-table attributes overrides the setting of the Temp-table Schema Marshal (-ttmarshal) startup parameter for an individual temp-table parameter. For more information about these temp-table attributes, see *ABL Reference*. For more information about the -ttmarshal startup parameter, see *Startup Command and Parameter Reference*.

Passing a ProDataSet with APPEND

You can include the APPEND keyword in a ProDataSet OUTPUT parameter just as you can for a temp-table. For example:

```
RUN getMoreData.p [ ON SERVER hXYZ ] ( OUTPUT DATASET MyDSet APPEND ).
```

In this example, suppose that ProDataSet MyDSet is defined within the calling procedure and has already been filled with some amount of data, normally in a previous call to a procedure on server XYZ. In that previous call, not all the data that could be used was filled. The procedure on the server might have deactivated some of the relations in the ProDataSet in order not to fill all levels of detail (on instructions from the client, perhaps).

Based on some event on the client, such as the user selecting a particular parent record for which detail has not yet been filled, the client procedure runs `getMoreData.p` on the server with the same ProDataSet as an `OUTPUT` parameter along with possibly one or more additional parameters that indicate how the ProDataSet should be further filled. The server procedure fills that part of the ProDataSet the caller wants, perhaps by retrieving the parent record the user wanted more detail for, and then doing a `FILL` on that buffer, which would fill all its children. The server procedure then returns the ProDataSet to the caller.

The `APPEND` keyword tells the AVM to append all the data passed back to the data that is already in the client's ProDataSet. Duplicate rows will result in an error when the data is appended, and thus must be eliminated in advance, before making the call. This happens regardless of the setting of the `FILL-MODE` attribute on the ProDataSet's buffers. The `FILL-MODE` thus applies only on `FILL`, not on parameter passing.

If you need to empty one or more of the receiving procedure's temp-tables during this kind of operation, you can simply empty it before the call, rather than using the `EMPTY FILL-MODE`.

If you want to refresh records with later versions from the called procedure, you must empty the table or delete the selected records to be refreshed before making the call.

Another typical use case beyond filling in levels of detail not yet retrieved is to define a form of batching of data by requesting more records for one or more tables not yet completely filled.

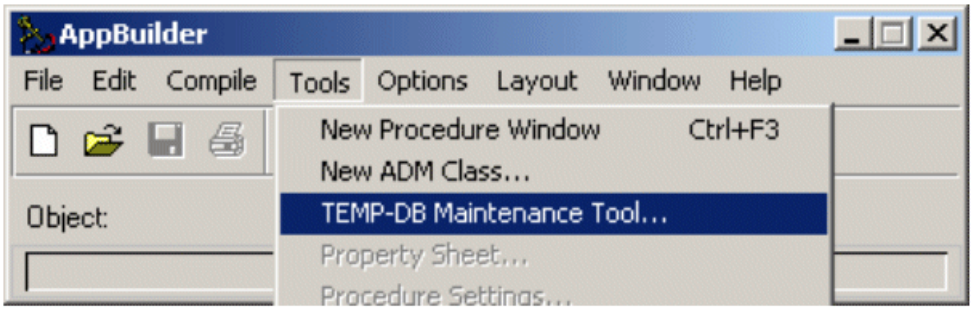
For an exploration of batching and selective filling use cases, see Batching Data with ProDataSets on page 205 and Advanced Events and Attributes on page 175.

Extending the sample procedure to pass a parameter

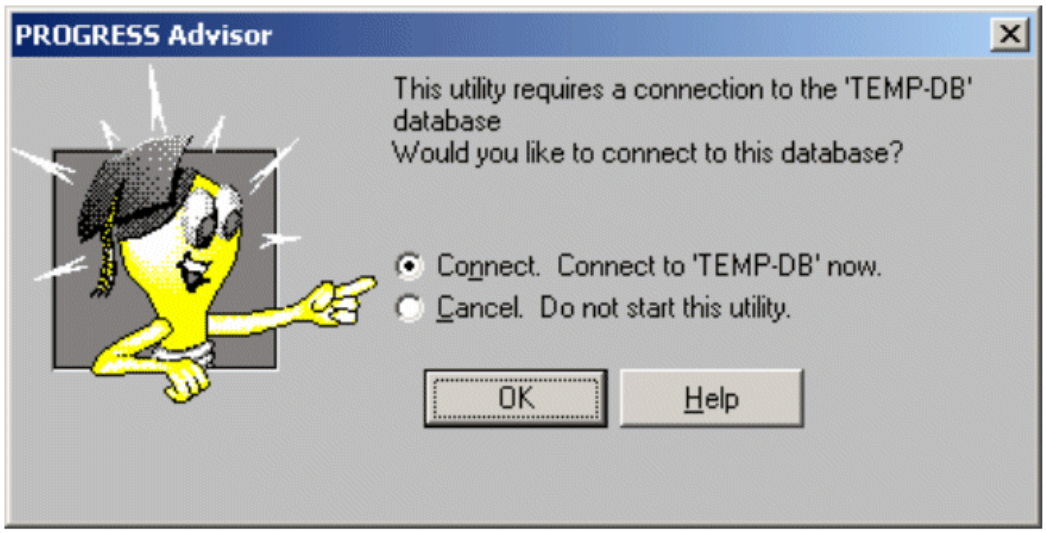
Now you can try passing a ProDataSet as a parameter, by building a new procedure in the AppBuilder to display the contents of your `Order` ProDataSet in a window, and calling the procedure that fills the ProDataSet from the new procedure. The AppBuilder uses a special database called `Temp-DB` to hold temp-table definitions. It uses these definitions to understand which fields are in which tables when you reference those fields in your code or use them to build windows with browses and fields in them. The AppBuilder contains a utility that provides support for creating and maintaining temp-table definitions using the Temp-DB to store the definitions. This section walks you through the process of using this utility to generate a temp-table include file that the AppBuilder can process. This procedure is a replacement for the original `dsOrderTT.i` include file that you built by hand in Introducing the OpenEdge DataSet on page 19.

To generate the temp-table include:

1. If you already have a database called `Temp-DB` where you have stored temp-table definitions, you can continue to use it and make its definitions available to the AppBuilder by connecting the database at design time. If you have not done this, you must create an empty database called `Temp-DB`, place it in your ProPath, and then follow the rest of the steps in this procedure.
2. In the AppBuilder, create a new **Window** procedure.
3. From the **Tools** menu, select the **TEMP-DB Maintenance Tool**:

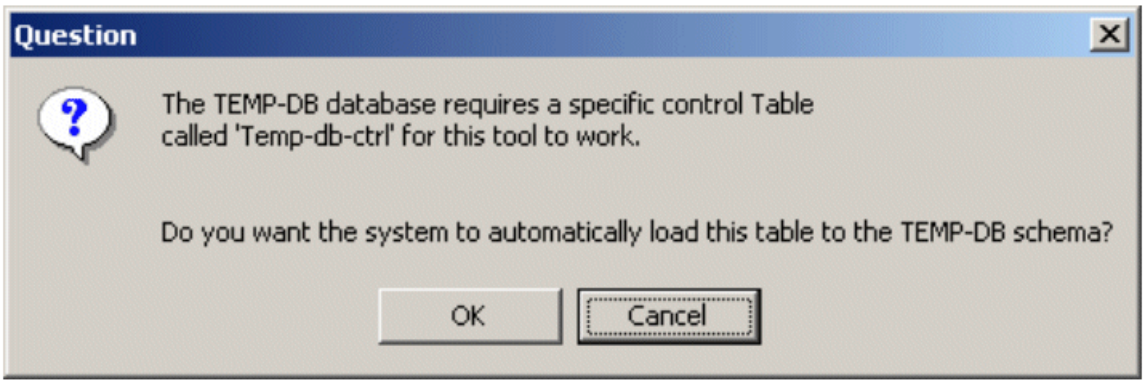


4. If the `Temp-DB` is not already connected, the AppBuilder prompts you to connect it:



   Select **Connect** and click **OK**.

5. The first time you use the **Temp-DB Maintenance Tool**, it must add a special control table to the `Temp-DB` where it holds meta-information about all the temp-table definitions it manages. If you get this prompt, press **OK** to load the definitions for this meta-table:

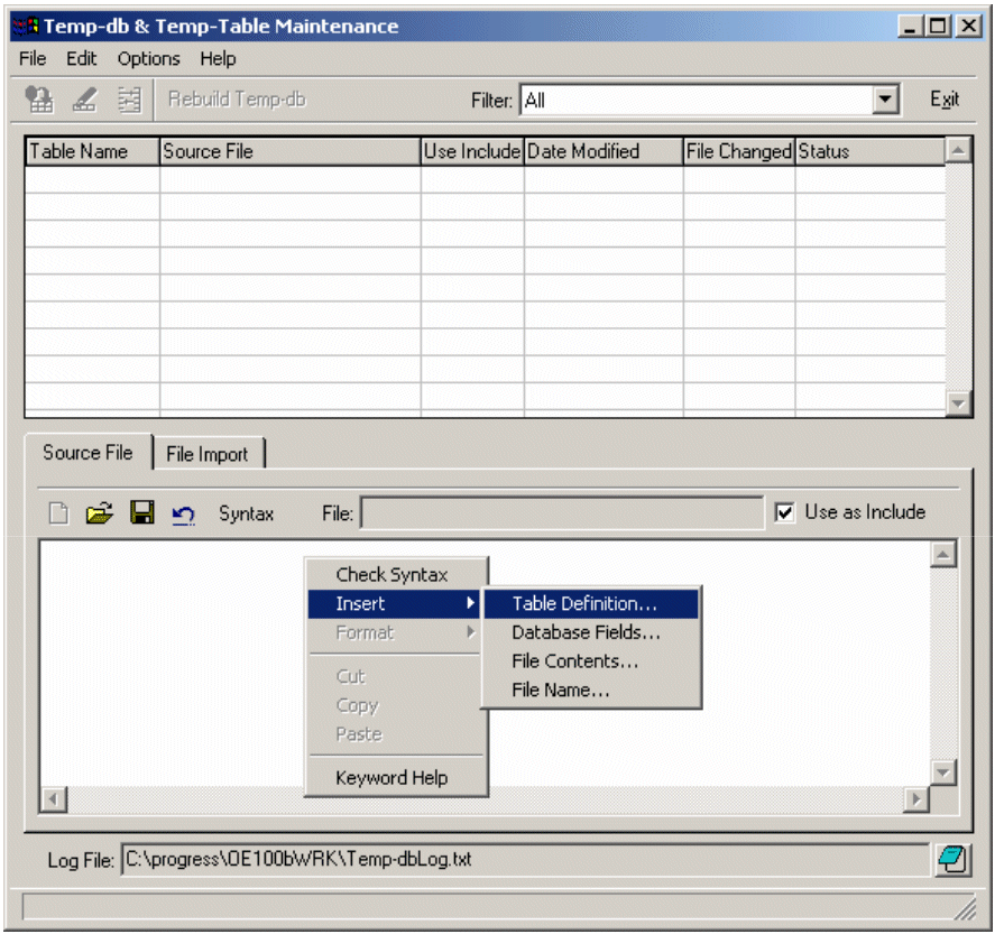


   Next you need to create temp-table definitions in the `Temp-DB` for the three temp-tables in your `Order` ProDataSet. You could import the include file you already have and start from that, but there is a good reason why you should not. Creating temp-table definitions that are defined to be `LIKE` the corresponding database tables is a simple shortcut, but it is not a good way to build definitions for a real application. It ties the temp-table definitions to any later changes that are made to the database tables, and it creates a requirement that the application database be connected at compile time, even though it should not be connected to the client side of the application at run time.

   The **Temp-DB Maintenance Tool** can easily generate a temp-table definition that is initially the same as a database table definition, but it does this with each field independently defined, so that you can edit the definition to be exactly what the internal data definition should be, and remove any ongoing dependencies on the external table definition. For this reason, you will replace `dsOrderTT.i` in these steps.

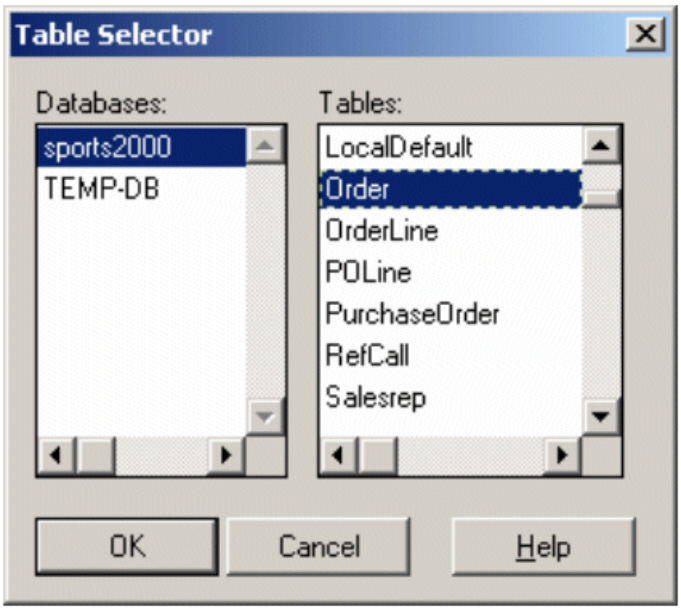
   The **Temp-DB Maintenance Tool** lets you create temp-table definitions based on database tables, or using your own field definitions, or any combination of the two. You do this in the editor part of the window. You can also import and manage any number of existing include files using the **File Import** tab, which displays all the temp-tables in the tool's browse control.

6. Right click in the editor to bring up its popup menu and select **Table Definition** > **Insert**, as shown:

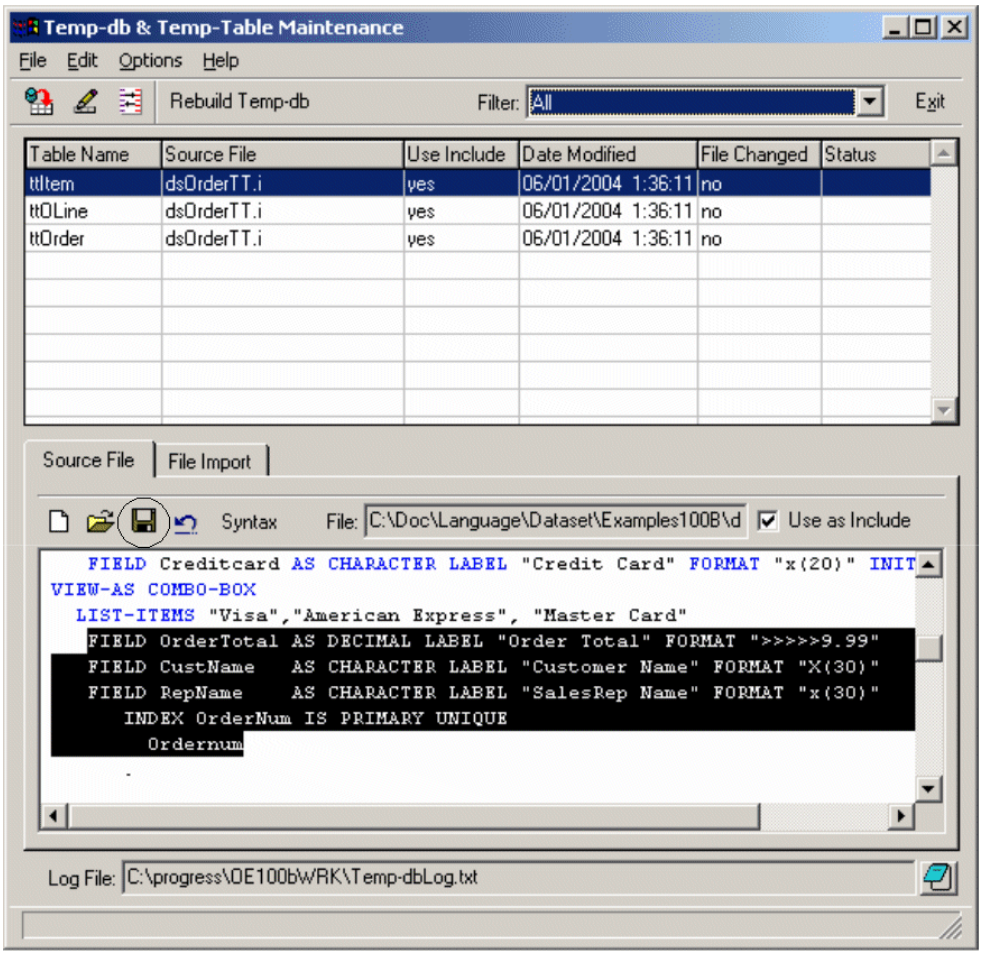


Note: You can also access this option through the **Edit** menu.

7. Select the **Order** table from the **sports2000** database as your first temp-table:



8. The tool builds a complete description of the `Order` table as a series of temp-table field and index definitions and displays it in the editor. Modify this to change the table name to `ttOrder`.
9. Add the three field definitions for the additional fields `OrderTotal`, `CustName`, and `RepName` as you did when you created the first version of `dsOrderTT.i`, as shown in the editor.
10. Remove all the index definitions except the unique primary index `OrderNum`.
11. Repeat these steps for the `OrderLine` table. Select **Table Definition** > **Insert** to select `OrderLine` and add its temp-table definition to the end of the editor contents. Rename the temp-table `ttOline`. Remove all but its primary index definition.
12. Repeat these steps again for the `Item` table. Name the temp-table `ttItem` and remove all but its unique primary index definition.
13. When you are done, click the **Save** button above the editor, as shown below. Select your `dsOrderTT.i` filename as the file to save the definitions to. The tool saves the include file, analyzes it, and displays the three temp-tables it contains in the browse, as shown:



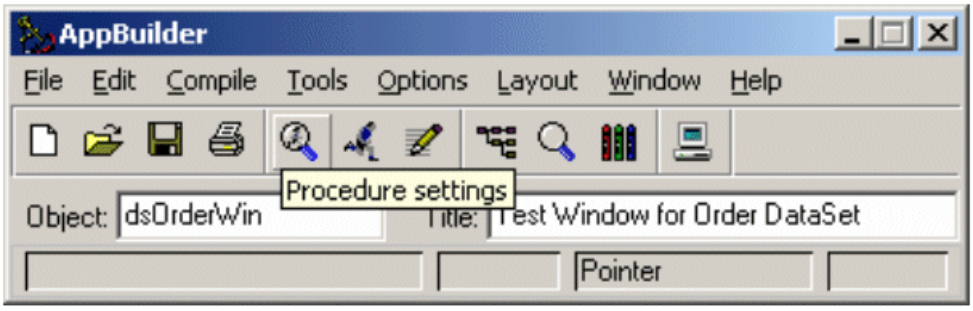
The **Use as Include** toggle box above the editor (which is also shown as the **Use Include** browse column for the temp-tables) indicates that you want the tool to save the definition as an include file to be included in your AppBuilder procedures, rather than having the AppBuilder generate the temp-table definitions inline. This is beneficial for keeping your temp-table maintenance independent of the procedures the temp-tables are used in.

14. In the **Definitions** section of your new **Window** procedure, include the `Order` ProDataSet definition include file:

```
/* ************************  Definitions  ************************/
/* Parameters Definitions ---                                      */
/* Local Variable Definitions ---                                  */
{dsOrder.i}
```

The AppBuilder will automatically generate a reference to the temp-table include file for you, when you bring those temp-tables into the procedure in a later step.

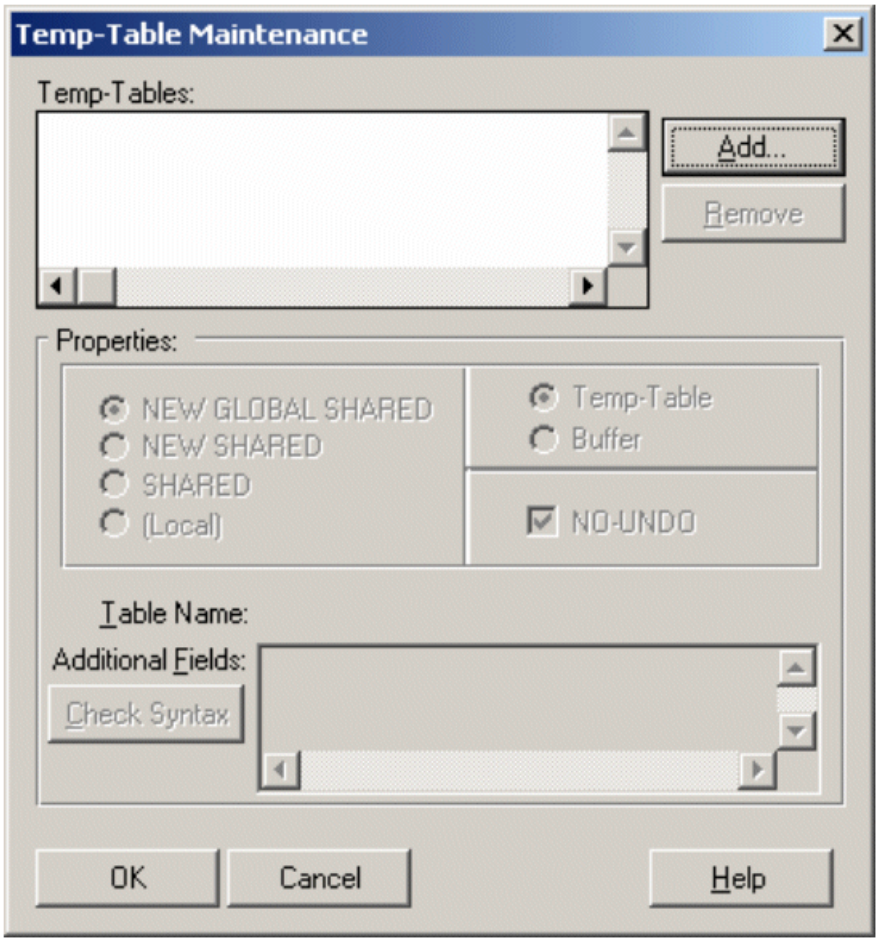
15. The temp-table definitions you just created are independent of any particular procedure that you build in the AppBuilder. To tell the AppBuilder that you want to use particular temp-table definitions in your new window, you use the **Temp-Table Maintenance** utility. To access this, click the **Procedure settings** button in the AppBuilder main window, as shown:



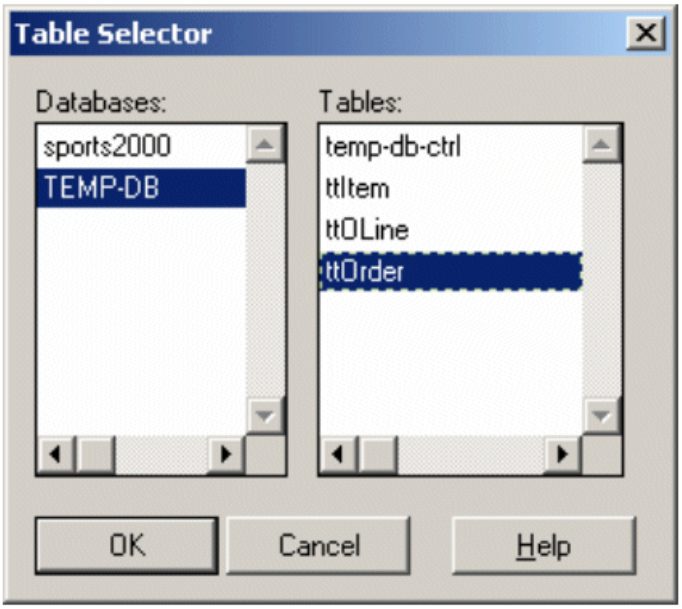
16. Click the **Temp-Table definitions** button (



17.) in the **Procedure Properties** dialog. The **Temp-Table Maintenance** dialog appears, as shown:

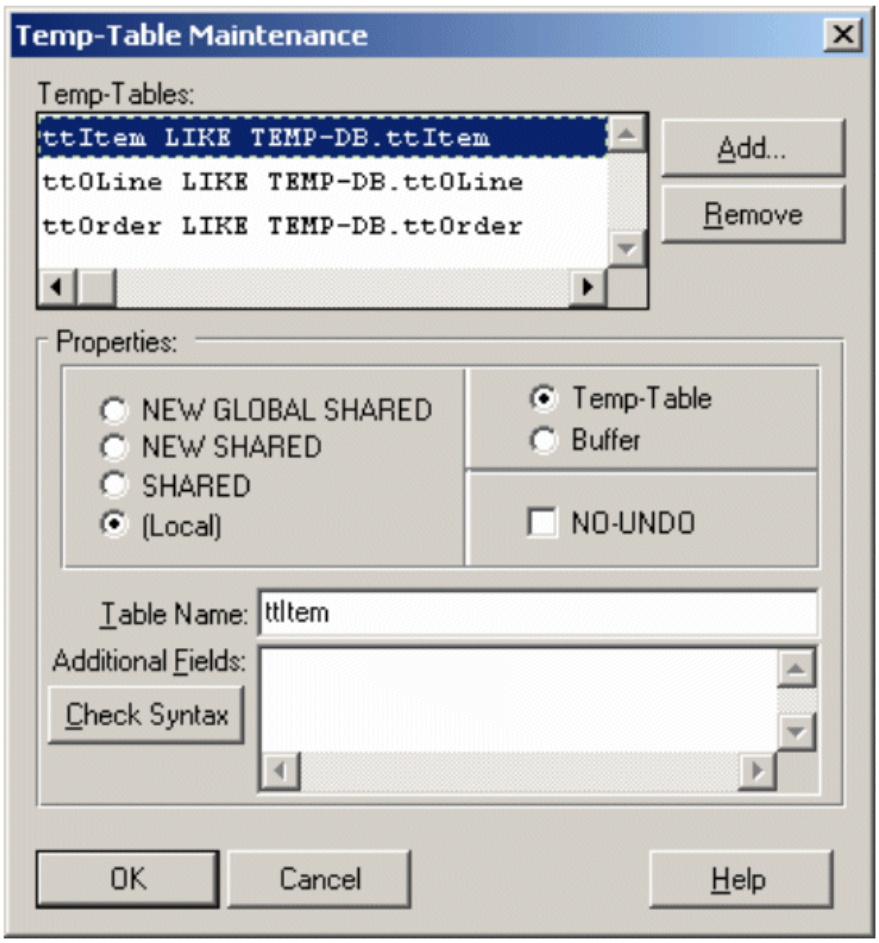


18. Add a temp-table `LIKE ttOrder` from the `Temp-DB` database. Because you have already defined the temp-tables exactly as they should be, you do not need to change anything in the definition, including the temp-table name, as shown:



19. Check off the **NO-UNDO** toggle for the table. Your tables will want to be able to take advantage of the AVM undo capability when you allow updates to the tables.

20. Do the same for the `ttOline` and `ttItem` tables:

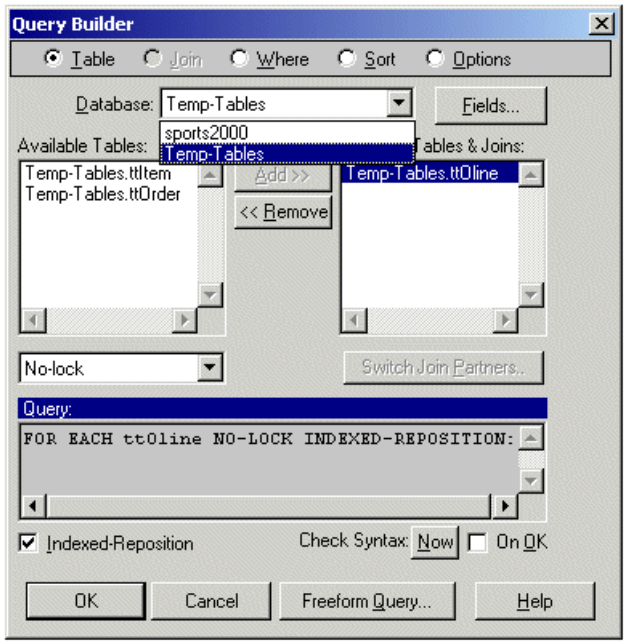


21. Click **OK** to exit the **Temp-Table Maintenance** dialog and then the **Procedure Settings** dialog.
22. Name the window `dsOrderWin`, and its default frame `dsFrame`.
23. Define several fill-in variables to display some of the `ttOrder` fields in: `integer` fill-ins `iOrderNum` and `iCustNum`, `character` fill-ins `cCustName` and `cRepName`, and a `decimal` field `dOrderTotal`.
24. Make each of these fields **Enabled**, and also set the **Read-Only** toggle box in the field property sheet for each of them except the `iOrderNum` field.

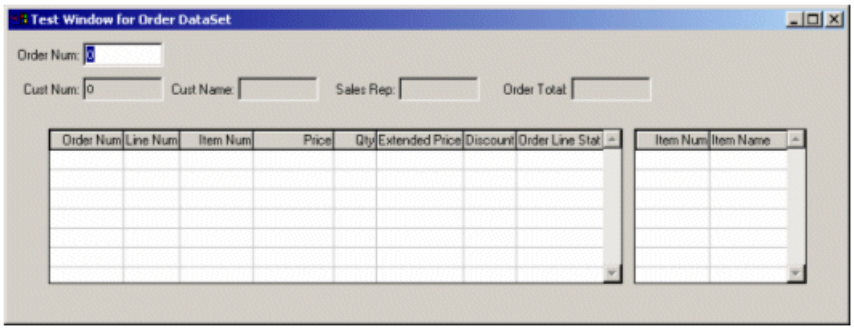
    You will use `iOrderNum` to enter an `Order Number` to pass to the ProDataSet fill procedure. The other fields simply display the `Order` values that come back. Making them enabled but read-only gives them a field border that makes them easier to read.

25. Drop a browse object from the AppBuilder palette onto the window and use it to define a browse on the temp-table `ttOline`.

    You do this by selecting **Temp-Tables** from the database list. The AppBuilder represents the temp-table definitions as if they were in a dummy database called `Temp-Tables`, as shown:



26. Click the **Fields…** button and add some or all of the `ttOline` fields to the browse.
27. Call the new browse `OlineBrowse`.
28. Create another browse for the `ttItem` table and add the `ItemNum` and `ItemName` fields to it. Name the browse `ItemBrowse`. When you are done, your window should look something like this:



Now you need to run a separate procedure that fills the `Order` ProDataSet with all the records for an `Order`, and then displays what that procedure returns.

29. Define this `LEAVE` trigger for the `OrderNum` field:

```
DO:
  ASSIGN iOrderNum.
  IF iOrderNum NE 0 THEN DO:
    DATASET dsOrder:EMPTY-DATASET.
    RUN fillDSOrder.p (iOrderNum, OUTPUT DATASET dsOrder BY-REFERENCE).
    FIND FIRST ttOrder.
    DO WITH FRAME dsFrame:
      ASSIGN
        iCustNum:SCREEN-VALUE  = STRING(ttOrder.CustNum)
        cCustName:SCREEN-VALUE = ttOrder.CustName
        cRepName:SCREEN-VALUE  = ttOrder.RepName
        dOrderTotal:SCREEN-VALUE = STRING(ttOrder.OrderTotal).
      {&OPEN-BROWSERS-IN-QUERY-{&FRAME-NAME}}
    END.
  END.
END.
```

If the user has entered an `Order` number, the trigger first empties the ProDataSet in case it still has data from a previous request. It then runs the procedure to fill the `Order` ProDataSet, passing in the `Order` number and getting the ProDataSet back as an `OUTPUT` parameter with the `BY-REFERENCE` qualifier. Passing it `BY-REFERENCE` eliminates the need to copy all the data from `fillDSOrder.p` back to the window procedure, but requires that you explicitly empty the ProDataSet first so as not to append data to the existing ProDataSet in the window procedure when the call is made locally.

The trigger then displays the `Order` fields and uses the AppBuilder-generated preprocessor to open the browses for `OrderLines` and `Items`.

30. Save this window procedure as `dsOrderWin.w`. Now you need to modify the fill procedure to accept the parameters.

31. Add these parameter definitions to `fillDSOrder.p` following the temp-table and ProDataSet definitions:

```
{dsOrderTT.i}
{dsOrder.i}

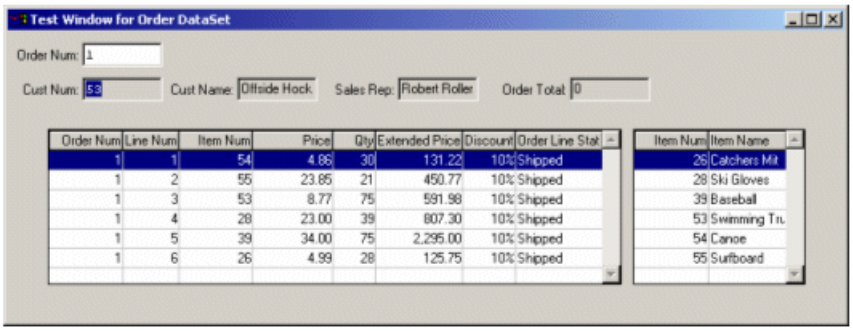
DEFINE INPUT  PARAMETER piOrderNum AS INTEGER NO-UNDO.
DEFINE OUTPUT PARAMETER DATASET    FOR dsOrder.
```

The parameters need to follow the ProDataSet definition because the `OUTPUT` parameter references the ProDataSet `dsOrder`.

32. Change the `FOR EACH` statement in the `Order` query to use the `Order` number passed in:

```
QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.OrderNum = " +
  STRING(piOrderNum) +
  ", FIRST Customer OF Order, FIRST SalesRep OF Order").
```

33. Remove the `FOR EACH:…DISPLAY` blocks from `fillDSOrder.p`. Now when you run the window you can enter an `Order` number and see all its data, as shown:



The window shows that the ProDataSet parameter has returned, through a single handle, the fields from the `ttOrder` record, the set of `OrderLines` in the `ttOline` table, and all their `Items` in the `ttItem` table. When the ProDataSet comes back as an `OUTPUT` parameter in a remote call, all the local copies of its temp-tables are automatically populated.

The `Order Total` is not filled in yet because that is a calculated field. In the next chapter, you will learn about event procedures for ProDataSets and how to write code to calculate that `Order Total` value for each `Order` as it is filled. Later, you will also learn how to synchronize the `OrderLine` browse and the `Item` browse so that the `Item` browse is repositioned to the currently selected `OrderLine`.

3

ProDataSets Events

This chapter describes the capabilities of the ProDataSet to respond to named events and execute custom internal ABL procedures.

For details, see the following topics:

- Event procedures for ProDataSets
- Defining FILL events
- Using event procedures in the sample procedure
- Summary

Event procedures for ProDataSets

In addition to carrying out default behavior for events such as filling or updating a ProDataSet, ABL defines named events to which you can attach internal procedures. Then you can define callbacks within your application to register a mapping of an event name with an internal procedure in a running procedure handle. For each defined event on each ProDataSet or ProDataSet temp-table where there is a callback, the AVM runs the internal procedure defined in the callback and passes in the ProDataSet as a parameter. This allows you to define business logic to validate or manipulate the data, or to extend or replace the default behavior of the ProDataSet.

The `SET-CALLBACK-PROCEDURE` method lets you associate a named event with an internal procedure to run when the event occurs, and a persistent procedure handle to run it in.

If there is a registered callback procedure for an event/ProDataSet combination, the AVM runs the internal procedure and passes the ProDataSet parameter to it. If there is no registered callback, then the AVM does nothing. This eliminates the possible overhead of searching up the procedure stack on every event to see if there is anything defined to run for it. Rather than searching through a stack of procedures that have all been attached to an object, the AVM can immediately go to the correct handle (if any) for each event.

Because the procedure handle for the event handler is part of the callback definition, the event handler can be in a logic procedure completely separate from where the ProDataSet is defined or used. In addition, many different procedures that all use the same ProDataSet can reference the same business logic that need only be running once within a session. Business logic for different parts of a ProDataSet can be handled by different procedures, so you can organize that logic to be as flexible as is needed. For example, it is entirely possible for you to use a single "pass-through" event procedure as the registered handle for all events, and then build a super procedure stack of procedures where the actual processing logic resides. Other organizational techniques are also possible; the mechanism provides efficiency and flexibility.

The SET-CALLBACK-PROCEDURE method uses this syntax:

Syntax

```
[ logical-var = ] object-handle:SET-CALLBACK-PROCEDURE
  ( event-name-expr, internal-proc-expr [ , proc-handle ] ).
```

Where:

- *object-handle* is the handle of a ProDataSet, a query, a ProDataSet temp-table buffer, depending on the event.
- *event-name-expr* is a character expression representing the name of the event as defined by the AVM for this object. The specific event names the AVM supports are described below.
- *internal-proc-expr* is a character expression representing the name of the internal procedure to run in response to this event.
- *proc-handle* is the procedure handle of a running persistent procedure where the *internal-proc-expr* is located. In order to provide a valid handle, the procedure must be running within the session before the callback is registered. The default is THIS-PROCEDURE.

The ProDataSet is always passed in as an INPUT parameter. The event procedure can receive this either into a static definition, using the parameter form DATASET PARAMETER FOR*dataset-name*, or into a handle using the parameter form DATASET-HANDLE*handle-var*. Within the callback event procedure, the SELF handle function evaluates to the ProDataSet handle or buffer handle associated with the event.

Note that you can only have a single active callback procedure for an event/object combination at any time. If you execute the SET-CALLBACK-PROCEDURE for an event name and object handle that already has a callback for that event, the latest one defined replaces the earlier one.

There are two methods you can use to retrieve callback procedure names:

- **GET-CALLBACK-PROC-CONTEXT( ) method** — Returns the handle of the procedure that contains the internal procedure associated with the AVM callback for the specified event. For example:

```
GET-CALLBACK-PROC-CONTEXT ( event-name )
```

If the object does not have a callback procedure for the specified event, this method returns the Unknown value (?).

- **GET-CALLBACK-PROC-NAME( ) method** — Returns the name of the internal procedure associated with the the AVM callback for the specified event. For example:

```
GET-CALLBACK-PROC-NAME ( event-name )
```

If the object does not have a callback procedure for the specified event, this method returns the `Unknown value (?)`.

There is an `APPLY-CALLBACK` method, described later, to force callback procedures to run at a time other than when the built-in events for them occur.

Defining FILL events

There are three levels of `FILL` events:

1. Events for the ProDataSet itself
2. Events for one of the ProDataSet's temp-table buffers
3. Events for each individual record created in a temp-table

Because the `FILL` can apply (either explicitly or by cascading) to the whole ProDataSet or to individual temp-table buffers, there are separate events for each of those levels.

The ProDataSet is passed into each event procedure as an `INPUT` parameter `BY-REFERENCE`. This allows the event procedure to operate on the ProDataSet using static ABL to reference its buffers and fields, without the ProDataSet being physically copied. This also means that because the ProDataSet is not copied, changes made to the ProDataSet by the event procedure are made to the same copy all procedures are using.

These are the `FILL` events for the ProDataSet and its members:

- **BEFORE-FILL on a ProDataSet handle** — Fires at the very beginning of a `FILL` on the ProDataSet before any records are read or populated. It lets you make a server or database connection or do other preparatory work. Alternatively, you could use the event to intercept and fully replace the default behavior, in a case where the data relationships are such that the standard Data-Relations cannot define them, or where there are perhaps no standard Data-Source objects at all, because the data comes from a non-database source.
- **BEFORE-FILL on a ProDataSet temp-table buffer handle** — Fires at the very beginning of a fill for a ProDataSet's temp-table. This lets you do preparatory work for the individual table. For the parent table in a set of related tables where the `FILL` event is applied to this top-level table, it could be the same kind of connection code as for the ProDataSet as a whole. Or you could prepare the query for a top-level table in its `FILL` event. For a child table, the event is fired once for each parent record that is created, and gives you the opportunity to adjust the query for the child table, or possibly cancel the fill for children of that parent altogether.
- **AFTER-FILL on a ProDataSet handle** — Fires at the very end of a `FILL` of a ProDataSet and can be used to adjust the contents of the ProDataSet, possibly to reject the entire `FILL` operation or to detach Data-Sources or disconnect from other resources.
- **AFTER-FILL on a ProDataSet temp-table buffer handle** — Fires at the end of a `FILL` of a ProDataSet buffer's temp-table and can be used to adjust the contents of the table, detach the Data-Source, and so on. As with the `BEFORE-FILL` event on a buffer, for a child table the event is fired once for each parent record that is created.

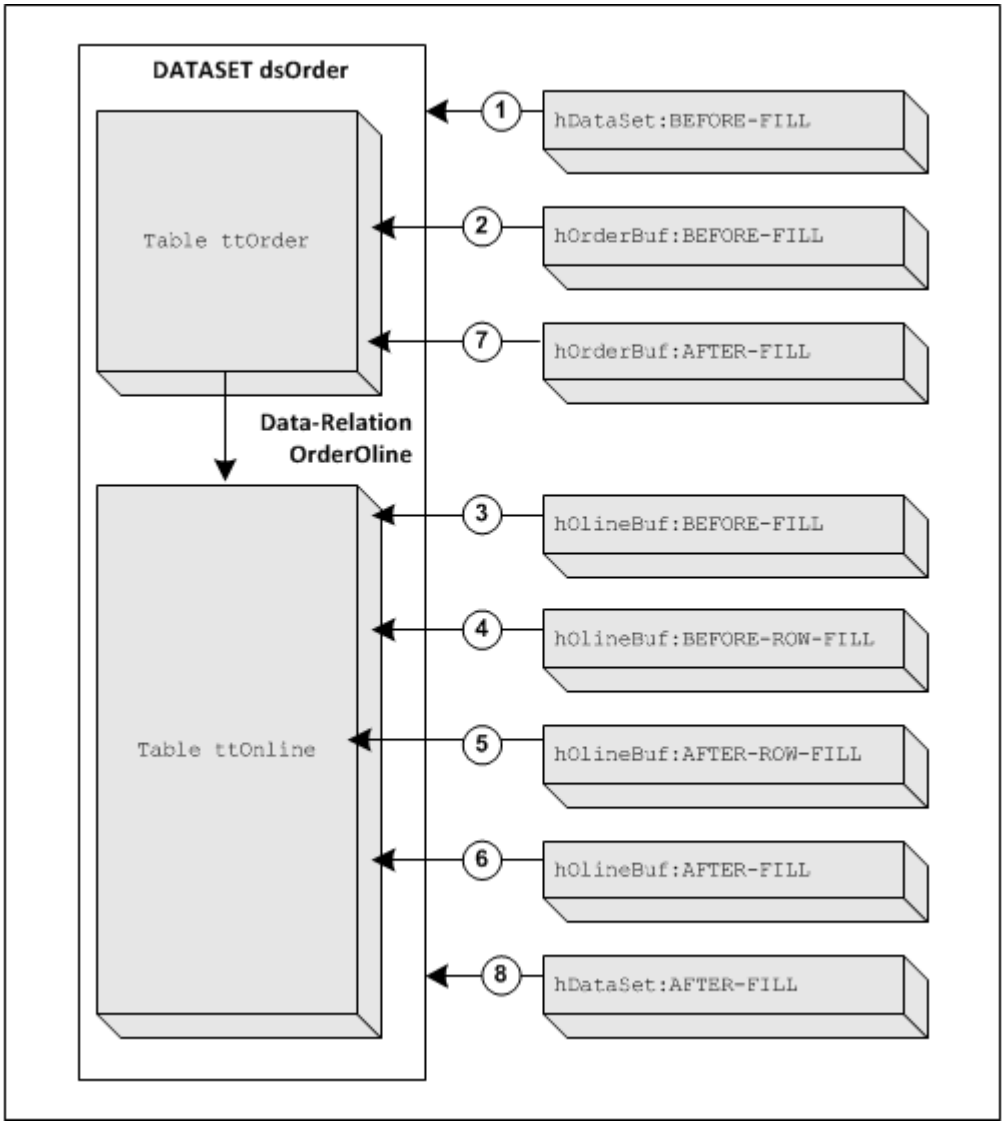
You use the `SET-CALLBACK-PROCEDURE` method to register each of these events for a ProDataSet handle or a ProDataSet buffer handle, which identifies the code to run for each object.

The lowest event level is for the individual record, fired once for each record created in each table during a FILL. These are the ROW-FILL events for the ProDataSet temp-table buffer:

- **BEFORE-ROW-FILL on a ProDataSet temp-table buffer handle** — Fires before the row is created in the temp-table, but after the Data-Source records for it have been read. For example, this code can examine the database buffers or other information and decide not to create the record using the RETURN NO-APPLY statement.
- **AFTER-ROW-FILL on a ProDataSet temp-table buffer handle** — Fires after a row is created in the temp-table. For example, the code can modify field values in the row by supplying values for calculated fields. It can also perform filtering and reject a row simply by deleting it, if the logic determines that it should not be part of the ProDataSet. The event procedure cannot modify record currency using the ProDataSet's buffers in any other way. It can use a separately defined buffer for the temp-table (or for other tables in the ProDataSet) to modify the ProDataSet in other ways. The code can RETURN ERROR to abort the entire FILL or RETURN NO-APPLY to cancel the cascade of the FILL operation down to children of this current record, if any.

The following figure illustrates the order in which these nested events are triggered, using a single parent-child relationship as an example. Multiple levels of nesting would result in multiple levels of nesting of the events for each table. If there are multiple top-level tables, the process is repeated for each top-level table. Events for top-level tables are triggered in the order in which the tables are defined in the ProDataSet definition or added to a dynamic ProDataSet.

Figure 10: Trigger order for nested events



This is the sequence shown in the above figure:

1. The ProDataSet `BEFORE-FILL` event fires first, before any record reads begin.
2. The table `BEFORE-FILL` event for the top-level table fires once before each row in that table is populated.
3. A nested table `BEFORE-FILL` event fires once for each parent row, before any rows in the child table are populated.
4. A `BEFORE-ROW-FILL` event fires once for each row in the table before it is populated, but after the corresponding records in the Data-Source have been read into their database buffers.
5. An `AFTER-ROW-FILL` event fires once for each row in the table after it has been created and its field values assigned.

6. The AFTER-FILL event on a nested table fires once for each parent row, after all the rows in the child table have been created and populated.
7. The AFTER-FILL event on a parent table fires once for each parent row, after it and all of its children have been populated.
8. The AFTER-FILL event on the ProDataSet itself fires last of all, after all rows have been populated.

If a callback procedure attempts to raise error for a FILL event, either through the traditional RETURN ERROR or through the structured UNDO, THROW, the ProDataSet ERROR attribute is set to true. Error is not raised to the caller.

As a recursive ProDataSet FILL is proceeding, it creates a clone of the relevant buffers, relations, queries and Data-Sources for each level of recursion. As a new record is added to the ProDataSet, it fires FILL events on the recursed, cloned buffer. From inside the event handler, you might want to see previous iterations of the buffer; its parent, grandparent, great-grandparent, and so on. These iterations are available through several attributes:

- CURRENT-ITERATION returns the level of iteration for the cloned buffer handle.
- GET-ITERATION returns the buffer handle at a specified recursion level.
- NUM-ITERATIONS indicates how many levels deep you are in a recursive FILL.

For more information about recursive Data-Relations, see Recursively filling a ProDataSet on page 43.

Using event procedures in the sample procedure

Let us move some of the supporting code to event procedures to test the callback facility.

To modify the code:

1. Create a new procedure called OrderMain.p that acts as the defining procedure for the ProDataSet, as shown below.

```
/* OrderMain.p -- Main procedure for an Order Dataset */
{dsOrderTT.i}
{dsOrder.i}

DEFINE INPUT  PARAMETER piOrderNum AS INTEGER NO-UNDO.
DEFINE OUTPUT PARAMETER DATASET    FOR dsOrder.

DEFINE VARIABLE hDSOrder AS HANDLE NO-UNDO.
DEFINE VARIABLE hEvents  AS HANDLE NO-UNDO.
DEFINE VARIABLE hDataSet AS HANDLE NO-UNDO.

hDSOrder = DATASET dsOrder:HANDLE.
RUN OrderEvents.p PERSISTENT SET hEvents (piOrderNum, hDSOrder).

hDSOrder:FILL().

DELETE PROCEDURE hEvents.
```

This new procedure simply defines the ProDataSet, accepts the `Order` number, and then runs a new event procedure where all the rest of the work is done. It passes the `Order` number and the ProDataSet handle in as `INPUT` parameters. `OrderEvents.p` binds the supporting events to the ProDataSet handle passed in as part of its main block, using the `SET-CALLBACK-PROCEDURE` method. `OrderMain` then does a `FILL` on the ProDataSet, which triggers the various events in `OrderEvents.p`. Finally, it deletes the persistent event procedure. In a real application, of course, it is likely that you would start event procedures like this one when you first need them and then leave them running to serve any caller.

2. Modify `dsOrderWin.w` to run this new procedure instead of `fillDSOrder.p` in the `LEAVE` trigger for `iOrderNum`.
3. Create the event handling procedure `OrderEvents.p`. Include the temp-table and ProDataSet definitions, and define the two `INPUT` parameters it needs, as shown:

```
/* OrderEvents.p -- FILL events for OrderDset.p */
{dsOrderTT.i}
{dsOrder.i}

DEFINE INPUT  PARAMETER piOrderNum AS INTEGER NO-UNDO.
DEFINE INPUT  PARAMETER phDataSet  AS HANDLE  NO-UNDO.
```

There is a rule that states you cannot define a static ProDataSet parameter at the top main block level of a procedure that you are going to run persistent, like this one. This is because the AVM needs an enclosing procedure block to pass a static ProDataSet reference into a persistent procedure by reference. For this reason, only internal procedures can have a static ProDataSet parameter. The static ProDataSet definition in `dsOrder.i` is used in these internal procedures later on, but the initial parameter at the top-level must be just a ProDataSet handle.

4. You need your top-level `Order` query definition, which you use to prepare a query for the `Order` number passed in. For example:

```
DEFINE QUERY qOrder FOR Order, Customer, SalesRep.
```

5. The two variables shown are needed to identify temp-table buffers based on the ProDataSet handle:

```
DEFINE VARIABLE iBuff AS INTEGER NO-UNDO.
DEFINE VARIABLE hBuff AS HANDLE  NO-UNDO.
```

6. Following the variables, Data-Source definitions from the first test procedure for `dsOrder` are found. For example:

```
DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder
  Order KEYS (OrderNum), Customer KEYS (CustNum),
  SalesRep KEYS (SalesRep).
DEFINE DATA-SOURCE srcOline FOR OrderLine.
DEFINE DATA-SOURCE srcItem FOR ITEM  KEYS (ItemNum).
```

7. The main block of the procedure establishes all the callbacks, so that when `OrderMain.p` does its `FILL`, they will be ready to respond to the events that happen as the `Order`, `OrderLine`, and `Item` records are read in and temp-table records are created for them. The first two callbacks are for the start and the end of the entire `FILL` at the level of the ProDataSet, so they are executed on the ProDataSet handle itself, as shown:

```
phDataSet:SET-CALLBACK-PROCEDURE
  ("BEFORE-FILL", "preDataSetFill", THIS-PROCEDURE).
phDataSet:SET-CALLBACK-PROCEDURE
  ("AFTER-FILL", "postDataSetFill", THIS-PROCEDURE).
```

The first of these procedures, which you will define in a moment, prepares the `Order` query. The second one detaches all the Data-Sources.

The remaining callbacks attach procedures to the temp-table buffers. Since the temp-table and ProDataSet definitions are included in the `OrderEvents.p`, it is natural to think that you can simply reference a buffer such as `ttOrder` in the callback definition. For example:

```
/* You think this will work but it will not… */
BUFFER ttOrder:SET-CALLBACK-PROCEDURE
  ("BEFORE-FILL", "preOrderFill", THIS-PROCEDURE).
```

Let us explore why this cannot work the way you might expect it to. The code in `preOrderFill` attaches all the Data-Sources to the buffer. The `preOrderFill` event procedure looks like this:

```
PROCEDURE preOrderFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE,
                                    "Customer.Name,CustName").
  BUFFER ttOline:ATTACH-DATA-SOURCE(DATA-SOURCE srcOline:HANDLE).
  BUFFER ttItem:ATTACH-DATA-SOURCE(DATA-SOURCE srcItem:HANDLE).
END PROCEDURE. /* preOrderFill */
```

The `SET-CALLBACK-METHOD` method along with its event procedure compiles just fine, because there is indeed a local `ttOrder` buffer the compiler can refer to. But before we go any further, you will get the following error message if you run the window with the callbacks defined in this way:



This message is telling you that the AVM was unable to fill the ProDataSet because when it got to the first table, `ttOrder`, there was no Data-Source for it. Also, there is no callback procedure to take the place of the Data-Source and fill the table. But the code defines a callback procedure, and the callback procedure attaches the Data-Sources. So what went wrong?

The answer is the same as in the example which showed the side effects of ProDataSets passed BY-REFERENCE (Local parameter passing example on page 65). The ttOrder buffer definition in the SET-CALLBACK-PROCEDURE method in the main block has no relationship to the ttOrder buffer for the ProDataSet handle phDataSet passed into OrderEvents.p. The ProDataSet definition in dsOrder.i and its temp-table definitions in dsOrderTT.i are strictly local at this point, and define what amounts to a separate instance of the same temp-tables and ProDataSet. Thus, when the code is attached to a callback to BUFFER ttOrder, it is attaching it to a handle for a temp-table the procedure is not really using and that the caller is not aware of.

8. To get the right buffer handle from the ProDataSet handle, you need to use one of the ProDataSet methods, GET-BUFFER-HANDLE, to access the buffer handle through the ProDataSet handle. This is the correct block of code that the main block of OrderEvents.p must use to attach the remaining callback events:

```
phDataSet:GET-BUFFER-HANDLE("ttOrder"):SET-CALLBACK-PROCEDURE
          ("BEFORE-FILL", "preOrderFill", THIS-PROCEDURE).
phDataSet:GET-BUFFER-HANDLE("ttOline"):SET-CALLBACK-PROCEDURE
          ("AFTER-FILL", "postOlineFill", THIS-PROCEDURE).
phDataSet:GET-BUFFER-HANDLE("ttItem"):SET-CALLBACK-PROCEDURE
          ("AFTER-ROW-FILL", "postItemRowFill", THIS-PROCEDURE).
```

You will learn about all the ProDataSet methods and attributes in following chapters. For now, let us look at all the remaining callback procedures.

9. The first one is for the BEFORE-FILL event of the ProDataSet itself. It prepares the Order query based on the OrderNum that was passed in to OrderEvents.p. For example:

```
PROCEDURE preDataSetFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.
  QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.OrderNum = " +
    STRING(piOrderNum) +
    ", FIRST Customer OF Order, FIRST SalesRep OF Order").
END PROCEDURE. /* preDataSetFill */
```

Remember that this procedure is not run when OrderEvents.p is run, but only later when the FILL event occurs. The piOrderNum parameter value is still available only because in this simple example the callback is only used by one caller, and its value is set when the persistent procedure is first run. In a real application you should construct your callbacks so that they can be shared by multiple instances of the objects that use them.

10. The second procedure is the AFTER-FILL event for the ProDataSet. It detaches all the Data-Sources, again using the NUM-BUFFERS attribute and the GET-BUFFER-HANDLE method to walk through the ProDataSet, as shown:

```
PROCEDURE postDataSetFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.
  DO iBuff = 1 TO DATASET dsOrder:NUM-BUFFERS:
    DATASET dsOrder:GET-BUFFER-HANDLE(iBuff):DETACH-DATA-SOURCE().
  END.
END PROCEDURE. /* postDataSetFill */
```

You have seen the first buffer-level callback procedure, preOrderFill, which is the BEFORE-FILL event for the ttOrder table. Take another look at the first lines of this procedure:

```
PROCEDURE preOrderFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.
  BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE,
                                    "Customer.Name,CustName").
```

If it was not correct to refer to BUFFER ttOrder in the SET-CALLBACK-PROCEDURE method in the main block, then why is it correct to do it here?

The answer to this is one of the key points you must keep in mind as you build applications with ProDataSets. The internal procedure preOrderFill receives a static reference to the ProDataSet dsOrder as an INPUT parameter. This is valid because you can pass a static ProDataSet reference to an internal procedure, whereas you cannot pass a static ProDataSet reference to the main block of a persistent procedure such as OrderEvents.p. Because the AVM passes the ProDataSet dsOrder into preOrderFill **by reference**, it simply points this internal procedure to the instance of dsOrder defined in the calling procedure. The local temp-table and ProDataSet definitions in dsOrder.i and dsOrderTT.i that the compiler uses to compile the reference to BUFFER ttOrder are automatically mapped, at the time the internal procedure is run, to a completely separate temp-table and ProDataSet definition. Therefore, **within this internal procedure**, the expression BUFFER ttOrder refers correctly to the buffer handle of the ttOrder temp-table, which is part of the ProDataSet dsOrder that is passed into the procedure. By contrast, in the main block the same reference is not correct because the only thing available to the main block is the handle of the caller's ProDataSet, not the ProDataSet itself. This is very important to understand as you start to work with ProDataSets.

Always keep in mind as you develop your applications whether you have a local ProDataSet, a reference to a ProDataSet defined in another procedure, or simply a handle to a ProDataSet.

The callback for the AFTER-FILL event on the ttOline buffer calculates the extra field OrderTotal in the ttOrder record, as shown:

```
PROCEDURE postOlineFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.
  DEFINE VARIABLE dTotal AS DECIMAL NO-UNDO.
  FOR EACH ttOline WHERE ttOline.OrderNum = ttOrder.OrderNum:
    dTotal = dTotal + ttOline.ExtendedPrice.
  END.
  ttOrder.OrderTotal = dTotal.
END PROCEDURE. /* postOnlineFill */
```

There are two more important points that this very simple calculation illustrates:

- Since the ttOline table has just been filled with all the OrderLines for the Order, the code can refer to the temp-table rather than the database records. This helps you write business logic that refers to the internal definition of your data, as distinct from the details of how it is stored in the database. If you change the nature of the mapping between the OrderLine data in the database and the fields in the ttOline table in the future, or even replace it with a completely different data source, the code that does the calculation does not need to change.

Tip: Always write your ProDataSet business logic to use the temp-table definitions in your ProDataSets wherever possible, because this is the definition that should remain constant and consistent regardless of how the mapping to the underlying database tables or other data source might change.

- The `ttOrder` record for the current `ttOlines` is immediately available to you because of the way in which the AVM executes the fill. For each `ttOrder` it populates, it immediately goes down a level in the relations and fills the children of that parent. This buffer currency is available to you even here where the local temp-table definitions for `ttOrder` and `ttOline` are actually pointing to a ProDataSet defined elsewhere.

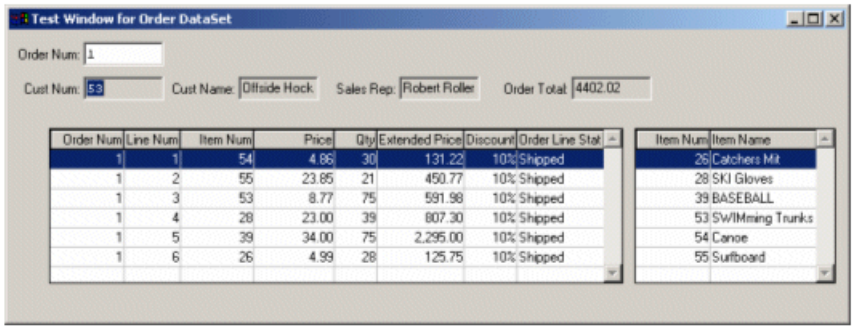
Tip: Always remember that all the contents of the ProDataSet are available to you in every event procedure. You can freely refer to parent records of the current table, and the buffer for the parent table for an event executed during a `FILL` will hold the parent record for the current children that triggered the event. Once the `FILL` is complete, a ProDataSet reference can give you access to any data in any of the ProDataSet's tables.

11. The final procedure is different from the others in that it is executed at the level of a single row fill, the `AFTER-ROW-FILL` event for the `ttItem` table. The procedure is executed once for every row in the `ttItem` table, just after the row is created and the fields from the `Item` Data-Source copied in. For example:

```
PROCEDURE postItemRowFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.
  DEFINE VARIABLE iType      AS INTEGER    NO-UNDO.
  DEFINE VARIABLE cItemTypes AS CHARACTER  NO-UNDO
    INITIAL "BASEBALL,CROQUET,FISHING,FOOTBALL,GOLF,SKI,SWIM,TENNIS".
  DEFINE VARIABLE iTypeNum   AS INTEGER    NO-UNDO.
  DEFINE VARIABLE cType      AS CHARACTER  NO-UNDO.
  DO iType = 1 TO NUM-ENTRIES(cItemTypes):
    cType = ENTRY(iType, cItemTypes).
    IF INDEX(ttItem.ItemName, cType) NE 0 THEN
      ttItem.ItemName = REPLACE(ttItem.ItemName, cType, cType).
  END.
END PROCEDURE.
```

This bit of code looks at the `Item`Name field, identifies whether it contains one of several key strings such as `BASEBALL` or `CROQUET`, and highlights those strings by replacing the string in the name with all uppercase. This is a simple illustration of the usefulness of the row-level event. You can use it to fill in calculated fields, to filter records beyond the default record selection, and for other row-level purposes.

12. Now if you run the `Order` ProDataSet window, you can see first of all that the code to attach Data-Sources and other such things that was moved into the event procedures executes correctly. In addition, the special event processing code that calculates the `OrderTotal` field and highlights the key words in the `Item Name` are working as well. For example:



Summary

Now you understand how to define business logic that will execute whenever your ProDataSet is filled. Dynamic ProDataSet Basics on page 91 covers the dynamic forms for ProDataSets, Data-Sources, and other objects, and the methods and attributes you can use to inspect and control a ProDataSet through its handle.

4

Dynamic ProDataSet Basics

Like other ABL objects, ProDataSets can be either static or dynamic. You define a static ProDataSet and its related objects, such as Data-Sources, using the statements you learned earlier. In this chapter, you will learn how to create a dynamic ProDataSet, add buffers dynamically, add Data-Relations dynamically, and create dynamic Data-Sources.

You can also access a ProDataSet, Data-Relation, or Data-Source through its object handle.

For details, see the following topics:

- Creating a dynamic ProDataSet
- Deleting a dynamic ProDataSet
- Specifying member buffers
- Creating Data-Relation objects
- Creating a dynamic Data-Source
- Duplicating ProDataSets with the CREATE-LIKE method
- Sample procedure: creating a dynamic ProDataSet
- Summary

Creating a dynamic ProDataSet

Here is the syntax for dynamically creating a ProDataSet:

Syntax

```
CREATE DATASET dataset-handle [ IN WIDGET-POOL poolname ].
```

Instead of defining a static ProDataSet, you create a dynamic ProDataSet when part or all of its definition has to be determined at run time. For example, you might need data in the application or from specific user requests to define your ProDataSet. The CREATE DATASET statement gives you a handle in the variable *dataset-handle*, which you can then use to build up the ProDataSet through additional statements that add buffers and relations to it, much as you build up a dynamic query or temp-table.

Note: If you do **not** specify the WIDGET-POOL*poolname* syntax to allocate storage in a specific named widget pool, you might expect that the ProDataSet will be created in the closest unnamed widget pool. This is the default behavior for other objects such as dynamic queries. Instead, the ProDataSet goes into the session's unnamed widget pool. This is necessary because the ProDataSet object must be able to serve as an output parameter. Therefore, the object cannot disappear when its defining procedure returns. Deletion must be delayed until the object can be handed off as an output parameter to the calling procedure. Likewise, the ProDataSet cannot go into a pool that is deleted when the procedure is deleted. If you give a pool name for an output parameter ProDataSet, remember that the pool must outlive the procedure where it is created.

The CREATE statement simply allocates storage for the ProDataSet description and puts a pointer to that storage into the *dataset-handle* variable. You must then run methods on the ProDataSet handle to define its buffers and relations. At this point, you can:

- Manipulate this new dynamic ProDataSet as a dynamic object
- Pass it locally or remotely into a static definition
- Pass it to another procedure that can receive and operate on it as a dynamic object

Passing ProDataSets

If you want to pass only the ProDataSet handle to another local procedure, which can then access methods and attributes through the handle, use the HANDLE parameter form. As with any other object, this form cannot be used in a remote call.

If you want to pass a reference to the entire ProDataSet structure and data to another procedure, whether local or remote, use the DATASET-HANDLE parameter form. In this case, the receiving procedure can also use the DATASET-HANDLE form to receive the ProDataSet as a dynamic object and access it through its handle.

If the receiving procedure has a static ProDataSet definition, then it can receive the ProDataSet as a static object using the DATASET parameter form, even though the ProDataSet was created dynamically.

In either of these latter two cases, where you pass the ProDataSet as a DATASET-HANDLE, you can pass it by reference when the call is local. Just like when you pass static ProDataSets, the call incurs no overhead from copying the ProDataSet.

Remember that when you pass a ProDataSet remotely or by value (the default), the AVM instantiates the ProDataSet, along with its temp-tables, buffer, relations, and all the temp-table data, in the procedure that receives the ProDataSet. If you use the dynamic DATASET-HANDLE form in the procedure that receives the ProDataSet, then the procedure can access all the ProDataSet elements through its attributes and methods.

Deleting a dynamic ProDataSet

When you are finished using a dynamic ProDataSet, you must delete it explicitly with the same statement you use for other ABL dynamic objects, as shown in the following syntax:

Syntax

```
DELETE OBJECT dataset-handle [NO-ERROR].
```

You can also delete the ProDataSet by deleting its named widget pool if it was created in one. If the ProDataSet is being passed as an output parameter, its widget pool should not be deleted in the called procedure. It is okay to delete the ProDataSet object itself in the called procedure. If the ProDataSet is in the unnamed session widget pool, the deletion will be delayed in the same way that TEMP-TABLE object deletions are delayed when they are output parameters. If the ProDataSet is in a named widget pool, it will be deleted when the widget pool is deleted.

Dynamic buffers and tables in a dynamic ProDataSet are normally deleted automatically when the ProDataSet is deleted. If you do not want one of a dynamic ProDataSet's buffers or temp-tables to be deleted, you can set the LOGICAL buffer attribute AUTO-DELETE to false to prevent auto-deletion of the buffer and its temp-table. This applies only to dynamic buffers.

Specifying member buffers

Every member of a ProDataSet is identified by a buffer for the temp-table holding that member's data. The same buffer cannot be added to more than one ProDataSet concurrently, although a member table can be a part of more than one ProDataSet, using separate buffers for independent record currency.

You can specify one or more buffers for a dynamic ProDataSet with the SET-BUFFERS method, which has the following syntax:

Syntax

```
[logical-var = ] dataset-handle:SET-BUFFERS
  ( {buffer-handle-expression | buffer-name-expression }
  [ , {buffer-handle-expression | buffer-name-expression } ] … )
```

This statement defines all the buffers for the ProDataSet in a single method call, whether there are one or more temp-tables in the ProDataSet.

Alternatively, you can add buffers to a dynamic ProDataSet one at a time, using the ADD-BUFFER method. This is the syntax for the ADD-BUFFER method:

Syntax

```
[ logical-var = ] dataset-handle:ADD-BUFFER
  ( buffer-handle-expression | buffer-name-expression )
```

These definitions apply to both the ADD-BUFFER and the SET-BUFFERS methods:

- *buffer-handle-expression* can be either a temp-table handle or a buffer handle. If it is a temp-table handle, then the temp-table's *default-buffer-handle* is used as the buffer handle.
- *buffer-name-expression* is the name of a static buffer for a temp-table scoped to the procedure that contains the statement.

Both ADD-BUFFER and SET-BUFFERS return an optional logical status, which is true if the method was successful and false if not. Use SET-BUFFERS just once to set the entire buffer list for the ProDataSet. If there were any buffers already added, SET-BUFFERS removes them first. ADD-BUFFER adds a single buffer to those already there.

If you want to reset your ProDataSet to have no buffers, you can use the following syntax:

Syntax

```
dataset-handle:CLEAR()
```

This removes all elements from the ProDataSet definition, including buffers, relations, and so on. It restores the state of the handle to exactly what it was after the CREATE DATASET statement.

As noted, by default, dynamic buffers that are added to a ProDataSet are deleted when the ProDataSet is deleted, unless you set the buffer's AUTO-DELETE attribute to false.

Creating Data-Relation objects

The Data-Relation object only exists with respect to a ProDataSet. Therefore, a dynamic Data-Relation is created by executing a ProDataSet object method, not with a CREATE statement for a separate object. You cannot delete a Data-Relation. When the ProDataSet object is deleted or cleared, its Data-Relation objects are automatically deleted.

Use the ADD-RELATION method to add a relation to a dynamic ProDataSet. This is the syntax for the ADD-RELATION method:

Syntax

```
dataset-handle:ADD-RELATION( parent-buffer-handle, child-buffer-handle,
  [ pairs-list [,reposition-mode [, nested [, recursive [, not active ]
  ] ] ] ]) .
```

In the ADD-RELATION method:

- *parent-buffer-handle* is the buffer-handle of the Data-Relation parent.
- *child-buffer-handle* is the buffer-handle of the Data-Relation child.
- *pairs-list* is an expression that evaluates to a comma-delimited list of parent-field, child-field pairs to describe the relation between parent and child, using the same form as the Data-Relation phrase in a static DEFINE DATASET statement. Make sure that the list does not contain any embedded spaces. The AVM does not trim the elements in the list.

- *reposition-mode* is a logical value. If true, it makes the Data-Relation a reposition relation; if it is false, the Data-Relation is a selection relation, which is the default.
- *nested* is an optional logical value. If true, the AVM nests child rows of ProDataSet buffers within their parent rows when serializing the data into XML or JSON.
- *recursive* is an optional logical value. If true, this mode instructs the ProDataSet FILL to load self-referencing elements. That is, an element can reference a child element that already either directly or indirectly references the parent element in the hierarchy. You can check to see if a Data-Relation is recursive using the readable RECURSIVE attribute of the Data-Relation object handle.
- *not active* is an optional logical value. If false, the Data-Relation is inactive. This allows you to have two relations between the same two ProDataSet temp-table buffers, but only have one active at a time.

You can add a multiple Data-Relation involving the same parent member temp-table. A table can be a child in only one active relation.

If the buffer handles specified are not in the ProDataSet, or fields supplied are not in the indicated tables, the AVM raises an error at run time.

The ADD-RELATION method returns a handle to the Data-Relation object, or the Unknown value (?) if there is an error.

A buffer for a temp-table does not have to have any Data-Relations at all. In this case, it is treated as an independent top-level data table within the ProDataSet. It must therefore be filled independently, either individually or when the ProDataSet as a whole is filled. There can be any number of top-level data tables (tables that are not the child in a Data-Relation). Each top-level table can have child tables or not.

You cannot take a static ProDataSet and add a buffer to it using the ADD-BUFFER method, or replace its buffers using SET-BUFFERS, or erase its buffer definition using the CLEAR method. However, you **can** add a dynamic Data-Relation to a static ProDataSet. This could be useful in the case where you need to use relations to navigate the ProDataSet but which are not necessary for filling it. Another case could be where a single ProDataSet might require different relations, depending on how it is used.

A difference in the ProDataSet's relations does not cause an error when the ProDataSet is passed as a parameter. If the ProDataSet is received dynamically using the DATASET-HANDLE form, then the AVM creates the Data-Relations that are defined in the caller as dynamic relations in the ProDataSet in the called procedure.

If the ProDataSet is received statically using the DATASET parameter form, then the AVM ignores the relations in the caller and uses the Data-Relation definitions in the receiving procedure's static ProDataSet definition. This means, for example, that you could pass a static ProDataSet from server to client, add a Data-Relation to it dynamically on the client, and then pass the ProDataSet back to the server without error. If the server-side definition is static, the extra relation on the client is simply ignored when it arrives on the server.

Creating a dynamic Data-Source

Because a ProDataSet and a Data-Source are independent objects, there is no need to create dynamic Data-Source objects for a dynamic ProDataSet. If you have static Data-Sources available, you can attach them to a dynamic ProDataSet handle just as easily as you can to a static ProDataSet handle. For example, you could have something in the way of a collection of static Data-Source definitions for a set of database tables, but have a procedure that assembled those tables in a variety of ways into various dynamic ProDataSets. The Data-Source definitions could be static, and the ProDataSet definition dynamic.

You can also create a dynamic Data-Source when you need one. You can attach a dynamic Data-Source to either a static or dynamic ProDataSet. The CREATE DATA-SOURCE statement is used to create a dynamic Data-Source, as shown using this syntax:

Syntax

```
CREATE DATA-SOURCE data-source-handle [ IN WIDGET-POOL poolname ] .
```

Like other database objects, the dynamic Data-Source is created in the closest unnamed widget-pool unless the IN WIDGET-POOL phrase is used. It is automatically deleted when the widget-pool is deleted. If there is no WIDGET-POOL phrase and no local CREATE WIDGET-POOL statement, then it will go into the session's unnamed widget pool. In this latter case, it must be deleted specifically, using the DELETE OBJECT statement. By contrast, a static Data-Source is automatically deleted when the procedure it is defined in is destroyed.

When you create a dynamic Data-Source, you get a handle and an empty structure to fill in. You can then associate a query with the dynamic Data-Source by setting its QUERY attribute, as shown in the following syntax:

Syntax

```
data-source-handle:QUERY = query-handle.
```

To disassociate the query and Data-Source, set the QUERY attribute to the Unknown value (?), as shown in the following syntax:

Syntax

```
data-source-handle:QUERY = ?.
```

Just like a static Data-Source, a dynamic Data-Source must have a set of buffers that can be deduced from the query, or you can supply them separately from the query.

The ADD-SOURCE-BUFFER method lets you build up a Data-Source at run time. This is the syntax for the ADD-SOURCE-BUFFER method:

Syntax

```
data-source-handle:ADD-SOURCE-BUFFER( buffer-handle, key-fields ).
```

In this method:

- *buffer-handle* is the handle of a database buffer, or a temp-table buffer that you want to use as a Data-Source for a temp-table in a ProDataSet.
- *key-fields* is a character expression that evaluates to a comma-separated list of key component fields for finding a record using the buffer, just as in the KEYS phrase in the static DEFINE DATA-SOURCE statement. This argument can be passed as the Unknown value (?) when the AVM can deduce a unique primary key based on the index definitions for the table, or when the Data-Source will not be used for updates, and therefore a unique key is not needed.

As with ProDataSets, you cannot use the ADD-SOURCE-BUFFER method to add a buffer to a static Data-Source dynamically. The DEFINE DATA-SOURCE statement must contain the complete definition of the static Data-Source.

Duplicating ProDataSets with the CREATE-LIKE method

At times you need to create a new dynamic ProDataSet that has exactly the same structure as another ProDataSet, either static or dynamic. The CREATE-LIKE method does this for you, much as the same method name does for temp-tables. This is the syntax for the CREATE-LIKE method:

Syntax

```
second-dataset-handle:CREATE-LIKE( { first-dataset-handle |
                                     first-dataset-name-expr }
                                      [ , prefix ] ).
```

The important parameters are:

- *second-dataset-handle* is the handle of a dynamic ProDataSet object with no definition, which you want to have inherit the entire definition of the *first-dataset*.
- *prefix* is an optional prefix string to be added to the beginning of the table names in the second-dataset. If you do not specify this, then the tables in the *second-dataset* have the same names as the tables in the *first-dataset*.

To use this method, you must first create a dynamic ProDataSet, and then use CREATE-LIKE to extract the definition of another where hDataSet is the handle of an existing ProDataSet, as shown:

```
DEFINE VARIABLE hDataSet2 AS HANDLE NO-UNDO.

CREATE DATASET hDataSet2.
hDataSet2:CREATE-LIKE(hDataSet).
```

The temp-table buffers in the second ProDataSet have the same name as those in the first. Because these are dynamic temp-tables and dynamic buffers, there is no name scoping conflict with the names of the original ProDataSet. You must, in any case, reference the temp-tables and buffers in hDataSet2 by their handles in order to identify them properly. However, if you wish to have distinct table names, you must supply the optional *prefix* argument.

You can use CREATE-LIKE any time you need a second dynamic ProDataSet within a procedure that has the same structure as another already defined or created. The ProDataSet structure, including temp-table and relation definitions, is copied to the second ProDataSet; the source ProDataSet's data, however, is not copied.

If you need two static ProDataSets with the same structure within a single procedure, you need to define them individually using two different names, along with temp-tables with distinct names. Unlike the dynamic temp-tables in a ProDataSet you build using CREATE-LIKE, two static temp-tables scoped to the same procedure cannot have the same name. So, you could use a sequence of statements to create two equivalent static ProDataSets, as shown:

```
DEFINE TEMP-TABLE TableA NO-UNDO
  FIELD KeyField1 AS INTEGER
  FIELD Field2    AS CHARACTER.

DEFINE TEMP-TABLE TableB NO-UNDO
  FIELD KeyField1 AS INTEGER
  FIELD Field3    AS CHARACTER.

DEFINE DATASET DataSet1 FOR TableA, TableB
  DATA-RELATION Relation1 FOR TableA, TableB
  RELATION-FIELDS (KeyField1, KeyField1).

DEFINE TEMP-TABLE TableA1 NO-UNDO LIKE TableA.
DEFINE TEMP-TABLE TableB1 NO-UNDO LIKE TableB.

DEFINE DATASET DataSet2 FOR TableA1, TableB1
  DATA-RELATION Relation1 FOR TableA1, TableB1
  RELATION-FIELDS (KeyField1, KeyField1).
```

An include file with an argument that adds a standard prefix or suffix to the temp-table and ProDataSet names could, of course, simplify the job of defining multiple sets of temp-tables and ProDataSets with equivalent structures, like those in the preceding example.

Keep in mind that these static ProDataSets and their temp-tables must have distinct names because they are top-level, unqualified objects that share the same name space within the procedure. Data-Relations and Relation-Fields can have the same names in both ProDataSets because they are implicitly qualified within their definitions by the ProDataSet name, just as the temp-table fields are implicitly qualified by the temp-table name. Any reference to the fields within the procedure needs to explicitly qualify the names so that the AVM knows which one you are referring to. A relation can be accessed by name through its parent ProDataSet, as in `DATASET DataSet1:GET-RELATION("Relation1")` or in `DATASET DataSet2:GET-RELATION("Relation1")`.

Alternatively, you can use a persistent procedure that defines a ProDataSet as a kind of factory for multiple instances of that ProDataSet. Each running instance of the procedure has its own ProDataSet, scoped to that procedure, each with its own data.

You typically use the CREATE-LIKE method to create a second ProDataSet for holding just the changes that have been made so that you can pass them to another procedure or another session for processing. This topic is discussed in later chapters along with the operation of before-tables where the original field values for changed rows are held. For now, note that if the original ProDataSet has defined before-tables, the CREATE-LIKE method creates a before-table for the tables in the new ProDataSet as well.

For cases where you want to copy the data in one ProDataSet to another ProDataSet, you can use the COPY-DATASET method, which is described in COPY-DATASET and COPY-TEMP-TABLE methods on page 193. COPY-DATASET also lets you copy the ProDataSet structure and definition as well, if the target is a dynamic ProDataSet handle with no structure. In this way, it lets you combine what the CREATE-LIKE method does with copying data in the same operation.

Sample procedure: creating a dynamic ProDataSet

Let us create a simple example to see how these dynamic statements work together. Create a new procedure called `DynamicDataSet.p`. This procedure:

- Takes several input parameters that define the elements of a ProDataSet
- Creates the ProDataSet
- Prepares a query for its top-level table
- Fills the ProDataSet
- Returns it to the caller

The ProDataSet can have any number of buffers, but (for the sake of simplicity) creates a single Data-Relation between the top two buffers. Any additional buffers are considered independent buffers not related to others in the ProDataSet.

Here are the parameters for the procedure:

```
/* DynamicDataSet.p -- creates a dynamic DataSet and Data-Sources, fills it
   for a key value passed in, and returns it. */
DEFINE INPUT  PARAMETER pcBuffers    AS CHARACTER  NO-UNDO.
DEFINE INPUT  PARAMETER pcFields     AS CHARACTER  NO-UNDO.
DEFINE INPUT  PARAMETER pcSources    AS CHARACTER  NO-UNDO.
DEFINE INPUT  PARAMETER pcSourceKeys AS CHARACTER  NO-UNDO.
DEFINE INPUT  PARAMETER pcKeyValue   AS CHARACTER  NO-UNDO.
DEFINE OUTPUT PARAMETER DATASET-HANDLE phDataSet.
```

The parameters provide the following information:

- **`pcBuffers`** — A list of buffer handles expressed as a comma-delimited string for temp-tables in the caller to be included in the ProDataSet
- **`pcFields`** — A list of fields to define the relation between the first two buffers passed in
- **`pcSources`** — A list of database table names to use as Data-Sources for the buffers, one for each buffer
- **`pcSourceKeys`** — A list of key fields for the Data-Source tables, one for each Data-Source
- **`pcKeyValue`** — A key value for the top-level table to use to fill the ProDataSet
- **`phDataSet`** — The procedure returns the dynamic ProDataSet as an `OUTPUT` parameter using the `DATASET-HANDLE` form, so that the caller can inspect and use the ProDataSet

In this first example, the ProDataSet is actually constructed from static temp-tables defined in the calling procedure, so it will work only within a single session. Later you will take advantage of additional dynamic ProDataSet methods and attributes to separate the caller from the called program entirely.

You need these local variables in the procedure:

```
DEFINE VARIABLE hBuffer     AS HANDLE  NO-UNDO.
DEFINE VARIABLE hDataSource AS HANDLE  NO-UNDO.
DEFINE VARIABLE hQuery      AS HANDLE  NO-UNDO.
DEFINE VARIABLE iEntry      AS INTEGER NO-UNDO.
```

The first executable statement in the procedure creates a dynamic ProDataSet.

The procedure then walks through the list of temp-table buffer handles passed into it as a comma-separated string and adds each in turn to the ProDataSet by converting each string back into a handle and executing the ADD-BUFFER method for it. For example:

```
CREATE DATASET phDataSet.
DO iEntry = 1 TO NUM-ENTRIES(pcBuffers):
  phDataSet:ADD-BUFFER(WIDGET-HANDLE(ENTRY(iEntry, pcBuffers))).
END.
```

Next, the procedure adds a single Data-Relation to the ProDataSet, using the first buffer as the parent and the second buffer handle as the child. The GET-BUFFER-HANDLE method returns the ProDataSet's temp-table buffers in the same order in which they were added. The pcFields parameter defines the parent and child fields to use to establish the relation, as shown:

```
phDataSet:ADD-RELATION(phDataSet:GET-BUFFER-HANDLE(1),
                       phDataSet:GET-BUFFER-HANDLE(2),
                       pcFields).
```

Next, the procedure walks through the list of source tables for the ProDataSet. For each one, it creates a dynamic Data-Source. It then creates a dynamic buffer for the table and uses ADD-SOURCE-BUFFER to add it to the dynamic Data-Source.

The final statement of this group uses a sequence of handle attributes to do several steps. Once again GET-BUFFER-HANDLE returns the handle of the temp-table buffer in the ProDataSet that corresponds to the Data-Source. The statement then uses this handle to attach the Data-Source to that buffer, as shown:

```
DO iEntry = 1 TO NUM-ENTRIES(pcSources):
  CREATE DATA-SOURCE hDataSource.
  CREATE BUFFER hBuffer FOR TABLE ENTRY(iEntry, pcSources).
  hDataSource:ADD-SOURCE-BUFFER(hBuffer, ENTRY(iEntry,pcSourceKeys)).
  phDataSet:GET-BUFFER-HANDLE(iEntry):ATTACH-DATA-SOURCE(hDataSource).
  …
```

There is no problem with using the same buffer handle for each dynamic buffer and the same handle for each dynamic Data-Source because as each is added to the ProDataSet, the dynamic ProDataSet can keep track of them internally. Attributes and methods such as GET-BUFFER-HANDLE let you walk through the ProDataSet after it has been built so that you can identify all of its components.

Now the procedure needs to prepare a database query for the table that is the source for the first, top-level temp-table in the ProDataSet so that the ProDataSet can be filled with all the records related to a single top-level record.

For this entry, the procedure creates a dynamic query and adds the dynamic buffer for this table's database source table as the query's buffer. Then it constructs a QUERY-PREPARE string from the table name, the key field for the table, and the key value to use to populate the ProDataSet. (Note that for simplicity the procedure expects only one key field for each Data-Source.) It makes this dynamic query the query for the first Data-Source, as shown:

```
IF iEntry = 1 THEN DO:
  CREATE QUERY hQuery.
  hQuery:ADD-BUFFER(hBuffer).
  hQuery:QUERY-PREPARE("FOR EACH " + ENTRY(1, pcSources) +
                       " WHERE " + ENTRY(1, pcSourceKeys) +
                       " = " + pcKeyValue).
  hDataSource:QUERY = hQuery.
END. /* END DO IF iEntry = 1 */
```

This ends the loop that walks through the list of Data-Sources.

Finally, the procedure issues a FILL on the ProDataSet handle, then deletes the dynamic query and each dynamic Data-Source, as shown:

```
phDataSet:FILL().
DELETE OBJECT hQuery.

DO iEntry = 1 TO phDataSet:NUM-BUFFERS:
  hBuffer = phDataSet:GET-BUFFER-HANDLE(iEntry).
  DELETE OBJECT hBuffer:DATA-SOURCE.
END.
```

After this method executes, the procedure returns the dynamic ProDataSet to the caller as an output parameter.

Now it is time to write a procedure, DynamicMain.p, that calls this one. This version of the procedure defines some static temp-tables for the ProDataSet to use, along with a handle to hold the ProDataSet that is returned to it. For example:

```
/* DynamicMain.p -- gets DynamicDataSet.p to create, fill,
   and return a dynamic DataSet for these temp-tables. */
DEFINE TEMP-TABLE ttCust     NO-UNDO LIKE Customer.
DEFINE TEMP-TABLE ttOrder    NO-UNDO LIKE Order.
DEFINE TEMP-TABLE ttSalesRep NO-UNDO LIKE SalesRep.

DEFINE VARIABLE hDataSet AS HANDLE NO-UNDO.
```

Here is the statement that runs the DynamicDataSet procedure:

```
RUN DynamicDataSet.p
  (INPUT STRING(BUFFER ttCust:HANDLE) + "," +
   STRING(BUFFER ttOrder:HANDLE) + "," +
   STRING(BUFFER ttSalesRep:HANDLE),
   INPUT "CustNum,CustNum",
   INPUT "Customer,Order,SalesRep",
   INPUT "CustNum,OrderNum,SalesRep",
   INPUT "1",
   OUTPUT DATASET-HANDLE hDataSet).
```

In this statement:

- The first parameter is a list of the buffer handles of this procedure's static temp-tables. Once again, this simplifies the example to the extent that these handles could, of course, not be passed across an AppServer call to be used on the server. Later in this chapter, you will see a fully dynamic version.
- The second parameter is a list of key fields for the parent and child of the ProDataSet's one Data-Relation.
- The third parameter is a list of the database source tables.
- The fourth parameter is a list of the (single) key fields for each of those tables.
- The fifth parameter is the key value to use to the top-level table, in this case `Customer` number 1.

The final `OUTPUT` parameter receives the ProDataSet back from the other procedure.

The following is a series of simple `FOR EACH` statements showing what we get back:

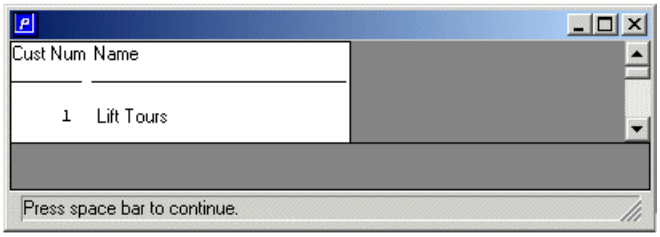
```
FOR EACH ttCust:
  DISPLAY ttCust.CustNum ttCust.Name.
END.
FOR EACH ttOrder:
  DISPLAY ttOrder.CustNum ttOrder.OrderNum.
END.
FOR EACH ttSalesRep:
  DISPLAY ttSalesRep.SalesRep ttSalesRep.RepName.
END.
```

And finally, remember to delete the dynamic ProDataSet that has been returned, as shown:

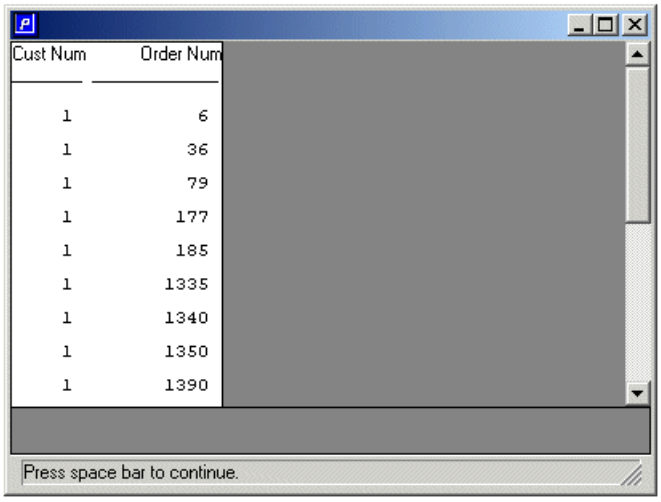
```
DELETE OBJECT hDataSet.
```

Tip: When your procedure creates a dynamic ProDataSet or receives a dynamic ProDataSet as an `OUTPUT` parameter, you must remember to take responsibility for deleting it when you are done using it. If you specify `BY-REFERENCE` in the parameter, your procedure might not know for sure whether it "owns" the ProDataSet or not. If the called procedure is run locally, then the calling procedure is actually using the ProDataSet owned by the called procedure. If the same procedure is run across an AppServer connection, then the ProDataSet is copied across the network and a new dynamic ProDataSet is created in the calling procedure. In such a case, you should use the statement `DELETE OBJECT` dataset-handle `NO-ERROR` to delete the object only if it is created for the calling procedure. The AVM prevents the calling procedure from deleting a ProDataSet passed `BY-REFERENCE` from another local procedure, and the `NO-ERROR` keyword suppresses the warning error in this case.

Run the main procedure to see what you get back. First is the single `Customer` record that satisfies the top-level query, as shown:



Next come all of its `Orders`, as shown in this screen shot:



These are identified by the AVM based on the dynamic Data-Relation between these two tables. The AVM has constructed a query automatically that selected just these `Orders` for the ProDataSet.

The `SalesReps` come last. Because there is no Data-Relation defined for this table, and no query for it either, the AVM retrieves all the `SalesReps` and loads them all into the ProDataSet, as shown:



Summary

This chapter introduced you to creating, duplicating, and deleting the following components of a dynamic ProDataSet:

- ProDataSet
- Member buffers
- Data-Relation
- Dynamic Data-Source

The next chapter describes the attributes and methods you can use to manipulate these components.

5

ProDataSet Attributes and Methods

This chapter introduces the attributes and methods of the ProDataSet, Data-Relation, and Data-Source objects.

For details, see the following topics:

- Accessing the handle of a ProDataSet
- Accessing a member buffer of a ProDataSet
- Sample procedures: using attributes and methods
- Accessing Data-Relations
- Using Data-Source attributes and methods
- Enhanced query support for ProDataSet buffers
- Session attributes for ProDataSets, Data-Sources, and queries
- Other ProDataSet methods
- Building a dynamic user interface from a ProDataSet
- Using the SYNCHRONIZE method
- Sample procedure: adding REPOSITION and SYNCHRONIZE
- Summary

Accessing the handle of a ProDataSet

Through the handle of a ProDataSet, whether it is static or dynamic, you can access many attributes that give you information about the ProDataSet and the handles of its components. A few of these attributes are settable at run time to let you adjust the behavior of the ProDataSet. The ProDataSet also has methods you use to invoke behavior using the same handle. Remember that to get the handle of a static ProDataSet, you simply precede the reference with the `DATASET` keyword, as shown:

```
hDataSet = DATASET dsOrder:HANDLE.
```

You can also use the expression `DATASET dsOrder:HANDLE` in place of a handle within a larger ABL statement without assigning it to a variable at all.

Accessing a member buffer of a ProDataSet

Temp-tables in a ProDataSet are normally accessed through their buffers. This is largely because, while a temp-table can possibly be a part of more than one ProDataSet, a buffer must be specifically associated with a single ProDataSet. The ProDataSet accesses its member temp-tables through the buffers for them. In some cases, as explained here, you can use either a temp-table handle or its buffer handle to access the table, but in general all access is through the buffer. This is important to keep in mind, especially because each static temp-table has by default a buffer of the same name, so the distinction can be confusing. Always remember that `TEMP-TABLE ttOrder:HANDLE` and `BUFFER ttOrder:HANDLE` are two different handles that point to different, though related, objects. This section explains the ProDataSet methods and attributes that give you access to the buffers of a ProDataSet, and the buffer attributes and methods for those buffers.

In order to access the member buffers in a ProDataSet, you use the `GET-BUFFER-HANDLE` method, which accepts either a numeric index or a buffer name, as shown in the following syntax:

Syntax

```
[handle-var = ] dataset-handle:GET-BUFFER-HANDLE
  ( buffer-index-expression | buffer-name-expression )
```

In this method:

- *`buffer-index-expression`* is the one-based index of the member-buffer in the ProDataSet.
- *`buffer-name-expression`* is a character expression that evaluates to the name of a buffer in the ProDataSet.

`GET-BUFFER-HANDLE` returns the buffer handle of the member buffer, or the `Unknown value (?)` if the buffer cannot be found.

To find out how many temp-tables there are in a ProDataSet, you use the `NUM-BUFFERS` attribute, as shown in the following syntax:

Syntax

```
[ integer-var ] = dataset-handle:NUM-BUFFERS
```

Of course, if the ProDataSet definition is local to the procedure seeking the buffer handle, you can simply get the handle directly if you know its name without going through the ProDataSet, as shown in the following syntax:

Syntax

```
hBuffer = BUFFER MyBuffer:HANDLE.
```

For example, the *buffer-name-expression* form is useful if you have only the handle to the ProDataSet available and need to locate one of its buffers by name to act on it, to set a callback procedure, or to attach a Data-Source.

The buffer handle provides the same access to the data in the temp-table as any other usage of a buffer handle.

The member buffers of a ProDataSet point back to the ProDataSet handle using the DATASET attribute, as shown in the following syntax:

Syntax

```
dataset-handle = buffer-handle:DATASET.
```

This returns the ProDataSet object handle of a member buffer object.

Sample procedures: using attributes and methods

This section extends the example procedures from Dynamic ProDataSet Basics on page 91 to make them more truly dynamic. It then shows small sections of code to illustrate the use of each of the methods and attributes.

To modify the example, create a copy of DynamicDataSet.p and name the new procedure DynamicDataSet2.p. Modify the parameter list to eliminate the pcBuffers parameter and to reorder the others to base the ProDataSet definition on the database tables it is filled from. For example:

```
/* DynamicDataSet2.p -- creates a dynamic DataSet and Data-Sources, fills it
   for a key value passed in, and returns it. */

DEFINE INPUT  PARAMETER pcSources    AS CHARACTER  NO-UNDO.
DEFINE INPUT  PARAMETER pcSourceKeys AS CHARACTER  NO-UNDO.
DEFINE INPUT  PARAMETER pcFields     AS CHARACTER  NO-UNDO.
DEFINE INPUT  PARAMETER pcKeyValue   AS CHARACTER  NO-UNDO.
DEFINE OUTPUT PARAMETER DATASET-HANDLE phDataSet.
```

Also, you need another handle variable to point to a series of dynamic temp-tables you will create for the database sources, as shown:

```
DEFINE VARIABLE hTable AS HANDLE NO-UNDO.
```

Modify the block of code that walks through the buffer handle list in `DynamicDataSet.p` to walk through the list of database source tables and create a dynamic temp-table `LIKE` each in turn.

Prepare each temp-table definition and then add the table's default buffer handle to the ProDataSet. The following block of code replaces the code beginning with `DO iEntry = 1 TO NUM-ENTRIES(pcBuffers)`:

```
DO iEntry = 1 TO NUM-ENTRIES(pcSources):
  CREATE TEMP-TABLE hTable.
  hTable:CREATE-LIKE(ENTRY(iEntry, pcSources)).
  hTable:TEMP-TABLE-PREPARE("tt" + ENTRY(iEntry, pcSources)).
  phDataSet:ADD-BUFFER(hTable:DEFAULT-BUFFER-HANDLE).
END.
```

As with other parts of these procedures, you can use the same temp-table handle variable for each of the temp-tables because once it has been added to the ProDataSet, the ProDataSet structure keeps track of the handle's value and position within the ProDataSet. You are then free to reuse the same handle variable to create another new dynamic temp-table that will have its own value for that handle.

There is one more small change. The `pcKeyValue` parameter used to retrieve data related to a single top-level table is changed to accept an expression such as `"= 1"` or `"< 10"`, by removing the equal sign literal (`" = "`) from the `QUERY-PREPARE` method. This lets you have one or more top-level records in the ProDataSet.

This will help illustrate some of the object attributes as we go along. For example:

```
hQuery:QUERY-PREPARE("FOR EACH " + ENTRY(1, pcSources) +
                     " WHERE " + ENTRY(1, pcSourceKeys) + pcKeyValue).
```

The rest of the procedure remains the same. Now the dependency on static temp-table definitions has been removed, and there is no need to pass any handles in the parameter list that would not survive being passed across the AppServer boundary.

Copy `DynamicMain.p` to a new variant called `DynamicMain2.p`. In this new procedure, you can delete the static temp-table definitions because you are making everything dynamic. This is why `dynamicDataSet2.p` now creates them as dynamic temp-tables instead of receiving their handles.

You will need several variables along the way to hold various attributes and other values, so define them at the top of the procedure, as shown:

```
DEFINE VARIABLE hBuffer   AS HANDLE  NO-UNDO.
DEFINE VARIABLE hQuery    AS HANDLE  NO-UNDO.
DEFINE VARIABLE hRelation AS HANDLE  NO-UNDO.
DEFINE VARIABLE iBuffer   AS INTEGER NO-UNDO.
```

Change the RUN statement to run DynamicDataSet2.p, and rearrange the parameters to match. Change the pcKeyValues parameter "1" to be "= 1", as shown:

```
RUN DynamicDataSet2.p
   (INPUT "Customer,Order,SalesRep",
    INPUT "CustNum,OrderNum,SalesRep",
    INPUT "CustNum,CustNum",
    INPUT "= 1",
    OUTPUT DATASET-HANDLE hDataSet).
```

Create a dynamic query that you will use in several parts of the procedure:

```
CREATE QUERY hQuery.
```

Remove all the rest of the code (the DISPLAY blocks) except for the final DELETE OBJECT statement.

At this point, you can add a series of blocks of code following the CREATE QUERY statement to illustrate how to access the ProDataSet dynamically.

This first example is a block of code that retrieves the number of buffers in the ProDataSet. For each one, it retrieves its buffer handle and then does a dynamic FIND-FIRST method on that handle to position to the first record in that temp-table. (FIND-FIRST is, of course, a standard ABL dynamic buffer method.) The MESSAGE statement shows the first two fields in each buffer:

```
/* This block shows how to access the DataSet's buffers and the data in their
   temp-table records. */
DO iBuffer = 1 TO hDataSet:NUM-BUFFERS:
  hBuffer = hDataSet:GET-BUFFER-HANDLE(iBuffer).
  hBuffer:FIND-FIRST().
  MESSAGE "Buffer " hBuffer:NAME SKIP
    hBuffer:BUFFER-FIELD(1):NAME
    hBuffer:BUFFER-FIELD(1):BUFFER-VALUE SKIP
    hBuffer:BUFFER-FIELD(2):NAME
    hBuffer:BUFFER-FIELD(2):BUFFER-VALUE
    VIEW-AS ALERT-BOX.
END.
```

When you run this, you can confirm that the dynamic temp-tables have the same data as the static temp-tables did before, as shown:







A ProDataSet buffer that is not the child of a relation is referred to as a top-level buffer. There might be more than one top-level buffer in a ProDataSet. The `NUM-TOP-BUFFERS` attribute gives you the number of those buffers, as shown in the following syntax:

Syntax

```
[ integer-var = ] dataset-handle:NUM-TOP-BUFFERS
```

The `GET-TOP-BUFFER` method returns the handle of one of those buffers using its index within the list of top-level buffers, as shown with this syntax:

Syntax

```
[ handle-var = ] dataset-handle:GET-TOP-BUFFER( buffer-index )
```

This example code for `DynamicMain2.p` shows that `ttCustomer` and `ttSalesRep` are both top-level buffers, because they do not participate in a relation:

```
/* This block shows the attribute and method that access the list of DataSet
   buffers that are not children in a relation.*/
DO iBuffer = 1 TO hDataSet:NUM-TOP-BUFFERS:
  hBuffer = hDataSet:GET-TOP-BUFFER(iBuffer).
  MESSAGE "Buffer " iBuffer hBuffer:NAME
    VIEW-AS ALERT-BOX.
END.
```

This code output proves the point:





Accessing Data-Relations

This section looks at attributes and methods that give you access to, and information about, the ProDataSet's Data-Relations. First, there is a NUM-RELATIONS attribute to return the number of Data-Relations, as shown in the following syntax:

Syntax

```
[ integer-var = ] dataset-handle:NUM-RELATIONS
```

You can retrieve the handle to a particular relation using the GET-RELATION method with its numeric index in the ProDataSet or its name. This is the syntax for the GET-RELATION method:

Syntax

```
[ handle-var = ] dataset-handle:GET-RELATION
  ( relation-index-expr | relation-name-expr)
```

Where:

- *relation-index-expr* is an integer expression that evaluates to the one-based index of the Data-Relation in the ProDataSet.
- *relation-name-expr* is a character expression that evaluates to the name of the Data-Relation.

Once you have a relation object, you can access its attributes. CHILD-BUFFER returns the buffer handle of the child member of the Data-Relation, as shown in the following syntax:

Syntax

```
[ buffer-handle = ] relation-handle:CHILD-BUFFER
```

Likewise, PARENT-BUFFER returns the parent buffer handle of the relation, as shown in the following syntax:

Syntax

```
[ buffer-handle = ] relation-handle:PARENT-BUFFER
```

Two attributes and method provide access to a relation through one of the member buffer handles. The NUM-CHILD-RELATIONS attribute returns the number of Data-Relations for which the buffer is the parent. There might be more than one because a buffer can be a parent in multiple relations, with different children. This is the syntax for the NUM-CHILD-RELATIONS attribute:

Syntax

```
[ integer-var = ] buffer-handle:NUM-CHILD-RELATIONS
```

As with the ProDataSet's GET-BUFFER-HANDLE method, you can use the GET-CHILD-RELATION method to walk through the list of child relations for a particular parent buffer using the buffer index within the list. This is the syntax for the GET-CHILD-RELATION method:

Syntax

```
[ handle-var = ] buffer-handle:GET-CHILD-RELATION( integer-expr )
```

You can also point back from the child buffer of a Data-Relation to its PARENT-RELATION, as shown in the following syntax:

```
[ handle-var = ] buffer-handle:PARENT-RELATION
```

Because a buffer cannot have more than one parent, there is no need for an attribute to return the number of parent buffers.

This code sample and output for DynamicMain2.p confirms the parent and child of the ProDataSet's one relation:




```
/* This block shows some of the Data-Relation methods and attributes. */
hRelation = hDataSet:GET-RELATION(1).
MESSAGE "Buffer" hRelation:CHILD-BUFFER:NAME "is the child of"
  hRelation:PARENT-BUFFER:NAME
  VIEW-AS ALERT-BOX.
```

The AVM creates a dynamic query representing each Data-Relation in the ProDataSet. This query provides filtering of child records in each relation so that you can use it if you want to walk the child records for the current parent. If you attach this query to a browse object, the browse is automatically refreshed with the right child records as the currently selected parent record changes. Several attributes provide access to and information about this query.

The first attribute is the relation's WHERE-STRING, which returns the current WHERE clause used to link the child of the relation to its parent, beginning with the keyword WHERE but not including the FOR EACH phrase of a QUERY-PREPARE method on a query. This attribute evaluates to the query string that the AVM generates for you based on the relation between the parent and the child. You could use this attribute to build an extended query of your own based on the default relationship but extending it in some way, as shown with this syntax:

Syntax

```
[ character-var = ] relation-handle:WHERE-STRING
```

The RELATION-FIELDS attribute returns the list of join fields between the parent and child as specified in the relation definition. This can be useful in code that exploits or extends the list of join fields without parsing the WHERE-STRING. In the default case, it provides essentially the same information but not necessarily in an ideal form for analyzing the relation. This is the syntax for the RELATION-FIELDS attribute:

Syntax

```
[ character-var = ] relation-handle:RELATION-FIELDS
```

You can access the dynamic query itself through the relation's QUERY attribute, as shown using the following syntax:

Syntax

```
[ handle-var = ] relation-handle:QUERY
```

This returns the handle of the navigation query that the AVM manages to filter children of the current parent when navigating the ProDataSet. This is not the same as the query defined for a Data-Source. This automatically generated query expresses the relation between parent and child temp-tables. This handle cannot be set, and the query cannot be modified. You can use this query to navigate the child records. This might be useful because it is automatically opened for you each time the parent changes. The AVM can insert the correct parent key field values directly into the child query each time the parent record changes, so it does not need to be fully prepared when the parent changes. This makes this default query more efficient than an ABL query you would re-prepare and reopen yourself each time the parent changes. You can also prepare and open your own query on any of the member buffers, or use FOR EACH or FIND syntax to access the rows in any member table.

As an example, you can add this MESSAGE statement to the procedure DynamicMain2.p:

```
MESSAGE "WHERE-STRING: " hRelation:WHERE-STRING SKIP
  "RELATION-FIELDS: " hRelation:RELATION-FIELDS
  VIEW-AS ALERT-BOX.
```

The code output shows you the WHERE-STRING and RELATION-FIELDS attributes for the ProDataSet's one relation:



As you can see, the WHERE-STRING of the query selects records in ttOrder whose CustNum matches ttCustomer. The RELATION-FIELDS attribute lists the fields used in the join. You can use either of these strings as a starting point, if you want, for a query of your own. The WHERE-STRING attribute is writeable, so you can either add to its value or replace it entirely. If you assign a new value to the attribute, the AVM uses the new WHERE expression when it re-opens the query during navigation to filter the child table of the data-relation.

So let us use this query to walk through the records in the ttOrder table:

```
hQuery = hRelation:QUERY.
hQuery:QUERY-OPEN().
hQuery:GET-FIRST().
hBuffer = hRelation:CHILD-BUFFER.

DO WHILE NOT hQuery:QUERY-OFF-END:
  MESSAGE hBuffer:BUFFER-FIELD(1):NAME
    hBuffer:BUFFER-FIELD(1):BUFFER-VALUE SKIP
    hBuffer:BUFFER-FIELD(2):NAME
    hBuffer:BUFFER-FIELD(2):BUFFER-VALUE
    VIEW-AS ALERT-BOX.
  hQuery:GET-NEXT().
END.
```

This block of code retrieves the relation's query, whose join fields and where-clause you have already seen. It opens the query, retrieves the first record, and displays the first two fields in the buffer for each record that satisfies the query.

If you run the procedure with this block of code, you get nothing. Why?

The reason is that there is no ttCustomer record selected. The ProDataSet does not automatically select any records in its temp-tables. You have to set the navigation in motion with queries or find statements of your own. The ProDataSet prepares queries for dependent tables, but again, you need to use the query to actually bring records into the buffers.

If you add this next statement just before the code in the previous code example, it brings the first (and, in this case, only) ttCustomer record into that parent table's buffer:

```
hRelation:PARENT-BUFFER:FIND-FIRST().
```

Now the query for ttOrder has a proper ttCustomer.CustNum value for its own query, and you get the results you expect, as shown:





There are several other relation attributes, as shown in the following syntax:

Syntax

```
[ logical-var = ] relation-handle:REPOSITION.
```

This method returns true if the relation is a REPOSITION relation, otherwise false. It can be set to change the mode of a relation.

By default, all relations in a ProDataSet are active. You can selectively disable a relation at compile time using the Data-Relation NOT-ACTIVE option. The ProDataSet also supports a logical RELATIONS-ACTIVE attribute. To deactivate all the Data-Relations in a ProDataSet, set the attribute value to false, as shown in the following syntax:

Syntax

```
dataset-handle:RELATIONS-ACTIVE = FALSE.
```

To reactivate them, set the attribute to true.

Alternatively, you can deactivate or reactivate a relation between two buffers in a ProDataSet by setting the ACTIVE attribute on the relation handle, as shown in the following syntax:

Syntax

```
relation-handle:ACTIVE = TRUE | FALSE.
```

Setting RELATIONS-ACTIVE to false for a ProDataSet is equivalent to setting ACTIVE to false, or specifying the NOT-ACTIVE option on all relations individually. Likewise, setting RELATIONS-ACTIVE to true for a ProDataSet sets the ACTIVE attribute to true for each individual relation. The most common use of RELATIONS-ACTIVE happens when you are operating in a mode where a FILL should operate on each buffer using its own individual query, without the nested filling of a parent and its children that usually occurs otherwise. This might be done for efficiency. Setting RELATIONS-ACTIVE to false is easier than traversing the ProDataSet's relations individually. Likewise, you might want to turn all relations back on after completing a FILL so that they are used to traverse the data after it has been loaded.

When a relation is inactive, there are several changes to the default behavior of the ProDataSet, depending on whether the FILL is done on the ProDataSet handle or on one of its buffer handles.

First, during a FILL on a ProDataSet **temp-table buffer**, if the AVM encounters a deactivated relation as it traverses the parent-child tree starting at that buffer, it does not fill the child of that relation and does not continue down that branch of the relation tree at all. In other words, a FILL on a ProDataSet buffer fills from that buffer down, stopping at any level that has no children and at any level where the relation to a child is deactivated.

By contrast, if the FILL is done on the ProDataSet **handle**, then every child of a deactivated relation is treated as a top-level table and filled either according to its query definition, if it has one, or with all records from its Data-Source.

Second, if a relation is inactive during navigation of a ProDataSet, the dynamic query for the child table is not prepared or opened as parent records are selected, even if there is a browse associated with the relation's query. Any access to the child temp-table must be through a query, FOR EACH, or other standard ABL construct in the application code.

No implicit behavior occurs when a relation is reactivated. There is no automatic synchronization of the hierarchy below the newly active relation. If you want to resync related buffers when you set the ACTIVE attribute to true, you can do this with the SYNCHRONIZE method on the parent buffer. Doing a partial ProDataSet FILL to return Order headers on page 185 shows examples of deactivating and activating Data-Relations.

Standard object attributes that are defined for the Data-Relation object include:

- ACTIVE
- ADM-DATA
- CHILD-BUFFER
- HANDLE
- INSTANTIATING-PROCEDURE
- MAXIMUM-LEVEL
- NAME
- NESTED
- PARENT-BUFFER
- PRIVATE-DATA
- QUERY
- RECURSIVE
- RELATION-FIELDS
- REPOSITION
- TYPE
- WHERE-STRING

Using Data-Source attributes and methods

You can identify the Data-Source currently attached to a buffer using the DATA-SOURCE attribute, as shown using this syntax:

Syntax

```
buffer-handle:DATA-SOURCE
```

The GET-DATASET-BUFFER attribute on the Data-Source handle returns the buffer handle for the Data-Source's associated ProDataSet buffer. This is the syntax for the GET-DATASET-BUFFER attribute:

Syntax

```
buffer-handle = data-source-handle:GET-DATASET-BUFFER.
```

The DATA-SOURCE-COMPLETE-MAP attribute returns a comma-separated list of field name pairs for all fields in a ProDataSet temp-table buffer that are mapped to corresponding fields in an attached Data-Source object. This list is formatted as a comma-separated list of field name pairs, qualified with the ProDataSet and Data-Source temp-table names, using the following syntax:

Syntax

```
tt-buffer-name.tt-field-name,db-table-name.db-field-name [, . . . ]
```

If the ProDataSet temp-table buffer does not have an attached Data-Source object, this attribute returns the Unknown value (?).

You can get the source buffers from a Data-Source with the NUM-SOURCE-BUFFERS attribute and the GET-SOURCE-BUFFER method, as shown in the following syntax:

Syntax

```
data-source-handle:NUM-SOURCE-BUFFERS
data-source-handle:GET-SOURCE-BUFFER [ (buffer-index) ]
```

Because a Data-Source most often has only one source buffer, the buffer-index argument is optional and defaults to 1.

You can get a comma-separated list of key fields defined in an associated KEYS clause for the specified buffer by using the KEYS attribute, as shown in the following syntax:

Syntax

```
data-source-handle:KEYS( buffer-sequence-number )
```

If there are no defined key fields, this attribute returns a comma-separated list of key fields in the buffer's unique primary index (if any). If there are no defined key fields and no unique primary index, this attribute returns the string "ROWID".

You can retrieve the ROWID of the data source row at which the next FILL operation should start with the NEXT-ROWID attribute, as shown in the following syntax:

Syntax

```
data-source-handle:NEXT-ROWID( buffer-sequence-number | buffer-name )
```

The AVM sets this attribute after each FILL operation in a series of FILL operations to retrieve data source rows in batches. You typically assign the value of this attribute to the RESTART-ROWID attribute before each FILL operation.

Note: This attribute is not marshalled between the client and the AppServer. You are responsible for retrieving, storing, and transporting this attribute value between the client and the AppServer.

You can retrieve the ROWID of the data source row at which a FILL operation will start with the RESTART-ROWID attribute. Set this attribute before each FILL operation in a series of FILL operations to retrieve data source rows in batches, as shown in the following syntax:

```
data-source-handle:RESTART-ROWID( buffer-sequence-number | buffer-name )
```

To position the FILL query to an absolute row number, use the RESTART-ROW attribute of the Data-Source object handle, as shown in the following syntax:

Syntax

```
data-source-handle:RESTART-ROW
```

Similar to RESTART-ROWID, this attribute facilitates batching during a FILL on a ProDataSet temp-table. For example, setting this attribute is helpful when you are paging back and forth in a table and want to retrieve page 3. In this case you would set RESTART-ROW to (3-1) * BATCH-SIZE to fill the third group of records.

You can retrieve the current where-clause for a buffer's query, whether it has been set explicitly or derived automatically from the fields in its relation, using the FILL-WHERE-STRING attribute, as shown in the following syntax:

Syntax

```
data-source-handle:FILL-WHERE-STRING
```

You might want to use the string to help you construct a query of your own to retrieve records that might require more filtering than the relation provides by default. The default is the Unknown value (?).

As with other objects, you can get a handle to a static Data-Source. Precede the Data-Source name with the keyword DATA-SOURCE, as shown in the following syntax:

Syntax

```
[ handle-var = ] DATA-SOURCE data-source-name:HANDLE
```

This block of code added to DynamicDataSet2.p retrieves the Data-Source for each of the ProDataSet's buffers and displays its database source buffer's name and the *where-clause* that the AVM generates automatically for the table based on the relation:

```
DO iEntry = 1 TO phDataSet:NUM-BUFFERS:
  hBuffer = phDataSet:GET-BUFFER-HANDLE(iEntry).
  MESSAGE "Data-Source: "
    hBuffer:DATA-SOURCE:GET-SOURCE-BUFFER:NAME SKIP
    "WHERE-STRING:" hBuffer:DATA-SOURCE:FILL-WHERE-STRING
    VIEW-AS ALERT-BOX.
  DELETE OBJECT hBuffer:DATA-SOURCE.
END.
```

The output for these MESSAGE statements follow:



The first database table for the ProDataSet is the Customer table. There is no default where-clause for it because it is the top-level table and has no parent. Even though you have defined a where-clause for this top-level table in the DynamicDataSet procedure, that is not assigned to the FILL-WHERE-STRING attribute. This attribute shows only the default selection the AVM generates for you:



The second Data-Source is the Order table. It **does** have a FILL-WHERE-STRING because it is the child of a relation. If you compare its FILL-WHERE-STRING attribute with the WHERE-STRING of its Data-Relation, you can see the difference between the job the Data-Source has to do during a FILL and the job the relation does after the FILL is complete and you are navigating the ProDataSet.

This FILL-WHERE-STRING retrieves Order records from the Order database table by matching the CustNum of the ttCustomer temp-table record that has just been retrieved from the database Customer table and created in the ProDataSet's temp-table.

The relation's WHERE-STRING compares the ttCustomer temp-table with the records already in the ttOrder temp-table, because it is used to navigate the filled ProDataSet:



Finally, you see the Data-Source for the SalesRep table. It also has no FILL-WHERE-STRING because it is not involved in a relation at all.

Standard object attributes that are valid for the Data-Source and accessible through its handle include:

- ADM-DATA
- FILL-WHERE-STRING
- HANDLE
- INSTANTIATING-PROCEDURE
- KEYS
- MERGE-BY-FIELD
- NAME
- NEXT-ROWID
- NEXT-SIBLING
- NUM-SOURCE-BUFFERS

- PREFER-DATASET
- PRIVATE-DATA
- QUERY
- RESTART-ROW
- RESTART-ROWID
- SAVE-WHERE-STRING
- TYPE

Enhanced query support for ProDataSet buffers

As you have seen from looking at the WHERE-STRING attribute for the Data-Relation and FILL-WHERE-STRING for the Data-Source, the AVM generates default queries for the purpose of copying database data into the ProDataSet during a FILL, and navigating and filtering that data after the FILL. Note that a query expression, such as WHERE OrderLine.OrderNum = ttOrder.OrderNum, is not usable in a QUERY-PREPARE for a query that is not part of a ProDataSet. In a standard query, you have to construct the where-clause out of multiple strings, one of which evaluates to the value of the field in the parent table at the time the QUERY-PREPARE is done. You then have to re-execute the QUERY-PREPARE and the QUERY-OPEN each time the value changes. For example, look at this stand-alone procedure:

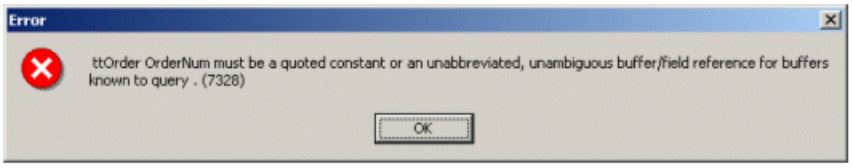
```
DEFINE VARIABLE hQuery AS HANDLE NO-UNDO.

DEFINE TEMP-TABLE ttOrder NO-UNDO LIKE Order.

CREATE ttOrder.
FIND Order WHERE Order.OrderNum = 1.
BUFFER-COPY Order TO ttOrder.

CREATE QUERY hQuery.
hQuery:ADD-BUFFER(BUFFER OrderLine:HANDLE).
hQuery:QUERY-PREPARE
  ("FOR EACH OrderLine WHERE OrderLine.OrderNum = ttOrder.OrderNum").
```

This compiles successfully, but generates the following error at run time:



This has been a limitation ever since dynamic queries were introduced into the language. The AVM cannot parse out `ttOrder.OrderNum` from the rest of the where-clause and insert the current value of the field. Instead you must code it this way:

```
DEFINE VARIABLE hQuery AS HANDLE NO-UNDO.

DEFINE TEMP-TABLE ttOrder NO-UNDO LIKE Order.

CREATE ttOrder.
FIND Order WHERE Order.OrderNum = 1.
BUFFER-COPY Order TO ttOrder.

CREATE QUERY hQuery.
hQuery:ADD-BUFFER(BUFFER OrderLine:HANDLE).
hQuery:QUERY-PREPARE
  ("FOR EACH OrderLine WHERE OrderLine.OrderNum = " +
  STRING(ttOrder.OrderNum)).
```

However, when a query is associated with a buffer that is a member of a ProDataSet, it has enhanced rules for what will compile in its dynamic predicate. As you can see from this example, the predicate of a normal query on a normal buffer is limited to an expression involving exclusively either constants or fields from the buffer's table itself. But if the buffer is part of a ProDataSet, this restriction is relaxed to allow references to any **other** member buffers in the ProDataSet. Also, if the buffer is part of a Data-Source, you might be able to do a dynamic `FIND` on it using the tables in any attached ProDataSet buffer. If the buffer is part of a query that is attached to a Data-Source, you also might be able to do a dynamic `QUERY-PREPARE` on it, referencing the Data-Source's ProDataSet target buffers.

This means, for example, that given a ProDataSet for `ttOrder` and its `ttOrderlines`, the Data-Source query for populating the ProDataSet temp-table `ttOrderline` from the `OrderLine` database table can reference the parent temp-table `ttOrder` directly. In such a case, the AVM automatically generates and manages queries that use the information in the Data-Relations to filter the current set of `ttOrderLine` records for a `ttOrder`. Internally, the AVM can take advantage of internal buffer references to insert a key value such as `ttOrder.OrderNum` into the `ttOrderline` query without having to re-prepare the `ttOrderline` query each time the selected `ttOrder` changes. This greatly increases the run time efficiency compared to what you would have to do in ABL code to re-prepare and re-open a child query each time the parent row changes.

QUERY attribute for Data-Sources, Data-Relations, and buffers

To associate a query with a dynamic Data-Source object, use the `QUERY` attribute. To disassociate the query and Data-Source object, set this attribute to the `Unknown value (?)`. You can also use the `FILL-WHERE-STRING` attribute to override the `WHERE` clause in the query.

For a Data-Relation object, the `QUERY` attribute returns the handle to the default dynamic query for a child buffer in the relation. This automatically-generated query expresses the relation between parent and child temp-tables, and lets you navigate the child records. You cannot set this handle nor can you modify the query.

For a buffer object, the `QUERY` attribute returns the handle to the query currently associated with the buffer (if any). If the buffer does not have an associated query, this attribute returns the `Unknown value (?)`. This attribute is also read-only for a buffer object.

KEYS attribute for buffers

You can also us e the KEYS attribute to return a comma-separated list of key fields for a buffer, using this syntax:

Syntax

```
buffer-handle:KEYS.
```

This attribute provides a list of the fields in the unique primary index for a temp-table, and can be useful for assigning those key values to related rows. For a temp-table buffer, the KEYS attribute evaluates to a list of fields in the temp-table's unique primary index. If it has no unique primary index, then the attribute evaluates to the string ROWID.

TOP-NAV-QUERY attribute

You can get or set the default navigation query for a top-level ProDataSet temp-table buffer using the TOP-NAV-QUERY attribute. A top-level buffer is a ProDataSet object buffer that is not a child in any active Data-Relation. There may be one or more top-level buffers in a ProDataSet object.

This syntax is used to assign a query to the top-level buffer:

Syntax

```
TOP-NAV-QUERY( index | buffer-name ) [ = query object handle ]
```

To identify a top-level buffer, you must supply an integer expression that evaluates to the 1-based buffer index, or a character expression that evaluates to the buffer name.

Session attributes for ProDataSets, Data-Sources, and queries

You can identify the first dynamic Data-Source in a session, using the FIRST-DATA-SOURCE session attribute, as shown:

```
SESSION:FIRST-DATA-SOURCE
```

You can then follow the chain of all dynamic Data-Sources using the SESSION:NEXT-SIBLING attribute.

You can identify the first dynamic ProDataSet in a session, using the new session attribute FIRST-DATASET, as shown:

```
SESSION:FIRST-DATASET
```

You can then follow the chain of all dynamic ProDataSets, using the SESSION:NEXT-SIBLING chain.

Although it is somewhat independent of the ProDataSet support, there is also a new session attribute FIRST-QUERY to identify the first dynamic query in a session. For example:

```
SESSION:FIRST-QUERY
```

You can again follow the chain of all dynamic queries in a session using the SESSION:NEXT-SIBLING attribute. This chain will include all auto-generated queries, such as those for Data-Relations, as well as ones you have created in your procedures.

Other ProDataSet methods

You are already familiar with the most important ProDataSet method, which is FILL. There are two other useful ProDataSet methods.

The CLEAR method destroys the entire definition of a dynamic ProDataSet, returning it to the state it was in just after the CREATE statement for it. You can then reuse the ProDataSet handle to build another dynamic ProDataSet. This method cannot be used on a static ProDataSet.

By contrast, the EMPTY-DATASET method empties all records from all the ProDataSet's tables, but does not destroy the ProDataSet definition. You can use this method on a static ProDataSet's handle or on a dynamic ProDataSet. This is similar to the EMPTY-TEMP-TABLE method for a temp-table. It is much more efficient than looping through the records and deleting them one at a time. Unless there is an active transaction, the records in each temp-table of the ProDataSet are eliminated in a single operation. If there is an active transaction, then the slower record-by-record operation is done. Normally, you should empty a ProDataSet only when there is no active transaction.

Building a dynamic user interface from a ProDataSet

In most cases, you expect that server-side ProDataSet definitions will be static in most procedures of your application. This is because each ProDataSet is likely to have a unique combination of tables and relations. Also, it is much easier to write business logic using static ABL statements that can reference table and field names directly rather than using indirect references such as BUFFER-FIELD(x):BUFFER-VALUE. Because business logic tends to be unique to the tables involved in most cases, a general purpose dynamic procedure that can handle many different ProDataSets is less likely to be useful on the server-side of an application, where you are loading ProDataSets from your database, doing validation, and saving back updates. There will, of course, be exceptions to this. The ability to mix and match DATASET and DATASET-HANDLE parameters makes it easy to pass a ProDataSet to another procedure, either locally or remotely, without being concerned about whether the other procedure wants to match the parameter up to a static ProDataSet definition or inspect it in a more general way through its handle.

In some cases, it is more likely that a user interface procedure on the client side of the application wants to accept a variety of different ProDataSets and display their data in a consistent way, by inspecting the ProDataSet structure and creating dynamic user interface objects such as fields and browses appropriate to the ProDataSet.

This section shows you an example of how you can build these kinds of procedures. You will run the same DynamicDataSet2.p procedure you built earlier, but from an AppBuilder-built window procedure that creates a series of dynamic browses for the tables in the ProDataSet.

Attaching a browse to a ProDataSet temp-table is very easy. If the table the browse displays is the child of a Data-Relation, then you can simply associate the browse with the dynamic query the ProDataSet generates for the relation. In this way the browse automatically shows the right records for the parent of the relation, because the dynamic query is filtered based on the parent key values and automatically reopens itself internally each time the record in the parent buffer changes. This can happen either programmatically by using a `GET-NEXT()` method or similar statement on the query, or by selecting a record in a browse for the parent table.

To create the new window procedure:

1. Create a new **Window** procedure in the AppBuilder.
2. Name its default window **BrowseWin** and its default frame **BrowseFrame**.
3. Make the window and frame 16 rows by 120 columns.
4. In the procedure's **Main Block**, add a statement to run an internal procedure called `showDataSet`, as shown:

```
MAIN-BLOCK:
DO ON ERROR   UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
   ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
  RUN enable_UI.
  RUN showDataSet.  IF NOT THIS-PROCEDURE:PERSISTENT THEN
    WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.
```

5. Define the internal procedure `showDataSet`. The procedure gets a dynamic ProDataSet back from `DynamicDataSet2.p`, and it creates three browses, each of which has a query and a buffer handle, so it needs these variables:

```
DEFINE VARIABLE hBuffer1 AS HANDLE NO-UNDO.
DEFINE VARIABLE hBuffer2 AS HANDLE NO-UNDO.
DEFINE VARIABLE hBuffer3 AS HANDLE NO-UNDO.
DEFINE VARIABLE hBrowse1 AS HANDLE NO-UNDO.
DEFINE VARIABLE hBrowse2 AS HANDLE NO-UNDO.
DEFINE VARIABLE hBrowse3 AS HANDLE NO-UNDO.
DEFINE VARIABLE hDataSet AS HANDLE NO-UNDO.
DEFINE VARIABLE hQuery1  AS HANDLE NO-UNDO.
DEFINE VARIABLE hQuery2  AS HANDLE NO-UNDO.
DEFINE VARIABLE hQuery3  AS HANDLE NO-UNDO.
```

   Note that this example is somewhat simplified in that it is designed to accept a ProDataSet of exactly the kind that `DynamicDataSet2.p` creates, with three tables and a single Data-Relation between the first two of these tables. By extending the procedure somewhat, you can make it work properly for virtually any combination of tables and relations that might come from another procedure. You can also make a decision as to which data to display with dynamic fill-ins or other single field objects, and which to display in browses.

6. Add the same `RUN` statement you used in `DynamicMain2.p`, as shown:

```
RUN DynamicDataSet2.p
  (INPUT "Customer,Order,SalesRep",
   INPUT "CustNum,OrderNum,SalesRep",
   INPUT "CustNum,CustNum",
   INPUT "= 1",
   OUTPUT DATASET-HANDLE hDataSet).
```

The top-level buffer does not have a query created for it automatically because these Data-Relation queries are only for the child table of a relation, where the filtering occurs. So, retrieve the buffer handle for the ProDataSet's first buffer and create a dynamic query for it.

7. Prepare the query for each record in the top-level temp-table and open it:

```
hBuffer1 = hDataSet:GET-BUFFER-HANDLE(1).
CREATE QUERY hQuery1.
hQuery1:ADD-BUFFER(hBuffer1).
hQuery1:QUERY-PREPARE("FOR EACH " + hBuffer1:NAME).
hQuery1:QUERY-OPEN().
```

8. Create a dynamic browse for this query:

```
CREATE BROWSE hBrowse1 ASSIGN
  QUERY       = hQuery1
  FRAME       = FRAME BrowseFrame:HANDLE
  HIDDEN      = FALSE
  NO-VALIDATE = TRUE
  WIDTH       = 120
  HEIGHT      = 5
  SEPARATORS  = TRUE
  SENSITIVE   = TRUE.
```

The `QUERY` attribute connects the browse to the query on the top-level table.

The `FRAME` attribute parents it to the frame in your window.

The `HIDDEN` attribute makes sure it is displayed when the window is displayed.

The `NO-VALIDATE` attribute disables any field-level validation expressions that are built into the schema definitions for the database tables and inherited, by default, by the ProDataSet's temp-tables. For example, if there is a field validation expression for the `ttOrder` table that tries to check that the `CustNum` is in the `Customer` table, this might not work because the ProDataSet, and potentially your window as well, are not connected to the database.

The `HEIGHT` and `WIDTH` attributes set the browse's size.

The `SEPARATORS` attribute provides vertical and horizontal lines between columns and rows to improve readability.

The `SENSITIVE` attribute allows you to scroll the browse, even though its columns are not enabled.

9. Complete the browse definition by adding all the columns from the top-level table's buffer to it. For example:

```
hBrowse1:ADD-COLUMNS-FROM(hBuffer1:NAME).
```

10. You need to identify the query for the second table in the ProDataSet (the `ttOrder` table in this example). Because this is the child buffer of a Data-Relation, there is a query already available for you to use. You will recall that this query has a `WHERE-STRING` that filters `ttOrder` records for the currently selected `ttCustomer`, as shown:



The AVM inserts the correct value for `ttCustomer.CustNum` each time the `ttCustomer` record in the buffer changes.

To use this query, simply retrieve its handle from the Data-Relation and open it:

```
hQuery2 = hDataSet:GET-RELATION(1):QUERY.
hQuery2:QUERY-OPEN().
```

11. Create a second dynamic browse to display its records, positioning it below the first one:

```
CREATE BROWSE hBrowse2 ASSIGN
  QUERY       = hQuery2
  FRAME       = FRAME BrowseFrame:HANDLE
  HIDDEN      = FALSE
  NO-VALIDATE = TRUE
  ROW         = 6
  WIDTH       = 120
  HEIGHT      = 5
  SEPARATORS  = TRUE
  SENSITIVE   = TRUE.
```

12. Get the temp-table's buffer handle and add its columns to the second browse:

```
hBuffer2 = hDataSet:GET-RELATION(1):CHILD-BUFFER.
hBrowse2:ADD-COLUMNS-FROM(hBuffer2:NAME).
```

You could use either the expression `hDataSet:GET-RELATION(1):CHILD-BUFFER` to identify the right buffer, or `hDataSet:GET-BUFFER-HANDLE(2)`. That is to say, the buffer you want is the child of the first (and only) relation, and also the second buffer overall in the ProDataSet.

13. Create a query for the third table. Because this is not in a relation, there is no Data-Relation query for it, so you need to create and prepare it yourself:

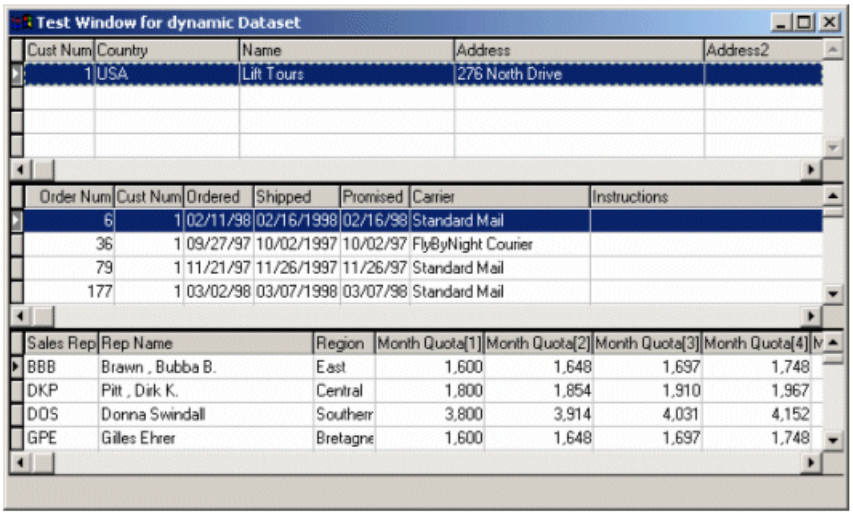
```
hBuffer3 = hDataSet:GET-BUFFER-HANDLE(3).
CREATE QUERY hQuery3.
hQuery3:ADD-BUFFER(hBuffer3).
hQuery3:QUERY-PREPARE("FOR EACH " + hBuffer3:NAME).
hQuery3:QUERY-OPEN().
```

14. Create a third browse to display the third buffer's records, positioning it below the second one, and add the buffer's columns to the browse:

```
CREATE BROWSE hBrowse3 ASSIGN
  QUERY        = hQuery3
  FRAME        = FRAME BrowseFrame:HANDLE
  HIDDEN       = FALSE
  NO-VALIDATE  = TRUE
  ROW          = 11
  WIDTH        = 120
  HEIGHT       = 5
  SEPARATORS   = TRUE
  SENSITIVE    = TRUE.

hBrowse3:ADD-COLUMNS-FROM(hBuffer3:NAME).
```

15. Save this procedure as `DynDataSetWin.w` and run it:

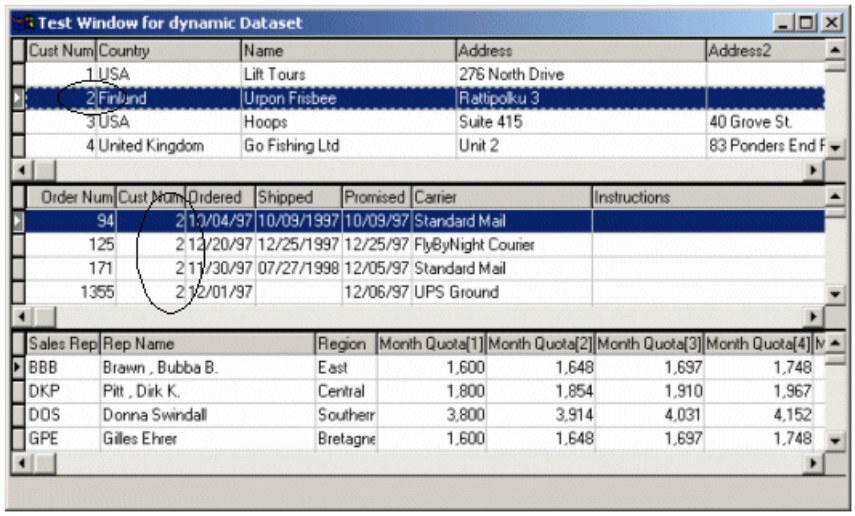


Voilà! The top-level browse confirms that you selected just one `Customer`. The second one displays the `Orders` for that `Customer`. Because displaying the `Customer` browse automatically positions to that one record in the `ttCustomer` buffer, the Data-Relation's query for the `ttOrder` query finds the value it needs there. The third browse shows all `SalesReps`, unrelated to the other tables.

16. Just to confirm what the Data-Relation query does for you, and how this is especially useful when you attach it to a browse, modify the `RUN` statement to request all `Customers` less than 10, along with their `Orders`. For example:

```
RUN DynamicDataSet2.p
  (INPUT "Customer,Order,SalesRep",
   INPUT "CustNum,OrderNum,SalesRep",
   INPUT "CustNum,CustNum",
   INPUT "< 10",
   OUTPUT DATASET-HANDLE hDataSet).
```

17. Run the procedure again:

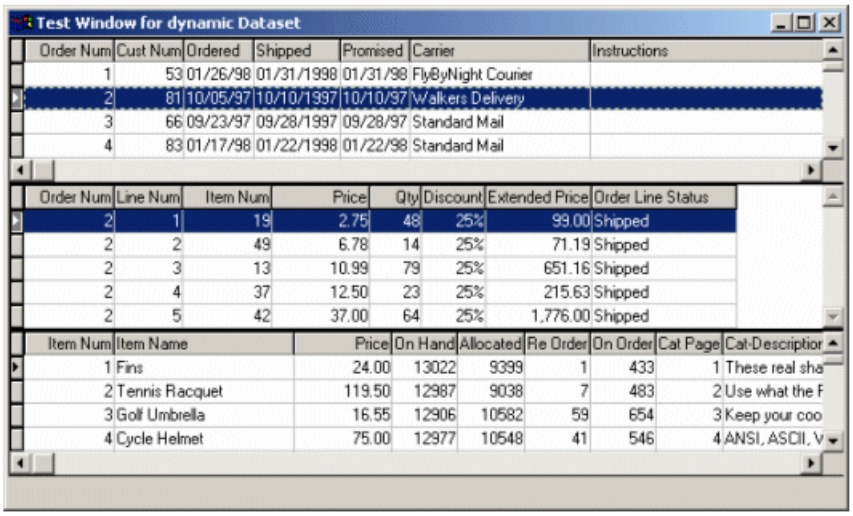


As you select different `Customers`, the `Order` browse reopens automatically with its `Orders`. Remember that the `ttOrder` table in the ProDataSet actually contains all the `Orders` of all the selected `Customers`, but its default query filters them for you when this is useful, as it is when you use a browse to display them.

18. Just to prove that the dynamic nature of the procedure really works, change the parameters of the `RUN` statement to use a different set of tables, as shown:

```
RUN DynamicDataSet2.p
  (INPUT "Order,OrderLine,Item",
   INPUT "OrderNum,LineNum,ItemNum",
   INPUT "OrderNum,OrderNum",
   INPUT "< 10",
   OUTPUT DATASET-HANDLE hDataSet).
```

There is one small problem here. Our simplified example expects a single key field for each table, and the key for `OrderLine` is, in fact, `OrderNum` and `LineNum` together. However, this does not affect the example in this case:



So what you have created here is a factory for a particular class of ProDataSets (the procedure `DynamicDataSet2.p`) and a general-purpose display mechanism for any ProDataSet of that type (the window procedure `DynDataSetWin.w`). This is a simple example of a very powerful capability.

Feel free to extend the window to prompt the user for the table and field names the parameters need, to make the example more fully dynamic.

Using the SYNCHRONIZE method

The Data-Relation queries in a parent-child hierarchy are synchronized automatically only when there is a browse object attached to the query. This attachment is done by assigning the relation's QUERY attribute to the browse's QUERY attribute. If you want to synchronize a hierarchy of queries under other circumstances, you use the SYNCHRONIZE method on any parent buffer handle:

```
buffer-handle:SYNCHRONIZE().
```

This causes the ProDataSet to traverse the ProDataSet hierarchy starting at buffer *buffer-handle*. It reopens each relation query for the current parent at each level, just as it happens automatically when you select a record in a browse or perform some ABL action that changes the record position in a parent buffer whose children are attached to browses. The synchronize behavior is not provided automatically in all cases because there are simply too many different ways in which the position could be changed and too many different responses that a developer might want. Always reopening all related queries is not appropriate, because of the expense involved.

You can decide when to synchronize by reacting to an event such as ON VALUE-CHANGED, or simply in conjunction with a language statement or method such as GET-NEXT, and explicitly doing the synchronize when necessary. Note that this synchronization affects **only** the implicit dynamic queries associated with Data-Relations when you are navigating a ProDataSet that has already been filled. It has nothing to do with the FILL operation itself, and using these queries is entirely optional. In many cases (perhaps even in most cases), your own ABL logic will instead use conventional nested FOR EACH blocks or queries to navigate through the levels of a ProDataSet, without using or caring about these implicit queries at all. This is part of why the overhead of opening them does not happen unless the relation queries have explicitly been attached to a browse.

As part of the SYNCHRONIZE() method, the AVM automatically positions to the first row in each relation query, in addition to reopening it on children of the current parent row. This is not done if the query is being browsed (with an ABL client browse widget) because the browse effectively forces a GET-FIRST already.

Doing this automatically spares the developer from having to write a GET-FIRST method on each child query. A typical block of code to navigate through the children of the current parent looks like this:

```
hChildQuery:GET-FIRST().
DO WHILE NOT hChildQuery:QUERY-OFF-END:
  hChildQuery:GET-BUFFER-HANDLE(1):SYNCRHONIZE().
  /* If there are further children */
  /* Processing code for the current row goes here. */
  hChildQuery:GET-NEXT().
 END.
```

If you forget the GET-FIRST, the loop exits immediately because the query is "OFF-END" if there is no row at all in its buffer.

In addition, this is the only way to propagate a SYNCHRONIZE() through multiple parent-child levels. Consider the example of a three-level ProDataSet with tables ttCustomer, ttOrder, and ttOrderLine. The application does a SYNCHRONIZE() on the ttCustomer table when the user selects a different row in that table. Without doing an implicit GET-FIRST on the ttOrder table to position it to the first ttOrder for the newly selected ttCustomer, the relation query for the ttOrderLine table will not be properly reopened and filtered for OrderLines of the Customer's first Order, because there would be no current row in the ttOrder table.

Independent of all this, you can freely define queries of your own to navigate the tables in the ProDataSet after it has been filled, or even as part of custom code to populate one or more tables of a ProDataSet independent of its FILL method.

Sample procedure: adding REPOSITION and SYNCHRONIZE

In Introducing the OpenEdge DataSet on page 19 you learned about the REPOSITION mode for a Data-Relation. You can include this in a DATASET definition as a keyword at the end of the relation definition, or you can set it at run time as an attribute (true or false) on the relation.

When you fill a ProDataSet, REPOSITION mode on a relation causes the AVM to treat the relation as deactivated and to populate the child table with all records from its Data-Source buffer, or with all those you specify in your own Data-Source query.

When you are navigating, REPOSITION mode causes the AVM to reposition its default query on the relation to the correct record, rather than reopening the query to select only that record.

In this section, you will extend the dsOrderWin.w example to show the effect of the REPOSITION mode on a relation. You will also use the SYNCHRONIZE method to readjust the queries the browses for OrderLines and Items use.

First, open dsOrder.i and add the keyword REPOSITION at the end of the LineItem relation definition, as shown:

```
DEFINE DATASET dsOrder FOR ttOrder, ttOline, ttItem
  DATA-RELATION OrderLine FOR ttOrder, ttOline
    RELATION-FIELDS (OrderNum, OrderNum)
  DATA-RELATION LineItem FOR ttOline, ttItem
    RELATION-FIELDS (ItemNum, ItemNum) REPOSITION.
```

Next, get the window procedure to use the relation queries rather than the static queries the AppBuilder generates. When you created dsOrderWin.w, you created two static browses to show the contents of ttOline and ttItem. The AppBuilder generated queries for the temp-tables for you when you did this, and associated them with preprocessor values, as shown:

```
&SCOPED-DEFINE OPEN-QUERY-ItemBrowse OPEN QUERY ItemBrowse FOR EACH ttItem
  NO-LOCK INDEXED-REPOSITION.
&SCOPED-DEFINE OPEN-QUERY-OlineBrowse OPEN QUERY OlineBrowse FOR EACH ttOline
  NO-LOCK INDEXED-REPOSITION.
…
&SCOPED-DEFINE OPEN-BROWSERS-IN-QUERY-dsFrame ~
  ~{&OPEN-QUERY-ItemBrowse}~
  ~{&OPEN-QUERY-OlineBrowse}
```

You then used the OPEN-BROWSERS preprocessor in the LEAVE trigger for the Order Number field:

```
{&OPEN-BROWSERS-IN-QUERY-{&FRAME-NAME}}
```

However, now you want to use the dynamic queries the ProDataSet provides for you so that you do not have to bother using those the AppBuilder defines.

In the Main Block of dsOrderWin.w, add these lines to associate the two browse objects with the ProDataSet relation queries:

```
MAIN-BLOCK:
DO ON ERROR   UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
   ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
  RUN enable_UI.
  /* Replace the default AppBuilder-generated static queries with
     the ones that are part of the Data-Relations. */
  OlineBrowse:QUERY = DATASET dsOrder:GET-RELATION("OrderLine"):QUERY.
  ItemBrowse:QUERY  = DATASET dsOrder:GET-RELATION("LineItem"):QUERY.
  ItemBrowse:SET-REPOSITIONED-ROW(4, "CONDITIONAL").

  IF NOT THIS-PROCEDURE:PERSISTENT THEN
    WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.
```

The OrderLine relation query is filtered automatically for ttOlines of the currently selected ttOrder. In the case of this sample window, there is only one ttOrder in the ProDataSet at a time. As you have seen from other examples, if there is more than one parent, the relation query filters for the current parent.

Because of the REPOSITION qualifier on the relation, the LineItem relation query does not select just the ttItem for the current ttOrder. Instead, this qualifier tells the AVM to leave the ttItem query open for all ttItem records, and to reposition it to the correct ttItem for the current ttOline.

The SET-REPOSITIONED-ROW method tells the AVM to make the selected row the fourth row in the viewport, so it is in the middle, unless it is already being displayed (that is the CONDITIONAL part).

In the LEAVE trigger for iOrderNum, you need to add statements that close the relation queries that the ttOline and ttItem browses are now using. If you do not do this, you might see anomalous behavior when they are reopened for the ProDataSet on another Order, as shown:

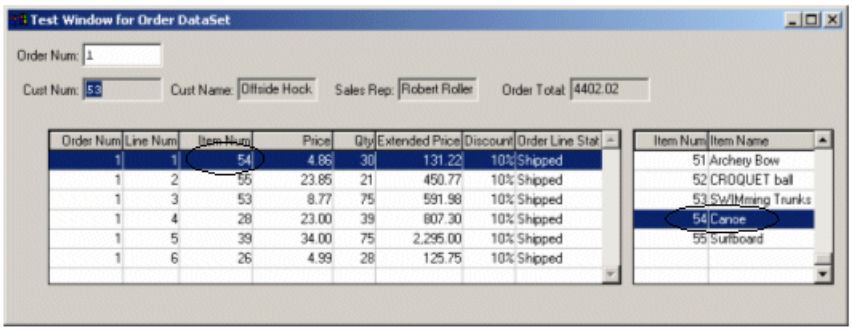
```
IF iOrderNum NE 0 THEN DO:
  DATASET dsOrder:GET-RELATION(1):QUERY:QUERY-CLOSE().
  DATASET dsOrder:GET-RELATION(2):QUERY:QUERY-CLOSE().
   . . .
```

Remove or comment out the `OPEN-BROWSERS…` preprocessor. In addition, add a line to run the `SYNCHRONIZE` method on the top-level buffer, for `ttOrder`, as shown:

```
DO WITH FRAME dsFrame:
  ASSIGN
    iCustNum:SCREEN-VALUE  = STRING(ttOrder.CustNum)
    cCustName:SCREEN-VALUE = ttOrder.CustName
    cRepName:SCREEN-VALUE  = ttOrder.RepName
    dOrderTotal:SCREEN-VALUE = STRING(ttOrder.OrderTotal).
    /* {&OPEN-BROWSERS-IN-QUERY-{&FRAME-NAME}}    */
    DATASET dsOrder:GET-BUFFER-HANDLE(1):SYNCHRONIZE().
END.
```

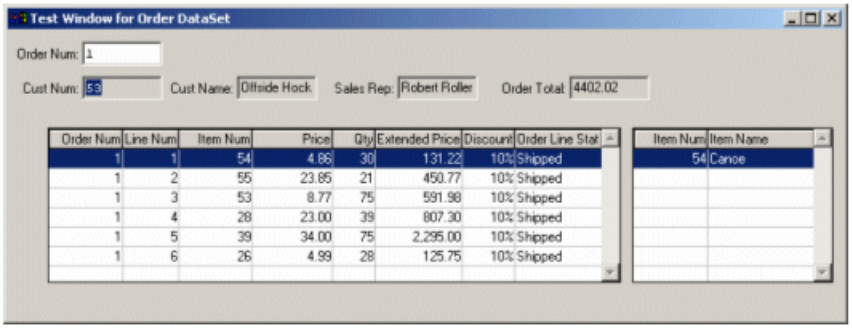
Since the two browses have been connected to the relation queries, it is not necessary to use the static ones anymore. But because there is no browse at the top level, for the `ttOrder` fields, you must nudge the AVM and direct it to go through the steps to reopen or reposition the related queries for the current `ttOrder` record. This is what `SYNCHRONIZE` does.

When you rerun the window procedure, you can see the effect of the `REPOSITION` relation:



You can scroll up and down in the `ItemBrowse` to see that the AVM has retrieved all `Items` into the temp-table. That is what `REPOSITION` does at `FILL` time. As you can see, the relation query the browse is using is automatically positioned to the correct `ttItem` each time you select a `ttOline` record. That is the part that `REPOSITION` does when you are navigating a filled ProDataSet. If you want one behavior without the other, you can turn the `REPOSITION` attribute on and off at run time.

Just to reinforce what is happening here, try removing the `REPOSITION` keyword from `dsOrder.i` again and rerunning the window:



At fill time, the AVM is loading only `Items` that are in one or more of your `OrderLines` into `ttItem`. When you view the data, it is filtering `ttItem` to show only the one `ttItem` for the current `ttOline` in the browse. So there is really nothing to browse in this case.

Summary

At this point, you have been introduced to all the important code features of the ProDataSet. The following chapters will focus on common use cases for updating, reading, and writing ProDataSet data.

6

Updating Data with ProDataSets

Once you have filled a ProDataSet with data and passed it to a client for display, you will likely have to process changes to the data. The ProDataSet supports a number of features to help you capture changes made on the client, pass them back to the server, and then apply those changes to the Data-Source tables. Of course, you can also make changes within a single session, even within a single procedure, and write them back to the database without passing the ProDataSet from session to session. The ProDataSet update features map to similar support for capturing updates in Microsoft's ADO .NET environment, so that changes made to a .NET DataSet, for example, in a Visual Basic or C# application, can be passed directly to the AVM, and the record of all the changes is reproduced on the OpenEdge side with no loss of information. The same process works in the other direction as well.

For details, see the following topics:

- Tracking changes in the temp-tables of a ProDataSet
- Processing changes
- Setting and using ERROR, ERROR-STRING, and REJECTED
- ProDataSet change events

Tracking changes in the temp-tables of a ProDataSet

A ProDataSet can be in one of three states:

- **Fill mode** — The ProDataSet is being filled with data. Normally you do this using the `FILL` method on the ProDataSet or on one or more of its buffers. You can create records in its temp-tables using standard ABL statements as well, and even add a temp-table that already has data in it to a ProDataSet. In any case, the AVM does not keep track of any of these additions or changes to the ProDataSet while you are filling it.

- **Navigation mode** — The ProDataSet has been filled and you are navigating the data. This might be in a user interface, such as the one you built in ProDataSet Attributes and Methods on page 105 or it could be in a business logic procedure that walks through the data to examine or process it in some way.
- **Change mode** — You are making changes to the ProDataSet data that you want to save back to the database or other data source. It is important that the AVM keep track of these changes because they can be made in a different procedure or even a different session from the one that writes them back to the database.

The fill and navigation modes are not strictly differentiated. You can fill a ProDataSet with data, navigate and display that data, and add more data at will.

However, the change-tracking mode must be clearly differentiated from the others because the AVM needs to know whether a statement needs to be tracked for later update to the database or is just part of an ongoing fill process. This applies to a statement that adds records to a ProDataSet temp-table or a statement that modifies existing records.

For this reason, there is a `TRACKING-CHANGES` logical attribute for temp-tables that are part of a ProDataSet. `TRACKING-CHANGES` tells the AVM when to track changes to the data in the temp-table so that the changes can later be properly made to the database tables that are the Data-Source for the temp-table.

`TRACKING-CHANGES` is initially false for any temp-table. This means that the temp-table is implicitly in `FILL` mode. Any changes made to the data in the temp-table while `TRACKING-CHANGES` is false are considered part of the process of filling the temp-table with data for the ProDataSet. This can be done by means of a `FILL` method on the table if it is part of a ProDataSet, on the ProDataSet itself, or simply by adding, changing, or deleting records in the temp-table using standard ABL syntax. In addition, any data already in the temp-table when it becomes part of the ProDataSet is considered part of the fill process. This allows the procedures managing the ProDataSet to populate it in any way they like without the AVM keeping track of changes to the ProDataSet's temp-tables.

You can set `TRACKING-CHANGES` independently for each temp-table in a ProDataSet for which changes are to be tracked.

You set `TRACKING-CHANGES` to true when you want the AVM to start tracking changes to the data in a temp-table or an entire ProDataSet. This means that any changes made from that point onward are intended to be written back to the Data-Source for the table, if there is one, or used by application code to process changes in some other way if there is no Data-Source. This means that although `TRACKING-CHANGES` is an attribute (not a method) that you can set and query, it has more serious side-effects than setting most attributes. This especially refers to the creation of a special companion table for each affected temp-table for keeping track of before images of changed rows, if one does not already exist.

Changes are tracked only for those temp-tables for which `TRACKING-CHANGES` is true. Any `FILL` method that is invoked on a temp-table whose `TRACKING-CHANGES` attribute is true, or on a parent temp-table or ProDataSet of that temp-table that cascades the fill down to that temp-table, will cause an error. You cannot mix use of the `FILL` method and change tracking.

You normally set `TRACKING-CHANGES` back to false for a temp-table after you have executed the `ACCEPT-CHANGES` method, which marks the changed versions of rows as the new current versions. Once you set `TRACKING-CHANGES` back to false, you can once again run a `FILL` method, or otherwise add records to the ProDataSet's temp-tables. Those new or modified records will not be recorded as changes until you set `TRACKING-CHANGES` back to true.

Within this chapter, we refer to the temp-tables that hold the current versions of records that are seen by the user interface or by logic that walks through the ProDataSet as the after-tables. This is because when changes are made they are applied to these tables so as to be immediately visible. Thus, they are the current or "after-change" version of the records.

Changes to each after-table are tracked using a companion temp-table referred to as the before-table, which is created (if it does not already exist) and managed automatically by the AVM when you set TRACKING-CHANGES to true. This table is used to hold before-images of modified records, deleted records, and a placeholder for each newly created record. The latest version of each record is always held in the after-table, so that it properly reflects all changes, creates, and deletes. Any attempt to directly modify the records in the before-table results in a run-time error.

ROW-STATE attribute

There is a ROW-STATE attribute on the temp-table buffer in both the after-table and the before-table to allow you to determine how, if at all, each record has been changed. ROW-STATE is an integer value representing one of these constants, which you can and should use to identify the meaning of the ROW-STATE:

- 0=ROW-UNMODIFIED
- 1=ROW-DELETED
- 2=ROW-MODIFIED
- 3=ROW-CREATED

Note that these are un-quoted literals that correspond to integer values, much like the values NO-LOCK, SHARE-LOCK, and EXCLUSIVE-LOCK for LOCK-MODE. You use the literals in ABL logic, in statements such as this:

```
IF hTTCust:ROW-STATE = ROW-MODIFIED THEN . . .
```

If you display the ROW-STATE, it is displayed as an integer.

Each record in the after-table that has been modified or added has an internal pointer to its counterpart in the before-table. These after-table records have a ROW-STATE value of ROW-MODIFIED or ROW-CREATED, depending on whether the row has been added to the temp-table since TRACKING-CHANGES was set to true. Deleted records do not appear in the after-table, because it reflects the current state of the data. Records in the after-table that have not been added or changed have a ROW-STATE of ROW-UNMODIFIED. These records have no counterpart in the before-table.

Every record in the before-table has a nonzero ROW-STATE because every record is the before-image for a deleted, created, or modified record. Records in the before-table can have a ROW-STATE equal to one of these values:

- **ROW-DELETED** — A deleted row in the before-table holds the original field values of the record at the time TRACKING-CHANGES was set to true. It has no counterpart in the after-table. A record is moved from the after-table to the before-table when it is deleted. Thus, the only way to locate deleted records is in the before-table. If a record is modified and then later deleted before changes are processed, then its state is ROW-DELETED and there is no record of the changes made before that.
- **ROW-MODIFIED** — A modified row in the before-table holds the field values of the record at the time that TRACKING-CHANGES was last set to true. The row is copied from the after-table when it is first changed. No intermediate changes between the "original" values and the latest values are saved anywhere.
- **ROW-CREATED** — A newly created row is created in the before-table at the same time that the equivalent record is created in the after-table. The before-table row for a created row does not hold any meaningful field values. It serves merely as a placeholder for the row so that you can identify created rows along with modified and deleted ones by looking at the before-table. The field values in the after-table can then be changed as the record is edited, just as for changed rows. The ROW-STATE remains ROW-CREATED until

changes to the table have been processed, regardless of how many times field values have been modified in the meantime.

Records in the after-table can have a ROW-STATE of ROW-UNMODIFIED, ROW-CREATED, or ROW-MODIFIED. This means that new records and changed records can be tracked through either table. However, because deleted records are removed from the after-table, they can be tracked only through the before-table. In general, you should process changes through the before-table for this reason.

Note: Editing a record by replacing its value with the exact same value changes the ROW-STATE attribute changes to ROW-MODIFIED, in spite of the fact that the record's value remains the same. As a result, GET-CHANGES can return records that don't appear to have been changed.

ROW-STATE function

It is not possible to check the value of an attribute such as ROW-STATE in a where-clause. For this reason, there is also a ROW-STATE function that can be referenced in a where-clause to let you filter ProDataSet temp-table rows by their ROW-STATE. The ROW-STATE function takes a buffer name as its argument and returns the same integer value as the ROW-STATE attribute. Ordinarily you should apply the function to a before-table buffer, but you can also use the after-table buffer name as well with the same effect. If there is no record in the buffer passed, or it is not a before-table, or is not an after-table that has a before-table associated with it, then the ROW-STATE value returned is "?". An error is not raised.

The function lets you construct a *where-clause*, such as in the following example:

```
FOR EACH <before-table-buf>
  WHERE ROW-STATE(<before-table-buf>) = ROW-MODIFIED:
  ...
END.
```

Special restrictions on TRACKING-CHANGES

These are some general restrictions on the use of TRACKING-CHANGES that mostly exist as sanity checks to prevent people from doing unexpected things and expecting something useful to happen:

- If you try to do a FILL or an EMPTY-DATASET while TRACKING-CHANGES is true, the AVM raises an error.
- If you pass a ProDataSet remotely or locally by value and TRACKING-CHANGES is true for the ProDataSet or any of its tables, the before- and after-tables are passed in their current state, but the value of the TRACKING-CHANGES attribute is not passed. That is, it is not automatically set to true on the receiving side.

Normally, if you set TRACKING-CHANGES to true, make changes, and then pass the ProDataSet as a parameter, the receiver processes the changes without making further changes that should be recorded in the before-table. If the receiving procedure wants to make further changes, it must set TRACKING-CHANGES to true after receiving the ProDataSet parameter.

Creating or defining the before-tables

For a dynamic temp-table, a before-table is created only when the temp-table's TRACKING-CHANGES attribute (or the corresponding ProDataSet attribute) is first set to true. After that the before-table remains in existence for the life of the after-table, even as TRACKING-CHANGES is set to false and back to true.

When you pass a ProDataSet parameter, if the receiving procedure receives the ProDataSet as a dynamic object (as a DATASET-HANDLE), then the AVM automatically creates a before-table for each modified temp-table if one does not already exist.

In the case of a static temp-table that might be used to track changes as part of a ProDataSet, you must provide a static name for its before-table by naming it in the DEFINE TEMP-TABLE statement, as shown in the following syntax:

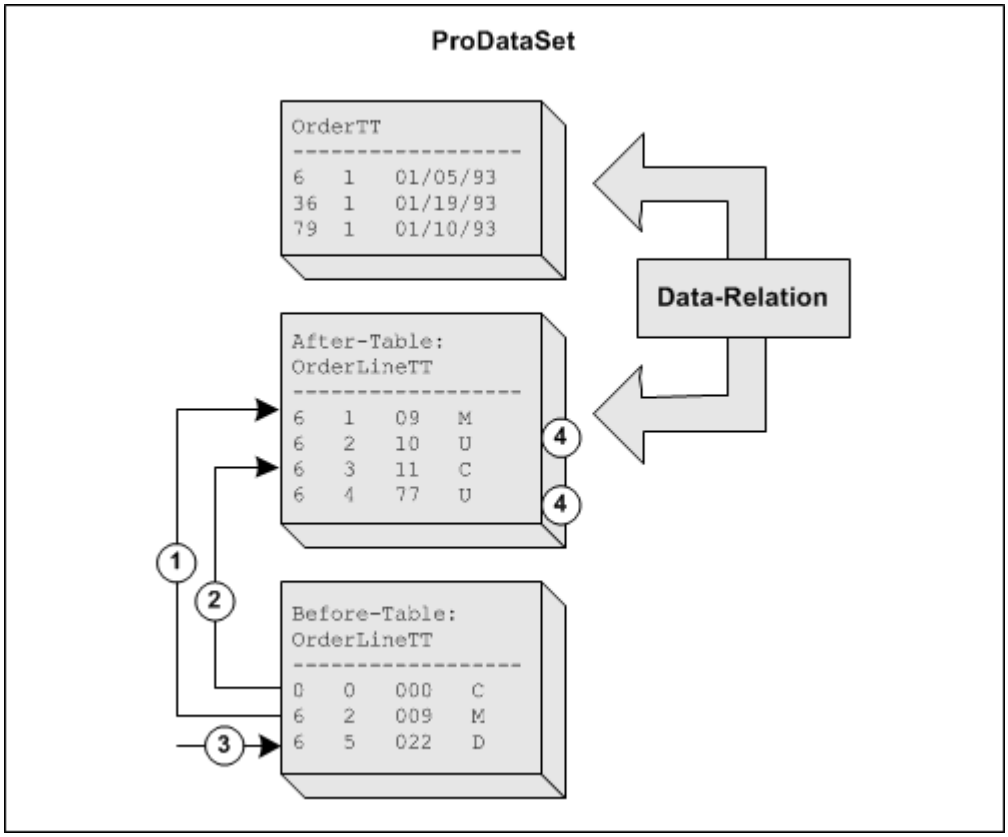
Syntax

```
DEFINE TEMP-TABLE temp-table-name [ BEFORE-TABLE before-table-name ] …
```

The AVM cannot create a before-table for you at run time as it does for dynamic tables. Procedure code can refer to the before-table by name as it does other temp-tables, except that any attempt to directly modify its records results in a run-time error. For a static definition, the before-table is instantiated along with the after-table, and you can freely refer to it in procedural code.

When you pass a ProDataSet parameter, if the receiving procedure receives the ProDataSet as a static object, then its static definition must include a static name for the before-table. Otherwise, the AVM does not create a before-table and raises an error. The following figure shows how the before-table and after-table are related.

Figure 11: Relationship of before- and after-tables



Actually, the before-table has a non-unique index on the ROW-STATE, so that changes are in order with delete operations first, then modify operations, and then create operations.

Both the before-table and after-table are part of the ProDataSet. They are passed together as part of the ProDataSet parameter and have the same scope and lifetime, with the exception that a before-table for a dynamic temp-table is created only when first needed.

The arrows illustrate the four states and their relationships:

1. When `OrderLine 09` is modified, the before image is copied to a new record in the before-table. The `ROW-STATE` for both records is `ROW-MODIFIED`, represented by the `M` in the illustration. The `ROW-STATE` is not actually stored as a user-accessible temp-table column. It is accessible only through the `ROW-STATE` attribute.
2. When row `11` is created, a placeholder for it is created in the before-table. It serves only to log the creation and point to the record in the after-table.
3. When row `22` is deleted, it is removed from the after-table and moved to the before-table.
4. Rows `10` and `77` have a `ROW-STATE` in the after-table of `ROW-UNMODIFIED`, represented by the `U`. There is no corresponding row in the before-table.

Locating rows in the before- and after-tables

There is a pair of complementary attributes for the before- and after-table buffers that point to the record in one table that corresponds to the record in the other table:

- *`after-buffer-hdl`* `:BEFORE-ROWID` returns the `ROWID` of the record in the before-table that is the before-image for the record in the after-table held in the buffer handle. This attribute is set to the `Unknown value (?)` for records that have not been changed.
- *`before-buffer-hdl`* `:AFTER-ROWID` returns the `ROWID` of the record in the after-table that is the current version of the added or changed record in the before-table held in the buffer handle. Deleted records have an `AFTER-ROWID` of the `Unknown value (?)`.

These attributes allow you, for example, to execute a static `FIND` statement with a *`where-clause`* such as `WHERE ROWID(ttBeforeTable) = ttAfterBuffer:BEFORE-ROWID`, or a dynamic `FIND-BY-ROWID` method, to retrieve the buffer handle for the corresponding record in the other table, to do comparisons of field values.

The `ROW-STATE`, `AFTER-ROWID`, and `BEFORE-ROWID` attributes are all read-only.

There are also attributes to point back and forth between the temp-tables themselves:

- *`after-table-handle`* `:BEFORE-TABLE` returns the handle of the before-table for this after-table.
- *`before-table-handle`* `:AFTER-TABLE` returns the handle of the after-table for this before-table.

These attributes are also read-only.

In addition, you can point directly from the buffer for one table to the buffer for the other using these attributes on a buffer handle:

- *`after-buffer-handle`* `:BEFORE-BUFFER` returns the handle of the default buffer for the before-table that is associated with the after-table for *`after-buffer-handle`*. Note that *`after-buffer-handle`* can be any buffer for the after-table, while the buffer handle you get back as the attribute value is always the default buffer for the other table.
- *`before-buffer-handle`* `:AFTER-BUFFER` similarly returns the handle of the default buffer for the after-table, relative to any buffer for the before-table.

When you pass a ProDataSet as a parameter remotely, any before-tables associated with its temp-tables are automatically and transparently marshaled along with it. If passed locally by value, then the before-tables are copied along with the after-tables. If passed locally `BY-REFERENCE`, then all references to the before-table are available to the procedure receiving the ProDataSet.

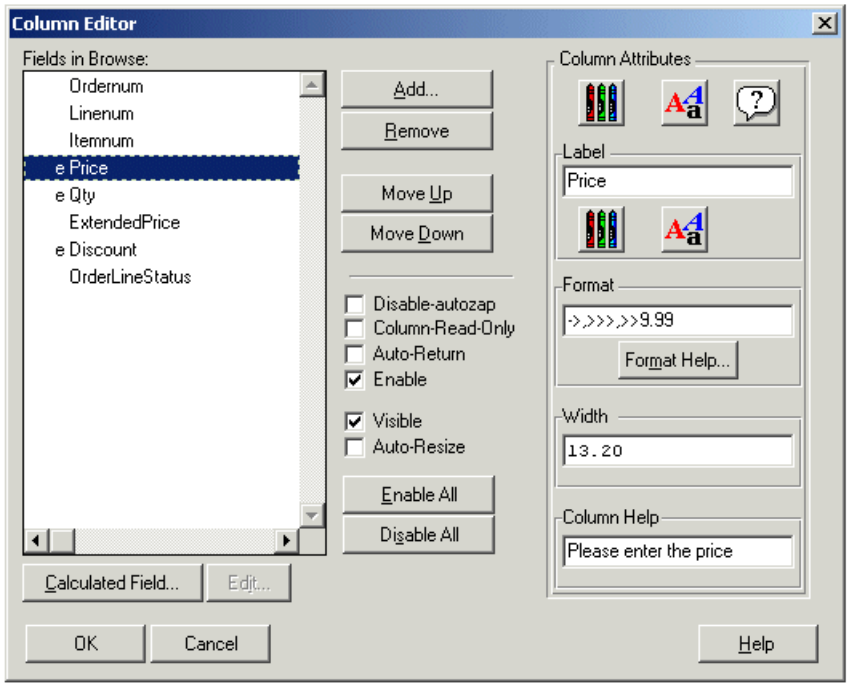
Extending the sample procedures to track changes

You can track changes in a ProDataSet to extend the sample window `dsOrderWin.w` and its supporting procedures:

To track changes in the ProDataSet:

1. Save a new version of the window procedure `dsOrderWin.w` as `dsOrderWinUpd.w`.
2. Enable the `Price`, `Qty`, and `Discount` columns in the `ttOline` browse.

   To do this, double-click on the browse, choose the **Fields** button in its property sheet, and double-click on each of the three fields. An **'e'** appears next to each field to indicate that it is now enabled for input, as shown:



   You have to define a before-table for the `ttOline` temp-table. Because this is a static temp-table, its before-table must be defined statically as well.

3. Edit the include file `dsOrderTT.i` to add the `BEFORE-TABLE` definition to the temp-table for `OrderLines`, as shown:

```
. . .
DEFINE TEMP-TABLE ttOLine BEFORE-TABLE ttOlineBefore
. . .
```

   Using this definition, the AVM creates the companion temp-table `ttOlineBefore` along with `ttOline`. The two temp-tables share the same scope. You can reference `ttOlineBefore` in your procedure code just as you can `ttOline`, but you cannot directly make any changes to its records. The before-table remains empty until you set `TRACKING-CHANGES` to true for `ttOline` and start making changes to `ttOline` records.

   You need to set the `TRACKING-CHANGES` attribute to enable tracking of any updates to the `OrderLines`.

4. In the LEAVE trigger for field iOrderNum, add a line to set TRACKING-CHANGES to true after receiving an Order ProDataSet back from OrderMain.p. In addition, set TRACKING-CHANGES to false before requesting the ProDataSet, as shown:

```
TEMP-TABLE ttOline:TRACKING-CHANGES = FALSE.

DATASET dsOrder:GET-RELATION(1):QUERY:QUERY-CLOSE().
DATASET dsOrder:GET-RELATION(2):QUERY:QUERY-CLOSE().
DATASET dsOrder:EMPTY-DATASET.

RUN OrderMain.p (INPUT iOrderNum, OUTPUT DATASET dsOrder BY-REFERENCE).

TEMP-TABLE ttOline:TRACKING-CHANGES = TRUE.
FIND FIRST ttOrder.
```

   When you set TRACKING-CHANGES to true, you enable the user to change the Price, Qty, and Discount fields of one or more ttOline records and to have those changes recorded in the client window procedure. For each change, the AVM adds a record to ttOlineBefore with the record's values before it was first changed.

   To make sure that you can select and view one Order and then another without getting an error that says you are doing a FILL when TRACKING-CHANGES is true, you need to set TRACKING-CHANGES to false before making the call to OrderMain.p.

   You need to assign changes made to the browse columns.

5. Define this ROW-LEAVE trigger for the OlineBrowse, as shown:

```
DO:
  DEFINE VARIABLE hCol AS HANDLE NO-UNDO.

  IF OlineBrowse:MODIFIED THEN
    ASSIGN
      INPUT BROWSE OlineBrowse
      {&ENABLED-FIELDS-IN-QUERY-OlineBrowse}
      /* Disable the Order Number until changes are saved. */
      iOrderNum:SENSITIVE IN FRAME dsFrame = FALSE.
END.
```

   In this trigger block, if the browse row has been modified, you assign the list of enabled columns. This is available as the preprocessor value ENABLED-FIELDS-IN-QUERY-OlineBrowse, which allows you to make changes to the list of enabled columns without having to remember to go back to change a hard-coded field list in this trigger.

   Once the user has made any changes, you prevent them from switching Orders until the changes have been saved back to the database, by disabling the iOrderNum field.

6. The AVM does not automatically set the MODIFIED browse attribute to false when you switch rows, so you must define a VALUE-CHANGED trigger block for the OlineBrowse, as shown:

```
DO:
  OlineBrowse:MODIFIED = FALSE.
END.
```

The ROW-LEAVE event fires **before** the browse switches rows, when you press the up or down arrow or select a different row with the mouse, for example. This lets you capture changes to the current row. The VALUE-CHANGED trigger fires **after** the browse switches rows, so it would be too late to capture changes to the row you left. All you can do here is reset the MODIFIED attribute in preparation for the next change.

Start to build the trigger code that sends changes back to the database.

7. In the AppBuilder, drop a new button called BtnSave onto the window and label it **Save Changes**.
8. To illustrate how ROW-STATE and the BEFORE/AFTER-ROWID attributes work, add this code to the CHOOSE trigger for the button:

```
FOR EACH ttOlineBefore:
  FIND ttOline WHERE ROWID(ttOline) = BUFFER ttOlineBefore:AFTER-ROWID.
  MESSAGE "Before ROW-STATE: " BUFFER ttOlineBefore:ROW-STATE
    " and Price: " ttOlineBefore.Price SKIP
    "After ROW-STATE: " BUFFER ttOline:ROW-STATE
    " and Price: " ttOline.Price.
END.
```

For each OrderLine that you change, there should be a record in ttOlineBefore. That record points to its companion in ttOline using the AFTER-ROWID attribute. The MESSAGE displays the value of ROW-STATE for each record, along with the Price before and after the change.

9. To try this out, run the window procedure, select an Order, and modify one or two of the Price values of its OrderLines.
10. Choose the **Save Changes** button.

For each OrderLine you changed, you should see a message like the following for a Price change from 23.95 to 23.85:



This confirms that the ROW-STATE attribute is always set for both the before-table record and the after-table record, so that you can query it starting at either table. The ROW-STATE value of 2 represents the literal ROW-MODIFIED. In the case of a delete, of course, there will be a record with a ROW-STATE of 1 (ROW-DELETED) only in the before-table, and no record in the after-table.

The message also confirms that the before-table record holds the value of the fields before any changes were made, and the after-table holds the modified values.

Comparison with change tracking in .NET

For comparison it is important to note that .NET supports row states and versions similar to these, with certain exceptions. This is the mapping between .NET states and versions, and the ProDataSet records and row states:

- Record states in .NET can be Added, Deleted, Detached, Modified, or Unchanged. These correspond to ROW-STATE values in ABL except for Detached. Detached means created but not yet in the DataTable. This is a state that can be supported in the AVM using transaction semantics. If you CREATE a temp-table record inside a transaction, then until that transaction ends that record is Detached; it has been created

but it is not yet "officially" or "definitively" in the table. It can be backed out if the transaction fails or is otherwise undone.

- Records in `DataTables` in .NET also have a version, which is a qualifier on the record object (like an attribute reference). This can be `Current`, `Default`, `Original`, or `Proposed`.
- The `Current` version is like a record in the ProDataSet after-table. We have specifically decided to use the term (and keyword element) `AFTER` because `CURRENT` already has a very specific, and very different, meaning in ABL, namely in functions and statements such as `GET CURRENT`, `FIND CURRENT`, and so on, that normally refer to a database record buffer.
- The `Original` version is like a record in the ProDataSet before-table. Once again, we have decided to use the term and keyword `BEFORE` rather than "original" because in the AVM we create and maintain a separate, additional table for the before images of records. It is logical to think of this table as being created as a by-product of the actual temp-table containing the data as it is retrieved and viewed. Therefore, anyone referring to these objects is likely to think of what Microsoft calls the "current" records as the "original" table, not the other way around. Also, the word "before" complements "after," so if one term is going to be different from .NET both might as well be, and also it matches our use of the well-established term "before-image," which is precisely what the "before" table contains.
- The `Default` version is defined to be the `Current` version for an `Added` or `Modified` record, or the `Original` version for a `Deleted` record. This rather specialized identifier is not duplicated in ABL and provides no information that cannot be obtained otherwise.
- The `Proposed` version is one that exists, but has not yet been committed to the `DataTable`. This state is possible inside a `BEGIN-EDIT/END-EDIT` block, which is essentially like a transaction block in ABL. Like the `Detached` state, this can be represented using transaction semantics in ABL, as a record still within an unfinished transaction.

Processing changes

Once you have made a set of changes to a ProDataSet, you then need to process them in a session that is connected to the database, and where the ProDataSet is attached to the Data-Sources. If you made the changes in a separate client session, such as our little `Order` window application, you need to pass the ProDataSet back to the server session. To minimize the amount of network traffic, in most cases you only want to send back those rows that were changed in some way, along with their before-images, leaving out all the rows in the after-tables that were not changed. There might be some exceptions to this, such as when the server-side business logic needs all the records in the original ProDataSet to do its processing, but in most cases you should do everything possible to limit the number of rows sent back across the network in a distributed application.

GET-CHANGES method

The `GET-CHANGES` method extracts the changes for you into a second ProDataSet, normally one that you have just created using the `CREATE-LIKE` method. However, there is nothing to prevent you from defining two equivalent ProDataSets statically and using `GET-CHANGES` to move changes from one to the other. If the target ProDataSet is static, it must contain temp-tables whose field definitions are the same at least in field data type, extent, and order. The temp-table index and Data-Relation definitions do not need to match between the ProDataSets, but the fields of the temp-tables must match in name, order, data type, and extent.

We call the result of a `GET-CHANGES` method a change ProDataSet. We refer to the original ProDataSet that acts as the source for the change ProDataSet as the origin ProDataSet.

GET-CHANGES uses the handles of the two ProDataSets, just as CREATE-LIKE does, as shown in the following syntax:

Syntax

```
change-dataset-handle:GET-CHANGES( origin-dataset-handle ).
```

GET-CHANGES effectively does the following for each temp-table in the *origin-dataset* for which there is a before-table:

- For each row in the before-table, it creates a row in the before-table of the *change-dataset* and buffer-copies the *origin-dataset* row into the *change-dataset*.
- For each before-table row that is not for a delete, it locates the row in the after-table of the *origin-dataset* that is paired with the row in the before-table (using the AFTER-ROWID attribute), creates a row in the after-table in the *change-dataset*, and buffer-copies the *origin-dataset* row to the *change-dataset* table.
- Sets the ROW-STATE of the *change-dataset* row to the ROW-STATE of the *origin-dataset* row in the *target-dataset* before-table, and if applicable, the after-table.
- Saves the ROWID of the before-table record in the *origin-dataset* as an attribute, called ORIGIN-ROWID, of the same row in the *change-dataset*. This can be used later to allow the final versions of changed rows to be merged into the original *origin-dataset*.

As the examples in Advanced Update Operations on page 233 show, your procedure can then pass the target dynamic ProDataSet as an INPUT-OUTPUT parameter to the server-side procedure, get back the final versions of any records further changed on the server, and merge them back into the original ProDataSet, using the MERGE-CHANGES method.

If you want to include the parent row of each changed child row in the ProDataSet object or ProDataSet temp-table, use the optional *get-parent-mode* expression, as shown in the following syntax:

Syntax

```
change-dataset-handle:GET-CHANGES(origin-dataset-handle  [, get-parent-mode] ).
```

If there is more than one parent level above the changed row, the AVM includes the parent row at each level. In this case, the parent temp-tables must have a unique primary index that the AVM can use to find the corresponding rows. If a parent row has changed, the AVM copies both the before-image and after-image of the parent row. If a parent row has not changed, there is no before-image of the parent row, and its change state (ROW-STATE) is ROW-UNMODIFIED(0) or the Unknown value (?). When the relation mode of a parent is REPOSITION, no attempt is made to find that parent.

When this expression is false, the AVM does not include parent rows. The default value is false.

The ProDataSet objects associated with the change-handle and original-handle must have the same number of temp-table buffers, and the definition of the corresponding temp-tables must match (that is, in the number of columns, data types, and so on).

Once the changed rows are loaded, the AVM sets the ORIGIN-HANDLE attribute on the temp-tables in the receiving ProDataSet object to the corresponding temp-tables in the original source ProDataSet object. The AVM also sets the ORIGIN-ROWID attribute on each of the before-image table rows created in the receiving ProDataSet object to the ROWID of the corresponding before-image table row in the original source temp-table. The MERGE-CHANGES and MERGE-ROW-CHANGES methods use these values to match up temp-tables and temp-table rows during a merge operation.

MERGE-CHANGES and MERGE-ROW-CHANGES methods

Once you have passed such a change ProDataSet to the server for update processing, it is quite likely that the ProDataSet data will be further modified on the server. For one thing, database triggers or other server-side code can make changes to the data, such as assigning key values from database sequences, generating values for calculated fields, and so on. In addition, any errors that result from the validation logic or from attempting to write invalid values back to the database will be logged in the ERROR-STRING attribute of the temp-tables or their individual records. For this reason, your client-side procedures will most often pass a change ProDataSet as an INPUT-OUTPUT parameter to the server business logic procedure, in order to get the final versions of all the records back, as well as any error messages.

Your client-side procedure can then scan the ProDataSet for error messages as it is returned. It then needs to merge any changes into the origin ProDataSet on the client so that the user can see the final versions of records in the user interface. The MERGE-CHANGES method replaces all the after-table values in the origin ProDataSet with the final values in the change ProDataSet. MERGE-CHANGES has the same syntax as other methods that operate on two ProDataSets, as shown:

Syntax

```
change-dataset-handle:MERGE-CHANGES( origin-dataset-handle
  [, copy-all-mode] ).
```

MERGE-CHANGES is almost always used in conjunction with GET-CHANGES. The ProDataSet that is the *origin-dataset* of the GET-CHANGES method must be the *origin-dataset* of a MERGE-CHANGES method, and the *change-dataset* for the MERGE-CHANGES method must be the *change-dataset* of the GET-CHANGES method that populated it. The GET-CHANGES method sets attribute values that allow a later MERGE-CHANGES method on the same pair of ProDataSets to move changes back into the origin ProDataSet.

If you want to merge all after-image table rows whether or not they contain changes, use the optional logical expression, copy-all-mode, In this case, the temp-table in the original ProDataSet object must have a unique primary index that the AVM can use to find each corresponding row from the after-image table (since unchanged rows do not have a corresponding row in the before-image table). When a corresponding row is not found in the original ProDataSet object, the AVM creates a new row using the row from the after-image table. When false, the AVM merges only after-image table rows that contain changes. The default value is false.

In some cases, you might need to construct a change ProDataSet yourself that is not the result of a call to GET-CHANGES, for example, if you need to do specialized processing that is not handled by GET-CHANGES. For this reason, we expose the attributes that MERGE-CHANGES uses to reconcile the two ProDataSets and allow you to set their values so that you can simulate in your own custom ABL code everything that GET-CHANGES does for you. These attributes are the ORIGIN-HANDLE of each modified temp-table and the ORIGIN-ROWID of each row, and they are explained later in this section.

Likewise, you might in some cases need to do your own merge between two ProDataSets instead of using the MERGE-CHANGES method. In this case, if you have a way to identify the corresponding rows reliably between the two ProDataSets, you can do this in your own ABL code if you need behavior different from what MERGE-CHANGES does for you.

GET-CHANGES stores the ROWID of each before-table row from the *origin-dataset* as an attribute of the equivalent row in the *change-dataset*, so that MERGE-CHANGES can correctly buffer-copy final row values back to the origin ProDataSet. Relying on the primary key or other data in the records would not be reliable in the general case because the key value might have been changed in the business logic. In fact, in the case of a newly created record, it is **likely** to have been changed because key values are often assigned from database sequences that are available only where the database is connected.

The before-table ROWID attribute is called ORIGIN-ROWID. For each before-table row in the *change-dataset* of the MERGE-CHANGES method, this attribute holds the ROWID of the corresponding before-table row in the *origin-dataset*.

Temp-table ROWIDs are, of course, highly transitory. If you create the same set of temp-table rows twice in succession, they will almost always have different ROWIDs each time. To assure that the ROWIDs saved as the ORIGIN-ROWIDs are still valid, GET-CHANGES also stores the temp-table handle for the after-table of the *origin-dataset* as an attribute called ORIGIN-HANDLE on the equivalent temp-table in the *change-dataset*. When you apply the MERGE-CHANGES method to two ProDataSets, the AVM verifies that the ORIGIN-HANDLEs of any modified temp-tables in the *change-dataset* (the source of MERGE-CHANGES) match the temp-table handles in the *origin-dataset*. If they do not match, or if any ORIGIN-ROWIDs cannot be found in the origin ProDataSet, an error results.

If you set these attributes yourself to do your own custom version of what GET-CHANGES and MERGE-CHANGES do, you must exercise care to make sure that you identify, set, and use the values properly. Invalid ROWIDs can result in very unpredictable and undesirable behavior.

There is also a MERGE-ROW-CHANGES method to merge only a single changed row back into the *origin-dataset*, as shown in the following syntax:

Syntax

```
change-buffer-handle:MERGE-ROW-CHANGES( [original-buffer-handle] ).
```

The *change-buffer-handle* is the before-table buffer handle in the *change-dataset*. The *original-buffer-handle* is the corresponding buffer in the *origin-dataset*. This argument is optional because the AVM can normally determine the handle automatically from the change-table attributes.

These methods are normally executed after passing the change ProDataSet to another procedure to process the changes. MERGE-CHANGES verifies that the ORIGIN-HANDLE of each change table matches the object handle of the equivalent table in the original ProDataSet, and returns an error if they do not match. It finds each before-table record in the original ProDataSet based on the ORIGIN-ROWID attribute of each changed row, and uses that to merge changes back into the original ProDataSet. If there are any mismatches here, this likewise results in an error. MERGE-ROW-CHANGES does the same for a single row.

When changes are merged, any error flags in the data cause a change to be rejected instead of merged. Otherwise, the change is accepted and the table statuses are updated accordingly. The methods and attributes that control this are described just below and are summarized here with respect to their relation to the MERGE-CHANGES method.

If either the ERROR attribute or the REJECTED attribute is true for a row, then MERGE-CHANGES or MERGE-ROW-CHANGES performs a REJECT-ROW-CHANGES on the row instead of merging it. If the REJECTED attribute is true for a table, then a MERGE-CHANGES that is executed on the whole ProDataSet runs REJECT-CHANGES on the table instead of merging it. For each table or row that is merged back into the original ProDataSet, the merge operation does an ACCEPT-CHANGES or ACCEPT-ROW-CHANGES implicitly.

ACCEPT-CHANGES and ACCEPT-ROW-CHANGES methods

There is an ACCEPT-CHANGES method on the ProDataSet handle, and the ProDataSet temp-table buffer handle, which effectively tells the AVM that any changes tracked in the before-table(s) have been processed and those records should be accepted as the new origin ProDataSet versions. The method can be executed on the ProDataSet handle or a temp-table buffer handle in the origin ProDataSet.

If the method is executed on the ProDataSet, then all before-tables in the ProDataSet are emptied, and the pointers to those records in the after-table are removed so that the BEFORE-ROWID attribute of every record in the after-tables is the Unknown value (?) and the ROW-STATE of every record in the after-tables is set to ROW-UNMODIFIED(0). The before-tables are not deleted, only emptied for possible later use.

If ACCEPT-CHANGES is executed on a temp-table buffer handle, then it is applied only to that temp-table and its after-table. It is equally possible to execute APPLY-CHANGES on the after-table handle, in which case it applies to that table and its before-table, and the result is the same. However, you should normally ACCEPT-CHANGES on the before-table buffer, since that is the table that holds a record of every change and nothing else.

There is a separate ACCEPT-ROW-CHANGES method for the buffer handle, which executes only on the current row in the buffer. It is applied only to that one record in the table and its corresponding record in the other table. ROW-ACCEPT-CHANGES can be executed only on the before-table buffer handle of the temp-table in the origin ProDataSet. This is the syntax for the ACCEPT-ROW-CHANGES method:

Syntax

```
before-table-buffer:ACCEPT-ROW-CHANGES()
```

These methods effectively reset the ProDataSet, temp-table, or row so its current state is as if a FILL was just executed.

REJECT-CHANGES and REJECT-ROW-CHANGES methods

As a complement to ACCEPT-CHANGES, there is also a REJECT-CHANGES method to restore part or all of the origin ProDataSet to its original state before a change was made, as shown using this syntax:

Syntax

```
dataset-handle:REJECT-CHANGES().
```

Like ACCEPT-CHANGES, this method can also be executed on a single temp-table buffer handle within the ProDataSet, as shown with the following syntax:

Syntax

```
buffer-handle:REJECT-CHANGES().
```

This is normally done automatically by MERGE-CHANGES when a REJECTED row, table, or ProDataSet is encountered. You can also run it yourself to control just which changes are reflected in a ProDataSet and which are undone.

As with other methods, there is also a method to reject only a single row, as shown in the following syntax:

Syntax

```
buffer-handle:REJECT-ROW-CHANGES( ).
```

The methods REJECT-CHANGES and REJECT-ROW-CHANGES restore one more of the original ProDataSet rows by copying the before-table rows back to the corresponding after-tables, deleting the before-table rows and resetting the ROW-STATE in the after-table rows to ROW-UNMODIFIED. When the REJECT-CHANGES method is executed on a ProDataSet handle, all modified tables in the ProDataSet are rejected. When executed on a buffer-handle, only changes to that buffer's temp-table are rejected.

SAVE-ROW-CHANGES method

Some of the behavior and attributes described in this and the following section require the first Service Pack for OpenEdge 10, release 10.0A01.

Once a set of changes has been completed, you can use the SAVE-ROW-CHANGES method on a before-table buffer handle to save the changes recorded in that row back to the Data-Source, as shown in the following syntax:

Syntax

```
buffer-handle:SAVE-ROW-CHANGES( [ buffer-index | buffer-name
                                [ , skip-list ] ).
```

Where:

- *buffer-index* is the sequential position of the buffer within the buffer list for the table that is being updated, if there is more than one buffer in the Data-Source definition. The argument is optional and the default value is 1.
- *buffer-name* is the buffer-name as an alternative to the buffer index and defaults to the first (or only) buffer in the Data-Source.
- *skip-list* is an optional character expression that evaluates to a comma-separated list of names for fields that should not be assigned after the CREATE of a new record, because it is a key field or other field assigned a value by a CREATE database trigger. Because a CREATE trigger fires as soon as a new record is created, a key field that is assigned a value in the CREATE trigger, such as an integer field assigned from a database sequence, would be overwritten by the default value for the field when SAVE-ROW-CHANGES assigns fields to the new database record. This optional argument suppresses the ASSIGN for that one field.

This method does the default handling of a create, modify, or delete on that buffer's record. There is no SAVE-CHANGES method for an entire change ProDataSet or temp-table. Instead, you execute the row-level method for each row you want saved. This way you can determine the exact sequence, transaction scoping, and other coding details for handling a set of related changes. Note that although we speak of a set of related changes and provide methods to handle multiple changes at a time, you can just as easily make a single row change and save it back to the Data-Source.

In a distributed application, you will typically collect just the changes into a change-ProDataSet using GET-CHANGES, and then pass them back to the server to be processed by executing SAVE-ROW-CHANGES on each row. However, there is no need always to use GET-CHANGES to extract just the change rows from the original ProDataSet. This is simply an optimization step to minimize network traffic for a distributed application. If you are not passing changes across a network, you can just as easily execute SAVE-ROW-CHANGES on each before-table row in the origin ProDataSet. How you use the methods is entirely up to you.

Changes can be automatically saved only for a single buffer at a time. If there is more than one buffer in the Data-Source, you can specify which buffer to update as the first argument to SAVE-ROW-CHANGES. Typically, an update to a row that contains fields from multiple database tables updates only one of those tables. For example, in the example we have been using of a ttOrder table that joins in the Customer Name from the Customer table and the SalesRep name from the SalesRep table, your application would not normally allow the user to change the Customer Name by updating one of its Orders. This is because the Name is directly associated with the CustNum field that is the actual join field between the tables, and there would be no proper way to map a change to Customer Name for one Order with either the Customer table or with other Orders for the same Customer.

If you have a special situation where you are, in fact, updating both sides of a join through a ProDataSet (which might be the case if the join is one-to-one) then you can use SAVE-ROW-CHANGES to assign fields from one table and assign the others in your own code, or you can execute SAVE-ROW-CHANGES on each buffer in turn. Otherwise, if your update requirements are quite specialized, you can simply not use the method at all and assign all fields yourself. As with other methods, all the steps taken by SAVE-ROW-CHANGES can be duplicated in your own code when you need to change the default behavior.

You must assure that the Data-Source is attached before doing a SAVE-ROW-CHANGES. In cases where there is no Data-Source or where special processing is needed, you must write the update code yourself instead of using SAVE-ROW-CHANGES.

SAVE-ROW-CHANGES goes through these steps for a modified row:

- Finds the corresponding database record for the updateable buffer the row is derived from, based on its key values as defined in the Data-Source definition. These key values are not ROWIDs, unless the ROWID is explicitly identified as the KEYS field and mapped to a field in the temp-table that holds the ROWID . There must be a unique key defined on the database table for the Data-Source to support this. This find is done with an EXCLUSIVE-LOCK, so that the row can be updated. If the record is not available the method retries once after a pause of one second, and then returns an error if the record is still not available.
- Compares the saved-off before-image of the ProDataSet buffer specified by the *buffer-index* or *buffer-name* argument in the Data-Source with the corresponding database buffer to confirm whether the data has been changed since being read. A change is rejected if the underlying data has been changed. Note that only the fields that are mapped to the ProDataSet temp-table can be compared. Fields in the database table not present in the temp-table cannot be checked.
- Buffer-copies changed fields in the corresponding after-table buffer to the corresponding database buffer fields, using the same field mapping used to FILL the table (as defined in the ATTACH-DATA-SOURCE method).
- Validates the updated database record to force any WRITE or ASSIGN triggers to fire.
- Sets the ERROR logical attribute in the after-table row, as well as in its temp-table and in the ProDataSet if any errors resulted from the attempted update, such as duplicate unique keys.
- Repopulates the after-table record from the database record, to catch any changes made by either event procedure code or trigger procedure code. If the procedure defines the ProDataSet as an INPUT-OUTPUT parameter sent from client to server, for instance, these changes are returned to the client where they can be displayed.
- Releases the after-table and database table records.

For a newly created row, SAVE-ROW-CHANGES creates the record in the database and buffer-copies all data table buffer fields except any *create-field* to the database buffer. It then validates and copies the database record back to the temp-table buffer as for a modified row.

For a deleted row, SAVE-ROW-CHANGES deletes the corresponding row from the Data-Source, based on the record's keys.

As with a modified row, SAVE-ROW-CHANGES can create or delete only a single database buffer at a time, not both sides of a one-to-one join. In most cases, the creation or deletion of related rows is likely to be implemented in the CREATE or DELETE trigger for the primary buffer anyway.

DATA-SOURCE-ROWID attribute

You can make your own version of SAVE-ROW-CHANGES, or access the source record in some way before or after SAVE-ROW-CHANGES, using the DATA-SOURCE-ROWID attribute of the buffer object. This attribute identifies the Data-Source row corresponding to either an after-table or a before-table buffer, as shown in the following syntax:

Syntax

```
rowid-variable = buffer-handle:DATA-SOURCE-ROWID ( arg )
```

If the Data-Source is a join, *arg* is the integer level of join, or it is a character expression that evaluates to the name of one of the Data-Source buffers.

MARK-ROW-STATE method

If a ProDataSet contains changes to be applied to a Data-Source, but no before-table records exist for it, you might still want to use the SAVE-ROW-CHANGES method to move the new data from the ProDataSet to the source database. This might be the case if the client does not have the ability to create before-table records, such as a third-party .NET application or some incoming XML or JSON with changed data, but no before-table records. The MARK-ROW-STATE method of the buffer handle creates the before-table records needed by SAVE-ROW-CHANGES, usually from independently pre-saved records on the server, as shown in the following syntax:

Syntax

```
MARK-ROW-STATE( row-state [, before-image source buffer handle] )
```

row-state is an integer expression that evaluates to one of the row-state values or one of the compiler constants ROW-CREATED, ROW-MODIFIED or ROW-DELETED as shown in the following table.

Compiler constant	Value	Description
ROW-UNMODIFIED	0	The row was not modified.
ROW-DELETED	1	The row was deleted.

Compiler constant	Value	Description
ROW-MODIFIED	2	The row was modified.
ROW-CREATED	3	The row was created.

before-image source buffer handle is an optional handle to the before-image source buffer.

If *row-state* is ROW-UNMODIFIED or if there is already a before-table record, no action is taken.

If *row-state* is ROW-CREATED, a before-table record is created and its ROW-STATE is marked as ROW-CREATED. The before-table record is associated with the buffer handle record.

If *row-state* is ROW-MODIFIED and the second optional parameter is defined, a before-image record is created by copying the before-image source buffer. If the second parameter is not defined, then a Data-Source must be attached to the buffer handle. In this case, the Data-Source mapping is used to locate the Data-Source record(s) related to the buffer handle, which is used to create a before-table record. The before-table record is marked ROW-MODIFIED and is associated with the buffer handle record.

Note: If the second parameter is not defined and if the Data-Source top level record cannot be located, then a ROW-CREATED before-table record will be created and associated with the buffer handle record.

If *row-state* is ROW-DELETED and the second optional parameter is defined, a before-image record is created by copying the before-image source buffer. If the second parameter is not defined, then a Data-Source must be attached to the buffer handle. In this case, the Data-Source mapping is used to locate the Data-Source record(s) related to the buffer handle, which then can be used to create a before-table record.

If a second parameter is not defined, then the buffer handle must contain a temporary record whose only purpose is to locate the Data-Source. Otherwise, it can be empty.

Note that if buffer handle is for a static table that was defined without a BEFORE-TABLE option, MARK-ROW-STATE causes the AVM to raise ERROR. If buffer handle is for a dynamic table that has no before-table, one is automatically created.

MARK-NEW method

If a ProDataSet temp-table is known to contain only new data, the MARK-NEW method of the buffer handle can be used to create a before-table record for those buffer records that do not have one, as shown with this syntax:

Syntax

```
buffer-handle:MARK-NEW()
```

Using this method is easier than using MARK-ROW-STATE(ROW-CREATED) because you do not have to manually loop through all the records in the table. ROW-STATE is set to ROW-CREATED. MARK-NEW does not support record deletes or updates.

If *buffer-handle* points to a static temp-table that was defined without a BEFORE-TABLE option, MARK-NEW causes the AVM to raise ERROR. If *buffer-handle* is for a dynamic temp-table that has no before-table, one is automatically created.

Special support for change conflicts

Because the ProDataSet with its independent FILL and save operations necessarily relies on an optimistic locking strategy, you need to consider what behavior you want in the event of a conflict with another change made since the ProDataSet was filled from its Data-Source. The AVM supports a number of options to give you automated support for almost any circumstance.

The first decision you need to make is whether a conflict with another change should be overwritten by the current ProDataSet's change, or rejected when the AVM detects a conflict.

In some cases, you might not care whether a record has been changed by another user since it was read. Although this is certainly not typical or recommended, for some non-transactional tables that contain fields that can safely be changed independently of one another (possibly address information, for example), you might not want to reject a change if the record has been changed elsewhere. There is a PREFER-DATASET logical attribute for the Data-Source to support this option. The attribute is false by default. If PREFER-DATASET is false for the table's Data-Source, then the AVM makes the comparison between the before-table row in the database and the corresponding Data-Source fields before applying the change. If there is any conflict, then the change is rejected and the ERROR attribute is set. If PREFER-DATASET is true, this check is not made. The ProDataSet row changes are written to the Data-Source without regard to any other changes made to the same row. As noted previously, only the database fields that are present in the ProDataSet temp-table, either implicitly or by being mapped to a field with a different name, can be compared.

You also may want to apply changes and resolve conflicts on a field by field basis. This is somewhat more expensive than either applying or rejecting all changes, but minimizes the overwriting of either your changes or another user's changes when a conflict occurs. There is a MERGE-BY-FIELD logical attribute on the Data-Source to specify this preference. This attribute is true by default. It gives you the ability to define whether a change is wholly rejected if there are any conflicting changes, even to different fields than the ones the ProDataSet is changing.

If MERGE-BY-FIELD is false and if another user has changed a record, the ProDataSet changes are either entirely accepted or entirely rejected, depending on the setting of PREFER-DATASET. If MERGE-BY-FIELD is true, then the AVM "blends" the two sets of changes where possible, if they are to different fields. If it is true, then changes are applied field-by-field only for modified fields, and there is no conflict unless the same field has been modified both by the current update and by another transaction. A logical DATA-SOURCE-MODIFIED attribute on the temp-table buffer is set to true if any field changes by another transaction are detected, regardless of whether they result in a conflict and error or not. Note that it is somewhat slower to have MERGE-BY-FIELD set to true, so it can be set to false to improve performance where a field-by-field check is really not necessary, in addition to cases where it is not wanted because of the application logic.

Here are the details of how changes are applied in the face of these attributes. In this description, "current changes" refers to modifies and deletes that are part of a ProDataSet the procedure is doing a SAVE-ROW-CHANGES on (the "current ProDataSet"). Obviously create operations do not need to check for conflicting field values. "Other changes" refers to changes made to the fields in the same rows by another user or procedure since the current ProDataSet was filled, which will be detectable by comparing the database buffers with the before-tables in the current ProDataSet.

If MERGE-BY-FIELD is false, then these are the actions:

- If PREFER-DATASET is true, then the modified or deleted row is updated or deleted without regard to whether there were other changes. In the case of a modify, all fields in the current ProDataSet row are copied to the database buffer, whether they were changed in the current ProDataSet or not. In the case of a delete, the database record is simply deleted.
- If PREFER-DATASET is false (the default), then the AVM compares the ProDataSet before-table with the database buffers. If there are any other changes, then the current change is rejected and the AVM sets the ERROR attribute. If the action was a delete, the row is not deleted from the database or the current ProDataSet. The AVM also sets the DATA-SOURCE-MODIFIED attribute on the row. It buffer-copies the entire database buffer back to the current ProDataSet's after-table row, to allow the changes to be seen back on the client.

If MERGE-BY-FIELD is true (the default), then rather than doing a wholesale buffer-copy in one direction or the other, the AVM does a field-by-field compare of the two buffers. In this case:

- This is the action if PREFER-DATASET is true:
  - If the operation is a delete, then the database row is deleted without regard to whether there were other changes.
- Otherwise, for a Modify:
  - If the field has not been changed in the current ProDataSet, then nothing is copied. This means that a change made by another user to a field that was not changed in the current ProDataSet will not be overwritten by the current change.
  - If the field has been changed in the current ProDataSet, it is copied to the database field. There is no check of whether there were other changes.
- These are the actions if PREFER-DATASET is false:
  - If the operation is a delete, and there are no other changes to the row, then the database record is deleted.
  - If the operation is a delete, and there **are** other changes to the row, then the delete fails, the AVM buffer-copies the entire database record into the ProDataSet buffer, and sets the ERROR and DATA-SOURCE-MODIFIED attributes.
- Otherwise, for a Modify:
  - If the field was changed in the current ProDataSet and not by other changes, then it is copied to the database.
  - If the field was not changed in the current ProDataSet, and also not by other changes, then nothing is copied.
  - If the field was changed in the current ProDataSet, and by other changes, then the AVM checks whether the other changes are the same as the current ProDataSet change. If this is the case, the AVM sets the ERROR flag. If the changed values are different, the AVM buffer-copies that one field from the database record back into the ProDataSet buffer, and sets the ERROR and DATA-SOURCE-MODIFIED flags. The current ProDataSet changes are not applied to the database, but the field by field comparison continues so that only those fields changed by other changes overwrite the current ProDataSet changes.
  - If the field was not changed in the current ProDataSet, but was changed by other changes, then the other changes are copied into the ProDataSet buffer, and the AVM sets the DATA-SOURCE-MODIFIED flag. This is not an error, however.

The net result of all this is, when PREFER-DATASET is false and MERGE-BY-FIELD is true, a modified row in the current ProDataSet is rejected in its entirety and causes an error only when one or more of the same fields have been changed by another user. Otherwise changed fields in the current ProDataSet row are successfully written to the database. Changes made by others are copied to the ProDataSet buffer to be returned to the client, and the CHANGED flag is set.

This may all sound complicated, but in effect it means that the AVM "does the right thing" based on whether you want the new change or the old change to take precedence, and whether or not you want to apply changes field by field or as a whole. The following table summarizes the results.

PREFER-DATASET setting	MERGE-BY-FIELD setting	If field conflict…	ERROR flag	DATA-SOURCE-MODIFIED flag	What is copied to the DB
TRUE	FALSE	ProDataSet wins	FALSE	TRUE	All fields in the temp-table row
TRUE	TRUE	ProDataSet wins	FALSE	TRUE	Changed fields only
FALSE	TRUE	Data-Source wins	TRUE	TRUE	Only non-conflicting fields
FALSE	FALSE	Data-Source wins	TRUE	TRUE	Nothing

Using the SAVE-WHERE-STRING attribute

As outlined in the previous section, the first step in saving a change is to locate the database buffer the temp-table row was derived from. The AVM can always determine how to identify that row, based on the KEYS phrase of the Data-Source or the primary key of the database table, but it is not always trivial for you to determine the same information in your procedures.

For this reason, there is a SAVE-WHERE-STRING attribute on the Data-Source to return to you the where-clause you would need to use to retrieve the same record yourself, as shown:

```
data-source-handle:SAVE-WHERE-STRING( [ buffer-index | buffer-name ] ).
```

As with SAVE-ROW-CHANGES, SAVE-WHERE-STRING defaults to the first (or only) buffer in the Data-Source. The attribute value is set internally by the AVM to be the *where-clause* phrase (including the initial keyword WHERE) needed to find the database buffer identified by the *buffer-index* or *buffer-name* within the Data-Source. This can be useful when you need to build up custom code to retrieve database records of changed buffers, if using the SAVE-ROW-CHANGES method is not adequate. If you specify a *buffer-index*, it is the sequential position of the buffer within the list of multiple buffers for the Data-Source. If you specify the *buffer-name*, it must be the name of an after-table buffer in the list of buffers attached to the Data-Source. The buffer must be one that has a before-table, either static or dynamic. The attribute value qualifies field names in the buffer's temp-table with the before-table name, not the after-table name. This is because this string can be used to find database records based on field values in the before-table rows, as your code is cycling through the before-table, processing each change.

If an error occurs during the database update, the ERROR attribute is set to true for the ProDataSet, the temp-table where the error occurred, and the buffer where the error occurred.

The following table summarizes all these change methods and the handles they can be used on.

Method name	ProDataSet handle	Temp-table buffer handle	ROW version
GET-CHANGES	Yes	Yes	None
SAVE-ROW-CHANGES	No	No	Only
MERGE-CHANGES	Yes	Yes	MERGE-ROW-CHANGES

Method name	ProDataSet handle	Temp-table buffer handle	ROW version
ACCEPT-CHANGES	Yes	Yes	ACCEPT-ROW-CHANGES
REJECT-CHANGES	Yes	Yes	REJECT-ROW-CHANGES

In summary:

- You execute a method on a ProDataSet handle to affect the entire ProDataSet.
- You execute a method on a temp-table buffer handle to affect that one temp-table.
- You do not execute any of these methods on a temp-table handle. However, the TRACKING-CHANGES attribute is on the temp-table handle.
- There is no row-level method for GET-CHANGES. You execute this only for an entire temp-table or ProDataSet.
- There is only a row-level method for saving changes.
- The row-level method is executed on the temp-table buffer handle. The name tells the AVM whether to act on a single row or the entire table.

Extending the samples to GET, SAVE, MERGE, and ACCEPT changes

Now, you will extend dsOrderWinUpd.w to use these methods to return changes to the database. Note that you will make a whole series of changes to the procedures. In some cases, one change replaces a change made earlier. We do this to show how you can use the low-level methods and attributes, and then how the higher-level methods can do a lot of the work for you, replacing the code you wrote the first time through. The versions of these procedures that are saved with the other examples represent the final state of the procedures. ABL statements used in earlier stages of a procedure's development are commented out so that you can examine them if you need to. However, you should build your own versions of the procedures step by step in order to learn to use all the different levels of statements and attributes available to you.

To extend the dsOrderWinUpd.w procedure:

1. In dsOrderWinUpd.w, Remove or comment out the entire FOR EACH block with the MESSAGE statement in the CHOOSE trigger for BtnSave, and add these variable definitions to the top of the block:

```
DEFINE VARIABLE hBuffer    AS HANDLE NO-UNDO.
DEFINE VARIABLE hDSChanges AS HANDLE NO-UNDO.
DEFINE VARIABLE hDSOrder   AS HANDLE NO-UNDO.
DEFINE VARIABLE hQuery     AS HANDLE NO-UNDO.
```

2. This series of statements in the CHOOSE trigger creates the change ProDataSet:

```
hDSOrder = DATASET dsOrder:HANDLE.
CREATE DATASET hDSChanges.
hDSChanges:CREATE-LIKE(hDSOrder).
hDSChanges:GET-CHANGES(hDSOrder).
```

The CREATE DATASET statement simply allocates a structure for the ProDataSet definition and points the handle hDSChanges at that structure.

The CREATE-LIKE method creates the dynamic temp-tables and Data-Relations in that structure. If you provide a prefix as a second argument to CREATE-LIKE, then the temp-table names are prefixed by that string. Otherwise, they have the same names as in the original ProDataSet.

The GET-CHANGES method copies all the before-table rows and their after-table partners into the dynamic change ProDataSet.

3. Turn the TRACKING-CHANGES flag off for the ttOline table, before passing it to the update procedure for processing, as shown:

```
TEMP-TABLE ttOline:TRACKING-CHANGES = FALSE.
```

The MERGE-CHANGES method that you will use later to merge any final field updates back into the ttOline table the browse is displaying requires that TRACKING-CHANGES be false. In any case, you need to set it off at some point as part of preparing for the next set of changes the user makes.

4. Add a statement to run a new business logic procedure that accepts the changes and writes them back to the database, as shown:

```
RUN updateOrder.p (INPUT-OUTPUT DATASET-HANDLE hDSChanges BY-REFERENCE).
```

The change ProDataSet is an INPUT-OUTPUT parameter so that the window procedure gets back any final changes to the data, as well as any messages. The BY-REFERENCE qualifier tells the AVM to share the same ProDataSet instance when updateOrder.p is run locally.

5. Define the update procedure updateOrder.p:

```
/* updateOrder.p -- accepts a ProDataSet and saves changes to the
   OrderLine to the database. */
{dsOrderTT.i}
{dsOrder.i}

DEFINE INPUT-OUTPUT PARAMETER DATASET FOR dsOrder.
```

Because the procedure uses the static temp-table and ProDataSet definitions, it can receive the dynamic change ProDataSet that was passed into it as a DATASET-HANDLE using the static DATASET parameter form. If this procedure is run locally within the same session as the window, as it is in this simplified example, then the ProDataSet is passed by reference so that updateOrder.p is actually pointing to the dynamic ProDataSet as it was created in the window procedure. If the same call is made remotely across an AppServer connection, then the BY-REFERENCE qualifier is ignored and the ProDataSet is marshaled across the network in both directions. The net result is the same, so this single call lets the AVM run the procedure in the most efficient way whether or not the application is actually distributed.

The update procedure needs these variable definitions. You will need the Data-Source definition for the `OrderLine` table later in the procedure. Define and attach the Data-Source `srcOline`, as shown:

```
DEFINE VARIABLE hAfterBuf  AS HANDLE NO-UNDO.
DEFINE VARIABLE hBeforeBuf AS HANDLE NO-UNDO.
DEFINE VARIABLE hDSChanges AS HANDLE NO-UNDO.

DEFINE DATA-SOURCE srcOline FOR OrderLine.

BUFFER ttOline:ATTACH-DATA-SOURCE(DATA-SOURCE srcOline:HANDLE).
```

Just to verify that the right data got transferred into the change ProDataSet, you can get the handle to the ProDataSet and the after-table buffer for `ttOline`. For example:

```
hDSChanges = DATASET dsOrder:HANDLE.
hAfterBuf = hDSChanges:GET-BUFFER-HANDLE("ttOline").
hAfterBuf:FIND-FIRST().
```

You can then retrieve the corresponding before-table buffer in this way:

```
hBeforeBuf = hAfterBuf:TABLE-HANDLE:BEFORE-TABLE:DEFAULT-BUFFER-HANDLE.
```

This is a bit convoluted, as you have to go from the after-table buffer to its temp-table handle, then to the before-table temp-table handle, and then from there to the before-table's default buffer handle. Alternatively, you can accomplish the same thing with the `BEFORE-BUFFER` or `AFTER-BUFFER` attribute:

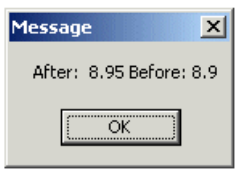
```
hBeforeBuf = hAfterBuf:BEFORE-BUFFER.
```

Remember that the `BEFORE-TABLE` and `AFTER-TABLE` attributes are on the temp-table handles and return a temp-table handle. The `BEFORE-BUFFER` and `AFTER-BUFFER` attributes are on the buffer handles and return a buffer handle.

You can likewise find the first row in the before-table and display the `Price` from both buffers to show that the procedure got the right records, as shown:

```
hBeforeBuf:FIND-FIRST().
MESSAGE "After: " hAfterBuf:BUFFER-FIELD("Price"):BUFFER-VALUE
        "Before:" hBeforeBuf:BUFFER-FIELD("Price"):BUFFER-VALUE.
```

You can save the `updateOrder.p` procedure and run the window to confirm this. Select an `Order`, modify the price of one of its `OrderLines` and click **Save Changes**:



Because there is a static definition of the ProDataSet and its temp-tables, you can access the before-table and its buffer directly by name. This code begins a CASE statement to process different kinds of changes:

```
/* Alternatively you can refer to the before-table and its buffer by
   name because they are statically defined. */
 FOR EACH ttOlineBefore:
   CASE BUFFER ttOlineBefore:ROW-STATE:
     WHEN ROW-MODIFIED THEN
       DO TRANSACTION ON ERROR UNDO, LEAVE:
```

In this example, you will just handle modified rows, not creates or deletes. Defining the scope of the transaction is your responsibility. There are so many different ways in which you might want to handle multiple related changes. You can accept each successful change and reject the ones that fail. You can reject the entire set of updates if anything fails. Or you can define anything in between. In this example each modified row is a separate transaction. This is appropriate if they are independent to the extent that you are not leaving the database in an invalid state if you allow some changes to be committed while returning an error status to be corrected for others.

Also, remember that as always, the default buffer name for a temp-table is the same as the temp-table name. Depending on which one you are referring to, you might need to qualify the reference with the keyword BUFFER or TEMP-TABLE. In this case, the FOR EACH statement always expects a buffer name, so there is no need to qualify the name to tell the AVM that this is a buffer reference. The CASE statement, however, does not know what to expect, so you have to provide an explicit reference to BUFFER ttOlineBefore so that the AVM knows to look for the ROW-STATE attribute on the buffer, not its temp-table. Remember, also, that the keyword ROW-MODIFIED evaluates to the integer value 2, which is the actual value the ROW-STATE attribute returns.

For each modified row, you need to assign the changes back to the database. To demonstrate what the SAVE-ROW-CHANGES method does for you, you can do the same work "by hand" in ABL statements so that you understand all the steps.

First, you need to find and lock the database record that was used to populate the changed row. In this case, the unique key is composed of the OrderNum and LineNum fields:

```
/* This is what SAVE-CHANGES will do for us. */
FIND OrderLine WHERE OrderLine.OrderNum = ttOlineBefore.OrderNum AND
  OrderLine.LineNum = ttOlineBefore.LineNum EXCLUSIVE-LOCK.
```

In the general case, it can be difficult to assemble the proper *where-clause* to retrieve the database record. This is what the SAVE-WHERE-STRING attribute on the Data-Source is for. You can substitute that value, which in this case is the same as the string in the FIND statement above, starting with the keyword WHERE. In order to access the SAVE-WHERE-STRING attribute, you must first attach the Data-Source. You did this at the top of the procedure. With that done, this statement can replace the FIND statement in the last code block:

```
BUFFER OrderLine:FIND-FIRST(DATA-SOURCE srcOline:SAVE-WHERE-STRING(1),
  EXCLUSIVE-LOCK).
```

Do not be confused by these two related Data-Source attributes:

- SAVE-WHERE-STRING is the where-clause needed to retrieve the right database record to match a before-table record that you are using as the basis of an update. This is why it compares the database buffer with the before-table buffer to identify a match. The SAVE-WHERE-STRING attribute requires an argument, which is the index of the database buffer you are trying to retrieve. In this example, OrderLine is the first (and only) database buffer for the Data-Source srcOrder.
- By contrast, FILL-WHERE-STRING is the where-clause needed to retrieve the right child database records for the current parent record when doing an automated FILL. In the *where-clause* for ttOline, for instance, FILL-WHERE-STRING joins the parent ttOrder temp-table to the OrderLine database table.

Use the attribute values such as SAVE-WHERE-STRING and FILL-WHERE-STRING wherever possible to generate ProDataSet-specific code for you. This way your procedures will be more flexible, more reusable, and less prone to error if the underlying table definitions change.

You need to compare the before-table record with what is in the database to make sure no one else changed it since your procedure read it into the original ProDataSet. You can set the ERROR-STRING attribute for the row if there is a conflict. For example:

```
IF NOT BUFFER OrderLine:BUFFER-COMPARE(BUFFER ttOlineBefore:HANDLE) THEN
  BUFFER ttOlineBefore:ERROR-STRING = "Someone else changed it.".
```

If there is no conflict with another change to the same database record, next you must find the after-table row for this change and copy its values into the database:

```
ELSE DO:
  FIND ttOline WHERE ROWID(ttOline) = BUFFER ttOlineBefore:AFTER-ROWID.
  BUFFER OrderLine:BUFFER-COPY(BUFFER ttOline:HANDLE).
```

It is important to examine why we used the BUFFER-COMPARE and BUFFER-COPY methods on the buffer handles here rather than the BUFFER-COMPARE and BUFFER-COPY statements. After all, we are dealing with static buffers, so the statements would have been usable.

The reason, as you should recall, is that the BUFFER-COMPARE and BUFFER-COPY methods have been extended to use the Data-Source field mapping list and field include list when they are used to compare a ProDataSet temp-table buffer to its Data-Source buffer. In this case the definition of ttOline is simple enough that the static statements would have worked correctly. There are no field name changes between OrderLine and ttOline, and no limited list of fields to include in the copy or compare. But in other cases where there is a field mapping or an include field list, the static statements would not work unless you went to the trouble of including the field mapping and include list as options on the static statement. If you use the methods instead, this is done for you.

Use the dynamic BUFFER-COPY and BUFFER-COMPARE methods wherever possible to copy rows into and out of ProDataSet temp-tables. Even if a table definition has no field mapping or include field list, your copy and compare will continue to work without change in the future if the table definitions ever change.

You need to retrieve any changes made by database triggers into the temp-table in preparation for returning it to the caller. To do this, you release the database buffer to force any triggers to fire, and then re-read the record, NO-LOCK this time, and buffer-copy it back into the after-table. You can use the SAVE-WHERE-STRING attribute again to find the record, this time NO-LOCK. For example:

```
/* Force execution of any triggers. */
VALIDATE OrderLine.
BUFFER ttOline:BUFFER-COPY(BUFFER OrderLine:HANDLE).
RELEASE OrderLine.
```

You then end all the procedure blocks with the END statement:

```
      END. /* ELSE DO IF not changed by someone else. */
    END. /* DO WHEN ROW-MODIFIED */
  END CASE.
END. /* FOR EACH ttOlineBefore */
```

6. Go back to the CHOOSE OF BtnSave trigger in the window procedure dsOrderWinUpd.w.

   It receives back the modified ttOline rows as part of the dsOrder INPUT-OUTPUT parameter. You have to merge these final changes back into your original ProDataSet so that they show up in the user interface. The MERGE-CHANGES method can do this for you, but again let us go through the steps in ABL code to confirm what the method will do for you.

7. Create a dynamic query to walk through the after-table for OrderLines in the change ProDataSet, as shown:

   ```
   CREATE QUERY hQuery.
   ```

   Because you used a dynamic ProDataSet to hold the changes by using the CREATE DATASET statement and the CREATE-LIKE method, all references to the ProDataSet need to be dynamic.

8. Add the one temp-table buffer to the query:

   ```
   hQuery:ADD-BUFFER(hDSChanges:GET-BUFFER-HANDLE(2)).
   ```

   You can refer to the buffer in the GET-BUFFER-HANDLE method by position, as in this example, or by name ("ttOline" in this case) as you did in updateOrder.p. If you are writing general-purpose code that needs to be reusable for a variety of ProDataSets, then the position option is more flexible, because it does not hardcode the buffer name into the procedure.

   One other important thing to note here is that you must not refer to the buffer name directly without getting it through its dynamic ProDataSet. That is, you can refer to the buffer name relative to the ProDataSet, like this:

   ```
   hQuery:ADD-BUFFER(hDSChanges:GET-BUFFER-HANDLE("ttOline")).
   ```

But you cannot refer to the buffer name directly in this way:

```
/* Do not refer to the buffer directly in this way: */
hQuery:ADD-BUFFER("ttOline").
```

The reason is that if you do not give the AVM any context for the reference to `ttOline`, it locates the static temp-table definition for `ttOline`, which is part of the original static ProDataSet `dsOrder`. This is the wrong buffer in this case. You must direct the AVM to use the buffer in the dynamic change ProDataSet. This is a result of the fact that it is acceptable to have multiple objects with the same name within a procedure if no more than one is statically defined, but you must refer to the dynamic objects that share the name through their handles or the handles of their parents. So in this example, any unqualified reference to `ttOline` refers to the static temp-table's buffer. Any references to the dynamic `ttOline` must be through a handle.

As these examples show, you will often need to refer to multiple different temp-tables and buffers with the same name when you are working with ProDataSets. Make sure that you properly reference dynamic references so that they point to the proper table or buffer.

If you had used the prefix argument to add a string to the beginning of each temp-table name, then you could safely refer to the buffer by name in an `ADD-BUFFER` method, because the name would be unique.

While we are on the subject of buffer names, there is one more thing you should understand about the buffer names in these dynamic ProDataSets. The before-table and its buffer in any dynamic ProDataSet are given the name `"BI"` plus the after-table name (up to `32` characters). So, for example, the before-table for `ttOline` in a dynamic ProDataSet that you `CREATE-LIKE dsOrder` is named `BIttOline`. If you had specified the prefix argument such as `"chg"` to the `CREATE-LIKE` method, then the before-table name would be `chgBIttOline`.

9. Prepare the dynamic query to walk through the after-table rows, as shown:

```
hQuery:QUERY-PREPARE("FOR EACH " + DSChanges:GET-BUFFER-HANDLE(2):NAME).
```

Here again the statement merits a brief discussion of the alternatives.

The `QUERY-PREPARE` method requires a string that evaluates to `FOR EACH ttOLine`. You could do this just by passing that literal string to `QUERY-PREPARE` directly. In the example above, the reference is more indirect, going through the buffer handle of the ProDataSet. This kind of reference is useful when you want to be able to reuse the same code for potentially different temp-table names.

You might wonder why it would be acceptable to refer to `ttOline` directly in the `QUERY-PREPARE` method, as in `hQuery:QUERY-PREPARE("FOR EACH ttOline")`, when it is not acceptable to refer to this dynamic buffer directly by name in the preceding `ADD-BUFFER` method, as in `hQuery:ADD-BUFFER("ttOline")`. The reason is that the `ADD-BUFFER` method has no context for the name. You could be adding any `ttOline` buffer to this dynamic query. This is why you need to make sure that it uses the right one by referencing it through its parent ProDataSet handle. However, having done this, the `QUERY-PREPARE` method can use the name of the buffer without a problem because it knows that this query is for the particular `ttOline` buffer that was established by the `ADD-BUFFER` method.

10. Open the after-table query and position to the first row in the table, as shown:

```
hQuery:QUERY-OPEN().
hQuery:GET-FIRST().
hBuffer = hQuery:GET-BUFFER-HANDLE().
```

Instead of getting the buffer handle from the query, you could also get it from the ProDataSet reference that was used to build the query. For example:

```
/* Alternative way of getting the query buffer handle. */
hBuffer = hDSChanges:GET-BUFFER-HANDLE(2).
```

Note that in the first instance, GET-BUFFER-HANDLE is operating on the query, which in this case has only one buffer. This makes the buffer number argument optional. The alternative uses the same method name to extract the second buffer handle (for ttOline) relative to the ProDataSet it is a part of.

11. Walk through the rows in the query and locate the corresponding row in the original ttOline table. Buffer-copy the final values to the original ProDataSet table.

    Here is how to do this with specific code to use the key fields from ttOline to identify the appropriate row in the other table:

```
DO WHILE hBuffer:AVAILABLE:
  FIND ttOline WHERE ttOline.OrderNum =
    INTEGER(hBuffer:BUFFER-FIELD("OrderNum"):BUFFER-VALUE)
    AND ttOline.LineNum =
      INTEGER(hBuffer:BUFFER-FIELD("LineNum"):BUFFER-VALUE).
  BUFFER ttOline:BUFFER-COPY(hBuffer).
  hQuery:GET-NEXT().
END. /* DO WHILE AVAILABLE */
```

    This technique has its problems, though. For one thing, it is difficult to generalize. This FIND statement is very specific to the OrderLine table and its key fields. Second, if the key field values have been assigned by the update logic (as is often the case for a newly created record), then the key values will not even match up.

    For this reason, the AVM provides the ORIGIN-ROWID attribute to point directly at the corresponding row in the origin ProDataSet. This value is assigned by the AVM for every modified row when you execute the GET-CHANGES method specifically, so that you can identify the right rows in the origin ProDataSet at the time of a merge. Naturally, you must merge changes back into exactly the same ProDataSet instance as GET-CHANGES was run on. Otherwise, the temp-table ROWID values will not match. The AVM uses the ORIGIN-HANDLE attribute internally when it executes the MERGE-CHANGES method to verify this. You can use ORIGIN-HANDLE yourself if you are doing the work of MERGE-CHANGES in your own ABL code and there is a need to verify that the ProDataSet you are merging changes back into is the appropriate one.

    The ORIGIN-ROWID attribute is set for both the before-table and after-table rows in the change ProDataSet, so that you can start with either one. It holds the ROWID of the before-table row in the origin ProDataSet. If you are applying final changes back to modified or created records, then you need to get to the corresponding after-table row in the origin ProDataSet. This makes identifying the right row a two-step process.

    Here are the statements that can replace the previous FIND statement:

```
/* This is where the ORIGIN-ROWID lets me get to the right ttOline record
   in the base ProDataSet. */
/* FIND ttOline WHERE ttOline.OrderNum =
     INTEGER(hBuffer:BUFFER-FIELD("OrderNum"):BUFFER-VALUE)
     AND ttOline.LineNum =
     INTEGER(hBuffer:BUFFER-FIELD("LineNum"):BUFFER-VALUE). */

BUFFER ttOlineBefore:FIND-BY-ROWID(hBuffer:ORIGIN-ROWID).
BUFFER ttOline:FIND-BY-ROWID(BUFFER ttOlineBefore:AFTER-ROWID).
BUFFER ttOline:BUFFER-COPY(hBuffer).
```

From the after-table buffer for `ttOline` in the change ProDataSet (which is the handle `hBuffer`), you use a dynamic `FIND-BY-ROWID` to position the before-table for `ttOline` in the origin ProDataSet to the before-image of the same row. Then you reference this record's `AFTER-ROWID` to identify and find the after-table row in `ttOline` in the origin ProDataSet. This is the row you want to copy final field changes to. The `BUFFER-COPY` method does this.

This two-step process is unavoidable. First, you need to back out a delete that has failed on the server in the origin ProDataSet. To do so, you must locate the before-table row in the origin ProDataSet to delete it after you re-create the un-deleted row in the after-table. This is all part of what the AVM does for you when you use the `MERGE-CHANGES` method.

12. Synchronize the top-level query (on `ttOrder`), as shown:

```
/* This forces the relation queries to re-open and refresh the browse. */
BUFFER ttOrder:SYNCHRONIZE().
```

This is necessary to force the relation queries to re-open and refresh the browse. Using the preprocessor `{&OPEN-QUERY-OlineBrowse}` would not work because the query the browse is using is the query on the Data-Relation, not the static `ttOline` query generated by the AppBuilder.

13. Use the `ACCEPT-CHANGES` method to clear all the before-table records and to accept the new values in any changed after-table records as the starting point for any further changes. Delete the dynamic change ProDataSet now that you are through with it:

```
hDSOrder:ACCEPT-CHANGES().
DELETE OBJECT hDSChanges.
```

14. Enable the `OrderNum` field again so the user can request another `Order` and disable the **SaveChanges** button until there are more changes to save. Also, turn the `TRACKING-CHANGES` flag back on for `ttOline` to capture any further changes that are made to this `Order` before another `Order` is selected. For example:

```
/* Re-enable the Order Number to select another Order. Also, set
   TRACKING-CHANGES back to TRUE to capture any further changes made to
   this Order. */
ASSIGN
  iOrderNum:SENSITIVE IN FRAME dsFrame = TRUE
  SELF:SENSITIVE                       = FALSE
  TEMP-TABLE ttOline:TRACKING-CHANGES  = TRUE.
```

Because this is the trigger for the button, you can refer to it as `SELF`, and no frame qualifier is needed.

15. Go into the property sheet for **BtnSave** and make it initially disabled. It will be enabled by the code when there are changes to save.

16. In the `ROW-LEAVE` trigger for `OlineBrowse`, enable the **Save** button if the row was modified, as shown:

```
IF OlineBrowse:MODIFIED THEN
  ASSIGN
    INPUT BROWSE OlineBrowse
    {&ENABLED-FIELDS-IN-QUERY-OlineBrowse}
    /* Disable the Order Number until changes are saved. */
    iOrderNum:SENSITIVE IN FRAME dsFrame = FALSE
    BtnSave:SENSITIVE IN FRAME dsFrame   = TRUE.
```

17. In the `LEAVE` trigger for field `iOrderNum`, disable the **Save** button when a new `Order` is selected:

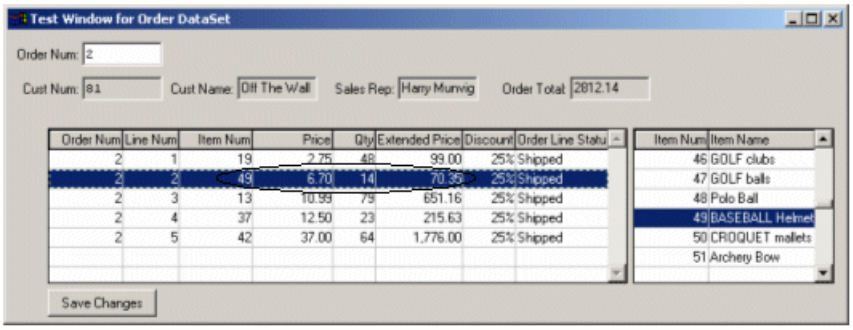
```
ASSIGN
  iCustNum:SCREEN-VALUE     = STRING(ttOrder.CustNum)
  cCustName:SCREEN-VALUE    = ttOrder.CustName
  cRepName:SCREEN-VALUE     = ttOrder.RepName
  dOrderTotal:SCREEN-VALUE = STRING(ttOrder.OrderTotal)
  BtnSave:SENSITIVE         = FALSE.
```

Here the `DO` block around the `ASSIGN` statement scopes the references to the `dsFrame`, so the `IN FRAME` phrase is not needed.

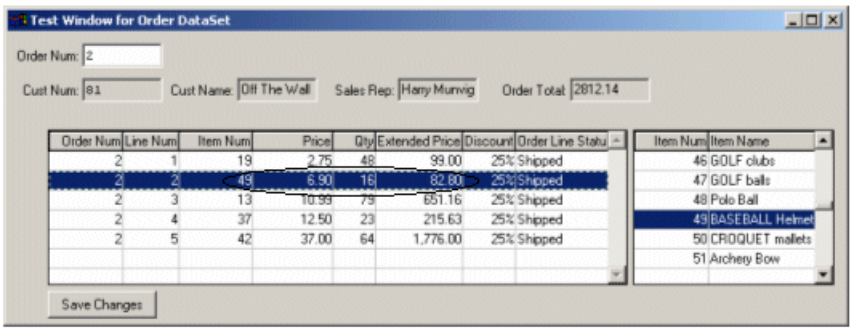
If you save this and rerun the window procedure, you can see the effect of the code that writes the changes back to the database.

18. Select one or more `OrderLines`, change the `Price` (and also `Qty` and `Discount` if you like), and choose the **Save Changes** button.

You see the confirmation that the records were written back to the database, because the `Extended Price` field, which is calculated by a database trigger procedure, it changed to reflect the new `Price`, `Qty`, and `Discount`. For example:



Changing the `Price` and `Qty` of line 2 recalculates the `Extended Price`, as shown:



Using the SAVE-ROW-CHANGES method in the update procedure

If you open up the updateOrder.p procedure again, you can replace the whole CASE statement with the SAVE-ROW-CHANGES method, as shown:

```
FOR EACH ttOlineBefore:
  BUFFER ttOlineBefore:SAVE-ROW-CHANGES().
END.
```

You can save this and rerun the window procedure to confirm that the save works the same as before. SAVE-ROW-CHANGES saves you all the work of the steps you coded individually in the CASE statement. You can still take advantage of the attributes and methods you used if you need to perform some part of the work of SAVE-ROW-CHANGES without using all of what it does. The default transaction handling is to treat each individual row save as its own transaction. If you want a larger transaction scope, you can define the DO TRANSACTION block at the level that is appropriate. Remember that there is no SAVE-CHANGES method to apply all changes in a single method call because there is so much variety both to the order in which the changes should be applied and to the transaction scope used for the changes.

Using the MERGE-CHANGES method in the window procedure

You can replace most of the code that follows the RUNupdateOrder.p statement in the BtnSave CHOOSE trigger with a single call to MERGE-CHANGES. In this case, you can apply all the changes back to the original ProDataSet with a single method call:

1. Comment out or remove all the code in the CHOOSE OF BtnSave trigger starting at the CREATE QUERY statement and ending with the end of the DO WHILE AVAILABLE block.
2. Replace this with the MERGE-CHANGES method on the ProDataSets to merge changes in the change ProDataSet hDSChanges back into the origin ProDataSet whose handle is hDSOrder.
3. Remove the ACCEPT-CHANGES method later in the trigger. MERGE-CHANGES does the work of ACCEPT-CHANGES, or in the case of a failed update, REJECT-CHANGES, as shown:

```
/* MERGE-CHANGES would go here and replaces all this code. */
/* CREATE QUERY hQuery.
. . .
  hQuery:GET-NEXT().
END.  /* END DO WHILE AVAILABLE */
-- end of code replaced by MERGE-CHANGES. */

hDSChanges:MERGE-CHANGES(hDSOrder).
```

MERGE-CHANGES replaces all the code that builds a dynamic query on the change ProDataSet's before-table, walks through it, finds the corresponding rows in the origin ProDataSet table, and copies final field values back into the origin ProDataSet. In addition, if there were any errors, MERGE-CHANGES would reject those changes by restoring the origin ProDataSet rows back to their state before the update was attempted. Using MERGE-CHANGES, you do not need to worry about using ORIGIN-HANDLE and ORIGIN-ROWID, following the chain of ROWIDs to the AFTER-ROWID, and so forth. The attributes are there for you to use only when you need to do something different from what MERGE-CHANGES does for you in a single statement.

Use the high-level methods like SAVE-ROW-CHANGES and MERGE-CHANGES whenever you can to take advantage of all the programming steps they take care of for you. However, don't hesitate to code your own alternatives when the default behavior is not what you need in a particular situation.

Setting and using ERROR, ERROR-STRING, and REJECTED

So far, you have seen how everything works when your updates succeed. In this section, you will build in a simple example of rejecting a change and having that reflected in the user interface. Let us examine the three attributes, ERROR, REJECTED, and ERROR-STRING, that let you log and check the status of your updates.

Note: The ERROR, ERROR-STRING, and REJECTED attributes are effective only if a before-table is NO-UNDO during a Data Source update using the SAVE-ROW-CHANGES( ) method. If the before-table is UNDO, ERROR, ERROR-STRING, and REJECTED attributes are lost when a SAVE-ROW-CHANGES block is undone.

If necessary, you can temporarily apply the NO-UNDO attribute to a temp-table and change it back to UNDO after a SAVE-ROW-CHANGES completes.

ERROR is a logical attribute on a Data-Set, ProDataSet temp-table, and temp-table buffer that indicates whether any errors have occurred in applying changes back to the database.

The attribute is set by ABL when an error is encountered during FILL or SAVE-ROW-CHANGES. The FILL error condition sets ERROR to true for the ProDataSet and temp-table for which the error occurred. SAVE-ROW-CHANGES sets it to true for the ProDataSet and temp-table as well, but also for the particular buffer that failed in the save attempt. This might be because of a unique index violation or other constraint violation that generates a native AVM error, if a trigger procedure for the database table returns error, or if the record was changed by another user.

The ERROR attribute can also be set programmatically to signal an error condition of any kind. If you set ERROR to true for a buffer, it is not automatically set to true for the buffer's temp-table and for the ProDataSet. You can set the attribute at those levels yourself, if you wish. Setting the attribute at the level of a temp-table or the entire ProDataSet allows you to tell immediately whether there is an error anywhere in a set of changes. You can then check each individual row in the updated temp-table to locate one or more specific rows that have errors.

ERROR-STRING is a character attribute on each ProDataSet temp-table and on each temp-table row. This attribute is never set by the AVM because it is not always clear what error message (among several, for example) to store into the attribute. Your program can store a value into ERROR-STRING to return a useful message to the caller. Setting ERROR-STRING is independent of setting the ERROR condition.

REJECTED is a logical attribute for the ProDataSet, temp-table, and temp-table buffer. This is never set by the AVM, but can be set programmatically to indicate that a change was not saved to the database because of an error condition. The AVM does not set the attribute because it is not possible for it to automatically determine the scope of a failed update. This depends on the transaction scoping of the update process when multiple records are involved. The REJECTED attribute can be used to signal to the caller (and most specifically to the MERGE-CHANGES method) which rows were and were not successfully updated, so that the proper adjustments can be made to the client interface or to other logic.

For example, if your transaction scoping is such that a single error causes the whole set of changes to be rejected, then either you or the AVM can set the ERROR status for the one row that actually failed. Because the entire transaction was undone, however, you could set the REJECTED attribute to true for every affected row. In this way, the calling procedure can tell that all the changes were rejected and reset the origin ProDataSet and the user interface accordingly. At the same time, you can identify the single row that actually caused the update to fail, whose ERROR attribute is true, and flag that as the row that needs to be corrected. This is why ERROR and REJECTED are separate attributes.

If you use the MERGE-CHANGES method (or MERGE-ROW-CHANGES), it checks the REJECTED and ERROR attributes for each row and does the merge accordingly. For each row marked REJECTED or ERROR, MERGE-CHANGES effectively does a REJECT-ROW-CHANGES internally to restore the row in the origin ProDataSet to its original values. You can take advantage of this default behavior, or you can use the REJECT-CHANGES and REJECT-ROW-CHANGES methods yourself to get exactly the behavior you require.

In the event of an error, you might not want to use MERGE-CHANGES at all. For example, say the user updates a whole series of OrderLines and saves them. One of the OrderLines causes an error, and the nature of the transaction is that none of the changes are accepted. If you run MERGE-CHANGES on the change ProDataSet when you get it back on the client, it will restore the ttOline table and its browse to the state of the rows before the user made the changes. This does indeed reflect the state of the database, but it is almost certainly not the behavior you want. Now the user has to remake all the changes to all the rows in order to resubmit them for update.

Instead, you might check the ERROR attribute in your own code and use the REJECT-ROW-CHANGES method to restore just those rows that generated errors to their original values, so that just those rows need to be corrected. Or you might not restore any rows to their original values, but simply display the errors and let the user correct them and resubmit the changes. Then you can run MERGE-CHANGES when the updates succeed. This is why these different methods exist, so that you can determine the behavior you want and program accordingly. Do not blindly use the top-level methods like MERGE-CHANGES when they do not give you the behavior you want.

In principle, when you set any of these attributes at the individual row level, you set them and check them on the before-table buffer. However, to provide maximum flexibility, you can set them on either the before-table or after-table buffer and the effect is the same. If you set one of the attributes on the before-table buffer, you can check its value on the corresponding after-table buffer and vice versa. Remember in particular that for deleted rows, there will only be a before-table row that records the delete, so in general you should do your error setting and error checking on the before-table buffers.

As you learned earlier, there is also a DATA-SOURCE-MODIFIED attribute that indicates whether there was a conflict between the before-table values for a row being saved and what is currently in the database, indicating that the database has been changed by another transaction since the ProDataSet was filled. If this attribute is set for any row in a temp-table, it is also set for the temp-table itself, and for the ProDataSet. If a conflict results in a field value or an entire row not being saved, then the AVM sets the ERROR attribute. You can use the DATA-SOURCE-MODIFIED attribute to identify conflicts and flag them in the user interface, whether it is to signal why a change was unsuccessful, or to draw the user's attention to other changes that were either overwritten by the user's changes or combined with the user's changes, when the changes are to different fields.

The ERROR, ERROR-STRING, DATA-SOURCE-MODIFIED, and REJECTED attributes are all cleared for all tables and rows affected by an ACCEPT-CHANGES, REJECT-CHANGES, MERGE-CHANGES, or FILL method on a table or ProDataSet, or for an EMPTY-DATASET method on the ProDataSet or EMPTY-TEMP-TABLE method on a table.

Using the error attributes in the sample procedures

In this section, you will add a simple validation check to the updateOrder.p procedure, set the ERROR and REJECTED flags and the ERROR-STRING, and check for these back in the window procedure.

To add a validation check to the updateOrder.p procedure:

1. In updateOrder.p, retrieve the after-table row for each before-table row in the change ProDataSet. Add this dynamic FIND statement at the top of the FOR EACH ttOlineBefore block:

```
FOR EACH ttOlineBefore:
  /* This code illustrates setting the ERROR status and the REJECTED
     status for a row. */
  BUFFER ttOline:FIND-BY-ROWID(BUFFER ttOlineBefore:AFTER-ROWID).
```

   Remember that even though the change ProDataSet is created as a dynamic ProDataSet in the window procedure, updateOrder.p receives it into a static definition, so you can reference both the before- and after-table buffers by their static names.

2. Now that you have both versions of the row, compare the price and generate an ERROR-STRING if the price has been increased by more than 10 percent. For example:

```
IF ttOline.Price > (ttOlineBefore.Price * 1.1) THEN
  ASSIGN
    DATASET dsorder:ERROR             = TRUE
    BUFFER ttOlineBefore:ERROR        = TRUE
    BUFFER ttOlineBefore:REJECTED     = TRUE
    BUFFER ttOlineBefore:ERROR-STRING = "Line " +
      BUFFER ttOlineBefore:BUFFER-FIELD("LineNum"):STRING-VALUE +
      " price change from " +
      TRIM(BUFFER ttOlineBefore:BUFFER-FIELD("Price"):STRING-VALUE) +
      " to " + TRIM(BUFFER ttOline:BUFFER-FIELD("Price"):STRING-VALUE) +
      " is too high.".
ELSE   /* else SAVE-ROW-CHANGES below */
```

   Here you are first setting the ERROR attribute for the whole change ProDataSet. This makes it easy to determine in the calling procedure if there were any errors.

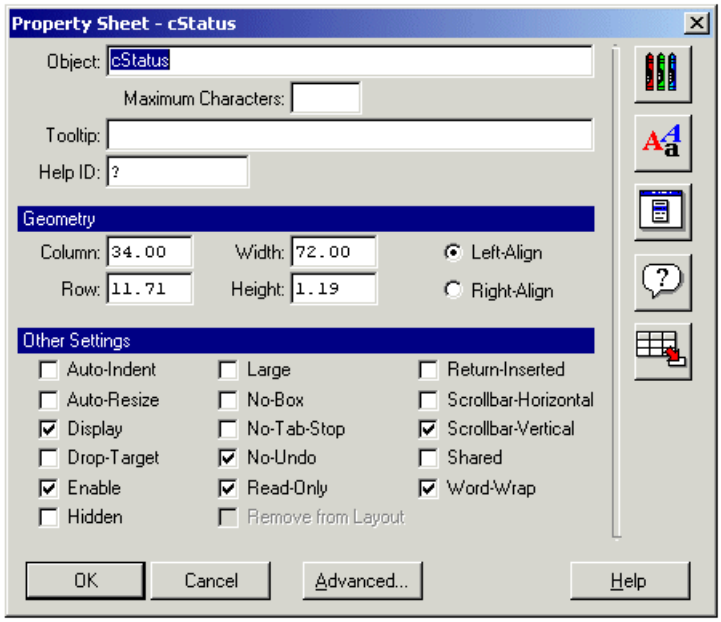
   Next, you set ERROR for the individual buffer along with the REJECTED attribute to signal to MERGE-CHANGES that this update did not make it into the database.

   Then you construct an error message and attach it to the buffer in error. The text of the message, when assembled, will look something like this:

```
Line 5 price change from 34.00 to 44.00 is too high.
```

   If the validation check does not fail, the ELSE keyword invokes the SAVE-ROW-CHANGES method to save the changes to the database. Note that this is one kind of error you can generate, one that is detected by your own code. The SAVE-ROW-CHANGES method itself can also generate errors if a native AVM error results from the change. In this case, the AVM sets ERROR at all levels, but not REJECTED or the ERROR-STRING.

3. Change dsOrderWinUpd.w to check for the error.
4. Add an editor control to the bottom of the window, called cStatus.
5. Give cStatus a vertical scrollbar, but no horizontal scrollbar.
6. Make cStatus enabled, but read-only.
7. Make cStatus tall enough to display one or two rows, as shown:



8. In the `CHOOSE OF BtnSave` trigger, add this block of code after you run `updateOrder.p`:

```
RUN updateOrder.p (INPUT-OUTPUT DATASET-HANDLE hDSChanges BY-REFERENCE).

/* Check the ERROR status that might have been returned. */
cStatus = "".
IF hDSChanges:ERROR THEN DO:
  /* There was an error somewhere in the updates. Find it. */
  CREATE QUERY hQuery.
  hBuffer = hDSChanges:GET-BUFFER-HANDLE(2).
  hQuery:ADD-BUFFER(hBuffer).
  hQuery:QUERY-PREPARE("FOR EACH " + hBuffer:NAME).
  hQuery:QUERY-OPEN().
  hQuery:GET-FIRST().
  DO WHILE NOT hQuery:QUERY-OFF-END:
    IF hBuffer:ERROR THEN
      cStatus = cStatus + hBuffer:ERROR-STRING + CHR(10).
    hQuery:GET-NEXT().
  END.
  hQuery:QUERY-CLOSE().
  DELETE OBJECT hQuery.
END.
DISPLAY cStatus WITH FRAME dsFrame.
/* END of Error status checking. */
```

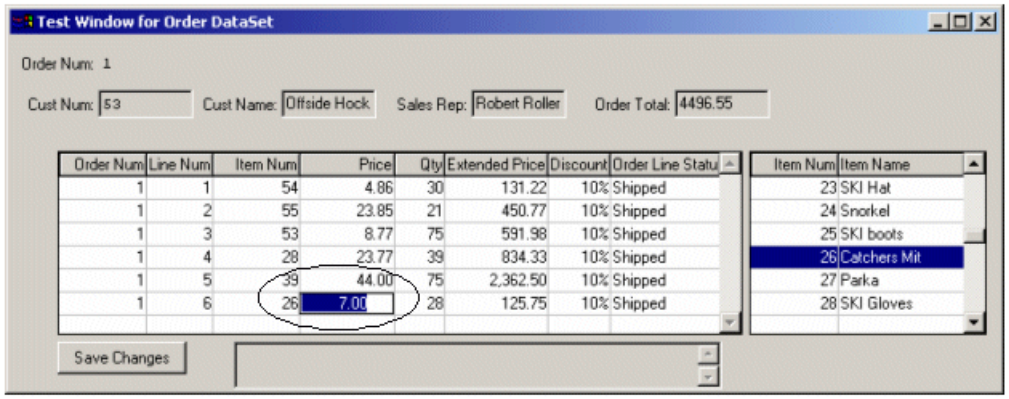
This first checks the `ERROR` attribute on the ProDataSet as a whole. This is why you set this attribute at this level, so that you know at once that there was an error in one of the updates.

To locate each error, you need to create a dynamic query for the `ttOline` buffer, prepare it, open it, and walk through all its rows. For each one with the `ERROR` status, you append the `ERROR-STRING` message to the editor text, with a line feed in between each one.

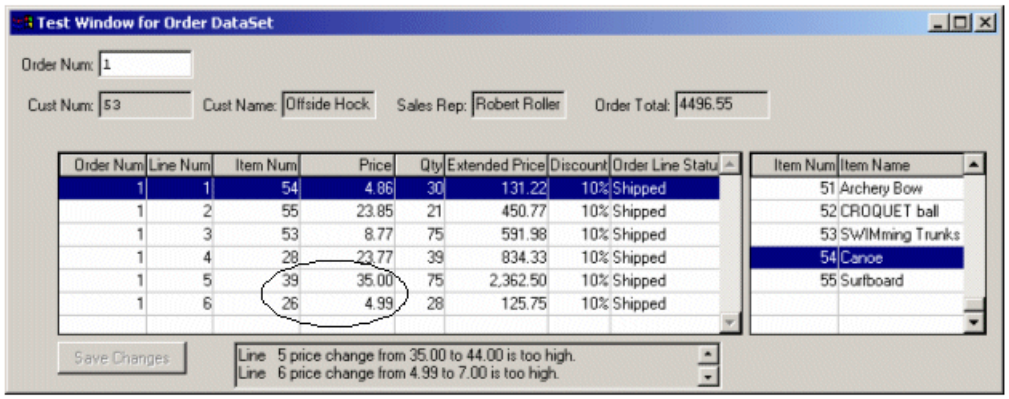
After closing and deleting the query, you display the status string.

Following this, the code already runs `MERGE-CHANGES` on the ProDataSet. When this happens, `MERGE-CHANGES` checks the `REJECTED` attribute of each row and restores the original row to the values it had before the update. Once again, this might not always be the behavior you want, but we use it here to illustrate what `MERGE-CHANGES` does for you.

9. Rerun the window to see the effect of your changes.
10. Select an `Order` and change the `Price` of one or more of its `OrderLines` to be more than 10% higher than before:



11. Select **Save Changes** to try to save these invalid changes:



The error strings are displayed in the status editor, and the changed prices have been rolled back in the `ttOline` table and its browse. Any successful updates would be displayed, along with the updated `Extended Price`, in the browse as well.

ProDataSet change events

There are events that you can define for the process of making local changes to the records in a ProDataSet.

The mechanism for defining change events is exactly the same as for defining FILL events. You use the SET-CALLBACK-PROCEDURE method to associate an internal procedure name and the handle of an active procedure instance containing that internal procedure with a fixed event name. Since these are events that occur when the temp-tables themselves are modified while TRACKING-CHANGES is true, it is reasonable that the event procedure could be located either on the client side or on the server side of a distributed application. This depends on whether the temp-tables are being changed by user actions on the client or by other business logic actions on the server. Note that there are no distributed calls to event procedures. The AVM will not automatically run an event in a server-side procedure from the client. The expectation is that if the temp-table is changed on the client, then the supporting event logic is running on the client. The application can, of course, make its own calls from the client event procedures to procedures on the server, but you must consider the expense of doing this and avoid it wherever possible.

In every case the event procedure receives the ProDataSet object as an INPUT parameter, just as FILL events do. The event procedure can define the parameter as DATASET or as DATASET-HANDLE, depending on whether it has a static definition of the ProDataSet. All of these events are defined on a temp-table buffer, not the ProDataSet itself. That is, the SET-CALLBACK-PROCEDURE method is executed on a temp-table buffer handle.

Event procedures are defined for create and delete events. There is no support for a modify event. This section uses the general term "change statement" to refer to any language statement that causes one of these events. The event procedures all have access to the before images of changed or deleted records using the attributes described earlier.

If you RETURN ERROR in a callback procedure, this raises the AVM error condition as it does elsewhere. You can also set the ERROR-STRING attribute of a row to signal an error internally. In all cases, if a callback procedure attempts to raise error, the ProDataSet handle ERROR attribute is set to true. Each event reacts to the error condition in a slightly different manner. The AVM supports the following events for ProDataSet row-level changes:

- **ROW-CREATE —** This is fired on a CREATE *temp-table* statement, after the record is created in the temp-table. The current buffer for the temp-table is available and contains initial values as defined in the temp-table definition (or inherited from the schema). This event could be used to calculate other initial values for other fields or to make changes to other records (such as to update a child record count in some parent record).

  You could also reject the create by deleting the new temp-table record. This will cause it to be expunged from the before-table as well as the after-table. Any CREATE trigger procedure is executed after this event is handled.

  When an error is raised in a ROW-CREATE callback procedure, usually through RETURN ERROR or a structured UNDO, THROW, error is raised to the caller.

- **ROW-DELETE—** This is fired on a DELETE*temp-table* statement, before the record is deleted. The event procedure could use this event to RETURN NO-APPLY to cancel the delete or to make adjustments to other records based on the delete (such as updating totals in other records). Since the record has not yet been deleted, the record is in the temp-table buffer and the code can look at its values. This event fires only after the AVM has verified that the delete operation is valid, for example, that there is a record in the buffer to delete. Therefore the event code can assume that the delete will go through unless canceled by the event procedure itself, and can take actions based on the record deletion while the record is still there to be looked at. Any DELETE trigger procedure is executed after this event is handled.

  When an error is raised in a ROW-DELETE callback procedure, usually through RETURN ERROR or a structured UNDO, THROW, error is raised to the caller.

- **ROW-UPDATE—** This is fired immediately before the record is updated in the temp-table. It typically occurs when the buffer scope ends, the transaction scope ends, the RELEASE statement or BUFFER-RELEASE( ) method is run on the buffer, or the buffer is needed for another record. The AVM sets the SELF system handle to the handle of the buffer on which the event procedure is running before calling the event handler. If the event handler returns NO-APPLY or ERROR, the return is ignored. The handler has run, and it is too late to undo any changes to the record. You can use this event to determine if and how a record has changed

by reading the buffer in the before-image table (using the SELF:BEFORE-ROWID attribute) and comparing it to the updated buffer. You can also use this event in the event handler to update fields in the record (for example, to supply a calculated field). You cannot read another record into the buffer on which the event procedure is running in the event handler. If you need to read another record, use a different buffer to avoid disturbing the record you are currently updating.

When error is raised in a ROW-UPDATE callback procedure, usually through RETURN ERROR or a structured UNDO, THROW, error is not raised to the caller.

Applying callback procedures programmatically

For both fill-related and change-related events, you might want to have any callback procedures applied programmatically at times other than when the AVM executes them automatically. For example, you might be writing code that populates a ProDataSet without doing an actual FILL method, or you might want to share logic between the FILL and the change process. To make the use of callback procedures as flexible as possible, there is an APPLY-CALLBACK method for use on a ProDataSet temp-table buffer handle, which you can use to make sure that any callback procedure that is defined for the event also executes at other times when you want it to. This spares you from having to write an explicit call that duplicates an event procedure definition.

This is the syntax for the APPLY-CALLBACK method:

Syntax

```
dataset-or-buffer-handle:APPLY-CALLBACK(event-name).
```

The AVM passes the ProDataSet into the callback procedure as a parameter. If there is no callback procedure, then nothing happens. The whole intent is to provide consistent behavior (including no behavior at all if there is no event code).

7

Advanced Events and Attributes

ProDataSets Events on page 79 introduced the `FILL` events you use to extend the default behavior of a ProDataSet or buffer `FILL`. This chapter describes more events and attributes. Some of the techniques shown include examples of:

- Doing a successive `FILL` of a ProDataSet, where an initial `FILL` populates the parent table of a ProDataSet and then later requests fill in more detail for selected top-level rows.
- Activating and deactivating Data-Relations and using `FILL-MODE` to control how much of a ProDataSet is filled at one time.
- Refreshing rows that have already been retrieved by re-reading the Data-Source.
- Using the `OUTPUT APPEND` option on a parameter to bring successively more data over.

For details, see the following topics:

- Query OFF-END Event
- Buffer BATCH-SIZE and LAST-BATCH attributes
- ProDataSet buffer FIND-FAILED event
- SYNCHRONIZE event for a ProDataSet buffer
- Successive loading of ProDataSet data
- Summary

Query OFF-END Event

There has long been an OFF-END GUI event for the ABL browse control. Developers can use this to detect the end of the available data in the query the browse is displaying. For example, you could then retrieve additional batches of data and append them to the rows in the table and re-open the query so that they are added to the browse. The ProDataSet supports an OFF-END event for its temp-table queries. It takes care of this function for you, so that you do not need to code an OFF-END browse trigger block to handle this, or even depend on there being a browse at all. In addition, ABL queries have a QUERY-OFF-END condition you can use to detect the end of the query's data when you are navigating through the data programmatically. The ProDataSet event can respond to this case as well, when there is no browse to trigger a GUI event. Regardless of how the end-of-data condition is detected, the query itself can respond to running out of rows so that an event handler can react appropriately, whether it is to retrieve more data from the server or take other action.

The OFF-END event is available for any query on a ProDataSet temp-table. The OFF-END event occurs when the query is positioned beyond the last row in the query, no matter how this is done. This can be because of a browse scroll to the end of the query, or a GET-NEXT() method or GET NEXT statement on the query beyond the last row.

This event can be attached to the query handle using the same SET-CALLBACK-PROCEDURE method the FILL events use, with this syntax:

Syntax

```
query-handle:SET-CALLBACK-PROCEDURE("OFF-END", event-procedure
                                    [, procedure-handle ] ).
```

Where:

- *query-handle* is the handle of a static or dynamic query.
- *event-procedure* is the name of an internal procedure to run that handles the event.
- *procedure-handle* is a running procedure instance containing that *event-procedure*. As for other events, the default is THIS-PROCEDURE.

In keeping with the calling sequence for the FILL and temp-table change events, the ProDataSet is passed in as an INPUT parameter implicitly BY-REFERENCE, providing the event procedure full access to all the data at no cost (because there is no copying of data involved). And as with other callback events, your code can use the APPLY-CALLBACK method to invoke the event handler programmatically.

A typical use of these events would be to fetch additional batches of data from the server if not all data has been retrieved and sent to the client. The event handler for OFF-END can find the last currently available row and pass its key to the server as a starting point for the next batch. There is an example of using this technique to provide data batching later in Batching Data with ProDataSets on page 205. The event procedure can of course also look at other information in the ProDataSet, including the current row in other tables (so that the query requesting more data could identify the parent for example), and whatever else is helpful.

Note that this event is similar to the existing QUERY-OFF-END query attribute. The difference is that the QUERY-OFF-END attribute is a condition that must be tested at a specific place in the application code, whereas the callback event procedure executes whenever the condition occurs regardless of where it happens in the application code or the user interface. This allows a single event handler to be executed whenever the condition occurs.

There are several pointers on the use of this event:

- The event can be attached only to a query that is on a single buffer for a ProDataSet temp-table; it is not possible to attach the events to a query that involves a join. Because ProDataSet temp-tables can already mask database table joins, this should not normally be a serious restriction.
- The query must be defined as SCROLLING (so that it has an internal result-list maintained by the AVM).
- The query open statement or method should not have the INDEXED-REPOSITION keyword; if present, it is ignored.
- The SET-CALLBACK-PROCEDURE method must come **before** the query is opened for the first time to assure that the event is triggered properly. Once the callback is registered, you can open and close the query as much as you need to and the callback procedure remains attached to the OFF-END event.
- If the event handler is able to add a row or rows to the end of the temp-table, it must RETURN NO-APPLY in order to prevent the QUERY-OFF-END (or browse OFF-END) condition from occurring. If the event handler returns NO-APPLY, the application event or code that triggered the event never even knows that the attempt to keep scrolling through the query initially failed.
- If the event handler is unable to add more rows to the temp-table, it should not RETURN NO-APPLY. That would result in an infinite loop, since the NO-APPLY will prevent the OFF-END condition from happening when it should.
- If you execute a GET LAST statement or GET-LAST() method on the query, the event handler is called repeatedly until it does not return NO-APPLY, signifying that all records have been retrieved. In the case of a very large set of rows this can result in a significant wait while all rows are retrieved. If you need to take advantage of the behavior that INDEXED-REPOSITION provides for you, allowing you to rebuild the query's result-list from the end, for example, so that you can jump directly to the last row, then you cannot use an OFF-END event handler to accomplish this.
- The handler will get an error if it does an OPEN, GET, or REPOSITION on the query itself, since the application is still in the middle of a query operation like NEXT or PREV that is suspended while the event handler executes. Also, any references to a different row in the query's temp-table must be done using a separate buffer.
- References to the ABL SELF keyword in the event handler evaluate to the query handle. From this the code can access the query's buffer if needed, using the construct SELF:GET-BUFFER-HANDLE.
- The query open can have the PRESELECT keyword or a BY clause for sorting, but note that in the case of a PRESELECT or non-indexed sort, the handler will be called repeatedly until all records are read before starting the post-select loop. This is not useful when the event handler is used for batching, but it is supported primarily so that dynamic cases do not have to check for special restrictions on the query.
- The OFF-END event fires before the QUERY-OFF-END condition is set, and before any other condition that signals the end of available data. This means, for example, that if there is an OFF-END event on a query, and the end of the rows in the query is reached because of some action (such as GET-NEXT), then the OFF-END event fires. If that event results in more rows being added to the query, then the QUERY-OFF-END condition does not occur. In this way, a single DO WHILE NOT hQuery:QUERY-OFF-END block, or the act of scrolling down through a browse, can continue until the OFF-END fails to add any new rows to the query.

The shows how to use the OFF-END event to providing batching for an application window when you need to be able to scroll through a large number of rows that cannot be retrieved all at once.

Buffer BATCH-SIZE and LAST-BATCH attributes

If you want to use the OFF-END event to support transparently retrieving successive batches of data from the server, you need to be able to specify how many rows to fill the table with for each batch, and also signal when all the data from the Data-Source has been retrieved.

The BATCH-SIZE attribute on a ProDataSet temp-table buffer lets you determine how many rows to fill at a time. If you set this integer attribute to a non-zero value, then whenever you execute a FILL on that buffer or its ProDataSet, the AVM copies no more than BATCH-SIZE rows from the Data-Source to the temp-table. If the BATCH-SIZE is reached before the end-of-data condition on the Data-Source, the FILL stops for that buffer.

If the end of data is reached before the BATCH-SIZE is exceeded, then the AVM sets the buffer's logical attribute LAST-BATCH to true. If the number of rows remaining to be read exactly matches the value of BATCH-SIZE, the AVM detects this and sets LAST-BATCH even without attempting to position beyond the last row of data. In this way, LAST-BATCH is set correctly when the total number of rows that satisfy the buffer's query is an exact multiple of the BATCH-SIZE.

LAST-BATCH is automatically set during the execution of the FILL method when the BATCH-SIZE on the buffer is non-zero and the end of the data is reached, and is cleared before the start of any FILL or EMPTY actions on the same buffer or ProDataSet. If this default behavior is not satisfactory, it can be set programmatically, right after the FILL or in an AFTER-FILL event. LAST-BATCH is always false when BATCH-SIZE is zero or the Unknown value (?), which means that data batching is not being done.

This attribute is marshaled along with the rest of the ProDataSet definition when the ProDataSet is passed as a parameter to another OpenEdge procedure, whether local or remote. If you define an OFF-END event handler for the buffer in the other procedure, it can check the LAST-BATCH flag for the buffer and determine whether it should make a call to get more rows or if there are no more rows to retrieve. Setting up an event handler for the OFF-END query event on page 208 shows an example of this.

BATCH-SIZE is primarily intended to limit the number of rows added to a top buffer in a ProDataSet, or to a non-top buffer whose parent table normally has only one row. However, it can be set at any level of the hierarchy. The counter used to compare the rows read against the BATCH-SIZE for a buffer is reset for every FILL action. If a buffer is not a top-level buffer in the ProDataSet, then FILL may be called on it many times, once for each parent row. The BATCH-SIZE limit is applied anew each time. For example, if you have ttCust and ttOrder as parent and child, and you put a BATCH-SIZE of 10 on ttOrder, then for each ttCust record that is added to ttCust during the FILL, you can have up to 10 ttOrder records added to the ProDataSet. Thus in a case such as this the BATCH-SIZE on the child table limits the number of child rows for each parent, not the total number of rows for the FILL.

Note: If there are multiple FILLs on the same temp-table, either because FILL is invoked multiple times in succession, or because FILL is invoked on a buffer that is a child of some parent buffer in a ProDataSet and the parent buffer has more than one row, the value of LAST-BATCH will be true if the last invocation of the FILL for that buffer reached the end of data, but false otherwise, and therefore cannot really be relied on. For example, let us go back to the case of a ProDataSet with ttCust and ttOrder tables, and multiple Customers and their Orders being read in the same FILL. If there is a BATCH-SIZE for ttOrder, then within a single ProDataSet FILL, that BATCH-SIZE may be reached for Orders of some Customers but not for others. LAST-BATCH will be true if the last Customer had more Orders than the BATCH-SIZE for ttOrder, and false otherwise. This is why we say that BATCH-SIZE is really intended for use on a top-level buffer, or on a child buffer when there is only one parent being FILLed at a time.

Though the LAST-BATCH attribute is marshaled with the ProDataSet, the BATCH-SIZE is not marshaled as part of a ProDataSet remote or local parameter. This value must be set and kept on the server side. Normally the client will need to know the value of LAST-BATCH, which is marshaled, but not be concerned about the BATCH-SIZE.

NEXT-ROWID attribute

The NEXT-ROWID attribute is of data-type ROWID, and is supported for each buffer in a Data-Source.

NEXT-ROWID is set by the AVM whenever a FILL is done. If the BATCH-SIZE attribute is not set for the Data-Source buffers, or the FILL completes the source query selection without running into a BATCH-SIZE limitation, NEXT-ROWID is set to the Unknown value (?). If the FILL runs into a BATCH-SIZE limitation, NEXT-ROWID is set to the ROWID of the next record in the selection for that source buffer. If there are multiple source buffers for a single ProDataSet row, then the NEXT-ROWID attribute is set for each of them. It is accessed through the Data-Source handle by specifying the buffer sequence number or buffer name, as shown with this syntax:

Syntax

```
data-source-handle:NEXT-ROWID( buffer-sequence-number | buffer-name )
```

The *buffer-sequence-number* is an INTEGER that represents the sequence number of the desired buffer. The *buffer-name* is a CHARACTER expression that evaluates to the name of a buffer in the Data-Source object. If no *buffer-sequence-number* or *buffer-name* is given, the attribute defaults to the first (or only) buffer in the Data-Source.

Like LAST-BATCH, NEXT-ROWID is intended for a temp-table that is either at the top of a ProDataSet relation hierarchy, or has at most one parent record at each level above the buffer that it applies to. This is because the attribute value is set only once for each buffer in each FILL. If a single FILL requires batching for children of multiple different parents, the value of NEXT-ROWID is set for each one in turn, so that the final value reflects only the NEXT-ROWID for the children of the last parent retrieved. If there is a need to capture NEXT-ROWID for multiple sets of dependent records during the FILL, then some other event handler is needed – for example an AFTER-ROW-FILL event procedure for the parent buffer, which can then store NEXT-ROWID into an array or some other field where is can later be retrieved and dealt with programmatically.

The NEXT-ROWID attribute is read-write. The default is the Unknown value (?). NEXT-ROWID is read/write because the application procedure may need to set NEXT-ROWID if the code is not using the FILL method, but instead is reading the source data from a flat file or some other unmanaged data store. Otherwise, this attribute is normally set by the AVM, not by the application.

RESTART-ROWID attribute

The second new attribute is RESTART-ROWID. This is also of data-type ROWID and is supported for each buffer in the Data-Source. RESTART-ROWID is accessed through the Data-Source handle by specifying the database buffer sequence number or buffer name, because in the event of a join, there may be multiple database buffers for a single Data-Source. This is the syntax for RESTART-ROWID:

Syntax

```
data-source-handle:RESTART-ROWID( buffer-sequence-number | buffer-name )
```

The *buffer-sequence-number* is an INTEGER that represents the sequence number of the desired buffer within the Data-Sources buffer list. The *buffer-name* is a CHARACTER expression that evaluates to the name of a buffer in the Data-Source object. If no *buffer-sequence-number* or *buffer-name* is given, the attribute defaults to the first (or only) buffer in the Data-Source.

The RESTART-ROWID attribute is read-write. Unlike NEXT-ROWID, RESTART-ROWID is normally set by application code. The default is the Unknown value (?). If the application sets RESTART-ROWID, any subsequent FILL using that Data-Source opens the Data-Source's query, and then tries to reposition the query to the ROWID given.

When the FILL method is executed on a Data-Source with one buffer, the AVM opens the Data-Source query. Then it checks to see if the RESTART-ROWID has a value other than the Unknown value (?). If it does, the AVM tries to reposition the Data-Source query to that ROWID. If the reposition is successful, the AVM proceeds with the FILL operation. If the reposition is not successful, the AVM puts out an error message, sets the RESTART-ROWID to the Unknown value (?), and continues the FILL from the top of the query at the current level. If the RESTART-ROWID is the Unknown value (?), the AVM proceeds with the FILL operation without repositioning the query.

When the FILL method is executed on a Data-Source with more than one buffer, the AVM opens the Data-Source query. Then it checks the RESTART-ROWIDs beginning with the top-most level of the join (the buffer whose sequence number is 1) and working downward. If the top-most RESTART-ROWID is the Unknown value (?), the AVM proceeds with the FILL operation without repositioning the query.

If the top-most RESTART-ROWID is not the Unknown value (?), the AVM continues to examine the RESTART-ROWIDs in sequence order. As soon as it finds a RESTART-ROWID equal to the Unknown value (?), it stops examining RESTART-ROWIDs and tries to reposition the query with the ROWIDs that it has. If the reposition is successful, the AVM proceeds with the FILL operation. If the reposition is not successful, the AVM puts out an error message, sets the RESTART-ROWID to the Unknown value (?), and continues the FILL from the top of the query at the current level.

If for any reason the specified RESTART-ROWID is not valid (possibly because it belongs to a child of a different parent than the current one), the ROWID is ignored and the FILL starts back at the first row satisfying the selection criteria or the Data-Relation. An error message results but the FILL continues. You can put NO-ERROR on the FILL to suppress the message, or allow it to go harmlessly into the server-side error log.

The NEXT-ROWID attribute is set by the AVM for each Data-Source at the end of a FILL to signal to the application what the ROWID of the next row to be retrieved at that level is. It is the responsibility of the application code to store this value and send it to the client. The client should then pass it back as an INPUT parameter to the next FILL request. The server code supporting the FILL request can then set RESTART-ROWID at that level and do the next FILL.

These two separate attributes are required to avoid situations where there could be conflicts between the AVM setting the attribute value and the application setting the attribute value. NEXT-ROWID is set by the AVM to provide information to the application. RESTART-ROWID is never set by the AVM. If the user sets RESTART-ROWID, it is an instruction to the AVM to start the next FILL at that level.

Neither attribute is marshalled between server and client.

RESTART-ROW attribute

Similar to RESTART-ROWID, the RESTART-ROW attribute of the Data-Source object facilitates batching when filling a ProDataSet temp-table. Unlike RESTART-ROWID, which repositions the query to a ROWID, this attribute allows the FILL query to be repositioned to an absolute row number.

For example, setting this attribute is helpful when you are paging back and forth in a table and want to retrieve page 3. BATCH-SIZE is used to calculate the row where a given page begins, and RESTART-ROW is set to that value just before the FILL. This causes the FILL query to be repositioned to the correct page. For example, to get page 3 you set RESTART-ROW to (3-1) * BATCH-SIZE to fill the third group of records.

Note: When using either the RESTART-ROWID or RESTART-ROW attribute, record sort order must be the same on the client and the server, otherwise the attribute behaves incorrectly or not at all. So, when batching between client and server, it is important that the ordering of records is identical on both the client and the server, and that records being created on the client are added to the end of the table.

ProDataSet buffer FIND-FAILED event

A similar condition to the OFF-END event that happens when you go past the end of a set of rows is when a FIND on a table fails. There is a FIND-FAILED event for a ProDataSet temp-table buffer, which is triggered when any attempt is made to find a row that is not in the table. This can be the result of a FIND statement, a FIND method on a buffer (FIND-FIRST and FIND-UNIQUE), or an action that implicitly finds a row in a ProDataSet temp-table. This is the syntax for the SET-CALLBACK-PROCEDURE method using FIND-FAILED:

Syntax

```
buffer-handle:SET-CALLBACK-PROCEDURE( "FIND-FAILED", event-procedure
                                    [, procedure-handle ] ).
```

If an application procedure is looking for a row that may have been cached already in a ProDataSet, and it is not there, this event gives the procedure the opportunity to suspend the FIND in order to retrieve the row in question, and to do anything else that might be appropriate. So the event procedure could retrieve just the row for which the FIND failed, and add it to the local cache, or it could retrieve a set of related rows, or anything else. The event handler has to be able to determine from the context of the ProDataSet, which as always is passed into it as an INPUT parameter, what the missing data is. For example, if the current row in the ttOrder table as passed in does not yet have its OrderLines, then the code can make a call back to the server to return OrderLines of the current Order and append them to the ProDataSet. However, the event handler does not have direct access to the where-clause used on the FIND that failed.

As with the OFF-END event, it is completely transparent to ABL code that causes the initial failure that the FIND ever failed in the first place. If the event handler is able to add the needed row to the temp-table and RETURN NO-APPLY, the original statement simply succeeds as if the row had always been there.

As with the OFF-END event, this event is supported only for FINDs on ProDataSet temp-table buffers, not other buffers.

FIND-FAILED can be triggered by the static FIND and FIND FIRST statements and the dynamic method equivalents FIND-UNIQUE and FIND-FIRST. FIND-FAILED does not occur for any FIND NEXT or LAST statement or their equivalent GET methods for queries, or for the CAN-FIND function. By contrast, FIND NEXT and FIND LAST conditions that do not yield a row of data are handled by the query OFF-END event.

If a unique find (that is, an unqualified FIND statement or a FIND-UNIQUE method) fails due to ambiguity, the event does not fire, since the problem in that case is not that the row does not exist, but that there are too many matches.

It is important that the event handler does not do any FINDs on the same buffer the event is defined on. Attempting to do this will cause an error.

Setting up an event handler for the FIND-FAILED buffer event on page 210 illustrates how to use the FIND-FAILED event to fill in Item rows in the client-side user interface as they are explicitly referenced for the first time by an OrderLine that uses that Item.

SYNCHRONIZE event for a ProDataSet buffer

There is also a SYNCHRONIZE event for ProDataSet temp-table buffers, which fires under these conditions:

- Whenever the application code does an explicit SYNCHRONIZE on that buffer or a parent of that buffer in the ProDataSet hierarchy that cascades the SYNCHRONIZE through the levels of the ProDataSet
- When there is an explicit browse row selection for a browse on a ProDataSet temp-table, such as when the user clicks on a row
- When the application uses the SYNCHRONIZE() method to force a synchronization of ProDataSet tables
- When the buffer's query is closed

This event allows the procedure code that responds to the event to display buffer values in a frame or take some other action. The event is set using the same SET-CALLBACK-PROCEDURE method and receives the ProDataSet as INPUT as other events do, as shown with this syntax:

Syntax

```
buffer-handle:SET-CALLBACK-PROCEDURE("SYNCHRONIZE", event-procedure
                                     [, procedure-handle ] )
```

The event handler is invoked just before the SYNCHRONIZE behavior occurs, that is, before the current buffer's child relation queries are re-opened. The handler procedure can RETURN NO-APPLY to cancel the effects of the SYNCHRONIZE in the event that it detects that the synchronization should not occur (to avoid the overhead of opening child queries when this is not wanted, for instance).

Successive loading of ProDataSet data

The following procedure updates your code, so that you can do an initial load of header information into a ProDataSet, let the user make a selection from those header rows, and then fill in detail for selected rows.

To update your code:

1. To get started, copy the final version of dsOrderWinUpd.w from Updating Data with ProDataSets on page 135 to a new procedure called PickOrder.w.
2. Delete the Order temp-table fill-ins from the window.
3. Define a browse for ttOrder with the columns OrderNum, CustNum, SalesRep, and OrderDate. Name the new browse OrderBrowse.

   Remember that you can do this in the AppBuilder by selecting the **Temp-Tables** dummy database in the query builder for the browse. Since you copied the procedure from the earlier one, it has all the same temp-table definitions.

4. Define new fill-ins called iCustNum, daOrderDate, and cSalesRep.

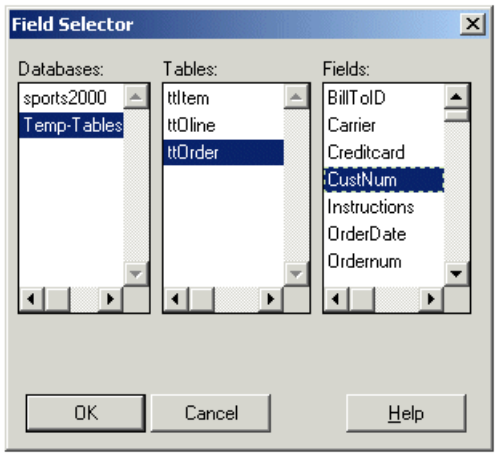
5. Make the initial value of `daOrderDate` blank (using the `Unknown value (?)`) so that it does not display today's date by default.

   An easy way to get the fill-ins to inherit the attributes of the fields they represent in `ttOrder` is to go through these steps for each one.

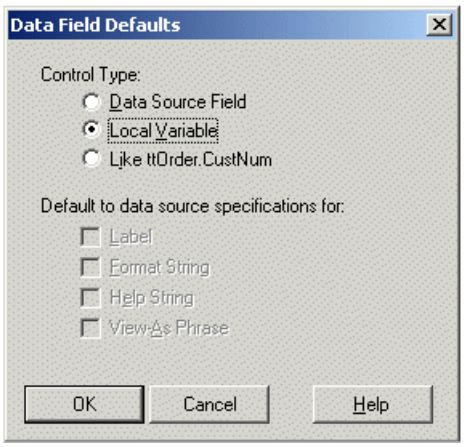
6. Select the fill-in from the **Palette** and drop it onto the design window.

7. Double-click on the fill-in to bring up its property sheet. Choose the **Database Field** button: 

8. From the **Field Selector** dialog box, select the field from the **Temp-Tables** database and `ttOrder` table, as shown:



9. Make the field's **Control Type**`Local Variable`, as shown:



10. Set the **Object name** in the property sheet to the variable name, such as `iCustNum`.

    This creates a local variable definition with the same attributes for label, format, and so forth as the temp-table field, but does not actually define the fill-in as the temp-table field. This is because you will not use these fill-ins to display fields from the current `ttOrder`, but rather to enter filter criteria for retrieving `Order`s through the ProDataSet.

    You will use these fill-ins to allow the user to filter `Order`s by entering a value into one or more of the fields. The window procedure will request a `dsOrder` ProDataSet with all the `Order`s that satisfy the selection, and then allow the user to select an `Order` and fetch `OrderLine` detail for it.

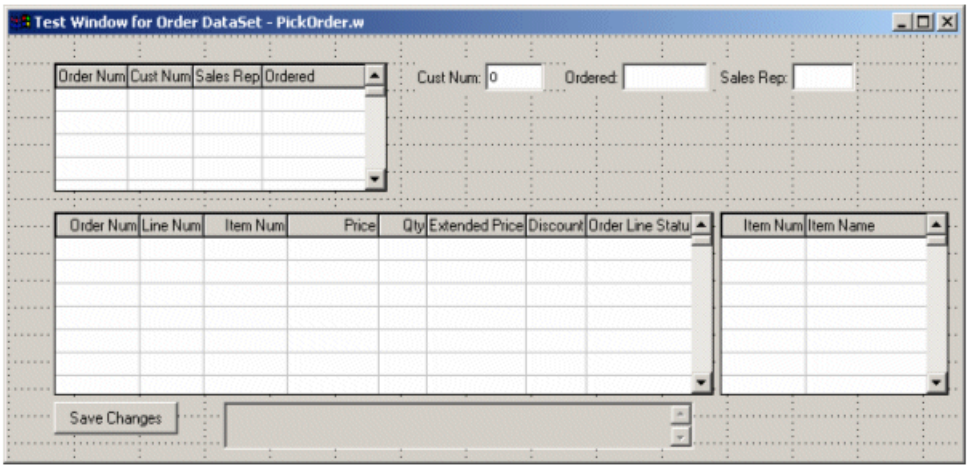
11. Remove all the commented-out code from the `CHOOSE trigger for BtnSave`. Change the statement at the end that re-enabled `iOrderNum` (which is no longer there) to re-enable the three filter fields after a `Save` has been processed, as shown:

```
/* Re-enable the filter fields to select another set of Orders. Also, set
   TRACKING-CHANGES back to TRUE to capture any further changes made to
   this Order. */
ASSIGN
  iCustNum:SENSITIVE IN FRAME dsFrame    = TRUE
  daOrderDate:SENSITIVE IN FRAME dsFrame = TRUE
  cSalesRep:SENSITIVE IN FRAME dsFrame   = TRUE
  SELF:SENSITIVE                         = FALSE
  TEMP-TABLE ttOline:TRACKING-CHANGES    = TRUE.
```

12. Change the `ROW-LEAVE` trigger code for the `OlineBrowse` to change the reference to `iOrderNum` to disable the three filter fields, in the same way as in Step 11:

```
IF OlineBrowse:MODIFIED THEN
  ASSIGN
    INPUT BROWSE OlineBrowse
    {&ENABLED-FIELDS-IN-QUERY-OlineBrowse}
    /* Disable the Order Number until changes are saved. */
     iCustNum:SENSITIVE IN FRAME dsFrame    = FALSE
     daOrderDate:SENSITIVE IN FRAME dsFrame = FALSE
     cSalesRep:SENSITIVE IN FRAME dsFrame   = FALSE
     BtnSave:SENSITIVE IN FRAME dsFrame     = TRUE.
```

At this point the window should look something like this:



Now you are ready to start writing the support logic to retrieve data into the window.

13. In the **Definitions** section, define a handle to hold the procedure handle of the procedure that contains the event logic and other support procedures for the data retrieval. For example:

```
DEFINE VARIABLE hOrderProc AS HANDLE NO-UNDO.
```

14. In the **Main Block**, add a statement to kill this procedure when the window exits. For example:

```
ON CLOSE OF THIS-PROCEDURE DO:
  DELETE PROCEDURE hOrderProc.
  RUN disable_UI.
END.
```

15. Add a statement to start the procedure when the window starts up. For example:

```
MAIN-BLOCK:
DO ON ERROR   UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
   ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
  RUN enable_UI.
  RUN orderSupport.p PERSISTENT SET hOrderProc.
  . . .
```

    The support procedure can be based on the procedure `OrderEvents.p` that you created earlier.

16. Copy `OrderEvents.p` to `OrderSupport.p`.
17. Remove the `INPUT` parameter definitions from `OrderSupport.p`.
18. Add variable definitions for a ProDataSet handle and a string to hold selection criteria. For example:

```
DEFINE VARIABLE cSelection  AS CHARACTER NO-UNDO.
DEFINE VARIABLE hBuff       AS HANDLE    NO-UNDO.
DEFINE VARIABLE hDataSet    AS HANDLE    NO-UNDO.
DEFINE VARIABLE iBuff       AS INTEGER   NO-UNDO.
```

19. Set the handle variable to the `dsOrder` ProDataSet handle and change the `SET-CALLBACK-PROCEDURE` references to the old input parameter `phDataSet` to be `hDataSet`, as shown:

```
hDataSet = DATASET dsOrder:HANDLE.
hDataSet:SET-CALLBACK-PROCEDURE
  ("BEFORE-FILL", "preDataSetFill", THIS-PROCEDURE).
. . .
```

Doing a partial ProDataSet FILL to return Order headers

This code example continues over the following sections, each of which illustrates and explains parts of the overall procedure. When the user enters values for one or more of the filtering fields, the window procedure will format that into a *`where-clause`* and pass that to a new internal procedure called `fetchOrders` that returns the ProDataSet.

Add this code for fetchOrders to OrderSupport.p:

```
PROCEDURE fetchOrders:
  DEFINE INPUT  PARAMETER pcSelection AS CHARACTER NO-UNDO.
  DEFINE OUTPUT PARAMETER DATASET     FOR dsOrder BY-VALUE.

  cSelection = pcSelection.
  hDataSet:EMPTY-DATASET.
  hDataSet:GET-BUFFER-HANDLE(2):FILL-MODE = "NO-FILL".   /* ttOline */
  hDataSet:GET-BUFFER-HANDLE(3):FILL-MODE = "NO-FILL".   /* ttItem  */
  hDataSet:FILL().
END PROCEDURE.
```

This saves off the selection criteria and empties the server-side ProDataSet in preparation for responding to the request.

When the user requests a set of Orders, you do not want to return all the OrderLine detail yet because that can be a lot of data. Instead, you just return the Order headers and wait for the user to select a specific Order to fill in. For this reason fetchOrders needs to set the FILL-MODE attribute for the ttOline and ttItem tables to "NO-FILL." In this way, when you then do a FILL on the ProDataSet, it fills only the ttOrder table and skips the other two tables.

The following four ways change the extent of a FILL, either to limit or to extend the amount of data loaded into the ProDataSet at one time:

- **Setting the FILL-MODE of one or more tables to "NO-FILL"** — When the AVM encounters a NO-FILL table, it does not fill that table and it does not continue down to any children of that table, in effect ending the FILL for that entire branch of the ProDataSet, if there are multiple levels of Data-Relations. In the case of the present example, the ttItem table (which is referenced in the code block above as buffer-handle 3 in the ProDataSet) is the child of ttOline, so ordinarily it would suffice to set ttOline to NO-FILL. But the ProDataSet definition makes the Data-Relation between ttOline and ttItem a REPOSITION relation, which means that at the time of the FILL, ttItem is treated as a top-level table, so that all Items are loaded into the ProDataSet. The REPOSITION relation tells the AVM to automatically select the current Item for each OrderLine in the user interface. For this reason you have to set the FILL-MODE to "NO-FILL" for ttItem, as well as ttOline, to keep the Items from being populated until they are needed.
- **Setting the ACTIVE attribute of one or more relations to FALSE** — Or setting the ProDataSet's RELATIONS-ACTIVE attribute to FALSE, which sets ACTIVE to false for all relations. When you do a FILL on the ProDataSet handle in this case, the AVM does not stop filling children when it encounters a deactivated relation. Instead, it treats every child of a deactivated relation as a top-level table and fills all such tables independently, that is, either with all records from the Data-Source or by using a query if you have defined one for the Data-Source. This method can, therefore, be useful when for some reason you want to define independent queries for a set of tables that are otherwise related, or that should be related after the FILL, when you are navigating the data.
- **Executing the FILL method on a temp-table buffer handle** — Do this rather than on the ProDataSet itself. In this case, the AVM fills only from that table down. This can limit the extent of the fill if there is more than one top-level table in the ProDataSet. Also, when you execute a FILL starting at a specific buffer, the AVM does not treat children of deactivated relations under that buffer as top-level tables. Instead, it simply ends the fill at the level of the parent of the deactivated relation.
- **Setting the MAXIMUM-LEVEL attribute on a Data-Relation handle** — This attribute forces a recursive Data-Relation to stop filling a ProDataSet at a specific number of iterations of a child buffer. When there is a need to return more child records for a parent buffer, the server-side ProDataSet query can be adjusted to return the child records in an OUTPUT APPEND parameter.

These various buffer FILL behaviors are designed to provide you with enough alternatives to satisfy just about any requirement. There is almost always more than one way to get the level of FILL that you want. Do not be unduly confused by all the alternatives. Just pick a way that works for your situation.

To show you an example of one alternative to the one the example uses, the code in fetchOrders could deactivate the first relation in the ProDataSet, between ttOrder and ttOline. In this case, you would have to do the FILL on the ttOrder buffer rather than on the ProDataSet. Otherwise, the ttOrder and ttItem tables would be filled with all the OrderLines and Items in the database, which is not what you want. So the code would look like this:

```
/* Instead of these statements…
hDataSet:GET-BUFFER-HANDLE(2):FILL-MODE = "NO-FILL". /* ttOline */
hDataSet:GET-BUFFER-HANDLE(3):FILL-MODE = "NO-FILL". /* ttItem  */
hDataSet:FILL().
--- you could use these statements to accomplish the same thing: */
hDataSet:GET-RELATION(1):ACTIVE = FALSE.
hDataSet:GET-BUFFER-HANDLE(1):FILL().
```

Note that in this case you do the FILL on the first buffer handle, not on the ProDataSet. Also, you must remember to set the ACTIVE attribute back to true when you need to fill the detail records later on.

Another point to remember in this case is that any event procedures defined at the level of the ProDataSet will not fire when you do a FILL on the ttOrder table. In our example, the support procedure prepares the Order query in the BEFORE-FILL event for the ProDataSet and detaches the Data-Sources in the AFTER-EVENT for the ProDataSet, so this code would need to be moved to the FILL events for the Order table. These are all things to consider when you are designing the structure of your procedures and event handlers.

Forcing the ProDataSet to be passed BY-VALUE

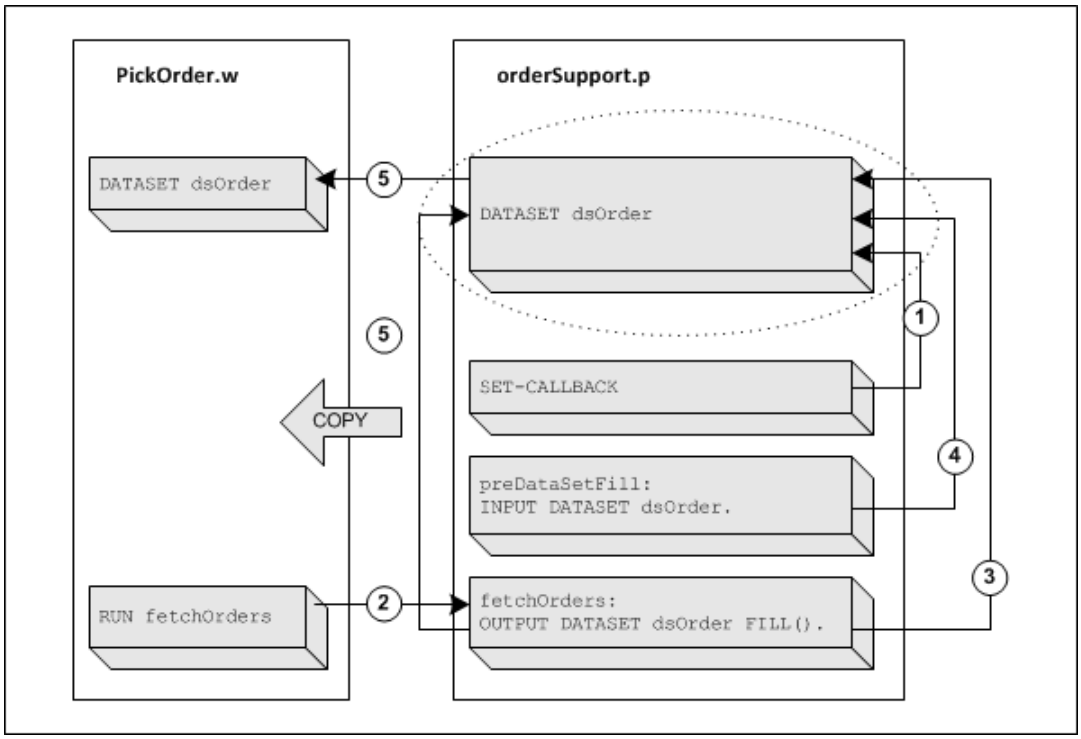
You might notice that there is an extra keyword on the ProDataSet parameter at the top of fetchOrders, as shown:

```
DEFINE OUTPUT PARAMETER DATASET FOR dsOrder BY-VALUE.
```

Why is this here? After all, as you learned earlier, passing ProDataSets by value (that is, by copying the data from one procedure to the other) is the default behavior, and you have to explicitly specify BY-REFERENCE in the RUN statement to pass the ProDataSet by simply pointing the called procedure at the same instance the calling procedure is using.

To answer the question, let us look at a sketch of how the ProDataSet definitions are used in these two procedures, the window procedure that represents the client and the support procedure that represents the server code, as shown in the following figure.

Figure 12: Use of ProDataSet definitions



Here are the steps these procedures go through using the default behavior of passing the ProDataSet by value:

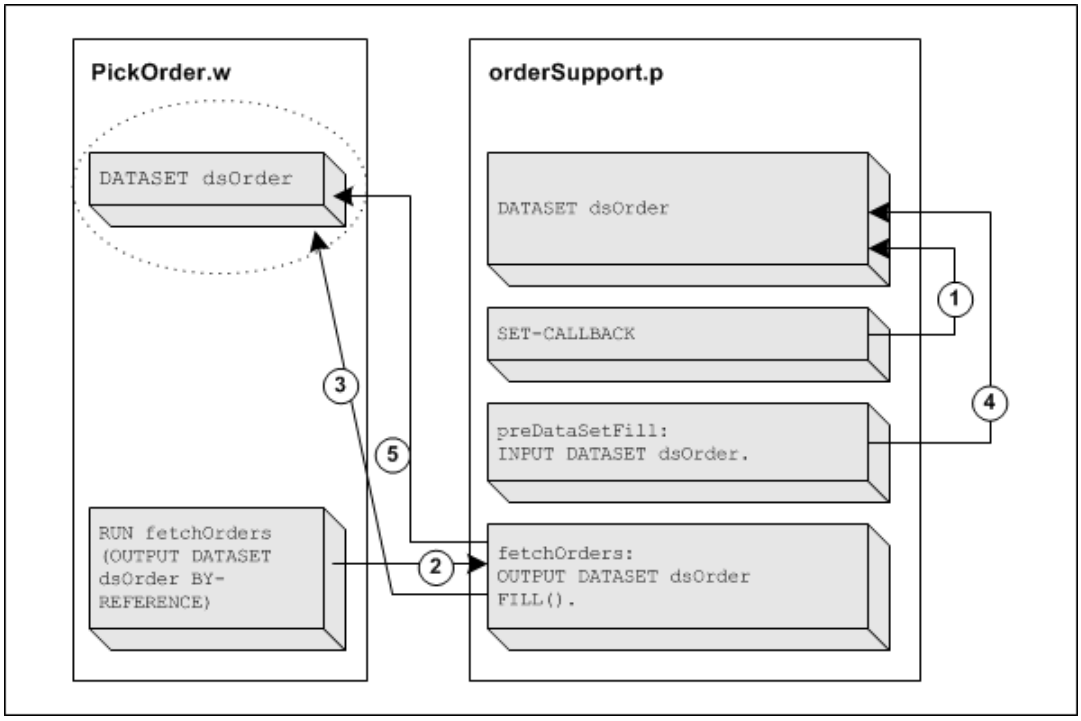
1. The ProDataSet that is going to supply the data to the client is defined in `orderSupport.p`. The `SET-CALLBACK-PROCEDURE` methods that attach its behavior to it all point to that definition by using the local ProDataSet handle or one of its buffer handles.
2. When the client procedure calls `fetchOrders`, the ProDataSet parameter in `fetchOrders`fetchOrders also references the local ProDataSet `dsOrder` whose definition it shares.
3. When `fetchOrders`fetchOrders does a `FILL`, that `FILL` also references the local ProDataSet `dsOrder`.
4. Because the `FILL` callback events have been attached to that local instance of `dsOrder`, they all execute correctly to prepare the query, attach Data-Sources, and take other actions as part of the `FILL`.
5. When `fetchOrders` returns ProDataSet `dsOrder` as `OUTPUT`, it again refers to the local ProDataSet, which is copied back to the window procedure's own instance of ProDataSet `dsOrder`.

In this way, the ProDataSet in (enclosed in the dotted line) maintains its integrity. All the behavior that has been attached to it executes properly. This is always how a procedure call like this operates when the application is actually distributed. If the window procedure truly executes in a client session and the support procedure in an AppServer session, then the ProDataSet is always copied back to the client from the server. Given this expectation, the way you have set up the server procedure is entirely appropriate.

However, you also want to code your procedures so that even when they are all run in the same session, even if only for initial testing purposes, they run correctly. Because ProDataSets are passed by value by default, the default behavior in a strictly local situation is also correct. The potential problem lies in the fact that `orderSupport.p` depends on the ProDataSet always being passed by value.

The following figure shows what happens if the client procedure in the same session decides to ask for the `OUTPUT DATASET dsOrder BY-REFERENCE`.

Figure 13: ProDataSet in an AppServer session



The following describes the execution flow in the above figure:

1. The `SET-CALLBACK-PROCEDURE` methods still act on the ProDataSet instance in `orderSupport.p`.
2. When the window procedure runs `fetchOrders` locally with the ProDataSet passed `BY-REFERENCE`, it is forcing the support procedure to use its instance of `dsOrder` in order to avoid copying the ProDataSet back.
3. Because the AVM adjusts the parameter reference to `dsOrder` in `fetchOrders` to point to the ProDataSet in the caller, the `FILL` is done relative to the ProDataSet in the window procedure `PickOrder.w`.
4. Unfortunately, the callback procedures are still attached to the (now unused) ProDataSet instance in `orderSupport.p`. Therefore they do not run when the `FILL` happens because the `FILL` is on a different instance of the ProDataSet.
5. When `fetchOrders` returns, it does not copy anything back to `PickOrder.w` because the `BY-REFERENCE` qualifier tells the AVM to use the caller's instance of the ProDataSet. Because the fill events never fire to prepare the query and attach the Data-Sources, nothing happened and there is no data on the client.

The design of `orderSupport.p` in this case requires that the ProDataSet be passed by value. For this reason, you can assure that the calls will be made properly by forcing the parameter to be passed by value by using the `BY-VALUE` keyword on the parameter definition in the called procedure. Then everything works properly even if a local caller tries to use `BY-REFERENCE`. The qualifier is simply overridden by the definition in the called procedure, without error.

As with the earlier discussion about when and how to use `BY-REFERENCE`, the implications of its use can seem confusing. The best guideline is to pass ProDataSets `BY-REFERENCE` when you expect that the procedure call will always, or at least sometimes, be made locally (within the same session) in the deployed application (where performance counts), and when you have structured your procedures such that you will not have references to the wrong ProDataSet instance, as could happen in the second example here. Passing a ProDataSet `BY-REFERENCE` can be a valuable optimization, but it is only an optimization, and you must always make sure that sharing the same ProDataSet instance will not have unintended consequences.

Now, let us return to completing the example.

Filtering the top-level query based on the user selection

You will recall from earlier examples that the ProDataSet `BEFORE-FILL` event procedure, called `preDataSetFill`, prepares the top-level query for `Order`s to select the one `OrderNum` passed in as an input parameter. Now you want to use the where-clause stored in the variable `cSelection` instead.

Modify `preDataSetFill` accordingly, as shown:

```
PROCEDURE preDataSetFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE " + cSelection +
    ", FIRST Customer OF Order, FIRST SalesRep OF Order").
END PROCEDURE. /* preDataSetFill */
```

Now the fill will load not just one `Order`, but typically a whole set of `Order`s into the top-level table.

Returning the partial ProDataSet to the client

When the user tabs out of the last of the filter fields, you want to pass the selection to `fetchOrders` and get the ProDataSet with selected `Order`s back.

To return the partial ProDataSet to the client:

1. In the procedure `PickOrder.w`, define this trigger block `ON LEAVE OF cSalesRep`:

```
DO:

DEFINE VARIABLE cSelection AS CHARACTER  NO-UNDO.

ASSIGN iCustNum daOrderDate cSalesRep.

  IF iCustNum NE 0 THEN
    cSelection = "CustNum = " + STRING(iCustNum).
  IF daOrderDate NE ? THEN
    cSelection = cSelection +
    (IF cSelection NE "" THEN " AND " ELSE "")+
    "OrderDate = " + QUOTER(daOrderDate).
  IF cSalesRep NE "" THEN
    cSelection = cSelection +
    (IF cSelection NE "" THEN " AND " ELSE "")+
     "SalesRep = " + QUOTER(cSalesRep).

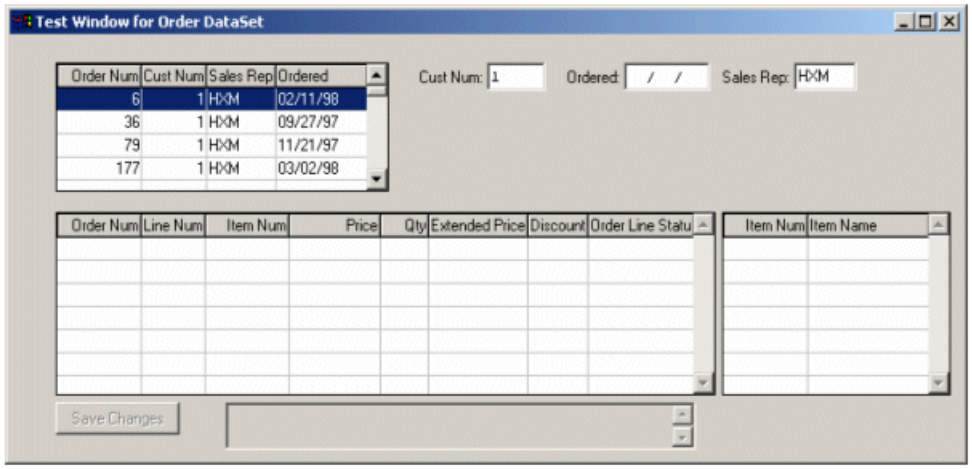
RUN fetchOrders IN hOrderProc (cSelection, OUTPUT DATASET dsOrder).

tnSave:SENSITIVE = FALSE.
   OPEN QUERY OrderBrowse FOR EACH ttOrder.
   DATASET dsOrder:GET-BUFFER-HANDLE(1):SYNCHRONIZE().
END.
```

The code first constructs a where-clause using whichever of the filter fields were filled in, and passes this to `fetchOrders`. It gets the ProDataSet back as an `OUTPUT` parameter. Since the ProDataSet is not passed by reference, any data that comes back replaces whatever might have been in the local ProDataSet before, so there is no need to empty the ProDataSet in advance. If you wanted to have the new set of `Order`s appended to the ones already there, then you could use the `APPEND` option on the parameter to do this.

2. Try running this much of the `PickOrder.w` procedure. Enter some selection criteria for `Order`s and tab through the `SalesRep` field.

   Here, we retrieve all the `Order`s for `Customer 1` and `SalesRep "HXM"`:



   You see that no `OrderLine` or `Item` information came back, because those tables were set to `NO-FILL`.

Retrieving detail for the ProDataSet

Now you want to get the `OrderLines` for a selected `Order`. You start by creating a trigger that fires in response to a double-click on an `Order` in the browse.

To get the `OrderLines` for a selected `Order`:

1. Define a trigger block `ON MOUSE-SELECT-DBLCLICK of OrderBrowse` (this is one of the Portable Mouse Events).

   When the user selects an `Order`, the trigger saves off the `Order` number for later reference. It must then delete the temp-table row for that `Order`. For example:

```
DO:
  DEFINE VARIABLE iOrderNum AS INTEGER NO-UNDO.
  iOrderNum = ttOrder.OrderNum.

  CLOSE QUERY OrderBrowse.
  DELETE ttOrder.
```

Why is this? When you pass the request to the support procedure to fill in the detail, it is going to prepare the top-level `Order` query for that one selected `Order`, set the `FILL-MODE` back to "`APPEND`" for `OrderLines` and `Items`, and then do a `FILL`. That `FILL` will read the `Order` into the top-level table along with its `OrderLines` and all `Items`. When this is returned to the client to be appended to the data already in the client's ProDataSet, that `ttOrder` row is going to be a duplicate of the row already there, and this will cause an error. Remember, when you use the `OUTPUT APPEND` parameter form, new data in each temp-table is always appended to the rows already there. There is no ability to use the `FILL-MODE` to define a different behavior for each temp-table.

Therefore, you must delete rows that will result in duplicates in advance, either on the sending side or the receiving side, before the new data is appended. In this case, deleting the existing row has the advantage of returning the latest field values for the `Order` along with its `OrderLines`, giving you what amounts to a `REFRESH` mode for the `Order`. This, in general, is how you accomplish a refresh of data in a ProDataSet, simply by deleting the rows you want to refresh and then requesting them again.

2. In case the user has selected an `Order` that has been selected before, you delete any `OrderLines` for it so that they are refreshed as well. For example:

```
FOR EACH ttOline WHERE ttOline.OrderNum = iOrderNum:
  DELETE ttOline.
END.
```

3. You run another support procedure in `orderSupport.p`, called `fetchOrderDetail`, passing in the selected `Order Number` and a flag telling the procedure whether `Items` have already been returned or not. If the `Items` are already on the client they do not have to be refilled and passed from the server. The ProDataSet is received in `APPEND` mode so that the `Order` and `OrderLines` that come back are added to what is already in the local ProDataSet, as shown:

```
RUN fetchOrderDetail IN hOrderProc
  (INPUT iOrderNum, INPUT NOT CAN-FIND(FIRST ttItem),
   OUTPUT DATASET dsOrder APPEND).
```

4. To reset the user interface, you need to open the top-level query, reposition the query to the selected `Order` (which repositions the browse as well), and do a `SYNCHRONIZE` to reset the dependent queries for `OrderLines` and `Items` as well. For example:

```
OPEN QUERY OrderBrowse FOR EACH ttOrder.
  FIND ttOrder WHERE ttOrder.OrderNum = iOrderNum.
  REPOSITION OrderBrowse TO ROWID ROWID(ttOrder).
  DATASET dsOrder:GET-BUFFER-HANDLE(1):SYNCHRONIZE().
END.
```

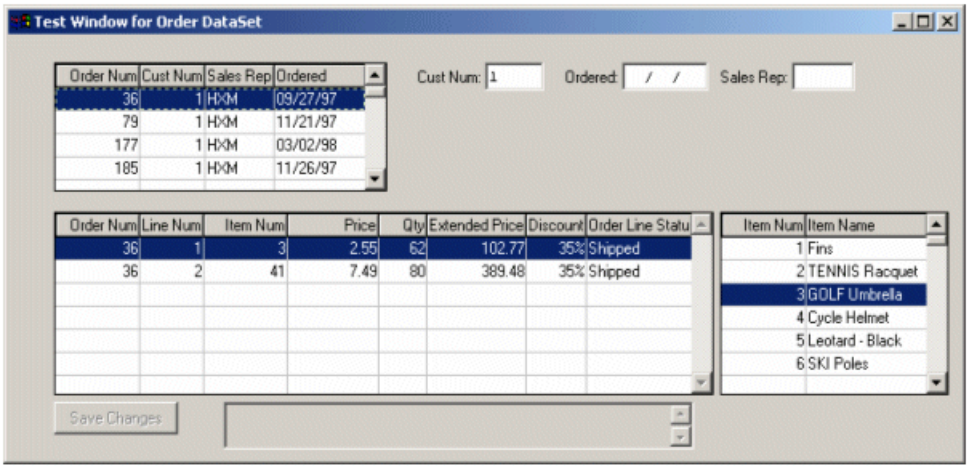
5. In OrderSupport.p, the fetchOrderDetail procedure empties the server-side ProDataSet and resets the selection to fill just the one selected Order. It resets the FILL-MODE for the ttOline table from "NO-FILL" to "APPEND" so that OrderLines are read for the Order, and sets the FILL-MODE for the ttItem table so that Items are filled the first time an Order is selected, and then left out of the fill after that. For example:

```
PROCEDURE fetchOrderDetail:
  DEFINE INPUT  PARAMETER piOrderNum AS INTEGER NO-UNDO.
  DEFINE INPUT  PARAMETER lFillItems AS LOGICAL NO-UNDO.
  DEFINE OUTPUT PARAMETER DATASET    FOR dsOrder BY-VALUE.

  hDataSet:EMPTY-DATASET.
  cSelection = "OrderNum = " + STRING(piOrderNum).
  hDataSet:GET-BUFFER-HANDLE(2):FILL-MODE = "APPEND".  /* ttOline */
  hDataSet:GET-BUFFER-HANDLE(3):FILL-MODE =            /* ttItem  */
    IF lFillItems THEN "APPEND" ELSE "NO-FILL".
  hDataSet:FILL().
END PROCEDURE.
```

6. Make sure that your dsOrder.i include file has the REPOSITION keyword for the LineItem relation so that you can see all the items in the ttItem browse.

7. Run PickOrder.w, select a set of Orders using the filter fields, and then double-click on one of the Orders, as shown:



The first time you double-click on an Order, its OrderLines and all Items are brought over from OrderSupport.p. After that, double-clicking on another Order brings over its OrderLines without refetching the Items. If you simply select an Order without double-clicking on it, this empties the ttOline browse, showing that its OrderLines are not yet part of the client's ProDataSet.

COPY-DATASET and COPY-TEMP-TABLE methods

So far this example has shown you how to append additional data to a single ProDataSet, and to control which parts of it get filled in at different times. In this section, you will learn about two additional ABL methods that can assist you when you need to copy or combine data in other ways.

The COPY-DATASET method copies all the data in one ProDataSet to another, with this syntax:

Syntax

```
target-dataset-handle:COPY-DATASET(source-dataset-handle [, append-mode ]).
```

The *source-dataset-handle* must be the handle of an existing static or dynamic ProDataSet. The *target-dataset-handle* can be the handle of another static or dynamic ProDataSet with the same temp-table and Data-Relation structure. It can also be a newly created dynamic ProDataSet with only a handle and no structure, that is, the result of executing the statement CREATE DATASET*target-dataset-handle*.

If the *target-dataset* has no structure, the table and Data-Relation structure of the *source-dataset* is copied to it along with the data. In this case the dynamic target ProDataSet is not given a name, but its temp-tables and their buffers are given the name "cpy_" plus the name of the temp-table or buffer in the source ProDataSet. This naming scheme can be overridden with your own prefix using the optional *name-prefix* argument, which is discussed in the .

COPY-DATASET copies the temp-tables from source to target in the order that they were defined or added in the two ProDataSets. As soon as it encounters a pair that do not have compatible table definitions, it puts out an error message and terminates the operation. As with other operations, two temp-table definitions are considered *compatible* if they contain the same number of fields with matching datatypes and extents for each field. Whether the field names are identical in the two tables does not matter. If either the source or target ProDataSet has an extra temp-table or temp-tables at the end of the ProDataSet, the COPY-DATASET succeeds because the unpaired tables are ignored.

By default, the tables in the target ProDataSet are emptied at the start of the COPY-DATASET operation. The end result is that the target ProDataSet temp-tables have the same rows as in the source ProDataSet. If you instead want to merge the data in the two ProDataSets, you include the optional second *merge-flag* parameter on the method call. If this logical parameter is present and evaluates to true, then data from the source ProDataSet temp-tables is merged into the target temp-tables according to the rules of the "MERGE" value for a temp-table buffer's FILL-MODE during a FILL. That is, rows from the two corresponding tables in each pair are combined so that the target ends up with all the rows that were in either the source or target at the start. If there is a unique index on the target temp-table, all rows from the source that violate that index constraint are silently eliminated during the COPY-DATASET operation, so that the target winds up with only one row for each index value. Note that there is no interleaving of data in related tables as it is copied from the source ProDataSet to the target. Each temp-table is copied individually in its entirety, starting at the top of the ProDataSet hierarchy. If duplicate rows are eliminated from a parent table in the course of the copy, this has no effect on the related rows in any child table.

COPY-DATASET has five additional positional, optional arguments. The *replace-mode*, *loose-copy-mode* and *current-only* options are LOGICAL datatypes, and the *pairs-list* and *name-prefix* options are character expressions. The full syntax of COPY-DATASET is:

Syntax

```
target-dataset-handle:COPY-DATASET( source-dataset-handle [ , append-mode ]
  [ , replace-mode ] [ , loose-copy-mode ] [ , pairs-list ]
  [ , current-only ] [ , name-prefix ]).
```

The COPY-TEMP-TABLE method works similarly for individual temp-tables. This is the syntax for COPY-TEMP-TABLE:

Syntax

```
target-table-handle:COPY-TEMP-TABLE(source-table-handle [ , append-mode ]
  [ , replace-mode ] [ , loose-copy-mode ] [ , name-prefix ]).
```

You can use the COPY-TEMP-TABLE method on any source or target temp-table, whether it is part of a ProDataSet or not. Therefore you can use this method to copy individual temp-tables from one ProDataSet to another, or for other temp-tables having nothing to do with ProDataSets.

As with COPY-DATASET, by default the target temp-table is emptied at the start of the operation. The rules for temp-table compatibility, the use of the optional merge-flag, and so on, are the same as for COPY-DATASET.

Replace-mode argument

The *replace-mode* option defaults to FALSE. If *replace-mode* is true, the AVM tries to find the corresponding row in the target table, through the target's unique primary index. If a corresponding row is not found in the target table, the AVM uses the source row to create a new target row. If the AVM finds the row in the target, it copies the source row values into that row. If the target row has a BEFORE-TABLE row, that row is left in place. There must be a unique primary index on each target member table in order for the *replace-mode* option to be used.

When *replace-mode* is true, it does not matter if *append-mode* is true or FALSE. The target table is not emptied prior to the copy operation.

Note: If the source or target (but not both) ProDataSet has any before-tables that have any rows in them, you can use COPY-DATASET on that ProDataSet. However, if *append mode* or *replace mode* is specified, the target ProDataSet may not have before-tables. These same rules apply to copying temp-tables using the COPY-TEMP-TABLE method.

Loose-copy-mode argument

By default, pairs of source and target temp-tables being copied by the COPY-DATASET or COPY-TEMP-TABLE method must have compatible definitions. The *loose-copy-mode* option overrides this default.

For the COPY-DATASET method, if *loose-copy-mode* is true, the parameter applies to each temp-table buffer in the ProDataSet. Thus, the remainder of this paragraph applies to both the COPY-DATASET and COPY-TEMP-TABLE methods. If *loose-copy-mode* is true for a temp-table buffer, the AVM effectively performs a BUFFER-COPY from source to target. BUFFER-COPY is more flexible than the default behavior of COPY-TEMP-TABLE or COPY-DATASET because it matches each source field by name to its corresponding field in the target, and ignores fields that do not match, without error. This is somewhat more expensive than the default behavior of COPY-DATASET and COPY-TEMP-TABLE, which assumes compatible definitions field-for-field between source and target. When *loose-copy-mode* is true, the AVM checks for an existing field-mapping between the two buffers specified in a previous ATTACH-DATA-SOURCE operation. If the source temp-table is in fact the Data-Source for the target, then its field-mapping is used to match up field pairs with differing names, just as for BUFFER-COPY. If there is no ATTACH-DATA-SOURCE between the two buffers, the copy is done by matching field names and only copying those fields that appear in both source and target. As with BUFFER-COPY, different field order and missing fields do not prevent the method from succeeding. There is also no requirement that indexes on the two temp-tables be identical.

Pairs-list argument

The *pairs-list* argument only applies to COPY-DATASET. The *pairs-list* is a comma-separated list of target and source tables to be copied, such as: targetT1,sourceT1,targetT2,sourceT2. If the *pairs-list* is present, the copy is done between only those member tables given in the *pairs-list*, by matching target and source table names. If the *pairs-list* is not given, the tables are copied in the order in which they were defined or added in the two ProDataSets.

If one ProDataSet has more tables than the other, the extra unpaired tables are ignored. In other words, you can use COPY-DATASET without the *pairs-list* option to copy one ProDataSet to another, even when one ProDataSet has more tables than the other, as long as the tables to be copied have the same relative positions within the ProDataSet structure. This is the same as the order in which the tables would be returned by the GET-BUFFER-HANDLE(i) method.

If the tables to be copied are not in the same relative order, then you must specify the *pairs-list* option to tell the AVM which source tables go with which targets. If the source and target tables do not match up in their relative order within their respective ProDataSets, you must specify the *pairs-list* even if the table names are the same.

For example, given a source ProDataSet with temp-tables T1, T2, and T3, and a target ProDataSet with just tables T1 and T3, you must define the *pairs-list* option to be T1,T1,T3,T3, because the relative position of T3 in the two ProDataSets is not the same, even though the table names are the same.

It is also worth noting that for static temp-table definitions, it is not possible for the source and target temp-tables to have the same name anyway, since each static name defines a single unique instance with that name.

Current-only argument

The *current-only* argument also applies only to COPY-DATASET. If *current-only* is present and is true, it tells the AVM to copy only the current buffer contents of each source ProDataSet buffer to the target ProDataSet. This option can be useful in situations where the business logic needs to deal with a modified row and its parent (and possibly grand-parent, etc.) without using any other rows. The *current-only* option is a way of getting this minimal set of data to pass to such an operation. Note that you might need to execute the SYNCHRONIZE( ) method to ensure that the contents of each buffer are what is needed. Since the usefulness of the *current-only* option is that it automatically copies a single row at multiple levels at once, it is supported only for COPY-DATASET. For a temp-table, you can use a single BUFFER-COPY statement or method to copy a single row.

Name-prefix argument

Available on both the COPY-DATASET and COPY-TEMP-TABLE methods, the *name-prefix* argument is an optional character expression used as the prefix for naming the target ProDataSet or temp-table, respectively, when the target has as yet no definition. If the parameter is not passed or is passed as the Unknown value (?), the AVM uses the default "cpy_" prefix. If the parameter is passed as an empty string "", then no prefix is added and the target ProDataSet or temp-table will have the same name as the source.

Using COPY-DATASET with a dynamic target ProDataSet

This first COPY-DATASET example shows how to copy a static source ProDataSet to a dynamic target, and have the AVM create the structure of the target before copying into it. The sample procedure is called DynCopy.p. It first includes the same static temp-table and ProDataSet definitions used elsewhere. It then defines variables to hold the handles of the target dynamic ProDataSet, its top-level buffer, and a dynamic query for that buffer, as shown:

```
/* DynCopy.p -- test procedure for COPY-DATASET to a dynamic Target. */
{dsOrderTT.i}
{dsOrder.i}

DEFINE VARIABLE hDataSet2 AS HANDLE NO-UNDO.
DEFINE VARIABLE hQuery2   AS HANDLE NO-UNDO.
DEFINE VARIABLE hBuffer2  AS HANDLE NO-UNDO.
```

The procedure next defines a static query for the Order table, along with the Data-Sources the ProDataSet uses, as shown:

```
DEFINE QUERY qOrder FOR Order.

DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder.
DEFINE DATA-SOURCE srcOline FOR OrderLine.
DEFINE DATA-SOURCE srcItem  FOR ITEM.
```

It then prepares the source ProDataSet query to retrieve Orders for Customer 1, along with their OrderLines and Items, and attaches the Data-Sources. The FILL brings all this data into the source ProDataSet, as shown:

```
QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.custnum = 1").
BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).
BUFFER ttOline:ATTACH-DATA-SOURCE(DATA-SOURCE srcOline:HANDLE).
BUFFER ttItem:ATTACH-DATA-SOURCE(DATA-SOURCE srcItem:HANDLE).
DATASET dsOrder:FILL().
```

A simple DISPLAY loop confirms that the Orders are in the source ProDataSet dsOrder and its ttOrder temp-table, as shown:

```
FOR EACH ttOrder:
  DISPLAY "Original Order: " ttOrder.OrderNum ttOrder.CustNum
    WITH FRAME Order1 20 DOWN.
END.
```

The procedure creates the dynamic ProDataSet using the handle hDataSet2, as shown:

```
CREATE DATASET hDataSet2.
hDataSet2:COPY-DATASET(DATASET dsOrder:HANDLE).
```

This is initially an empty structure for a ProDataSet. It has no table or relation definitions. The COPY-DATASET method copies the table and Data-Relation structure from dsOrder to the new dynamic ProDataSet, and then its data. To verify that both the definition and its data have been copied, the procedure creates a dynamic query for the top-level table in the new ProDataSet and prepares the query to navigate its rows.

As explained above, the COPY-DATASET method gives the new dynamic buffer the name "cpy_" plus the source buffer name, as shown:

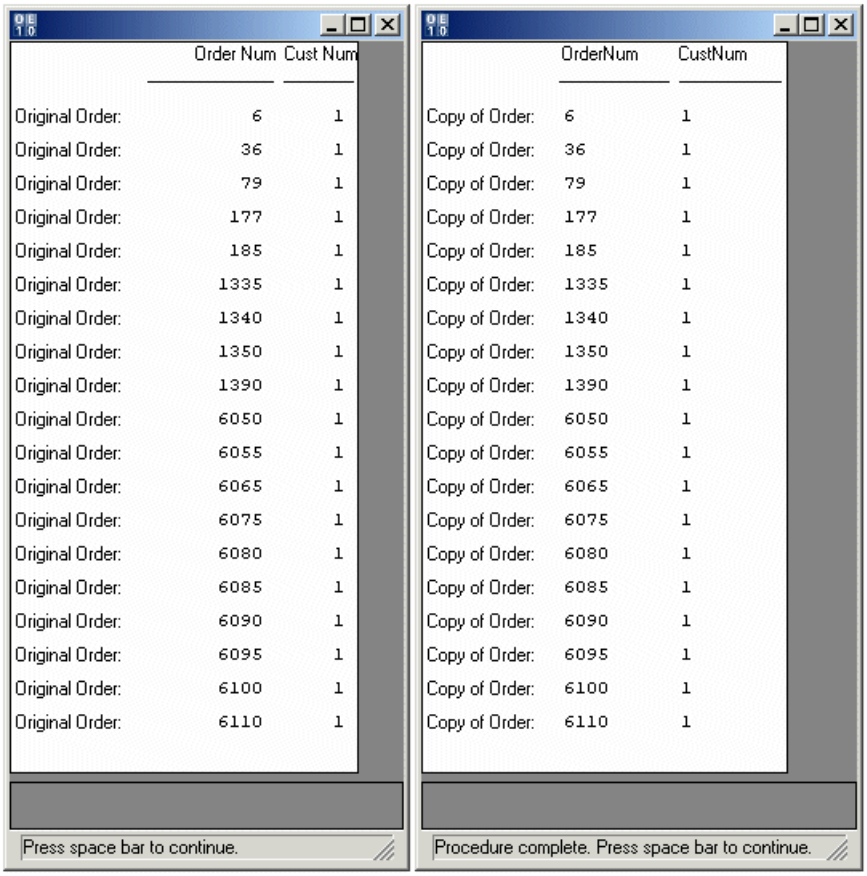
```
CREATE QUERY hQuery2.
hQuery2:ADD-BUFFER(hDataSet2:GET-BUFFER-HANDLE(1)).
/* Note: the buffer name is cpy_ttOrder: */
hQuery2:QUERY-PREPARE("FOR EACH " + hDataSet2:GET-BUFFER-HANDLE(1):NAME).
```

The procedure opens the dynamic query and walks through all the rows at the top level of the target ProDataSet, displaying the same OrderNum and CustNum fields to verify that it contains the same data as the source, as shown:

```
hQuery2:QUERY-OPEN().
hQuery2:GET-FIRST().
hBuffer2 = hQuery2:GET-BUFFER-HANDLE.

DO WHILE NOT hQuery2:QUERY-OFF-END:
  DISPLAY "Copy of Order: "
    hBuffer2:BUFFER-FIELD("OrderNum"):BUFFER-VALUE COLUMN-LABEL "OrderNum"
    hBuffer2:BUFFER-FIELD("CustNum"):BUFFER-VALUE  COLUMN-LABEL "CustNum"
    WITH FRAME Order2 20 DOWN.
  hQuery2:GET-NEXT().
  DOWN WITH FRAME Order2.
END.
```

When you run the procedure it shows you the Orders in both ProDataSets:



Using the COPY-DATASET method for successive FILLs

This second `COPY-DATASET` example provides a variation on the earlier `PickOrder` procedure. As it stands, `PickOrder.w` and its support procedure `OrderSupport.p` retrieve a set of `Order` headers, and then in follow-on calls, the `OrderLines` for those `Order`s. All the items are retrieved along with the first set of `OrderLines`. When you select a different set of `Order`s, the target ProDataSet is emptied and you start over again.

To show a possible use for `COPY-DATASET`, you start by making some changes to the way Pick`Order` works, which will cause a problem that must be corrected.

To update your code:

1. Create a copy of `PickOrder.w` called `PickOrderCopy.w`.
2. Create a variation on `dsOrder.i` called `dsOrderNoRepos.i`, which does not have the `REPOSITION` keyword on the `Item` relation, as shown:

```
/* dsOrderNoRepos.i -- include file definition of DATASET dsOrder with
   no REPOSITION qualifier. */
DEFINE DATASET dsOrder FOR ttOrder, ttOline, ttItem
  DATA-RELATION OrderLine FOR ttOrder, ttOline
    RELATION-FIELDS (OrderNum, OrderNum)
  DATA-RELATION LineItem FOR ttOline, ttItem
    RELATION-FIELDS (ItemNum, ItemNum).
```

This will be used in the OrderSupport procedure when it fills the ProDataSet, so that rather than retrieving all Items regardless of their relationship to the OrderLines (which is what REPOSITION does on a FILL), it will retrieve only those Items that match one of the OrderLines being filled at the same time.

3. Create a copy of OrderSupport.p called OrderSupportCopy.p. Include dsOrderNoRepos.i in place of dsOrder.i, as shown:

```
/* OrderSupportCopy.p -- FILL events for OrderDset.p */
{dsOrderTT.i}
{dsOrderNoRepos.i}
```

4. Edit the internal procedure fetchOrderDetail in OrderSupportCopy.p to eliminate the second parameter, which is the flag indicating whether Items have been retrieved already or not. In this variation, you will always retrieve Items for the current OrderLines (and only those Items). The FILL-MODE for the ttItem table needs to be changed to MERGE. APPEND mode is correct for the ttOline table, because there will not be any duplicate OrderLines for a given set of Orders (although MERGE mode would work just as well). MERGE mode is needed for the ttItem table in case the same Item is used in more than one OrderLine.

   MERGE mode eliminates the duplicates. For example:

```
PROCEDURE fetchOrderDetail:
  DEFINE INPUT  PARAMETER piOrderNum  AS INTEGER    NO-UNDO.
   /* DEFINE INPUT  PARAMETER lFillItems  AS LOGICAL    NO-UNDO. */
   DEFINE OUTPUT PARAMETER DATASET FOR dsOrder BY-VALUE.

  hDataSet:EMPTY-DATASET.
  cSelection = "OrderNum = " + STRING(piOrderNum).
  /* We need to reset the FILL-MODE from its initial setting of
     NO-FILL for these two tables. */
  hDataSet:GET-BUFFER-HANDLE(2):FILL-MODE = "APPEND".  /* ttOline */
  hDataSet:GET-BUFFER-HANDLE(3):FILL-MODE = "MERGE".   /* ttItem  */
  /* IF lFillItems THEN "APPEND" ELSE "NO-FILL". */
  hDataSet:FILL().

END PROCEDURE.
```

5. Back in PickOrderCopy.w, change the **Main Block** to run OrderSupportCopy.p instead of OrderSupport.p

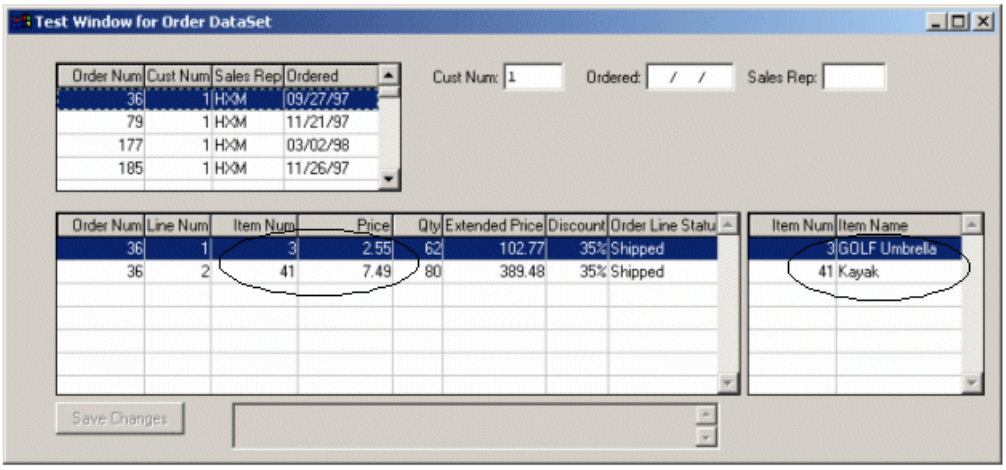
6. Change the SalesRep LEAVE trigger in PickOrderCopy.w to run the new version of fetchOrders, with only two parameters, as shown:

```
RUN fetchOrders IN hOrderProc (INPUT cSelection, OUTPUT DATASET dsOrder).
```

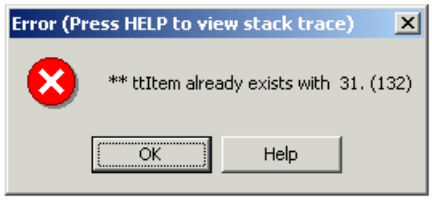
7. Change the MOUSE-SELECT-DBLCLICK trigger for the ttOrder browse, likewise eliminating the second argument to fetchOrderDetail, as shown:

```
RUN fetchOrderDetail IN hOrderProc
  (INPUT iOrderNum, OUTPUT DATASET dsOrder APPEND).
```

8. Now try running the window procedure. Enter `1` for the `Customer` and tab through the other fill-in fields. You will see `Order`s for `Customer 1` as before.
9. Double-click on the second `Order` (number `36`). You see that only its `Items` have been retrieved to be displayed in the `Item` browse:



10. Now double-click on the next `Order`, number `79`. Its `OrderLines` and `Items` are retrieved and added to those already in the window procedure's ProDataSet. So far so good.
11. Next, double-click on the next `Order`, number `177`. The following error appears:



What went wrong? Because the `APPEND` mode on the ProDataSet parameter in the `Order` browse trigger does not eliminate duplicates, the AVM attempted to add an `Item` to the window procedure's `ttItem` table that was already used in another `OrderLine`, and was therefore already there. This violates the unique index on the `ttItem` table, and the AVM complains accordingly.

Since you cannot get the `APPEND` parameter mode to do a merge for you, eliminating duplicates, how can you accomplish this?

One way is to use the `COPY-DATASET` method to combine the data from two ProDataSets, one which is already on the client, and the other which is retrieved into a separate ProDataSet from the support procedure.

12. To do this, insert the contents of the include files `dsOrderTT.i` and `dsOrder.i` into `PickOrderCopy.w`'s definitions section, and edit them to give unique names to the second copy of the ProDataSet and its temp-tables. The temp-tables should be named `ttOrder2`, `ttOline2`, `ttOlineBefore2`, and `ttItem2`. The field and index names can stay the same. The new ProDataSet definition should look like this:

```
DEFINE DATASET dsOrder2 FOR ttOrder2, ttOline2, ttItem2
  DATA-RELATION OrderLine FOR ttOrder2, ttOline2
    RELATION-FIELDS (OrderNum, OrderNum)
  DATA-RELATION LineItem FOR ttOline2, ttItem2
    RELATION-FIELDS (ItemNum, ItemNum).
```

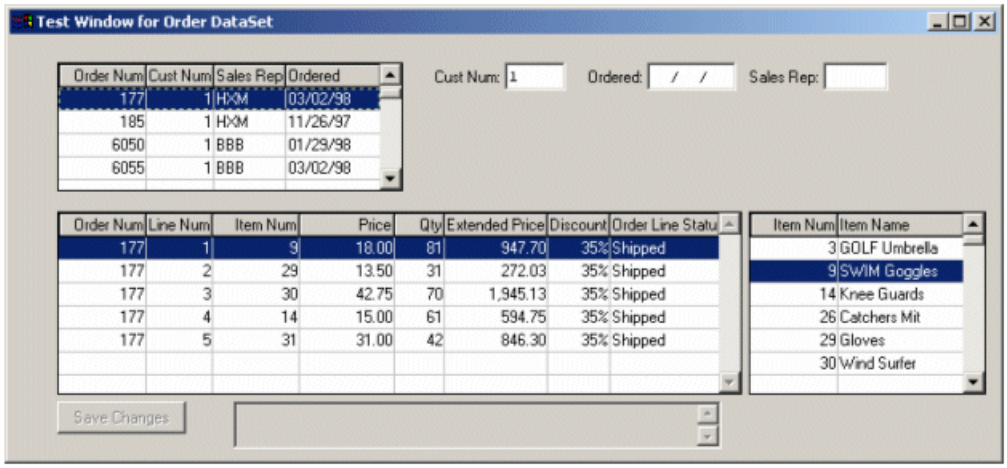
Remember that there is no way to have multiple instances of the same static ProDataSet or temp-table in a single OpenEdge procedure, so the second ProDataSet and its temp-tables need distinct names. You could, of course, also use a dynamic ProDataSet as the second ProDataSet, as shown in the previous example.

13. Now re-edit the `MOUSE-SELECT-DBLCLICK` trigger on the `Order` browse to remove the code that empties the target ProDataSet, and to receive the additional `Order`s, `OrderLines`, and `Items` into a second ProDataSet that is then copied into the first one. For example:

```
/* DELETE ttOrder.
   FOR EACH ttOline WHERE ttOline.OrderNum = iOrderNum:
     DELETE ttOline.
   END. */

RUN fetchOrderDetail IN hOrderProc (iOrderNum, OUTPUT DATASET dsOrder2).
DATASET dsOrder:COPY-DATASET(DATASET dsOrder2:HANDLE, TRUE).
```

The second argument to `COPY-DATASET`, the *`merge-flag`*`TRUE`, tells the AVM to merge data from ProDataSet `dsOrder2` into `dsOrder` rather than emptying `dsOrder` first. Now when you rerun the window, retrieve `Order`s for `Customer 1`, and select `Order`s `36`, `79`, and `177` in turn, the buffer handle is gone, because new data is being merged into the original ProDataSet and duplicates are automatically eliminated. For example:



Summary

The examples showed the following ways to use the ProDataSet to read data flexibly and efficiently:

- Doing successive fills, starting with header information and filling in detail when needed, can be much more efficient than initially filling a ProDataSet with a large amount of data that the user might never need to look at.
- You can populate a portion of a ProDataSet by deactivating relations or setting the `FILL-MODE` of one or more tables to "`NO-FILL`."
- You can refresh data on the client by deleting the rows to be refreshed and then requesting new data from the server.
- You can use the `OUTPUT APPEND` parameter mode to add data to a ProDataSet in multiple requests.

- Alternatively, you can use a second ProDataSet as a parameter and then use the `COPY-DATASET` method to combine data from multiple fills more flexibly.

The next chapter includes an example of how to batch large amounts of data from server to client.

8

Batching Data with ProDataSets

This chapter describes how to batch with a ProDataSet and includes an example of batching data by retrieving subsets of a large number of rows in successive requests.

For details, see the following topics:

- Overview
- Using the include-field list to limit the fields copied into the table
- Setting up an event handler for the OFF-END query event
- Setting up an event handler for the FIND-FAILED buffer event
- Summary

Overview

If you run `PickOrder.w` and tab through all the filter fields without entering a value into one, you will see a small but noticeable delay before the `Order` browse displays the `Order`s. After all, the AVM has to read nearly 4,000 `Order` records from the database, plus their `Customers` and `SalesReps`, create a temp-table record for each of them, and buffer-copy the database records to the temp-table. Then it copies the ProDataSet definition and the entire contents of the `ttOrder` table to the window procedure. Considering the amount of work it is doing, it is pretty amazing that it does not take a lot longer than it does. However, if the number of rows were even larger, or if you were running the support procedure across an AppServer connection on a different machine, the delay would be much greater.

Generally, you should try to avoid giving your users the opportunity to browse through very large numbers of rows on the client, instead prompting them to filter the data in advance as the example window does. However, in some cases you need to move a potentially large number of rows from server to client, and it is often better to do it in batches so the user can see some of the rows before every database record has been read and copied into the temp-table and across to the client.

Note: When batching records, ProDataSet temp-table buffer queries on both the client and server must use a sort that is synchronized. Lacking this sort order agreement, results are not predictable. In addition, records being created on the client must be added to the end of the table.

This section extends the example from Advanced Events and Attributes on page 175 to show you a way of batching records . At the same time we will show you how to limit the number of fields copied into the temp-table. After all, the window is only showing four fields from the `ttOrder` table, so there is really no point in copying every field into the temp-table, and more significantly, passing all those field values across to the client where they will never be seen or used.

To update the code:

1. To get started, copy the `PickOrder.w` procedure to `PickOrderBatch.w`, and the `OrderSupport.p` to `OrderSupportBatch.p`.
2. Change the `RUN` statement in the **Main Block** of `PickOrderBatch.w` to start `OrderSupportBatch.p`.
3. In the `LEAVE` trigger for `cSalesRep` in `PickOrderBatch.w`, change the `RUN` statement so that you run a different support procedure if the user does not enter a value into any of the filter fields, as shown:

```
/* There were selection criteria */
IF cSelection NE "" THEN
  RUN fetchOrders IN hOrderProc (cSelection, OUTPUT DATASET dsOrder).
/* No selection so retrieve (the first) batch of rows, starting at the
   first order. */
ELSE
  RUN fetchOrderBatch IN hOrderProc
    (0, "OrderNum,CustNum,SalesRep,OrderDate", OUTPUT DATASET dsOrder).
```

   The new procedure `fetchOrderBatch` takes an `Order Number` to start with, and a list of fields to populate the temp-table. Since you want to start fresh when the user tabs out of the `SalesRep`, you just pass `0` as the starting point in the `Order` table. The four fields in the second parameter are the fields the browse uses; those are the only ones you need values for on the client.

   The `OUTPUT` parameter is the same ProDataSet as before.

4. Switch over to `OrderSupportBatch.p`. First you need a new definition at the top, for example:

```
DEFINE VARIABLE cFieldList AS CHARACTER NO-UNDO.
```

   The `cFieldList` is the list of fields to include in the `ttOrder` table, passed over to `fetchOrderBatch`.

5. Write the `fetchOrderBatch` procedure. It needs the three parameters you saw in the `SalesRep` trigger. For example:

```
PROCEDURE fetchOrderBatch:
  DEFINE INPUT  PARAMETER piLastOrder AS INTEGER   NO-UNDO.
  DEFINE INPUT  PARAMETER pcFieldList AS CHARACTER NO-UNDO.
  DEFINE OUTPUT PARAMETER DATASET     FOR dsOrder BY-VALUE.
```

   The selection that becomes the *where-clause* for the `Order` query needs to start with the first `Order Number` greater than the one passed in. For the call in the `SalesRep` trigger, this is the first `Order` in the database. In later calls, the `INPUT` parameter will be the highest `Order Number` retrieved so far.

   The `pcFieldList` parameter passed in is saved in the variable `cFieldList`, which can be seen throughout the procedure. For example:

```
ASSIGN
  cSelection = "OrderNum > " + STRING(piLastOrder)
  cFieldList = pcFieldList.
```

6. You use the `BATCH-SIZE` attribute on the `ttOrder` buffer to tell the AVM to fill only a maximum of 20 rows into the `ttOrder` temp-table at a time, as shown:

```
hDataSet:GET-BUFFER-HANDLE(1):BATCH-SIZE = 20.
```

7. The next four lines are the same as in `fetchOrder`, and you can copy them from there:

```
hDataSet:EMPTY-DATASET.
hDataSet:GET-BUFFER-HANDLE(2):FILL-MODE = "NO-FILL". /* ttOline */
hDataSet:GET-BUFFER-HANDLE(3):FILL-MODE = "NO-FILL". /* ttItem  */
hDataSet:FILL().
```

Using the include-field list to limit the fields copied into the table

You must make use of the field list passed in to limit the number of fields that are copied into the `ttOrder` temp-table.

Edit the `ATTACH-DATA-SOURCE` method for the `ttOrder` table in pre`Order`Fill to use `cFieldList` as the value for the include field list.

The first of the optional arguments to ATTACH-DATA-SOURCE following the Data-Source handle is the field mapping for fields whose names are changed. This argument is in the following example:

```
PROCEDURE preOrderFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE,
    "Customer.Name,CustName", ?, cFieldList ).
  BUFFER ttOline:ATTACH-DATA-SOURCE(DATA-SOURCE srcOline:HANDLE).
  BUFFER ttItem:ATTACH-DATA-SOURCE(DATA-SOURCE srcItem:HANDLE).
END PROCEDURE. /* preOrderFill */
```

The second and third optional arguments are a list of fields to exclude from the temp-table and a list of fields to include. You can specify one of these but not both.

Limiting the field values that are buffer-copied speeds up the creation of the temp-table rows somewhat. But the major reason for not filling fields that the client does not need is to cut down on the network traffic, as always anticipating the deployment situation where the support procedure is running on an AppServer and the window on a separate client machine. The field values that are not buffer-copied into the temp-table will be null, blank, or 0, unless the field has another specific initial value. Even though these values do go across to the client, this greatly reduces the number of bytes of data being passed.

Setting up an event handler for the OFF-END query event

Batching is supported by the OFF-END event on the query for the ttOrder temp-table. Whenever the client reaches the end of that query's rows, as the user scrolls down through the Order browse for example, the event allows your code to check whether there are more rows to retrieve.

To set up this event, add this SET-CALLBACK-PROCEDURE method to the LEAVE trigger for the SalesRep field, after getting the initial ProDataSet back from the support procedure:

```
BtnSave:SENSITIVE = FALSE.
/* Set up an OFF-END event handler for the Order buffer to do batching. */
QUERY OrderBrowse:SET-CALLBACK-PROCEDURE("OFF-END","OffEndOrder",
                                         THIS-PROCEDURE).
OPEN QUERY OrderBrowse FOR EACH ttOrder.
```

Code the OffEndOrder internal procedure itself, also in PickOrderBatch.w. Like all event handlers, it receives the ProDataSet as an INPUT parameter by-reference, as shown:

```
/*------------------------------------------------------------------------
  Purpose:   Procedure OffEndOrder handles the OFF-END event on the Order
             query. It asks the support procedure for another batch of rows
             unless the LAST-BATCH has already been returned.
  Parameters:  INPUT DATASET dsOrder
------------------------------------------------------------------------*/
DEFINE INPUT PARAMETER DATASET FOR dsOrder.
```

This procedure needs to check to see whether all the Orders have been retrieved in the window procedure yet. The LAST-BATCH attribute on the ttOrder buffer provides this information. If the last batch of rows has not been returned, it passes the highest OrderNum value received so far to the same fetchOrderBatch routine that brought the first batch of rows over to the SalesRep LEAVE trigger.

Whatever rows come back are appended to what is already in the local ProDataSet, as shown:

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

/* If LAST-BATCH flag doesn't indicate that all rows have been returned, pass
   the current last OrderNum to tell where to start, and get another
   batch. */

IF NOT BUFFER ttOrder:LAST-BATCH THEN DO:
  FIND LAST ttOrder.
  iOrderNum = ttOrder.OrderNum.

  /* Field list (2nd param) only needs to be set once. */
  RUN fetchOrderBatch IN hOrderProc (iOrderNum, "",
    OUTPUT DATASET dsOrder APPEND).
```

After disabling the **Save** button until the user has actually retrieved and made changes to an Order's OrderLines, the procedure simply executes a RETURN NO-APPLY statement to cancel the default effects of the OFF-END event on the query. In the background, the AVM has automatically reopened the OrderBrowse query on the ttOrder table, and repositioned it to the first new row returned. In this way, the rest of the client application does not even know that the event occurred or that the end of the set of rows on the client was temporarily reached. For example:

```
    BtnSave:SENSITIVE IN FRAME dsFrame = FALSE.
    RETURN NO-APPLY.
  END.   /* END DO IF NOT LAST-BATCH */
END PROCEDURE.
```

If the LAST-BATCH flag had already been set when the event occurred, none of the code would be executed and the procedure would return normally. Because there was no RETURN NO-APPLY statement to cancel the OFF-END, the QUERY-OFF-END event and any other related events such as OFF-END on the browse itself would occur.

To see the effects of the new event handler, save both procedures and rerun the window. Tab through the SalesRep field and see the first batch of 20 Orders come up in the Order browse. If you scroll down through those Orders, successive batches are transparently retrieved, added to the temp-table and its query, and therefore displayed in the browse. The browse's own OFF-END GUI event never occurs until the last batch of Orders is retrieved. (If you scroll patiently through all 4000+ Orders, you will see the scrolling eventually end. If you defined an OFF-END trigger for the browse, perhaps just to display a message, you would see that the message does not appear until you reach the end of all the Orders.) The same thing happens if the query's OFF-END event happens for any other reason, such as executing successive GET-NEXT methods on the query.

Setting up an event handler for the FIND-FAILED buffer event

We can extend the example even further to show how to use the FIND-FAILED event. This ProDataSet buffer event is similar to the OFF-END event on a query in that it gives your application the opportunity to retrieve missing data without the rest of the application even being aware that it was not in the ProDataSet in the first place. You can use the event to retrieve needed rows one at a time, or in batches, depending on the situation.

To show how the FIND-FAILED event works:

1. Add a callback for a FIND-FAILED event handler to the LEAVE trigger for SalesRep in PickOrderBatch.w, right after the callback for the OFF-END event. For example:

```
/* Set up an OFF-END event handler for the Order buffer to do batching. */
QUERY OrderBrowse:SET-CALLBACK-PROCEDURE("OFF-END","OffEndOrder",
                                         THIS-PROCEDURE).
/* Also a FIND-FAILED event handler for the Item table. */
BUFFER ttItem:SET-CALLBACK-PROCEDURE("FIND-FAILED","FindFailedItem",
                                     THIS-PROCEDURE).
```

   Note that the OFF-END event must be attached to a query, and the FIND-FAILED event to a ProDataSet buffer, in this case the ttItem buffer, which will allow the procedure to retrieve Items to add to the browse the first time they are referenced.

2. Change the call to fetchOrderDetail in the MOUSE-SELECT-DBLCLICK trigger for the Order browse by removing the second parameter that tells fetchOrderDetail whether items have been retrieved or not, as shown:

```
/* Don't get all Items */
RUN fetchOrderDetail IN hOrderProc
  (iOrderNum, OUTPUT DATASET dsOrder APPEND).
```

   In this case, fetchOrderDetail will never return any Items. They will be retrieved one at a time as they are needed.

3. Edit the fetchOrderDetail internal procedure in OrderSupportBatch.p to remove the second parameter. In addition, set the FILL-MODE for ttItem to be NO-FILL, unconditionally. This means that fetchOrderDetail will return only OrderLines for the current Order, and no Items. For example:

```
PROCEDURE fetchOrderDetail:
  DEFINE INPUT  PARAMETER piOrderNum AS INTEGER NO-UNDO.
  /* Removed lFillItems parameter -- never return all items */
  DEFINE OUTPUT PARAMETER DATASET    FOR dsOrder BY-VALUE.

  hDataSet:EMPTY-DATASET.
  cSelection = "OrderNum = " + STRING(piOrderNum).
  hDataSet:GET-BUFFER-HANDLE(2):FILL-MODE = "APPEND".  /* ttOline */
  hDataSet:GET-BUFFER-HANDLE(3):FILL-MODE = "NO-FILL". /* ttItem  */
  hDataSet:FILL().
END PROCEDURE.  /* fetchOrderDetail */
```

4. Now code the FindFailedItem internal procedure in PickOrderBatch.w, as shown:

```
PROCEDURE FindFailedItem:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  DEFINE VARIABLE cItemName AS CHARACTER NO-UNDO.

  RUN fetchItem IN hOrderProc (ttOline.ItemNum, OUTPUT cItemName).
  CREATE ttItem.
  ASSIGN
    ttItem.ItemNum = ttOline.ItemNum
    ttItem.ItemName = cItemName.
  RETURN NO-APPLY.
END PROCEDURE.
```

   As with all callbacks, this one gets the ProDataSet as INPUT. It runs a new procedure called fetchItem, which you will write in a moment, which accepts the needed Item number and returns its ItemName. As always with callbacks, the buffer in the appropriate temp-table in the ProDataSet parameter holds the row that was current when the event occurred. In this case the AVM is attempting to reposition the ttItem browse each time you select a ttOline row in its browse. To do this, the AVM is doing the equivalent of a FIND statement internally. When the Item is not already there, the FIND-FAILED event occurs, giving your event handler the opportunity to add the needed Item to the client.

   Since the browse shows only the ItemNum and ItemName fields, fetchItem only needs to return the name. You then create a new ttItem temp-table row for that item, and RETURN NO-APPLY. This tells the AVM to ignore the failure and try to locate the row again, transparent to wherever the find failed. This now succeeds, and the new Item appears in its browse and becomes the currently selected row.

5. Finally, write the new fetchItem procedure in OrderSupportBatch.p, as shown:

```
PROCEDURE fetchItem:
  DEFINE INPUT  PARAMETER piItemNum  AS INTEGER   NO-UNDO.
  DEFINE OUTPUT PARAMETER pcItemName AS CHARACTER NO-UNDO.

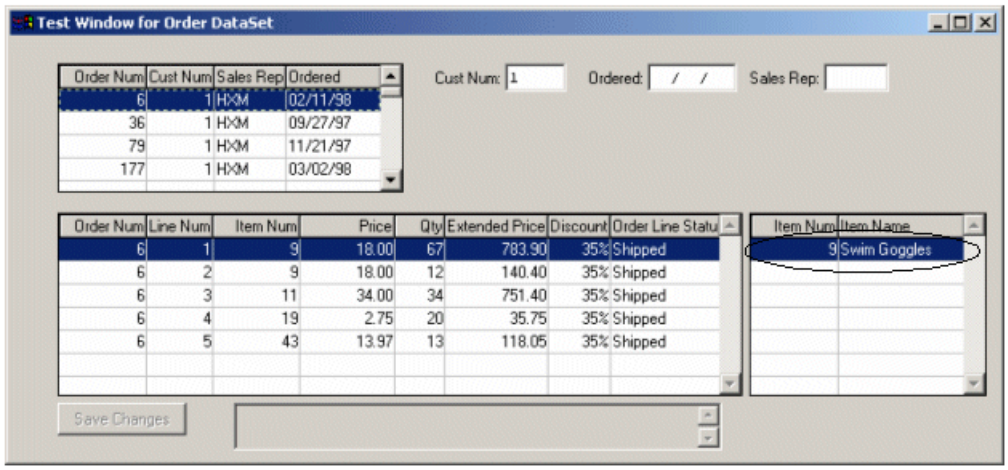
  FIND Item WHERE Item.ItemNum = piItemNum.
  pcItemName = Item.ItemName.
END.
```

   This accepts the current ItemNum, which the window procedure discovered that it did not have when it tried to reposition the Item browse when you select an OrderLine. It finds the database record for that Item and returns its ItemName. Since you are just returning a single field, there is no need to do a FILL. That would be overkill in this case.

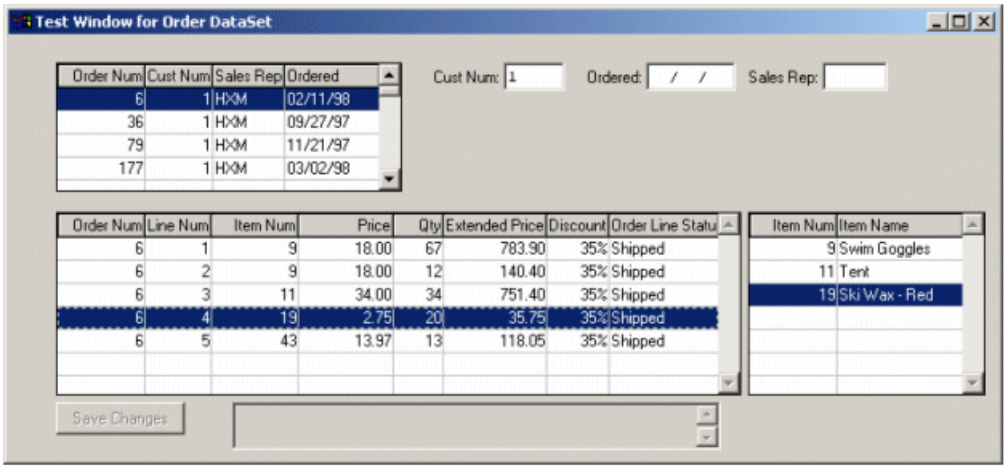
6. Save all of this and rerun PickOrderBatch.w. Enter some selection criteria for Orders, such as Customer 1, tab through the fill-ins, and then double-click on an Order to retrieve its OrderLines.

   When you do this, your modified version of fetchOrderDetail is not returning any Items. Only when you select an OrderLine does the AVM detect that its Item is missing, because there is no row to which it can reposition the Item browse. This fires your FIND-FAILED event handler, which retrieves that one Item, and adds it to the temp-table. By executing a RETURN NO-APPLY, the handler tells the AVM to ignore the failure and retry the reposition.

Now the `Item` is retrieved as expected, as shown:



As you continue to select `OrderLines`, the AVM continues to retrieve each `Item` the first time it is needed, adding it to the `ttItem` temp-table and to the browse, as shown:



Note that because you have created a general-purpose event handler for any `FIND` on the `ttItem` table that fails, any other action your client might take that tries to find a `ttItem` row unsuccessfully will automatically trigger the same handler, and will transparently add the row to the temp-table and make the row available to the statement that needed it, without that statement or its surrounding code needing to do anything special to allow for the possibility that the row is not yet there when it is first referenced. This is the value of the event handlers, that they execute whenever they are needed within the session.

Retrieving rows to cache on the client one at a time may not be a good strategy in many cases. Your `FIND-FAILED` handler could also retrieve batches of rows up to the one that is needed, or batches of related rows that are likely to be needed by the client. In any case, the `FIND-FAILED` event is just one more tool you can use to make batching and caching data in your ProDataSets as transparent as possible.

Summary

This chapter has illustrated several techniques you can use to access data more flexibly using ProDataSets, including:

- Setting a batch size and retrieving large amounts of data one batch at a time, to avoid the overhead of loading all rows into a temp-table and sending all the data across a network at the same time

- Using the `OUTPUT APPEND` parameter mode for the ProDataSet to build up both header and detail information on the client one step at a time
- Using the include-field list to limit the number of fields that are actually populated with data from the Data-Source, and therefore reducing the amount of data sent across the network to the client
- Using `OFF-END` and `FIND-FAILED` event handlers to retrieve data transparently to add to the client's ProDataSet when it is needed

You could extend this example in many ways. For example, to support a `FIND LAST` or other client-side event to jump to a later batch of rows, and to enable scrolling and retrieving batches of rows backwards as well as forwards, and so on. Much of this simulates what the SmartDataBrowser object does in conjunction with a SmartDataObject for data retrieval in an ADM2 application.

9

Advanced Read Operations

Previous chapters introduced you to the language syntax, attributes, and methods that support processing database updates through a ProDataSet. That introduction was somewhat complicated, as it was necessary to cover all the different ways of handling ProDataSet changes, including the attributes and methods that you can use when the higher-level methods provided do not do everything you need to do. In most cases, however, the high-level methods such as `GET-CHANGES`, `SAVE-ROW-CHANGES`, and `MERGE-CHANGES` can make collecting changes and applying them to the database straightforward. This chapter takes this simplification a step further. It shows you how to create general purpose dynamic procedures that encapsulate all the client-side and all the server-side steps used in handling a set of changes. These include applying multiple changes to the database within a single procedure, with the looping mechanism provided to locate all the changed rows and save the changes in an appropriate order. The chapter also extends the sample procedures you have written so far for Order management into something more like a real business object or business entity. The samples encapsulate the Data-Source management procedure, validation logic, and the API other procedures use to access the entity.

For details, see the following topics:

- ProDataSets as a data access layer
- Caching data using a ProDataSet
- Creating views with ProDataSets as Data-Sources
- Summary

ProDataSets as a data access layer

This section shows you how you can encapsulate the data access layer of your application architecture using a type of procedure that defines and uses Data-Sources for your ProDataSets. This discussion is not a formal template, but it is a starting point for organizing the different parts of your application. To think about the data access layer as distinct from the rest of your business logic is taken up again later.

Defining the right internal representation

A ProDataSet definition should represent the appropriate internal representation of your application data, that is, how the application should view and manage the data. This can vary considerably from the way the data is actually stored in your database or other sources of data. You can use the ProDataSet as a way to mask the actual data structure and even the nature of the data source. The term data source written in this way is intended to be generic and can refer to any source at all. It could be an OpenEdge database, another database, a flat file, a stream of data coming from a scanning device, or anything else. By contrast, the term Data-Source refers to the specific ABL object that maps an OpenEdge database buffer or query to a ProDataSet buffer. You can use your own `FILL` logic to populate a ProDataSet from a non-database source or when the buffer and field mapping provided for it is not sufficient to define the data transformation to get the data from how it is represented externally to how it should be represented within the application. The goal is to free up your application logic from being constrained by the physical reality of your database definition. By defining ProDataSets that present your data to the rest of the application as you want it to be seen, you can greatly simplify the logic that handles that data. By properly separating out the logic that maps the external to the internal form, you allow yourself to change the external form as you need to, and then simply change the code that does the transformation. This lets you gradually clean up an older database that might have an inferior design, without changing all your application logic. Or, it lets you substitute an entirely different source for the same data without the application knowing or caring. This is the notion of the **data access** layer that we are introducing here.

Defining the right granularity for your ProDataSets

A ProDataSet can represent one or more tables (and any external data structure that can be mapped to it). Often, of course, a single object or document in your application has a complex structure that requires multiple levels of master-detail relationships. This is what the multiple tables of a ProDataSet can be used for. What then is the appropriate size for your ProDataSets?

There is no single right answer to this question, but if you think of your ProDataSets as representing business objects or documents your application manages, then you should be able to define the right scope for them. Even with such a simplified database as the OpenEdge Sports2000 database that we have been using to illustrate the use of the ProDataSet, you can identify some combinations of tables that can properly be thought of as single business objects and some that probably cannot.

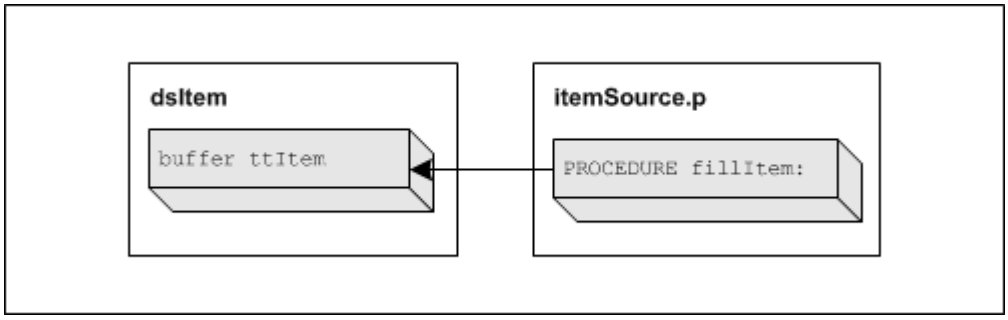
For example, an `Order` is a common object in many applications. Generally, an `Order` has a header record. This is likely the right choice for the top level of an `Order` ProDataSet. The `OrderLines` for an `Order` can then be a separate table in the same ProDataSet, if you think of the `OrderLines` as being a part of the `Order`.

Are the `Item` records then also a part of the `Order`? Well, probably not, even though we have added an `Item` temp-table to the `Order` ProDataSet in these examples so that we can demonstrate certain things about how ProDataSets work. The `Item` identifier is a part of each `OrderLine`, but the `Item` list or catalog itself is really independent of any particular `Order`. You might decide that having a separate temp-table in an `Order` ProDataSet that lets you pull in `Item` detail information for all the `Items` used in a particular `Order` is a useful way to represent the data, as we have done in some of these examples. This is perfectly legitimate if it suits your purposes, but that is different from how you think of the `Item` catalog as a whole.

In this situation, ProDataSets let you define multiple levels of granularity and then combine them as you need to. For example, you can define an `Item` temp-table `ttItem` and an `Item` ProDataSet `dsItem` to represent `Items` as objects in their own right. You could use this ProDataSet to present a list of all `Items` or all `Items` that satisfy some selection. Having a ProDataSet that contains just one table is perfectly reasonable. Even though there are no relations within the ProDataSet, the ProDataSet structure still provides services for you to manage the `Items`.

For example, it could provide a common internal definition for the `ttItem` table and perhaps common `FILL` logic to map the external `Items` onto the internal temp-table, as shown in the following figure.

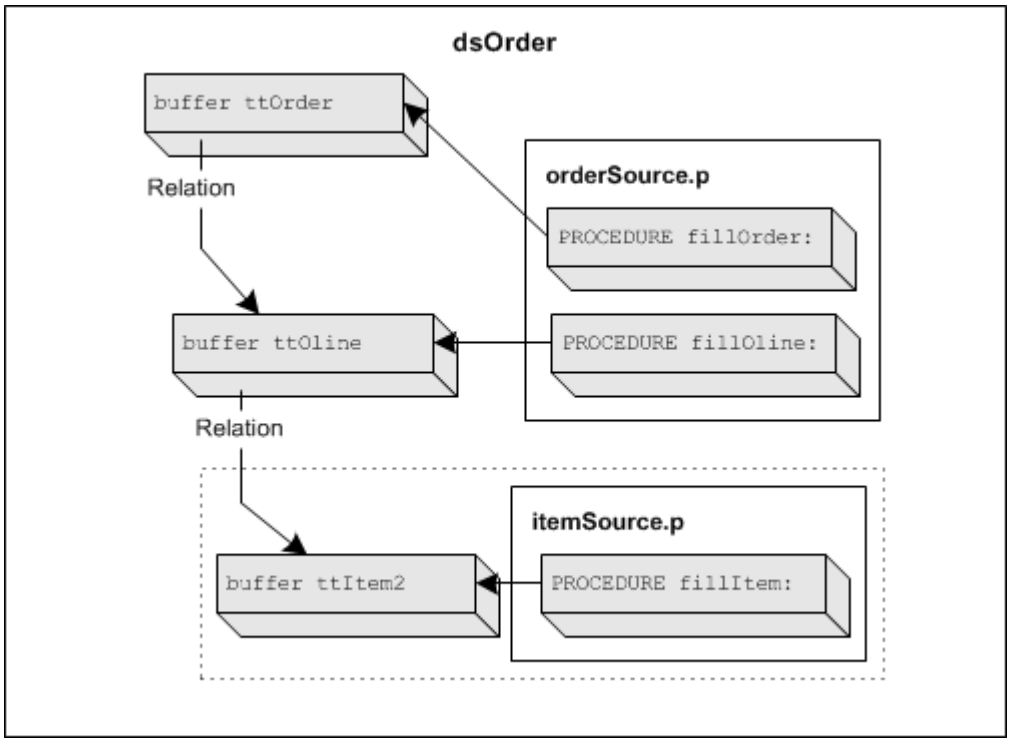
Figure 14: ProDataSets and data granularity



In the above figure, `itemSource.p` represents a procedure that manages the Data-Source and `FILL` logic for the `ttItem` table. This is the kind of procedure we will use to build a simple example later in this section.

You could then use this same `ttItem` temp-table and any of its logic as part of a larger ProDataSet, such as an `Order` ProDataSet that wants to present `Items` used in the `Order`, as shown in the following figure.

Figure 15: Reusing temp-table buffers



Note: Note that while you cannot actually make a ProDataSet such as `dsItem` a part of another ProDataSet, you can make its temp-tables part of another ProDataSet such as `dsOrder` by using a different buffer for the temp-table. Because the `SET-CALLBACK-PROCEDURE` mechanism lets you associate event procedures in any running procedure with a ProDataSet event, you can combine logic from multiple support procedures in a single ProDataSet.

Defining the right top-level table for a ProDataSet

In many simplified examples that use the Sports2000 database, we show one or more `Customers` and the `Customer's Order`s. Would `Customer-Order-OrderLine` then be a good basis for a ProDataSet? To answer this, ask yourself another question: Do you think of all of a `Customer's Order`s as being a part of the `Customer` object itself? Probably not. You might want to represent or work with a `Customer` object and its `Order` objects at the same time, but this does not mean that you should combine them into a single ProDataSet or think of them as a single business entity.

In a real database, a customer or client is typically represented by data in several tables. There might be a `Customer` header table, one or more addresses from an `Address` table, one or more contacts from a `Contacts` table, and so on. This is probably the right set of tables to combine in a `Customer` ProDataSet.

ProDataSets with more than one top-level table

A typical ProDataSet will likely have a single top-level parent table and perhaps one or more child tables beneath it, representing one or more levels of parent-child relationships. But there can be many other reasonable combinations as well. Some business entities (and therefore their ProDataSets) might have more than one top-level table, representing two or more subobjects that, for whatever reason, are always used together. If they are sometimes used independently, then you are likely best off building a ProDataSet for each one and then another ProDataSet that combines them. You might also have sets of tables that have no formal relationship at all, but are simply convenient or logical to access together. Sets of code tables that you need to load at the same time are one example of this. In this case all the code tables would be top-level tables with perhaps no children and no Data-Relations. The example later in this section is of this type, just to show how this kind of ProDataSet could be used.

Dynamic versus static ProDataSets

Chapters thus far have shown you various uses for dynamic ProDataSets, and these can be a powerful tool. Most of your business objects, however, should probably be defined as static temp-tables and static ProDataSets. This lets you write straightforward ABL logic to manage the data, referring to temp-tables and fields by their names rather than through handles.

Sharing ProDataSet and temp-table definitions between procedures

As we have shown in these examples, it is a good idea to put your definitions into include files so that they can be reused in as many procedures as need them and maintained by editing the one include file for a definition. It is also a good idea to separate your temp-table definitions from your ProDataSet definitions. This lets you reuse the temp-tables, for example, in multiple different ProDataSets, as the `ttItem` illustration above shows. It also lets you include ProDataSet definitions in places where the temp-tables need to be defined separately, such as in the AppBuilder.

In summary, there is no single right granularity or complexity for a business object or the ProDataSet that represents its data. Design your ProDataSets so that they represent useful collections of data, either single business documents or sets of data that are normally handled together. Remember as you do this that the AVM gives you a lot of flexibility in how you load and use the data in a single ProDataSet. As you have seen in earlier examples, an `Order` ProDataSet can contain a single `Order` with all its detail, or it can be used to browse `Order` headers and filling in the detail as you need it. Thus, you can make the scope of a business object larger or smaller, depending on your needs.

Building a data access support procedure

What then should go into a procedure that handles the data access layer of your application? In order to keep the layers of your application architecture as cleanly separated as possible, you want to think of the data access layer as having all the knowledge about the specific database structure or other data sources. References to database tables and fields and to other external data sources should be avoided everywhere else in your application if at all possible. At the same time, the data access layer should not be tied to the internal data representation more than it has to be. Here are a few basic guidelines:

- Put your Data-Source definitions into the data access support procedure for the ProDataSet. These identify the specifics of the relationship between the internal and external data representations.
- Put your `FILL` logic into this procedure as well. Your `FILL` logic can supplement the default loading of data into the ProDataSet's tables, or it can replace it entirely where there is no standard Data-Source.
- Attach the internal procedures that contain the `FILL` logic to the ProDataSet. You can do this by passing the ProDataSet handle into a specific `FILL` procedure handler as an input parameter, as shown in the .
- If you can code your `FILL` logic and other supporting logic without the static ProDataSet and temp-table definitions, leave them out of the data access procedures. This way you do not even have to recompile these procedures or check them for consistency with upper layers of the application when it is not necessary. In the example procedure, the temp-table and ProDataSet include files are left out and the few references to specific database fields that are needed are made dynamic, so that there are no dependencies on any fields other than the ones that need to be specifically mapped or transformed. If your data transformation is such that you would need many dynamic references, then go ahead and include the static temp-table definitions. Just consider omitting them when you can.

Data access procedure example

Let us write a very simple support procedure to handle the data access for one new ProDataSet. You will use this ProDataSet in other examples in this chapter.

To show something different from `Order`s and `OrderLines`, and to provide a basis for data caching and data sharing between procedures, the sample ProDataSet is a set of independent tables that have coded values of one kind or another. Within the limited confines of the Sports2000 database, likely candidates are the `State` table, the `Department` table, and the `SalesRep` table. These are all limited enough in scope that the tables can be completely populated when the ProDataSet is first filled.

Here is the dsCodeTT.i include file, with the temp-table definitions:

```
/* dsCodeTT.i --  caching object for various code tables */
DEFINE TEMP-TABLE ttSalesRep NO-UNDO
  FIELD RepCode      AS CHARACTER FORMAT "x(4)"
  FIELD RepName      AS CHARACTER FORMAT "x(20)"
  FIELD Region       AS CHARACTER FORMAT "x(12)"
  FIELD AnnualQuota  AS DECIMAL
  FIELD TotalBalance AS DECIMAL
  INDEX RepCode IS UNIQUE RepCode.

DEFINE TEMP-TABLE ttDept LIKE Department.
DEFINE TEMP-TABLE ttState LIKE State.
```

You can see that the ttSalesRep table is different from the database table it is derived from. It maps some database fields to different names in the temp-table and generates two calculated fields as well.

Here is the dsCode.i include file with the ProDataSet definition:

```
/* dsCode.i --  caching object for the temp-tables in dsCodeTT.i */
DEFINE DATASET dsCode FOR ttSalesRep, ttState, ttDept.
```

This could not be any simpler. There are no relationships between the tables, so the ProDataSet definition only needs to list them.

Now let us start to build the data access support procedure that defines the Data-Sources and handles the FILL logic. First, it defines the Data-Sources, which are also quite simple, as each one names just a single database table the ProDataSet temp-table is derived from. Thus, there is no need for query definitions. The AVM can generate the queries it needs for loading data automatically, as shown:

```
/* CodeSource.p -- Data-Source definitions and FILL logic for code tables */
DEFINE DATA-SOURCE srcRep   FOR SalesRep.
DEFINE DATA-SOURCE srcDept  FOR Department.
DEFINE DATA-SOURCE srcState FOR State.
```

The other thing to note about the top of the procedure is that it does not include the temp-table or ProDataSet definitions. The ProDataSet is always referenced by its handle alone, which means that anything about the ProDataSet definition can change without requiring even a recompile of this procedure. The only dependencies are the specific fields that the procedures must map or reference for calculations. Even in those cases, the code that maps or references them could be made conditional so that it would not fail if the fields were removed from the ProDataSet, or this procedure was used with a version of the ProDataSet that did not have them.

Caching complex derived data in a ProDataSet

As you saw in the temp-table definitions, there are calculated fields in the SalesRep temp-table. We need a procedure called postRepRowFill to generate those calculated fields. To be sure, the calculations are not very complex in this case, but we use this as a placeholder for a more serious application where a table of complex data that is relatively expensive to derive is loaded into a ProDataSet once so that it can be used throughout a session.

There are various alternatives to this approach. If the derived data is common to all use cases, then it can be calculated once each time the data it depends on is changed, for example, by database trigger procedure code, and stored in permanent database tables of its own. But in many cases the calculations are specific to the immediate user of the data. For example, a price sheet might depend on various factors that change from session to session, including who the current `Customer` is, who the user is, what the nature of the product requirements are, and so forth. In a case like this, a ProDataSet instance can provide reusable calculations that can be kept available for the user's session or until a different `Customer` or product line is selected.

As with all `FILL` event procedures, this example receives the ProDataSet as an `INPUT` parameter, passed by reference. There are also several local variables, as shown:

```
PROCEDURE postRepRowFill:
  DEFINE INPUT PARAMETER DATASET-HANDLE phDataSet.
  DEFINE VARIABLE hSalesRep     AS HANDLE  NO-UNDO.
  DEFINE VARIABLE dTotalQuota   AS DECIMAL NO-UNDO.
  DEFINE VARIABLE dTotalBalance AS DECIMAL NO-UNDO.
  DEFINE VARIABLE iMonth        AS INTEGER NO-UNDO.
```

The ProDataSet supports several different code tables. You need to get the handle to the `SalesRep` table. For example:

```
hSalesRep = phDataSet:GET-BUFFER-HANDLE("ttSalesRep").
```

Tip: It's a good idea to keep any references to the specifics of the ProDataSet structure like this as flexible as possible. In this case, the `ttSalesRep` table is the first one in the ProDataSet, but referencing it as `GET-BUFFER-HANDLE(1)` can give you maintenance headaches as definitions change. Also you might find yourself able to reuse parts of this code with a very different ProDataSet structure if you don't assume any unnecessary dependencies. In some cases even the table name could be a parameter.

The first calculated field is the total `Annual Quota` for each `SalesRep`. To calculate this, you total the 12 elements of the `MonthQuota` array, as shown:

```
  DO iMonth = 1 TO 12:
    dTotalQuota = dTotalQuota + SalesRep.MonthQuota[iMonth].
  END.

  hSalesRep:BUFFER-FIELD("AnnualQuota"):BUFFER-VALUE = dTotalQuota.
```

Remember that because the temp-table definitions are not included, you need to reference the fields dynamically through the buffer handle. You need to decide when this makes the code too complex without sufficient benefit of flexibility.

Next, you calculate the `Total Balance` of all the `SalesRep's Customers`:

```
  FOR EACH Customer WHERE Customer.SalesRep =
    STRING(hSalesRep:BUFFER-FIELD("RepCode"):BUFFER-VALUE):
    dTotalBalance = dTotalBalance + Customer.Balance.
  END.
  hSalesRep:BUFFER-FIELD("TotalBalance"):BUFFER-VALUE = dTotalBalance.
END PROCEDURE. /* postRepRowFill */
```

The other entry points are functions that attach and detach the ProDataSets. The attachDataSet function takes the ProDataSet handle as an INPUT parameter, sets the callback procedure for the AFTER-ROW-FILL event for the ttSalesRep table, and attaches the Data-Sources, as shown:

```
FUNCTION attachDataSet RETURNS LOGICAL (INPUT phDataSet AS HANDLE):
  phDataSet:GET-BUFFER-HANDLE("ttSalesRep"):SET-CALLBACK-PROCEDURE
    ("AFTER-ROW-FILL", "postRepRowFill", THIS-PROCEDURE).
  phDataSet:GET-BUFFER-HANDLE("ttSalesRep"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcRep:HANDLE, "SalesRep.SalesRep,RepCode").
  phDataSet:GET-BUFFER-HANDLE("ttState"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcState:HANDLE).
  phDataSet:GET-BUFFER-HANDLE("ttDept"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcDept:HANDLE).
  RETURN phDataSet:ERROR.
END FUNCTION. /* attachDataSet */
```

Nothing special happens here except mapping the SalesRep field in the database table to the RepCode field in the temp-table.

Note these facts about the HANDLE parameter to this function:

- You can pass a HANDLE of a ProDataSet to a user-defined function, but you cannot pass a ProDataSet using the DATASET or DATASET-HANDLE parameter forms. This is simply not supported. It is likely that if you find yourself needing the actual ProDataSet to be instantiated, or you need to reference more of the ProDataSet than a single field or other element that you pass in as a parameter, you are probably overstepping the bounds of what is appropriate for a function as opposed to an internal procedure. Remember that because a function can appear anywhere within a larger expression or where-clause, there are some restrictions on the kinds of data manipulation that are permitted within a function, where indexes and transactions are concerned. Try to limit your use of functions to concise operations that return a useful value that you would want to reference in an expression. (This recommendation is really independent of ProDataSets, but can serve to explain why passing a DATASET to a function was not deemed essential to support.) As you can see from the example, you can still access anything within the ProDataSet through its handle.
- Remember the difference between the DATASET-HANDLE form in the event procedure postRepRowFill and the HANDLE form in the function. Both parameters are in fact handles, but in the former case the AVM is prepared to pass in the entire ProDataSet definition and data. In the latter case only the handle itself is passed. Because the AVM internally passes the ProDataSet to the event procedure by reference, what happens is effectively the same as when the HANDLE is passed to the function: the AVM simply supplies the handle of an existing ProDataSet that is defined somewhere else. You can reference any element in the ProDataSet by walking through it by starting with its handle. But because the effects of this could be extremely different if the call were remote, the AVM requires that the parameters be of the appropriate form. You can supply a static DATASET parameter or a dynamic DATASET-HANDLE parameter to pass the ProDataSet itself, but if the caller passes just the HANDLE, then the called procedure (or function) must also define the parameter as a HANDLE.

  The detachDataSet function simply detaches the Data-Sources, as shown:

```
FUNCTION detachDataSet RETURNS LOGICAL (INPUT phDataSet AS HANDLE):
  phDataSet:GET-BUFFER-HANDLE("ttSalesRep"):DETACH-DATA-SOURCE().
  phDataSet:GET-BUFFER-HANDLE("ttState"):DETACH-DATA-SOURCE().
  phDataSet:GET-BUFFER-HANDLE("ttDept"):DETACH-DATA-SOURCE().
  RETURN phDataSet:ERROR.
END FUNCTION. /* detachDataSet */
```

Summary of the data access procedure

To summarize the data access procedure:

- It defines the Data-Sources for the ProDataSet tables
- It provides any needed field mapping from database to ProDataSet
- It defines and attaches the `FILL` logic needed for calculations
- In short, it handles everything that needs to be aware of the database specifics and nothing at all that happens to the ProDataSet after it is filled

Caching data using a ProDataSet

This section explores a number of different ProDataSet capabilities that you can take advantage of in your applications. It uses the code table ProDataSet and data access support procedure you wrote in the last few chapters. The general theme to this chapter is providing a variety of views of the same data, retrieving and calculating it once, and then responding to different kinds of requests for subsets or other reuse of the data.

Using a subset of the tables in a ProDataSet

Writing the server procedure

You already know how to return an entire ProDataSet to another procedure. You also know how to deactivate relations or designate tables as `NO-FILL`. When you do this you return some tables as empty to the caller. Let us look at how you could return a dynamic ProDataSet that represents a subset of the tables that are defined in a larger one. In the case of the code table ProDataSet, some callers might not want all the tables at all, or might not even know of their existence or have any definitions to receive them into. In this case the server can dynamically subset the data at the table level according to the caller's request.

Let us write that server procedure now. As with other examples, the code is simplified to the extent that it runs in a single session, but the backend server procedures are clearly separated from the user interface procedures that run on the client.

To update the code:

1. Create the new procedure called `CodeSupport.p`, as shown in the following code block.

   It first includes the temp-table and ProDataSet definitions. The definitions, of course, also create an instance of that static ProDataSet and its temp-tables when `CodeSupport.p` is run. This means that in effect each run-time instance of `CodeSupport.p` "owns" its own instance of the ProDataSet as well. This allows any internal procedures to use static ABL to reference the ProDataSet and its data, as well as any other procedures to which the ProDataSet might be passed by reference. This is different from how `CodeSource.p` operates. Because it has no static definition and receives only the ProDataSet handle as input, it operates only on an instance of the ProDataSet actually created and managed elsewhere (namely here in `CodeSupport.p`).

CodeSupport.p starts an instance of CodeSource.p and then requests it to attach the Data-Sources using the attachDataSet function and fills the ProDataSet, as shown:

```
/* CodeSupport.p -- support procedures for dsCode tables */
{dsCodeTT.i}
{dsCode.i}

DEFINE VARIABLE hCodeSet    AS HANDLE  NO-UNDO.
DEFINE VARIABLE hCodeSource AS HANDLE  NO-UNDO.
DEFINE VARIABLE hSourceProc AS HANDLE  NO-UNDO.
DEFINE VARIABLE lError      AS LOGICAL NO-UNDO.

hCodeSet = DATASET dsCode:HANDLE.
RUN CodeSource.p PERSISTENT SET hSourceProc.
lError = DYNAMIC-FUNCTION("attachDataSet" IN hSourceProc, hCodeSet).
hCodeSet:FILL().
```

2. Next, create the internal procedure that actually generates a new ProDataSet with a subset of the tables in the original one, fetchCodeTables, as shown:

```
PROCEDURE fetchCodeTables:
  DEFINE INPUT  PARAMETER pcTables AS CHARACTER  NO-UNDO.
  DEFINE OUTPUT PARAMETER DATASET-HANDLE phDynData.

  DEFINE VARIABLE cTable    AS CHARACTER NO-UNDO.
  DEFINE VARIABLE hTableBuf AS HANDLE    NO-UNDO.
  DEFINE VARIABLE iTable    AS INTEGER   NO-UNDO.

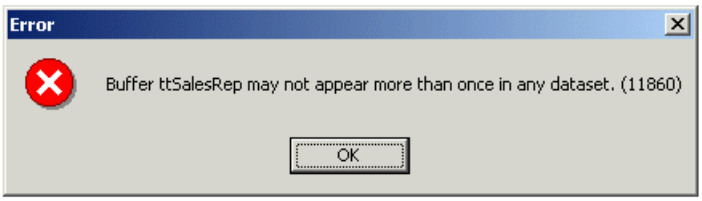
  CREATE DATASET phDynData.
  DO iTable = 1 TO NUM-ENTRIES(pcTables):
    cTable = ENTRY(iTable,pcTables).
    CREATE BUFFER hTableBuf FOR TABLE cTable.
    phDynData:ADD-BUFFER(hTableBuf).
  END.
END PROCEDURE. /* fetchCodeTables */
```

This takes a table list as input, creates a new dynamic ProDataSet along with new buffers for the static ProDataSet's tables, and adds the buffers to the ProDataSet. This makes the existing data (already retrieved and filled) in the ProDataSet dsCode part of the new ProDataSet without any need to copy it, because the caller wants all the data in the requested subset of the tables.

Since you are putting the same static temp-tables into the new ProDataSet, why do you need new dynamic buffers for them? Remember that there is a rule that a temp-table can be part of more than one ProDataSet at a time, but one temp-table buffer can only be part of one ProDataSet at a time. The AVM generally manages the temp-tables in a ProDataSet through their buffers, and it can do this only when each ProDataSet has its own distinct set of buffers, even when temp-tables are shared. If you were to leave out the CREATE BUFFER statement, you could try to add the existing static buffer from dsCode directly into the new ProDataSet, as shown:

```
/* You cannot use the same buffer in two ProDataSets: */
phDynData:ADD-BUFFER(DATASET dsCode:GET-BUFFER-HANDLE(iTable)).
```

If you did, you would get the following error when you run an application that uses `fetchCodeTables`:



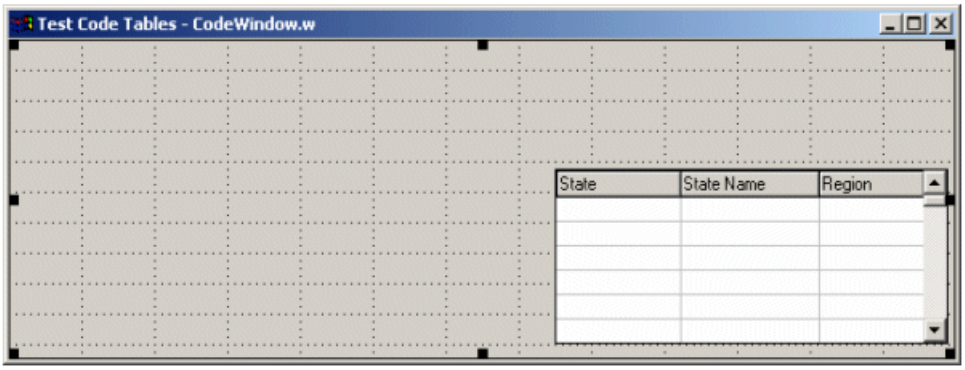
Building the user interface for the new ProDataSet

Now build a user interface for the new ProDataSet.

To build the user interface:

1. Create a new window procedure in the AppBuilder. Name it `CodeWindow.w`.
2. Make the window `10` rows by `130` columns.
3. Name the default frame `CodeFrame` and the window `CodeWin`.
4. Use the AppBuilder's temp-table utility to define a temp-table `ttState LIKE` the `State` database table.
5. Drop a browse called `StateBrowse` onto the design window and attach it the `ttState` table and its three fields.

   Your window should look roughly like this when you place the `StateBrowse` to leave room for other objects. For example:



6. Add a statement to the **Main Block** to run a procedure where all the code will go to start up the window. For example:

```
MAIN-BLOCK:
  DO ON ERROR   UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
     ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
    RUN enable_UI.
    RUN startupCodeWindow.
    IF NOT THIS-PROCEDURE:PERSISTENT THEN
      WAIT-FOR CLOSE OF THIS-PROCEDURE.
  END.
```

7. Add a line to the `CLOSE` trigger in the **Main Block** for a procedure to shut down the window support code. For example:

```
ON CLOSE OF THIS-PROCEDURE DO:
  RUN shutdownCodeWindow.
  RUN disable_UI.
END.
```

8. Add these variables to the **Definitions** section:

```
DEFINE VARIABLE hCodeSupport AS HANDLE NO-UNDO.
DEFINE VARIABLE hCodeSet     AS HANDLE NO-UNDO.
DEFINE VARIABLE hRepBrowse   AS HANDLE NO-UNDO.
DEFINE VARIABLE hRepQuery    AS HANDLE NO-UNDO.
DEFINE VARIABLE hStateQuery  AS HANDLE NO-UNDO.
```

9. Create the internal procedure `startupCodeWindow`. This starts the `CodeSupport` procedure and then asks it for a ProDataSet with just the `ttSalesRep` and `ttState` tables in it, as shown:

```
/*------------------------------------------------------------------
Procedure:   startupCodeWindow
Purpose:     Fetch needed code tables from server.
Parameters:  <none>
------------------------------------------------------------------*/
RUN codeSupport.p PERSISTENT SET hCodeSupport.
RUN fetchCodeTables IN hCodeSupport
  (INPUT "ttSalesRep,ttState", OUTPUT DATASET-HANDLE hCodeSet).
```

10. Create a dynamic query for the `ttState` table that comes back as part of the dynamic ProDataSet, attaches it to the `StateBrowse`, prepares it, and opens it. The variables you use in this procedure are the ones you added in the **Definitions** section:

```
CREATE QUERY hStateQuery.
  hStateQuery:ADD-BUFFER(hCodeSet:GET-BUFFER-HANDLE("ttState")).
  StateBrowse:QUERY IN FRAME CodeFrame = hStateQuery.
  hStateQuery:QUERY-PREPARE("FOR EACH ttState").
  hStateQuery:QUERY-OPEN().
```

Again, you might ask why you need a new dynamic query for this table. After all, you just defined a static temp-table `ttState` and a static browse `StateBrowse` against that temp-table.

Once again, the answer is that you are not really using that static temp-table. It only provides a definition to base the browse on. What comes back from `fetchCodeTables` is a separate dynamic temp-table that happens to have the same name and the same fields so that you can easily use it in place of the static temp-table you defined.

You cannot simply open the static query and use it. For example:

```
/* Can't do this:
   OPEN QUERY StateBrowse FOR EACH ttState. */
```

If you do, you will not see any data in the browse.

11. To reinforce how to do this properly, you can create a dynamic query for the other table you are getting back, `ttSalesRep`:

```
CREATE QUERY hRepQuery.
hRepQuery:ADD-BUFFER(hCodeSet:GET-BUFFER-HANDLE("ttSalesRep")).
```

Be sure to add the right buffer to the query, which is the one for the temp-table that comes back as part of the dynamic ProDataSet `hCodeSet`. If you had a local definition of `ttSalesRep`, its buffer would not do you any good for the same reason that your local definition of `ttState` cannot be used.

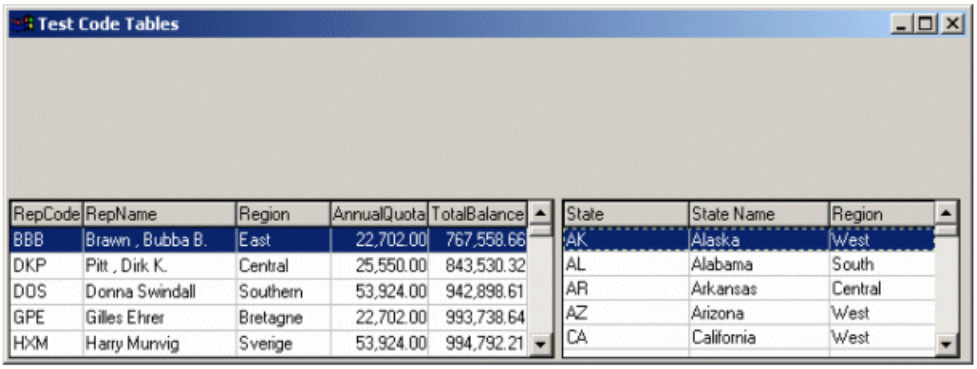
This dynamic browse uses the `ttSalesRep` query:

```
CREATE BROWSE hRepBrowse ASSIGN
  QUERY        = hRepQuery
  ROW-MARKERS  = FALSE
  FRAME        = FRAME CodeFrame:HANDLE
  HIDDEN       = FALSE
  NO-VALIDATE  = TRUE
  WIDTH        = 74
  HEIGHT       = 5
  ROW          = 6
  SEPARATORS   = TRUE
  SENSITIVE    = TRUE.
hRepBrowse:ADD-COLUMNS-FROM("ttSalesRep").
```

12. Finally, the procedure needs to prepare and open the dynamic query on `ttSalesRep`:

```
hRepQuery:QUERY-PREPARE("FOR EACH ttSalesRep").
hRepQuery:QUERY-OPEN().
```

13. Now, when you run the window, you see both the static browse and the dynamic browse. Both are, in fact, using dynamic temp-tables that came back as part of the dynamic ProDataSet `hCodeSet`, as shown:



14. Define the internal procedure `shutdownCodeWindow` to delete the supporting procedure instance:

```
/*------------------------------------------------------------------
Procedure: shutdownCodeWindow
Purpose:   Cleanup supporting procedure and any other objects when
           deleting
           the window.
------------------------------------------------------------------*/
  APPLY "CLOSE" TO hCodeSupport.
END PROCEDURE.
```

Rather than deleting it directly, applying the `CLOSE` event to it gives it a chance to clean up after itself. This is modeled on the standard code the AppBuilder generates to close a procedure by running `disable_UI`.

15. To handle the `CLOSE` event in `CodeSupport.p`, add the following trigger to its main block so that it can delete the other persistent procedure `CodeSource.p` that manages the Data-Sources, and then delete itself:

```
ON CLOSE OF THIS-PROCEDURE DO:
  DYNAMIC-FUNCTION("detachDataSet" IN hSourceProc, INPUT hCodeSet).
  DELETE PROCEDURE hSourceProc.
  DELETE PROCEDURE THIS-PROCEDURE.
END.
```

Summary of caching data using a ProDataSet

This part of the new window, along with its supporting procedure, shows you can make the same tables part of more than one ProDataSet, as long as each ProDataSet has its own unique set of buffers for those tables. You can fill commonly used data just once, on the server or on the client as needed, and then subset it in this way without any need to copy the data to each new view of that data.

In the next section, you will extend this procedure to show an alternative to this, namely, how to create a new ProDataSet that actually uses an existing one as a Data-Source, and creates new temp-tables with only a subset of the columns in the original ones.

Creating views with ProDataSets as Data-Sources

In this section, you will create a support procedure that creates a new ProDataSet with a custom temp-table with only specific fields the caller requests, and only certain rows from the table that satisfy the caller's selection criteria.

Sample procedure: creating a view

Because the caller is selecting a subset of the fields and rows, in this case the new ProDataSet has to copy data from the original one, rather than simply assigning new buffers to the existing temp-tables in their entirety.

To update the code:

1. Create the new procedure `fetchCustomTable` in `CodeSupport.p`. It takes the table name, field list, and selection *`where-clause`* as input parameters and returns the new dynamic ProDataSet, as shown:

```
PROCEDURE fetchCustomTable:
  DEFINE INPUT  PARAMETER pcTable     AS CHARACTER NO-UNDO.
  DEFINE INPUT  PARAMETER pcFields    AS CHARACTER NO-UNDO.
  DEFINE INPUT  PARAMETER pcSelection AS CHARACTER NO-UNDO.
  DEFINE OUTPUT PARAMETER DATASET-HANDLE phFilterData.
```

2. Add the variables the procedure uses:

```
DEFINE VARIABLE cField  AS CHARACTER NO-UNDO.
DEFINE VARIABLE hTable  AS HANDLE    NO-UNDO.
DEFINE VARIABLE iField  AS INTEGER   NO-UNDO.
DEFINE VARIABLE hNewBuf AS HANDLE    NO-UNDO.
DEFINE VARIABLE hOldBuf AS HANDLE    NO-UNDO.
DEFINE VARIABLE hQuery  AS HANDLE    NO-UNDO.
```

3. Create a dynamic ProDataSet and a dynamic temp-table to put into it, with the fields the caller requested:

```
/* Create a new dynamic ProDataSet based on the table and fields passed
   in. */
CREATE DATASET phFilterData.
CREATE TEMP-TABLE hTable.
DO iField = 1 TO NUM-ENTRIES(pcFields):
  cField = ENTRY(iField,pcFields).
  hTable:ADD-LIKE-FIELD(cField,pcTable + "." + cField).
END.

hTable:TEMP-TABLE-PREPARE(pcTable).
hNewBuf = hTable:DEFAULT-BUFFER-HANDLE.
phFilterData:ADD-BUFFER(hNewBuf).
```

4. Create a dynamic query for the temp-table in the original ProDataSet and prepare it using the where-clause passed in:

```
/* Next create a dynamic query for the selection criteria passed in. */
CREATE QUERY hQuery.
hOldBuf = DATASET dsCode:GET-BUFFER-HANDLE(pcTable).
hQuery:ADD-BUFFER(hOldBuf).
hQuery:QUERY-PREPARE("FOR EACH " + pcTable + " WHERE " + pcSelection).
```

5. Open the query and buffer-copy all the rows that satisfy the selection into the new temp-table:

```
hQuery:QUERY-OPEN().
hQuery:GET-FIRST().
DO WHILE NOT hQuery:QUERY-OFF-END:
  hNewBuf:BUFFER-CREATE().
  hNewBuf:BUFFER-COPY(hOldBuf).
  hQuery:GET-NEXT().
END.
```

Or, you can create a dynamic Data-Source for the temp-table in the original ProDataSet and attach that Data-Source to the new temp-table buffer in the new ProDataSet. This shows how one ProDataSet table that has already been filled can be used as a Data-Source for a table in another ProDataSet.

As in this example, this is appropriate if the original ProDataSet is filled with some set of generally useful data (and possibly, data that is expensive to regenerate and that needs to be used as a cache for the session), and if the second ProDataSet only wants a subset of its rows or fields. Remember that this approach does involve copying data from one ProDataSet to another.

To use this method, remove or comment out the lines in the previous code section and replace them with this code:

```
CREATE DATA-SOURCE hCodeSource.

/* NOTE: hOldBuf is the source temp-table buffer, and the KEYS list is
   not needed */
hCodeSource:ADD-SOURCE-BUFFER(hOldBuf, ?).

/* Because there is a specific query for selecting a subset of the rows
   in the source temp-table, the procedure uses the dynamic query defined
   above. Otherwise it could leave off the query and get all rows    automatically.
 */
hCodeSource:QUERY = hQuery.

/* Now when it attaches the Data-Source and fills the new ProDataSet it
   gets rows from its Data-Source, which is the table in the original
   ProDataSet. */
hNewBuf:ATTACH-DATA-SOURCE(hCodeSource).
phFilterData:FILL().
hNewBuf:DETACH-DATA-SOURCE().

/* This is the end of the alternative code to use the original ProDataSet
as a Data-Source for the custom subset. */
```

6. Delete the dynamic objects the procedure uses. Note that it's OK to delete the ProDataSet before returning because the AVM delays the actual delete until the parameter has been returned. For example:

```
  DELETE OBJECT phFilterData.
  DELETE OBJECT hCodeSource.
  DELETE OBJECT hQuery.
END PROCEDURE. /* fetchCustomTable */
```

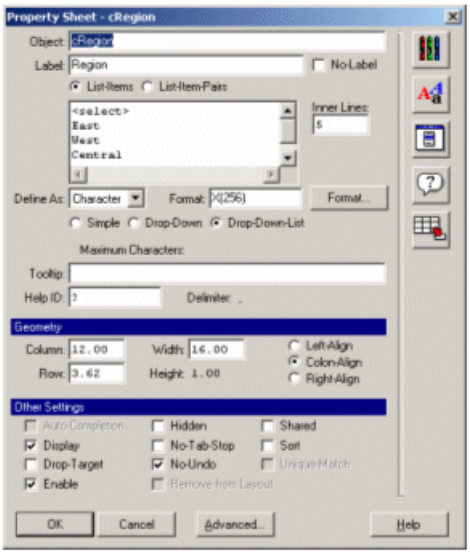
7. Return to the window procedure `CodeWindow.w` to create a user interface for the custom ProDataSet.
8. Add these variables to the **Definitions** section:

```
DEFINE VARIABLE hCustomQuery  AS HANDLE NO-UNDO.
DEFINE VARIABLE hCustomBrowse AS HANDLE NO-UNDO.
DEFINE VARIABLE hCustomSet    AS HANDLE NO-UNDO.
```

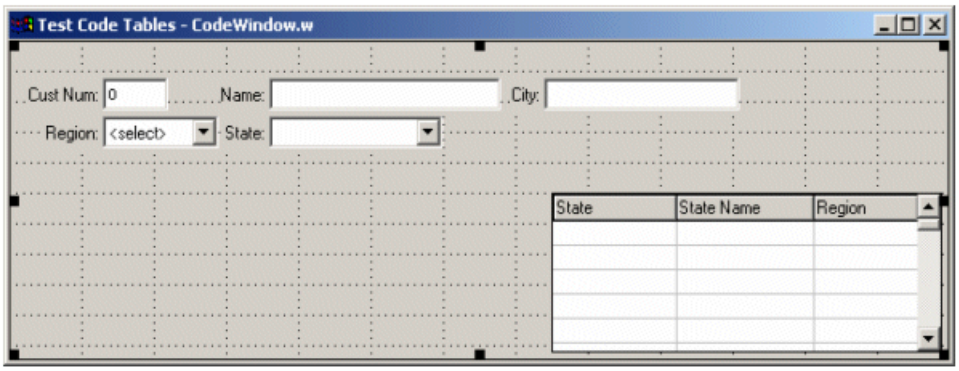
You are going to add some `Customer` fields to the window along with a **Region** combo box that lists the four regions in the US. When the user selects a region, the procedure runs `fetchCustomTable` to request a list of state codes and state names for that region. To reduce the size of the example somewhat, you will use fields from the `Customer` table directly rather than a `Customer` ProDataSet, which would be the proper way to do things.

9. From the AppBuilder palette, select the **DB Fields** icon and drop the fields `CustNum`, `Name`, and `City` from `sports2000.Customer` onto the window.

10. Select the combo box from the palette and create a combo box. It has the **Object** name `cRegion`, the **Label**`Region`, `5`**Inner Lines,** and the set of **List-Items**`<select>,East,West,Central,South`. The `<select>` choice prompts the user to select a region before seeing any `SalesReps` for it, as shown



11. Create another combo box called `cState`, with a **Label** of `State` and `5`**Inner Lines** as well. It has no initial **List-Items**. The design window should now look something like this:



12. Code a `VALUE-CHANGED` trigger for the **Region** combo box.

    If the user makes a region selection, the trigger runs `fetchCustomTable`, requesting a ProDataSet with the `ttState` table, two of the three fields from the table, and only those `SalesReps` where the `Region` matches the one chosen. For example:

```
DEFINE VARIABLE hStateBuf AS HANDLE NO-UNDO.

IF cRegion:SCREEN-VALUE NE "<select>" THEN DO:
  RUN fetchCustomTable IN hCodeSupport
   (INPUT "ttState",
    INPUT "State,StateName",
    INPUT "Region = '" + cRegion:SCREEN-VALUE + "'",
    OUTPUT DATASET-HANDLE hCustomSet).
```

It empties the **State** combo in case this is not the first request, creates a dynamic query for it, and adds each `StateName` that came back in the ProDataSet from `fetchCustomTable` to the **List-Items** for the **State** combo. It makes the first one the current choice, and deletes the query now that it is done with it. For example:

```
  cState:LIST-ITEMS = "".  /* Empty the old list if any. */

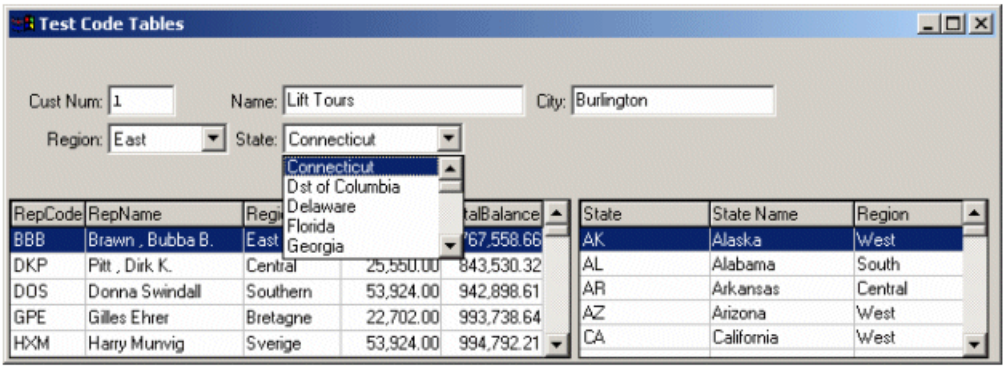
  CREATE QUERY hCustomQuery.
  hStateBuf = hCustomSet:GET-BUFFER-HANDLE("ttState").
  hCustomQuery:ADD-BUFFER(hStateBuf).

  hCustomQuery:QUERY-PREPARE("FOR EACH ttState").
  hCustomQuery:QUERY-OPEN().
  hCustomQuery:GET-FIRST().

  DO WHILE NOT hCustomQuery:QUERY-OFF-END:
    cState:ADD-LAST(hStateBuf:BUFFER-FIELD("StateName"):BUFFER-VALUE).
    hCustomQuery:GET-NEXT().
  END.

  cState:SCREEN-VALUE = cState:ENTRY(1).
  DELETE OBJECT hCustomQuery.
END.
```

Now your procedures are finished. When you run the window, the standard AppBuilder-generated code opens a `Customer` query and retrieves the first `Customer` for you because you added fields from that table to the window. You can then select a `Region` and see a list of all the `States` in that region to choose from, as shown:



This illustrates how a ProDataSet that is filled with a set of useful data can be divided in many ways by other procedures that need various subsets of the data in the same session or another session. Data held in a ProDataSet in a client session can act as a cache for visual objects or client-side business logic that needs to view or use the data or a subset of the data. Any object in the same session can define its own query to browse or otherwise use a subset of the rows in the data.

Summary

The sample procedure in this chapter described how to effectively create different internal views of data when using a ProDataSet as a Data-Source.

10

Advanced Update Operations

Previous chapters introduced you to the language syntax, attributes, and methods that support processing database updates through a ProDataSet. That introduction was somewhat complicated, as it was necessary to cover all the different ways of handling ProDataSet changes, including the attributes and methods that you can use when the higher-level methods provided do not do everything you need to do. In most cases, however, the high-level methods such as `GET-CHANGES`, `SAVE-ROW-CHANGES`, and `MERGE-CHANGES` can make collecting changes and applying them to the database straightforward. This chapter takes this simplification a step further. It shows you how to create general purpose dynamic procedures that encapsulate all the client-side and all the server-side steps used in handling a set of changes. These include applying multiple changes to the database within a single procedure, with the looping mechanism provided to locate all the changed rows and save the changes in an appropriate order. The chapter also extends the sample procedures you have written so far for `Order` management into something more like a real business object or *business entity*. The samples encapsulate the Data-Source management procedure, validation logic, and the API other procedures use to access the entity.

For details, see the following topics:

- Creating a data access procedure for the Order ProDataSet
- Building a business entity procedure to support the ProDataSet
- Building general update procedures for client and server
- Running standard validation procedures on update
- Summary

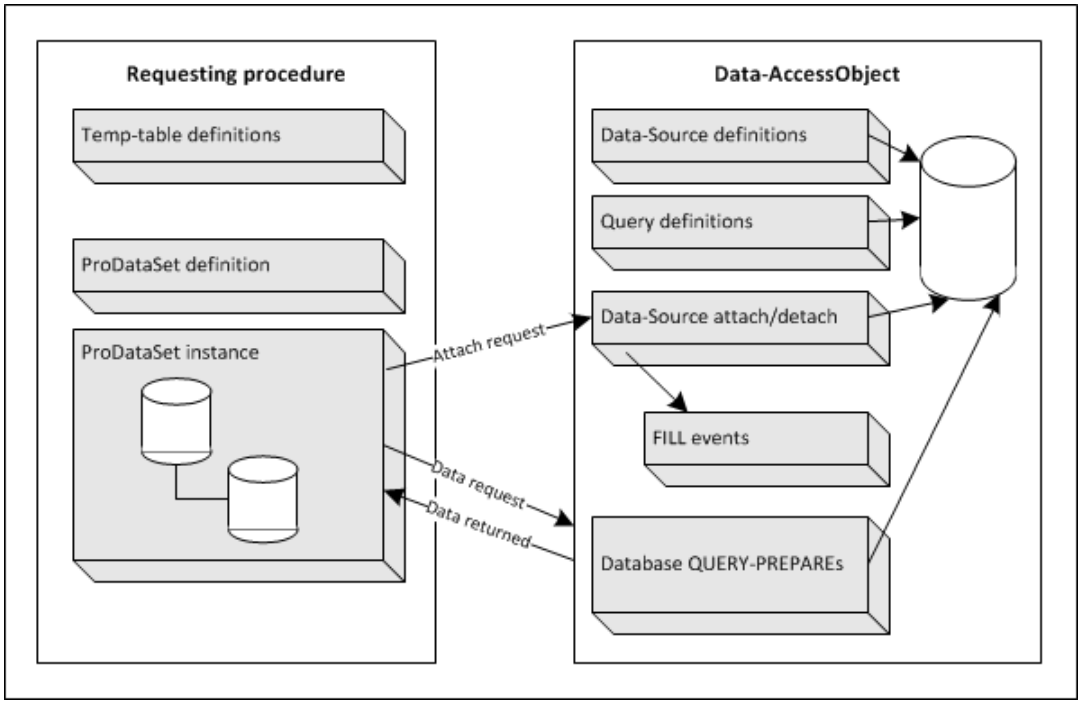
Creating a data access procedure for the Order ProDataSet

The first step is to create a data access procedure that handles all the code that requires knowledge of the data source. To do this, you create a procedure to act as the data access object for the `dsOrder` ProDataSet in the same way that `CodeSource.p` does for the code table ProDataSet from the previous chapter. The following figure illustrates:

- Data definitions and logic to consider as part of the data access object.
- The data access object encapsulates all database references. Therefore, query definitions are part of the object.
- Data-Source definitions that map database tables and queries to ProDataSet buffers are part of the object.
- Methods (internal procedures or functions) that attach and detach Data-Sources for the ProDataSet buffers are part of the object.
- Methods that prepare database queries or methods that in other ways reference database table and field names directly are part of the object. For example, one kind of request of the `Order` ProDataSet is to return all the data for a particular `Order` number. This request can be made of the higher-level business entity (as you will do later in this chapter), but the query itself should be prepared in the data access object. This is because that request requires defining a particular database query to get the right `Order` from the database.
- `FILL` event logic that determines the final form of what is in the ProDataSet is part of the object. These `FILL` event procedures can be associated with the ProDataSet instance at the time of the `ATTACH`, as in these examples.

In this way, all the definitions and code that reference the database are nicely captured in a single place. Here they can be maintained, as needed, when database definitions or data sources change. All the higher levels of access to the ProDataSet do not contain any such references, which isolates them from the specifics of the Data-Sources. Once the `FILL` event procedures have been associated with the caller's ProDataSet instance and the Data-Sources attached to it, the caller can simply execute the `FILL` method, and all the required logic is executed properly on that instance. The following figure is an example.

Figure 16: Data access procedure



So, let us create the data access procedure `OrderSource.p`.

Much of the code in this procedure comes from the `OrderEvents.p` procedure in Advanced Events and Attributes on page 175. You can copy code from there and adapt it as needed. The purpose of this exercise is to begin to isolate the code better, based on what role it plays, to begin to provide more of an architecture to the application's objects.

In this example, the data access procedure has a static definition of the ProDataSet and its temp-tables:

```
/* OrderSource.p -- Data-Sources and FILL events for Order ProDataSet */
{dsOrderTT.i}
{dsOrder.i}
```

As we noted in Advanced Events and Attributes on page 175, it can help you isolate your ProDataSet definitions better if you can avoid this, but if there are many references to ProDataSet tables and fields in the `FILL` logic or elsewhere, then this might not be practical. This example shows the alternative of having the definitions in the data access procedure so that they can be referenced in static ABL statements. As we work through the procedure, it will be important to note how the actual ProDataSet instance the code is operating on is **not** the one that this procedure gets by including the definitions. The support code is always using the instance from the requesting procedure.

The top-level definition and the Data-Source definitions will be familiar from OrderEvents.p. For example:

```
DEFINE QUERY qOrder FOR Order, Customer, SalesRep.
DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder
  Order KEYS (OrderNum), Customer KEYS (CustNum), SalesRep KEYS (SalesRep).
DEFINE DATA-SOURCE srcOline FOR OrderLine.
DEFINE DATA-SOURCE srcItem FOR ITEM  KEYS (ItemNum).
```

OrderEvents.p was intended to run as a stand-alone procedure with no internal procedures. Thus, it took the Order Number and OUTPUT ProDataSet as parameters directly. In this case, the data access procedure runs as a persistent procedure, so it has no parameters. Instead, there is an internal procedure, called fetchOrder, that implements this specific request for a single Order. Because it takes dsOrder as an INPUT-OUTPUT parameter, passed BY-REFERENCE, it is the caller's instance of the ProDataSet that is used, not the one represented by the include files at the top of OrderSource.p.

The procedure uses the QUERY-PREPARE method to get the right order, and then fills the ProDataSet, as shown:

```
PROCEDURE fetchOrder:
  DEFINE INPUT        PARAMETER piOrderNum AS INTEGER NO-UNDO.
  DEFINE INPUT-OUTPUT PARAMETER DATASET    FOR dsOrder.

  QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.OrderNum = " +
    STRING(piOrderNum) +
    ", FIRST Customer OF Order, FIRST SalesRep OF Order").
/* Note that this reference to dsOrder is not using the local definition but
   rather the actual dataset instance being passed in by reference. */
  IF VALID-HANDLE(DATASET dsOrder:GET-BUFFER-HANDLE(1):DATA-SOURCE) THEN
    DATASET dsOrder:FILL().
  ELSE DO:
    DATASET dsOrder:GET-BUFFER-HANDLE(1):TABLE-HANDLE:ERROR-STRING =
      "Data-Sources not attached".
    DATASET dsOrder:ERROR = TRUE.
  END.
  RETURN.
END PROCEDURE. /* fetchOrder */
```

The calling procedure runs the attach method (defined later) before running fetchOrder, so that everything has been set up properly for the FILL. The code checks to make sure that there is a Data-Source for the top-level buffer before proceeding with the FILL. If not, it sets the ERROR attribute for the ProDataSet and an error message on the top-level temp-table, which the caller can inspect.

Later in this chapter, you will write another procedure that represents the Order entity itself. This will have the ProDataSet instance that is actually used in the application, and it will define the API that other procedures, such as a client window, would use to access the ProDataSet. That API will include a fetchOrder procedure.

Why, then, is this version of fetchOrder here in the data access procedure? Since it needs to use a specific database query to prepare the top-level table, it is better to put the procedure into the data access object. The fetchOrder procedure in the Order entity itself will turn around and run this one to maintain the right level of encapsulation in the objects.

Two of the FILL event procedures from OrderEvents.p are preserved here, postOlineFill and postItemRowFill. Procedure postOlineFill calculates the OrderTotal in the ttOrder table. For example:

```
PROCEDURE postOlineFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  DEFINE VARIABLE dTotal AS DECIMAL NO-UNDO.

  /* Here as well "ttOline" uses the local definition for compilation, but
     points to the ttOline table in the input parameter at run time. */
  FOR EACH ttOline WHERE ttOline.OrderNum = ttOrder.OrderNum:
    dTotal = dTotal + ttOline.ExtendedPrice.
  END.
  ttOrder.OrderTotal = dTotal.
END PROCEDURE. /* postOlineFill */
```

Procedure postItemRowFill edits the ItemName field in the ttItem table. For example:

```
PROCEDURE postItemRowFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  DEFINE VARIABLE cItemTypes AS CHARACTER NO-UNDO
    INITIAL "BASEBALL,CROQUET,FISHING,FOOTBALL,GOLF,SKI,SWIM,TENNIS".
  DEFINE VARIABLE cType      AS CHARACTER NO-UNDO.
  DEFINE VARIABLE iType      AS INTEGER   NO-UNDO.
  DEFINE VARIABLE iTypeNum   AS INTEGER   NO-UNDO.

  DO iType = 1 TO NUM-ENTRIES(cItemTypes):
    cType = ENTRY(iType, cItemTypes).
    IF INDEX(ttItem.ItemName, cType) NE 0 THEN
      ttItem.ItemName = REPLACE(ttItem.ItemName, cType, cType).
  END.
END PROCEDURE.  /* postItemRowFill */
```

There is something important to note about these two procedures. They contain direct references to fields such as ttItem.ItemName, which is possible because the temp-table definitions are included in the procedure. But remember that the local instance of the ProDataSet and its temp-tables is used for definition only. When the AVM invokes these procedures during the FILL, it passes in the current ProDataSet instance implicitly BY-REFERENCE. The AVM not only adjusts all references to the ProDataSet itself to point to that external ProDataSet, but also all references to its temp-tables and their fields. This gives you the best of both worlds, as it were: a local static definition that makes the code simpler and clearer, but an automatic reference at run time to an externally defined ProDataSet that is passed into the procedure without any copying or other overhead.

The code to attach the Data-Sources has been separated out, however, so that it can be executed not only for a FILL but also on a save. This is in the function attachDataSet, which also runs the SET-CALLBACK-PROCEDURE for the two FILL events. For example:

```
FUNCTION attachDataSet RETURNS LOGICAL (INPUT phDataSet AS HANDLE):
  phDataSet:GET-BUFFER-HANDLE("ttOline"):SET-CALLBACK-PROCEDURE
    ("AFTER-FILL", "postOlineFill", THIS-PROCEDURE).
  phDataSet:GET-BUFFER-HANDLE("ttItem"):SET-CALLBACK-PROCEDURE
    ("AFTER-ROW-FILL", "postItemRowFill", THIS-PROCEDURE).
  phDataSet:GET-BUFFER-HANDLE("ttOrder"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcOrder:HANDLE, "Customer.Name,CustName").
  phDataSet:GET-BUFFER-HANDLE("ttOline"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcOline:HANDLE).
  phDataSet:GET-BUFFER-HANDLE("ttItem"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcItem:HANDLE).
END FUNCTION. /* attachDataSet */
```

There is also a corresponding detachDataSet function. For example:

```
FUNCTION detachDataSet RETURNS LOGICAL (INPUT phDataSet AS HANDLE):
  DEFINE VARIABLE iBuff AS INTEGER NO-UNDO.

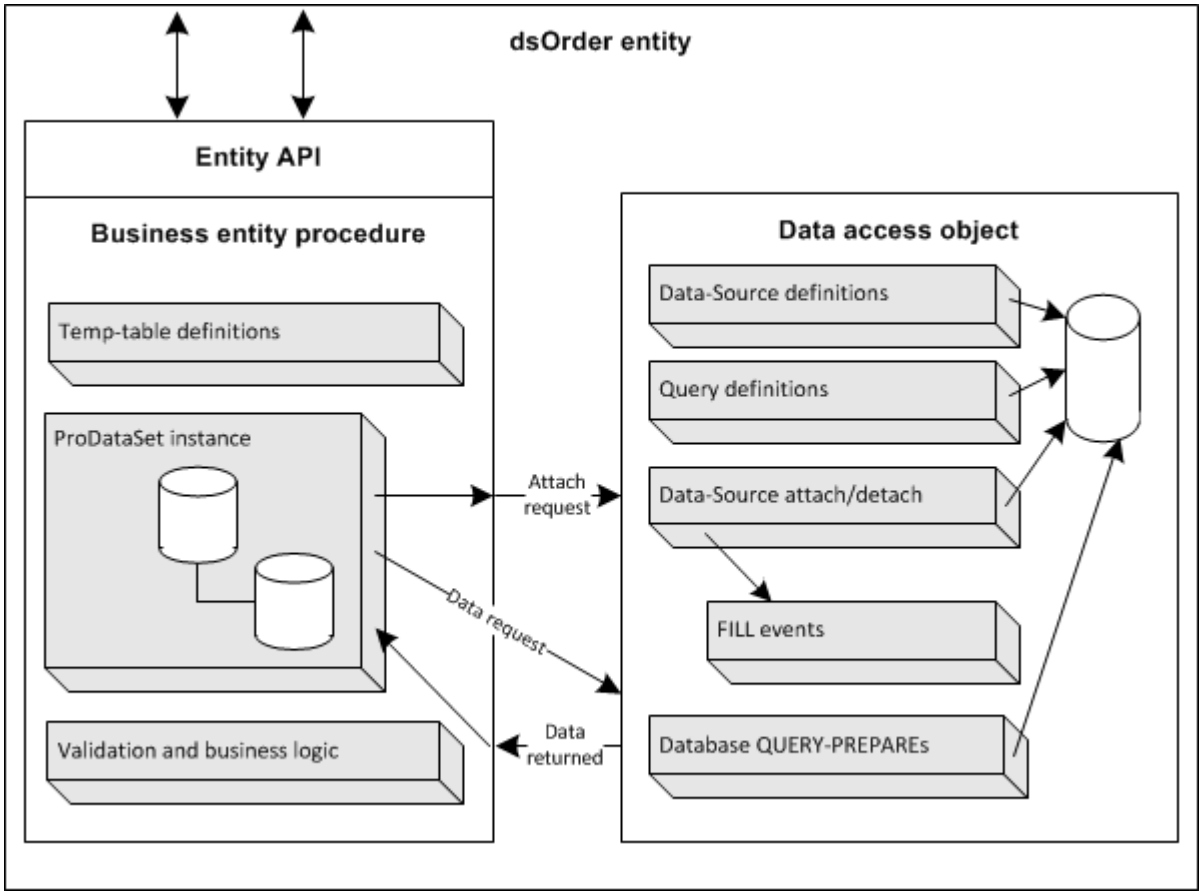
  DO iBuff = 1 TO DATASET dsOrder:NUM-BUFFERS:
    phDataSet:GET-BUFFER-HANDLE(iBuff):DETACH-DATA-SOURCE().
  END.
END FUNCTION. /* detachDataSet */
```

Building a business entity procedure to support the ProDataSet

Next, you will write the procedure that really represents the dsOrder ProDataSet itself. This is a variation of the OrderSupport.p procedure you have written before. Call it OrderEntity.p.

The goal of this part of the exercise is to create a business entity object that manages `dsOrder`, expanding on the data access procedure you created earlier. The following figure is a sketch of the larger entity object.

Figure 17: Business entity procedure



The data access object becomes part of a larger `Order` entity or business object. The entity presents an API to the outside world that supports specific types of access to the object. Any requests that need to fill or save data use the business entity procedure as a gateway to the data managed in the data access object.

Validation logic for saving changes and other business logic associated with the ProDataSet are also encapsulated within the entity so that the logic is always executed consistently whenever the ProDataSet is referenced. Now you are working with a true business object.

Procedure `OrderEntity.p` is a simple example of the business entity procedure. It manages the API the client window uses to test the procedures and coordinate access to the data. Later, you will attach validation logic to it as well.

There is not very much code in the procedure, partly because some of it is in `OrderSource.p` and partly because some of it will be moved to a new general purpose dynamic procedure to handle the whole save-changes process.

This is the procedure that actually "owns" the ProDataSet instance on the server side. It is this procedure's ProDataSet instance that the `FILL` events and the Data-Sources are attached to, and this instance that is actually filled with data to be passed back to the client.

OrderEntity.p will be run PERSISTENT in support of the client window, as a server side procedure, either locally in the client process or remotely. It first gets the handle of the ProDataSet and uses this to set up a CLOSE trigger for the procedure to clean up, as shown:

```
/* OrderEntity.p -- Entity procedure for ProDataSet dsOrder */
{dsOrderTT.I}
{dsOrder.i}

DEFINE VARIABLE hDataSet    AS HANDLE  NO-UNDO.
DEFINE VARIABLE hSourceProc AS HANDLE  NO-UNDO.
DEFINE VARIABLE lError      AS LOGICAL NO-UNDO.

hDataSet = DATASET dsOrder:HANDLE.

ON CLOSE OF THIS-PROCEDURE DO:
  DYNAMIC-FUNCTION("detachDataSet" IN hSourceProc, INPUT hDataSet).
  DELETE PROCEDURE hSourceProc.
  DELETE PROCEDURE THIS-PROCEDURE.
END.
```

Remember that you cannot obtain a handle such as hDataSet in the procedure main block and then use it inside internal procedures that have a ProDataSet as a parameter passed BY-REFERENCE. The handle will not be valid in those procedures because they are pointing to the external ProDataSet instance. This code gets the handle at the top of the main block simply because there is a limitation in the DYNAMIC-FUNCTION syntax that does not recognize the form DATASET dsOrder:HANDLE as a parameter. (This is actually the same restriction that already exists for buffer handle references.) Thus, you need to obtain the handle in advance so that you can simply name the handle variable in the function reference.

Next, the main block starts the data access procedure and runs the attachDataSet function for the local ProDataSet instance:

```
RUN OrderSource.p PERSISTENT SET hSourceProc.

lError = DYNAMIC-FUNCTION("attachDataSet" IN hSourceProc, hDataSet).
```

Because the methods in OrderSource.p operate on the ProDataSet instance passed into it, a single instance of the procedure itself could support any number of external instances of the ProDataSet. Therefore, in your application you could check whether there is already a running instance of a procedure such as this and use its handle if there is.

Remember that the fetchOrder procedure in OrderSource.p is intended to be called from its enclosing business entity procedure. Following is the version of fetchOrder that a procedure outside the Order entity calls to get an Order back. It defines its own OUTPUT parameter explicitly BY-VALUE to assure that no other procedure tries to pass in another ProDataSet instance BY-REFERENCE. This is because it is the instance in OrderEntity.p that has been attached to the FILL events and Data-Sources, and only that instance can be filled properly.

Procedure fetchOrder turns around and passes in the Order Number value to fetchOrder in OrderSource.p, along with its own ProDataSet instance BY-REFERENCE. In this way, it makes sure the local instance is used. For example:

```
PROCEDURE fetchOrder:
  DEFINE INPUT  PARAMETER piOrderNum AS INTEGER NO-UNDO.
  DEFINE OUTPUT PARAMETER DATASET    FOR dsOrder BY-VALUE.

  /* This turns around and runs an equivalent procedure in the Data-Source
     procedure, passing in the static dataSet. */
  DYNAMIC-FUNCTION("attachDataSet" IN hSourceProc, hDataSet).
  RUN fetchOrder IN hSourceProc (INPUT piOrderNum,
    INPUT-OUTPUT DATASET dsOrder BY-REFERENCE).
  DYNAMIC-FUNCTION("detachDataSet" IN hSourceProc, INPUT hDataSet).
END PROCEDURE. /* fetchOrder */
```

Finally, there is a saveChanges procedure in OrderEntity.p. This attaches the Data-Sources to make sure they are there for the save, and then runs a new procedure that you will write next, which captures all the standard save logic in a single dynamic procedure. It then detaches the Data-Sources from the ProDataSet, as shown:

```
PROCEDURE saveChanges:
  DEFINE INPUT-OUTPUT PARAMETER DATASET FOR dsOrder.

  DEFINE VARIABLE hDataSet AS HANDLE NO-UNDO.

  hDataSet = DATASET dsOrder:HANDLE.
  DYNAMIC-FUNCTION("attachDataSet" IN hSourceProc, INPUT hDataSet).
  RUN commitChanges.p (INPUT-OUTPUT DATASET dsOrder BY-REFERENCE).
  DYNAMIC-FUNCTION("detachDataSet" IN hSourceProc, INPUT hDataSet).
END PROCEDURE. /* saveChanges */
```

As you can see, saveChanges is actually entirely generic. It could operate on any ProDataSet, as long as it knows the procedure handle of the data access procedure that is stored in hSourceProc. If you had a generic session manager that coordinated running procedures on the server, your client could simply run saveChanges in that session manager, which could use the ProDataSet parameter to identify the ProDataSet type or name (dsOrder in this case), and set hSourceProc to the data access procedure for that ProDataSet. Since our simple example does not have such a manager, we put saveChanges into the Order entity itself.

This is the end of the code for OrderEntity.p.

Building general update procedures for client and server

In Updating Data with ProDataSets on page 135, we introduced you to all the syntax to support logging changes and applying them to the database. Some of that work was more complicated than it would normally need to be because we needed to show you all the attributes and methods that are available to you when you need to do something very specific in your application. However, the high-level methods such as GET-CHANGES, SAVE-ROW-CHANGES, and MERGE-CHANGES can do all the work of each major step in the process in many (perhaps most) cases, which removes the burden of having to pay strict attention to what obscure supporting attributes like ORIGIN-HANDLE and BEFORE-ROWID do. As we explained in Updating Data with ProDataSets on page 135, those attributes and methods are there so that you can do everything that the higher-level methods do when they do not provide exactly what you need.

In this section, we will show you how to simplify the update process even further, providing a simple dynamic procedure for the client side and for the server side of an application that can collect and apply changes for any ProDataSet, using its handle and the attributes that let you inspect its structure.

Building the client side change handler

The first of these procedures, called `clientChanges.p`, runs on the client side of the application. (As with earlier examples, these simplified procedures do not actually use the AppServer, but can very easily be extended to do so.) This can be run from a client procedure such as the `BtnSave` trigger in the `Order` update window, and later you will do that.

So, let us create procedure `clientChanges.p` based on code from the `CHOOSE` trigger for `BtnSave` in `dsOrderWinUpd.w`.

`ClientChanges.p` takes the ProDataSet handle as input, along with the supporting entity procedure handle. As we noted, this procedure handle parameter would be unnecessary if there were a single service procedure on the server to handle all updates. It returns any status messages as `OUTPUT`, as shown:

```
/* clientChanges.p -- client side of generic commitChanges support */
DEFINE INPUT  PARAMETER phDataSet     AS HANDLE    NO-UNDO.
DEFINE INPUT  PARAMETER phSupportProc AS HANDLE    NO-UNDO.
DEFINE OUTPUT PARAMETER pcStatus      AS CHARACTER NO-UNDO.
```

It defines variables for the handles of the change ProDataSet, query, buffer, and a buffer counter:

```
DEFINE VARIABLE hBuffer    AS HANDLE  NO-UNDO.
DEFINE VARIABLE hDSChanges AS HANDLE  NO-UNDO.
DEFINE VARIABLE hQuery     AS HANDLE  NO-UNDO.
DEFINE VARIABLE iBuffer    AS INTEGER NO-UNDO.
```

It creates a dynamic ProDataSet, makes it like the `INPUT` ProDataSet, and extracts changes from the input ProDataSet:

```
CREATE DATASET hDSChanges.
hDSChanges:CREATE-LIKE(phDataSet).
hDSChanges:GET-CHANGES(phDataSet).
```

It then runs this generic `saveChanges` procedure in the support procedure handle, passing the change ProDataSet as `INPUT-OUTPUT`. This is done `BY-REFERENCE` so that in the case where this procedure and the supporting entity procedure are in the same session, the change ProDataSet does not need to be copied:

```
RUN saveChanges IN phSupportProc
  (INPUT-OUTPUT DATASET-HANDLE hDSChanges BY-REFERENCE).
```

If any errors were logged, it places them into the status parameter for return to the caller. For example:

```
/* Check the ERROR status that might have been returned. */
IF hDSChanges:ERROR THEN
  DO iBuffer = 1 TO phDataSet:NUM-BUFFERS:
    CREATE QUERY hQuery.
    hBuffer = hDSChanges:GET-BUFFER-HANDLE(iBuffer).
    hQuery:ADD-BUFFER(hBuffer).
    hQuery:QUERY-PREPARE("FOR EACH " + hBuffer:NAME).
    hQuery:QUERY-OPEN().
    hQuery:GET-FIRST().

    DO WHILE NOT hQuery:QUERY-OFF-END:
      IF hBuffer:ERROR THEN
        pcStatus = pcStatus + hBuffer:ERROR-STRING + CHR(10).
      hQuery:GET-NEXT().
    END.

    hQuery:QUERY-CLOSE().
    DELETE OBJECT hQuery.
  END.
```

And finally, it merges the final change records back into the ProDataSet passed in, deletes the dynamic change ProDataSet, and does a synchronize on behalf of the user interface to make sure it redisplays any parts of the ProDataSet that are on the screen. For example:

```
hDSChanges:MERGE-CHANGES(phDataSet).
DELETE OBJECT hDSChanges.

phDataSet:GET-BUFFER-HANDLE(1):SYNCHRONIZE().
```

It is worth noting here that you need to decide when to use `MERGE-CHANGES` in your own applications. An additional parameter to `clientChanges.p` could provide some alternatives, or the generic procedure could leave the merge up to the caller entirely. If there were any errors, you might not want to do a `MERGE-CHANGES` because this rejects any change that has been marked with the `ERROR` or `REJECTED` attribute on the record, which restores the original record values. This overwrites whatever changes the user has made. Typically, you are more likely to want to leave the changes in the user interface alone, point them out to the user using the Status return messages, and give the user the opportunity to correct the errors without having to re-enter all changes. In this case, you should run `MERGE-CHANGES` only when the updates all succeeded. This is entirely up to you. In some cases the `SYNCHRONIZE` method might also better be left up to the caller.

So now you have a general client-side change handler that you can call from any procedure that has accumulated changes in a ProDataSet. Now, move on to the server side of the update.

Building the server side change handler

On the server side, there can also be a generic procedure that applies all changes to a ProDataSet. As we explained in Updating Data with ProDataSets on page 135, there is a `SAVE-ROW-CHANGES` method to apply one row of changes to the database, but not a `SAVE-CHANGES` method for a temp-table or for the entire ProDataSet. The reason for this is there might be many ways in which you need to control the update process, including:

- **The transaction scope** — Are all updates made as part of a single transaction that fails in its entirety if any change is not valid? Are all updates to a single table made together? Or is each individual change a separate transaction that can succeed or fail independent of others? These questions are the reason why we support both an `ERROR` and a `REJECTED` attribute for ProDataSet rows. A change that fails or conflicts

with a change made by another user is REJECTED and also made an ERROR. Any other change that is backed out because it is part of the same transaction is REJECTED whether it caused an error or not. You can check these flags back in the caller to understand what made it into the database and what did not, and why.

- **The order of applying changes to different tables** — In a parent-child ProDataSet or when there are multiple top-level tables, which changes should be applied first? Normally, you would apply changes from the top down, but there might be exceptions to that. And if there are multiple top-level tables, does it matter to you which order changes are applied in? By writing your own code you have complete control over this. No default would satisfy all situations.
- **The order of creates, deletes, and modifies to a single table** — Do you want deletes applied before or after creates and updates? This might depend on your data definitions or business logic. The ROW-STATE attribute has a numeric value that orders rows so that unmodified rows are first, followed by deleted rows, modified rows, and created rows, in that order. You can take advantage of this default ordering, but you can also change it as necessary.

This section shows you a sample for a general purpose procedure that makes default decisions about these variants. Each row is a single transaction. Tables are processed from the top of the hierarchy down (and multiple top-level tables are processed in the order in which they appear in the ProDataSet definition). And changes are processed in ROW-STATE Order. You could, of course, extend or modify this procedure for other defaults and to accept parameters to change the behavior.

The example procedure is commitChanges.p. It handles changes to any ProDataSet with any number of tables in its hierarchy. It takes the ProDataSet handle as its INPUT-OUTPUT parameter. The variables represent the handle to the current top-level buffer and a buffer counter, as shown:

```
/* commitChanges.p */
DEFINE INPUT-OUTPUT PARAMETER DATASET-HANDLE hDataSet.

DEFINE VARIABLE hTopBuff AS HANDLE  NO-UNDO.
DEFINE VARIABLE iBuff    AS INTEGER NO-UNDO.
```

The procedure starts its main loop through all top-level buffers, using the NUM-TOP-BUFFERS counter and the GET-TOP-BUFFER method to return each one in turn. These are the temp-table buffers with no parent. They are returned in the order they appear in the ProDataSet definition. There is one special case for which the loop must check. Because of the FILL behavior, children of a REPOSITION Data-Relation show up in the list of top-level buffers, even though they are not really top-level for the purpose of walking through the hierarchy of the ProDataSet as this procedure does. For this reason, the loop contains a check to see if there is a PARENT-RELATION for the buffer. If there is, then it is really a child of a reposition relation, not a true top-level buffer, and it is skipped.

For each top-level buffer, the iterative procedure traverseBuffers is run to walk down that branch of the ProDataSet definition, as shown:

```
DO  iBuff = 1 TO hDataSet:NUM-TOP-BUFFERS:
  hTopBuff = hDataSet:GET-TOP-BUFFER(iBuff).

  /* Skip the reposition children. */
  IF hTopBuff:PARENT-RELATION NE ? THEN NEXT.

  RUN traverseBuffers (hTopBuff).
END.  /* END DO iBuff */
```

Procedure traverseBuffers serves only to iterate down through any number of levels of parent-child tables. It runs another internal procedure saveBuffer on itself and then loops through each child of the current buffer. NUM-CHILD-RELATIONS returns the number of relations for which this buffer is the parent, and GET-CHILD-RELATION returns each one in turn.

Since GET-CHILD-RELATION returns the handle of the Data-Relation object, you need to follow that to its CHILD-BUFFER to get the buffer handle of each of the current's buffer's children, as shown:

```
PROCEDURE traverseBuffers:
  DEFINE INPUT  PARAMETER phBuffer AS HANDLE NO-UNDO.

  DEFINE VARIABLE iChildRel AS INTEGER NO-UNDO.

  RUN saveBuffer(phBuffer).

  DO iChildRel = 1 TO phBuffer:NUM-CHILD-RELATIONS:
    RUN traverseBuffers
     (phBuffer:GET-CHILD-RELATION(iChildRel):CHILD-BUFFER).
  END. /* END DO iChildRel */
END PROCEDURE.   /* traverseBuffers */
```

Procedure saveBuffer does the actual work of running SAVE-ROW-CHANGES, as shown:

```
PROCEDURE saveBuffer:
  DEFINE INPUT  PARAMETER phBuffer AS HANDLE NO-UNDO.

  DEFINE VARIABLE hBeforeBuff AS HANDLE NO-UNDO.
  DEFINE VARIABLE hBeforeQry  AS HANDLE NO-UNDO.

  hBeforeBuff = phBuffer:BEFORE-BUFFER.
```

The buffer handle passed in is actually the after-table buffer for the current table. You want to run SAVE-ROW-CHANGES on the **before-table** buffer, especially since in the case of a Delete there is no after-table row to process. Variable hBeforeBuff points to that buffer.

The VALID-HANDLE test checks whether there is a before-table for this temp-table at all. If the table does not have a BEFORE-TABLE in its definition and has not been enabled for update by setting TRACKING-CHANGES to true, then there will not be a before-table. Otherwise, the before-table could contain zero, one, or many rows, depending on how many rows in that table have been changed. The query simply walks through all those rows and runs SAVE-ROW-CHANGES on each one. SAVE-ROW-CHANGES starts its own transaction if there is none active, so each change is saved independently. If the save fails because of invalid data or because it had been changed by another user, the AVM sets the ERROR attribute on the row.

The procedure checks this and also sets the REJECTED attribute, so that the caller knows row by row which rows were successfully updated into the database and which ones were not. For example:

```
  IF VALID-HANDLE(hBeforeBuff) THEN DO:
    CREATE QUERY hBeforeQry.
    hBeforeQry:ADD-BUFFER(hBeforeBuff).
    hBeforeQry:QUERY-PREPARE("FOR EACH " + hBeforeBuff:NAME).
    hBeforeQry:QUERY-OPEN().
    hBeforeQry:GET-FIRST().

    DO WHILE NOT hBeforeQry:QUERY-OFF-END:
      hBeforeBuff:SAVE-ROW-CHANGES().
      /* If there was an error signal that this row did not make it into the
         database. */
      IF hBeforeBuff:ERROR THEN
        hBeforeBuff:REJECTED = TRUE.
      hBeforeQry:GET-NEXT().
    END.  /* END DO WHILE NOT QUERY-OFF-END */
    DELETE OBJECT hBeforeQry.
  END. /* END DO IF VALID-HANDLE */
END PROCEDURE. /* saveChanges */
```

Remember that SAVE-ROW-CHANGES takes two optional arguments:

- The buffer to save, if there is more than one buffer in the Data-Source and the modified buffer is not the first one.
- The name of a field not to copy to the database buffer because its value is assigned by a database trigger. Normally this is the primary key for a newly created row.

Because this is a generic save procedure, there is no straightforward way to specify these parameters if they are needed. This example uses the default.

That is the end of the code for commitChanges.p.

Changing the window procedure to use the new procedures

You can test these new procedures with a window procedure very much like dsOrderWinUpd.w.

To update the code:

1. Copy dsOrderWinUpd.w to dsOrderWinAdv.w (adv for "advanced").
2. Add this new **Definition** to it. This is the handle of the Order entity. This is now a persistent procedure, rather than running OrderSupport.p as a standalone .p:

```
DEFINE VARIABLE hOrderSupport AS HANDLE NO-UNDO.
```

3. Change the **Main Block** to run OrderEntity.p as a persistent procedure:

```
MAIN-BLOCK:
DO ON ERROR   UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
   ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
  RUN enable_UI.
  RUN OrderEntity.p PERSISTENT SET hOrderSupport.
  . . .
```

4. Change the CLOSE trigger in the **Main Block** to give the Order entity procedure a chance to clean up:

```
ON CLOSE OF THIS-PROCEDURE DO:
  APPLY "CLOSE" TO hOrderSupport.
  RUN disable_UI.
END.
```

5. In the LEAVE trigger for iOrderNum, change the run of OrderMain.p to run fetchOrder in the Order entity procedure. For example:

```
TEMP-TABLE ttOline:TRACKING-CHANGES = FALSE.

DATASET dsOrder:GET-RELATION(1):QUERY:QUERY-CLOSE().
DATASET dsOrder:GET-RELATION(2):QUERY:QUERY-CLOSE().
DATASET dsOrder:EMPTY-DATASET.

RUN fetchOrder IN hOrderSupport
  (INPUT iOrderNum, OUTPUT DATASET dsOrder).
```

Note that the OUTPUT DATASET dsOrder parameter is not passed BY-REFERENCE, and should not be. If OrderEntity.p were running in a separate session, it would be ignored and would make no difference. But when it is running in the same session, you want the ProDataSet that is initialized, attached, and returned by fetchOrder to be the one the client uses, not the client's locally defined ProDataSet, which is not attached to any Data-Sources.

6. In the CHOOSE trigger for BtnSave, remove the code that is now in clientChanges.p so that the trigger is reduced to this:

```
DO:
  DEFINE VARIABLE hDSOrder AS HANDLE NO-UNDO.

  TEMP-TABLE ttOline:TRACKING-CHANGES = FALSE.
  cStatus = "".
  hDSOrder = DATASET dsOrder:HANDLE.

  RUN clientChanges.p (hDSOrder, hOrderSupport, OUTPUT cStatus).

  DISPLAY cStatus WITH FRAME dsFrame.
  /* Re-enable the Order Number to select another Order. Also, set
     TRACKING-CHANGES back to TRUE to capture any further changes made
     to this Order. */
  ASSIGN
    iOrderNum:SENSITIVE IN FRAME dsFrame = TRUE
    SELF:SENSITIVE                       = FALSE
    TEMP-TABLE ttOline:TRACKING-CHANGES  = TRUE.
END.
```

You can see how much of what this trigger does has been replaced by the generic support procedure clientChanges.p.

That is it. If you save this and run it, it should work exactly as it did before, with much less code in the client procedure and much less code in the support procedures because of clientChanges.p and commitChanges.p. Also, your Order support logic has been better organized into procedures that do specific parts of the job in ways that you can use as a standard for other ProDataSets.

Running standard validation procedures on update

There are no standard event hooks for SAVE-ROW-CHANGES as there are for FILL because, generally, validation logic will be executed before or possibly after the SAVE-ROW-CHANGES method, so there is nowhere for the AVM to execute standard events. Instead, you can execute whatever validation logic or other business logic you need to before or after running SAVE-ROW-CHANGES.

Because commitChanges.p simulates what a SAVE-CHANGES method on the whole ProDataSet would do, there are places within this procedure where standard event hooks can go. This section shows you how to add event hooks of this kind to your commitChanges procedure.

To add event hooks to commitChanges.p:

1. Add a HANDLE variable hSourceProc and assign it to the SOURCE-PROCEDURE, as shown:

```
DEFINE VARIABLE hSourceProc AS HANDLE NO-UNDO.

hSourceProc = SOURCE-PROCEDURE.
```

   Validation procedures will be executed in the calling procedure from the internal procedure saveBuffer. At the time commitChanges.p is run, the calling procedure's handle is the SOURCE-PROCEDURE. By running validation in the caller, you can code validation logic directly into the entity procedure or into another procedure that you run as a super procedure of the entity. The problem is that from within an internal procedure like saveBuffer, called from the main block, the value of SOURCE-PROCEDURE is **commitChanges.p** itself. For this reason the new code saves off the value of SOURCE-PROCEDURE in the main block so that it can be referenced in saveBuffer.

   The rest of the changes are to the internal procedure saveBuffer.

2. Add a variable definition to saveBuffer for a character string to hold a procedure name:

```
DEFINE VARIABLE cLogicProc S CHARACTER NO-UNDO.
```

   The new code assumes a naming convention of the temp-table name plus Delete, Create, or Modify for validation logic procedures. For example, a procedure to execute when a ttOline record is modified is called ttOlineModify. You will code just such a procedure. Add this statement to generate the procedure name at the beginning of the DO block that walks through the dynamic query:

```
DO WHILE NOT hBeforeQry:QUERY-OFF-END:
  cLogicProc = phBuffer:TABLE-HANDLE:NAME +
  IF hBeforeBuff:ROW-STATE = ROW-DELETED THEN "Delete"
  ELSE IF hBeforeBuff:ROW-STATE = ROW-CREATED THEN "Create"
  ELSE "Modify".
```

Before the code runs the logic procedure it needs to find the right after-table row so that the validation procedure can see its values. Remember that the dynamic query `hBeforeQry` is navigating the rows of the before-table. The after-table rows are not automatically found, so this statement is required to bring the corresponding after-table row into its own buffer:

```
phBuffer:FIND-BY-ROWID(hBeforeBuff:AFTER-ROWID).
```

Now you can run the logic procedure if it exists. The code passes the ProDataSet as `INPUT BY-REFERENCE` just as the AVM does for `FILL` event procedures. Remember that an `INPUT` parameter passed `BY-REFERENCE` to a local procedure is effectively the same as an `INPUT-OUTPUT` parameter, because any changes made are visible to the caller. Nonetheless, we pass the ProDataSet as `INPUT` for consistency with the calling sequence of `FILL` events, as shown:

```
RUN VALUE(cLogicProc) IN hSourceProc
  (INPUT DATASET-HANDLE hDataSet BY-REFERENCE) NO-ERROR.
```

The validation procedure can set the `ERROR` attribute for the row if it fails validation. If this is not the case then it runs `SAVE-ROW-CHANGES` on the row.

After this, if the `ERROR` attribute has been set either by the validation procedure or by `SAVE-ROW-CHANGES` itself, the code sets `ERROR` on the ProDataSet itself. This is because when your code sets `ERROR` on a row programmatically, the AVM does not set it on the ProDataSet. This happens only when the AVM detects an error internally and sets `ERROR` on both the row and the ProDataSet.

For example:

```
IF NOT hBeforeBuff:ERROR THEN
  hBeforeBuff:SAVE-ROW-CHANGES().
  /* If there was an error signal that this row did not make it into the
     database. */
  IF hBeforeBuff:ERROR THEN
    ASSIGN hDataSet:ERROR = TRUE
  hBeforeBuff:REJECTED = TRUE.
  hBeforeQry:GET-NEXT().
```

As before, the code also sets the `REJECTED` flag.

Now you have a general-purpose save procedure that also runs general-purpose business logic at key points during the update process.

3. Open the procedure `OrderEntity.p`, where `Order`-specific business logic goes, and create an internal procedure `ttOlineModify` that compares some values in the before- and after-tables for the current `ttOline` row.

If the `Quantity` order has been doubled or more, then the procedure rewards the customer by increasing the discount by 20%. On the other hand, if the `Quantity` has been cut by half or more, this is considered an error and the procedure sets the `ERROR` flag and also an `ERROR-STRING` message. For example:

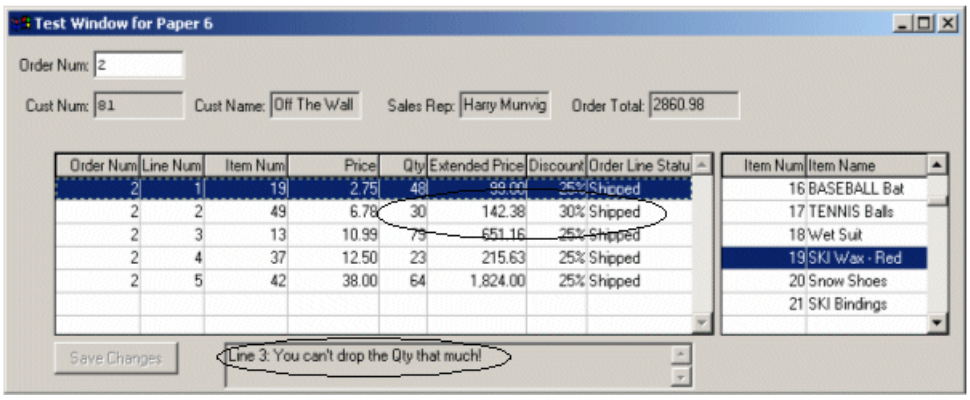
```
PROCEDURE ttOlineModify:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  /* If the customer doubled the quantity ordered, then increase the
     discount by 20%. */
  IF ttOline.Qty >= (ttOlineBefore.Qty * 2) AND
    ttOline.Discount = ttOlineBefore.Discount THEN
    ttOline.Discount = ttOlineBefore.Discount * 1.2.
  ELSE IF ttOline.Qty <= (ttOlineBefore.Qty * .5) THEN
    ASSIGN
      BUFFER ttOline:ERROR        = TRUE
      BUFFER ttOline:ERROR-STRING = "Line " +
        STRING(ttOline.LineNum) + ": You can't drop the Qty that much!".
  RETURN.
END PROCEDURE. /* ttOlineModify */
```

Even though `saveBuffer` passes the ProDataSet as a dynamic `DATASET-HANDLE` (because it has only a dynamic reference to the ProDataSet), `ttOlineModify` can receive it as a static `DATASET`, matching its local definition, so that you can code static ABL statements for your business logic.

Now if you re-run the window you can see the effects.

4. Run `dsOrderWinAdv.w`. Select an `Order`, increase one `Qty` by more than double, and cut another by more than half:



In this case, we doubled the `Qty` for `Line Number 2`, and the discount was increased accordingly. We decreased the `Qty` for `Line Number 3` by more than half, and that change was rejected and the message displayed. This is an example of why you might not want to run `MERGE-CHANGES` on the whole ProDataSet when not all changes were successful. The original values for the rejected row are redisplayed and the user's changes erased. It would probably be better to run `MERGE-ROW-CHANGES` just on successful updates and leave the incorrect values for other rows so that they can be seen and more easily corrected.

Summary

This chapter has introduced the concept of a business entity that manages a ProDataSet, along with a standard design for a business entity. You also learned how to create dynamic procedures that can be used to handle parts of the update process for almost any ProDataSet and how to execute standard validation procedures from those procedures.

ProDataSets have the advantage that you can attach an `ERROR-STRING` to each row independently rather than stringing together all errors for the transaction. In addition, you can set `ERROR-STRING` for messages that you do not want treated as `error`s (in which case the attribute name is a bit of a misnomer). In this way, you can return informational messages to the caller without signaling error by setting the `ERROR` flag.

11

Data Access and Business Entity Objects

This book has developed a series of increasingly advanced examples on how to use the ProDataSet object. As the later chapters illustrate, you can mask some of the advanced features of the ProDataSet by standardizing your use of ProDataSets and building parts of the data management behavior into generic procedures to handle any ProDataSet using dynamic access through its handle. The sample procedure `commitChanges.p` you developed earlier demonstrates these techniques.

You can also establish standard templates for procedures that provide and use static definitions of ProDataSets and their temp-tables as well as business logic procedures with object-specific ABL logic.

This chapter discusses in more detail some of the basic principles you should consider when you design your procedures in this way. In particular, it continues the discussion of how you should consider your Data Access objects and procedures as distinct from your Business Entity objects and procedures. (Data Access objects define and reference the database tables or other sources of data for your ProDataSets. Business Entity objects apply your business logic to the internal data definitions in your temp-tables and the ProDataSets themselves.)

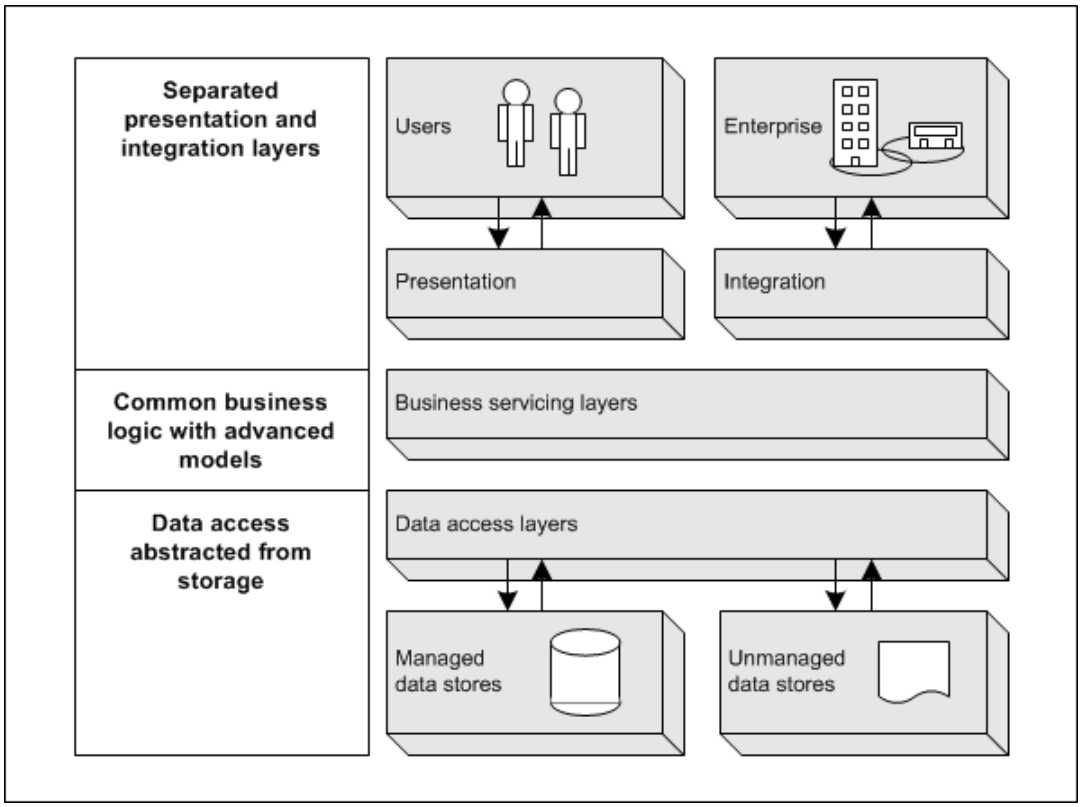
For details, see the following topics:

- ProDataSets and the OpenEdge Reference Architecture
- Data Access object
- Business Entity object
- Business logic options
- Summary

ProDataSets and the OpenEdge Reference Architecture

The OpenEdge Reference Architecture (ORA) represents an effort to formalize the recommended structure of a new or a transformed OpenEdge application, and the reasons for recommending that structure. The remainder of this chapter describes several of the layers of the architecture in more detail. The following figure shows the high-level layers of the architecture.

Figure 18: OpenEdge Reference Architecture



In particular, these are the elements of the architecture that pertain to how you use ProDataSets in your application.

The Data Access object discussed in the next section implements the data access layer of the architecture. Its goal is to provide a separation between the physical data in your database or in any other type of data source and the logic that uses that data in the rest of the application. The data itself may be in what we refer to as a managed or unmanaged data store. A managed data store is typically your OpenEdge database or other database you can access through an OpenEdge DataServer. An unmanaged data store could be a set of XML or JSON documents, a flat file, a data stream coming from a scanner or other device, or anything else that is not a true database.

Typically the ProDataSet represents the internal data definition used by the application. If the data source is in a managed data store, then typically you can define ABL Data-Source objects to specify what tables the data comes from, along with any joins or field mapping that must be done to transfer data to the ProDataSet. If the data source is in an unmanaged data store, then you have to write custom ABL logic that you use as `FILL` event procedures for the ProDataSet to define how it is populated. In either case, the overall goal is to make the ultimate source of the data as transparent as possible to the rest of the application.

The Business Entity object is a key part of the implementation of the Business Servicing Layer of the architecture. It can encapsulate one or more ProDataSet definitions that make up a logical application object at the level at which your application typically needs to deal with its data. This might be anything from a single table to a large number of related tables that typically have to be processed together. The Business Entity also includes basic validation logic for its data and an API for all the types of both read and update calls that other objects need to make when they use the object's data.

Multiple Business Entities together can be used as part of a larger business task or workflow. A Business Entity can also be packaged as a Web Service to be called from outside the OpenEdge environment altogether. A detailed discussion of these uses of Business Entities is beyond the scope of this chapter.

The presentation and integration layers of the architecture can then be defined independently of the business servicing layer, and use the APIs of the Business Entities and other components to retrieve, process, and update data.

The following sections discuss the principles behind each of the components that make up the data access and Business Entity parts of the architecture.

Data Access object

The first layer of the data management object hierarchy is the data access layer. Code and definitions at this level are aware of the actual source of data, whether it is a standard Data-Source object that maps to one or more database tables, or some other source. Other sources could be unmanaged data not stored in a database, or database data whose mapping to the internal data definition is complex enough that it cannot be expressed in a field mapping and database query. (The term *Data-Source* is used in this document to refer to an actual OpenEdge 10 Data-Source object, as defined in Data-Source object on page 35 and used throughout this book. The term *data source* or *source of data* is used more generically to refer to any source of data for the application, wherever it comes from.)

Isolating the data source from the internal view of data

The basic principle to keep in mind in deciding what definitions and ABL code go into this layer is that, to the greatest extent possible, you should isolate all code that has knowledge of the data source in the data access layer and its procedures. This allows you to define an internal data structure (an internal "schema", if you will) that is the best representation of your data for your application logic. In many cases, because of the evolution of a database design over time, the re-use of databases inherited from older applications, and other factors, your current database schema may not be an ideal way for your business logic to refer to and manage your data. An older database may have a number of shortcomings that you want to be able to mask in newly built business objects and business logic, including:

- The database may not be properly normalized, which can mean that you need to execute complex queries in order to relate data in one table to another, or that there is redundant data that requires fields in multiple tables to be updated when data in another table changes.
- The database may have inconsistent naming conventions, or similar data may be stored in different places in different ways. This could result in a single entity such as a customer number being stored in different tables under different names or in different formats or even in different data types. You will want your internal representation to be consistent.
- The database may contain overly large tables with many fields, which have been accreted over time as various special end-user features required more and more fields to hold values used by only a subset of your users, or for other reasons. Often these fields could be logically grouped according to when or by whom they are needed, so that the internal tables do not use or see all the fields at the same time.

- Conversely, sometimes multiple tables that really represent a single data entity must be joined together because they were developed at different times. Internally these might better be seen as fields in a single table.
- Often, you would like to denormalize the data internally but not in the database. For example, it may be helpful to include the customer name along with customer number internally, if they need to be viewed together, while you would not want to repeat the customer name in multiple tables that all used the customer number as a key.

These are just a few of the considerations that would lead you to want to provide a different internal view of data from how it is actually stored. Even when the database is well designed, there will likely be times when you still would want the internal view of the data to be different, for example, when you need to present a denormalized view or specially filtered subset of the data to the application.

If you have an existing application, it is likely that you cannot (and should not) face the prospect of doing an extensive cleanup of the database schema and data as an initial part of a transformation effort. There may be various reasons for this. If you have a large installed user base, it may not be feasible to convert their databases to a new form all at once. Also, you may have large numbers of reports and business logic procedures that you expect to be able to reuse or easily repackage in the newer version of your application, even as you adapt your application for a distributed environment and in other ways that help it conform with the guidelines of the OpenEdge Reference Architecture. If you make extensive changes to the database schema, you may improve the quality of the data representation at the expense of a great deal more up front design and development work, and with more disruption to your current installed base.

Isolating the database schema specifics in the data access layer of your modernized procedures lets you clean up or otherwise change the underlying schema independent of how the application logic uses the data. When you make changes to the schema, only the mapping code in the data access layer needs to change. If you have older procedures that reference the database directly, then these can be kept isolated from the new internal definitions until you are prepared to rework or replace them.

In addition, some of your data may come from a source other than an OpenEdge database or a database accessible through an OpenEdge DataServer. In this case the actual Data-Source object will not be useful to you. Your data may come from an unmanaged source such as a flat file, a spreadsheet, or an XML or JSON document. Or it might be read dynamically from a data streaming device. In this case, you can still define a data access procedure that uses `FILL` event procedures to populate the Business Entity's ProDataSet with your own custom code. This means that, as with database Data-Sources, the rest of your application does not need to know about the specifics of where the data comes from or how it is managed once it gets beyond the business logic of the application.

Elements of a Data Access object

Given these basic principles of separation of data source specifics from the rest of the application, what then are the elements that properly belong in a Data Access object? Keep in mind as we ask this question that a Data Access object is not an ABL construct or anything else with a specific meaning or structure. It is a concept that can be useful to you to think about as you architect your application. We capitalize the term only to identify it as an implementation of a part of the Reference Architecture.

The sample procedures from earlier chapters are a reasonable starting point to identify the proper elements to include in a Data Access object. In later documents we will extend these basic elements with a more detailed API to provide a template with additional standard behavior. In general you can think of a Data Access object as being paired with a Business Entity object that manages the actual ProDataSet instance data for the rest of the application and applies validation and other business logic. This mapping might not be one-to-one, however. For various reasons you might have multiple Business Entities that use the same data sources and therefore the same Data Access object.

There is also no reason why a Data Access object or any other "object" in your application must be thought of as a single ABL procedure. A base procedure can extend its behavior through the use of super procedures or other forms of procedure library. Or a single procedure could manage multiple Data Access objects, if that is appropriate to your situation. The important thing is how you think about organizing your definitions and the code to manage them, and how this fits in with the rest of your application.

So given these points, let us look at some of the elements of a typical Data Access object.

ProDataSet and temp-table definitions

As we have discussed in earlier chapters, in most cases your ProDataSet and temp-table definitions on the server-side of the application (at least) will be statically defined, because they represent specific sets of data with their own distinct structure and characteristics. Therefore the first element in a typical Data Access object will be the temp-table and ProDataSet definitions for the data it retrieves from the data sources.

There is no reason why some Data Access objects could not be based on dynamic ProDataSets, especially if they represent a collection of tables or sets of table in the database that all have a similar structure and are all processed in the same way. As always, if you support these kinds of variant objects, you should make sure you structure them in such a way that other objects that communicate with them do not need to know whether their data comes from static or dynamic objects. Since the AVM supports freely interchanging static and dynamic temp-tables and ProDataSets when you pass them as parameters, this should not be difficult.

Here are the include file references from the earlier example procedure `OrderSource.p`, which is really an example of a Data Access object:

```
{dsOrderTT.i}

/* …where the include file has these definitions:
DEFINE TEMP-TABLE ttOrder NO-UNDO LIKE Order
  FIELD OrderTotal AS DECIMAL
  FIELD CustName LIKE Customer.NAME
  FIELD RepName  LIKE SalesRep.RepName.

DEFINE TEMP-TABLE ttOline NO-UNDO LIKE OrderLine
  BEFORE-TABLE ttOlineBefore.

DEFINE TEMP-TABLE ttItem NO-UNDO LIKE ITEM
  INDEX ItemNum IS UNIQUE ItemNum.
*/

{dsOrder.i}

/* … where the include file has this definition:
DEFINE DATASET dsOrder FOR ttOrder, ttOline, ttItem
  DATA-RELATION OrderLine FOR ttOrder, ttOline
    RELATION-FIELDS (OrderNum, OrderNum)
  DATA-RELATION LineItem FOR ttOline, ttItem
    RELATION-FIELDS (ItemNum, ItemNum) REPOSITION.
*/
```

These are defined as two separate include files just to make it possible to include them independently in a procedure where you do not need or want both.

How are these definitions used in the Data Access object? If you remember the interaction from the sample procedure `OrderSource.p` and its Business Entity, `OrderEntity.p`, the definitions are used for compilation only, so that the AVM can understand references to temp-tables and their fields in the internal procedures inside the Data Access procedure.

Why is this? The Business Entity object, described later, "owns" the data for its instance of the ProDataSet, whether that is all the data for an `Order` or summary information for all of a `SalesRep's Order`s or whatever else it may be. Every running instance of the Business Entity represents a distinct instance of its ProDataSet and a distinct set of data rows.

By contrast, the Data Access object only serves to populate the Business Entity's ProDataSet with data, and where necessary to assist in getting updates back to the data source. In any call, the actual ProDataSet instance will be passed in from the Business Entity or other requesting procedure `BY-REFERENCE`, so that it replaces the locally defined instance that is used to compile the Data Access procedure. Since each call to the Data Access object passes in a ProDataSet instance, there should be no reason why a single running instance of the Data Access procedure should not be able to serve all requests. It needs to be designed to make sure that there is no context kept from call to call that would prevent this, or else context needs to be managed in some way if this is necessary.

The local temp-table and ProDataSet definitions also cause an instance of the ProDataSet and its temp-tables to be instantiated in the data access procedure. Since this instance is not actually used at run time, it is important to observe the guideline discussed in earlier chapters concerning ProDataSets passed `BY-REFERENCE`, namely that you should not use the handle or any other references to this ProDataSet instance from the procedure's main block in code located within one of the procedure's internal procedures that receives the ProDataSet as a parameter.

Data-Source queries

The next thing to define in a Data Access object is any database queries the procedure uses to fill the ProDataSet tables. In the sample procedure `OrderSource.p`, there is one query for the top-level table, which is needed because it involves a join of multiple database tables. For example:

```
DEFINE QUERY qOrder FOR Order, Customer, SalesRep.
```

Because these queries define the nature of the Data-Source, and because they are used in the `FILL` process and the `FILL` events that the Data Access object defines, they belong here rather than in the Business Entity.

In cases where there is no standard database Data-Source, there may be no queries of this kind. In that case, whatever other definitions may be needed to allow the `FILL` event procedures to do their job belong here.

Data-Source definitions

If the ProDataSet can be filled from an OpenEdge database or OpenEdge DataServer-managed tables, then you can define Data-Source objects to handle the fill. These name the database table that is the source of data, or the query that identifies one or more tables, along with the key fields for the tables. The `dsOrder` ProDataSet used in the sample procedures has these Data-Source definitions:

```
DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder
  Order KEYS (OrderNum), Customer KEYS (CustNum), SalesRep KEYS (SalesRep).
DEFINE DATA-SOURCE srcOline FOR OrderLine.
DEFINE DATA-SOURCE srcItem FOR ITEM  KEYS (ItemNum).
```

The Data-Sources in effect define the part of the `FILL` process that the AVM can handle automatically for you. In the simplest case this is everything, and no special `FILL` logic is needed. In other cases the standard `FILL` behavior must be supplemented by additional code. This goes into `FILL` event procedures that are also part of the Data Access object. In cases where there are no standard Data-Source objects, then **all** the `FILL` logic goes into the event procedures.

FILL event procedures

If you need special logic to supplement or to fully control the FILL process, the procedures that implement that logic also go into the Data Access object. They can be attached to any ProDataSet instance passed in to the procedure using the SET-CALLBACK-PROCEDURE method. They belong there because they have full knowledge of the specifics of the sources of data, whether they are standard OpenEdge Data-Sources or not, and how the data is mapped to the internal representation used by the rest of the application.

For example, OrderSource.p has these FILL event procedures:

```
PROCEDURE postOlineFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  DEFINE VARIABLE dTotal AS DECIMAL NO-UNDO.

  /* Here as well "ttOline" uses the local definition for compilation but
     points to the ttOline table in the input parameter at run time. */
  FOR EACH ttOline WHERE ttOline.OrderNum = ttOrder.OrderNum:
    dTotal = dTotal + ttOline.ExtendedPrice.
  END.
  ttOrder.OrderTotal = dTotal.
END PROCEDURE. /* postOlineFill */

PROCEDURE postItemRowFill:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  DEFINE VARIABLE cItemTypes AS CHARACTER  NO-UNDO
    INITIAL "BASEBALL,CROQUET,FISHING,FOOTBALL,GOLF,SKI,SWIM,TENNIS".
  DEFINE VARIABLE cType      AS CHARACTER  NO-UNDO.
  DEFINE VARIABLE iType      AS INTEGER    NO-UNDO.
  DEFINE VARIABLE iTypeNum   AS INTEGER    NO-UNDO.

  DO iType = 1 TO NUM-ENTRIES(cItemTypes):
    cType = ENTRY(iType, cItemTypes).
    IF INDEX(ttItem.ItemName, cType) NE 0 THEN
      ttItem.ItemName = REPLACE(ttItem.ItemName, cType, cType).
  END.
END PROCEDURE. /* postItemRowFill */
```

Procedure postOlineFill calculates the Order total, and procedure postItemRowFill reformats the Item Name for each Item.

Functions to attach and detach the Data-Sources from a ProDataSet

The reason why all the FILL event procedures can be defined in the Data Access object, even though the procedure's ProDataSet instance is not really used to hold data at run time, is that the Data Access object takes responsibility for attaching those procedures to any ProDataSet instance passed into the Data Access object. OrderSource.p has an attachDataSet function to do this, as shown:

```
FUNCTION attachDataSet RETURNS LOGICAL (INPUT phDataSet AS HANDLE):
  phDataSet:GET-BUFFER-HANDLE("ttOline"):SET-CALLBACK-PROCEDURE
    ("AFTER-FILL", "postOlineFill", THIS-PROCEDURE).
  phDataSet:GET-BUFFER-HANDLE("ttItem"):SET-CALLBACK-PROCEDURE
    ("AFTER-ROW-FILL", "postItemRowFill", THIS-PROCEDURE).
  phDataSet:GET-BUFFER-HANDLE("ttOrder"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcOrder:HANDLE, "Customer.Name,CustName").
  phDataSet:GET-BUFFER-HANDLE("ttOline"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcOline:HANDLE).
  phDataSet:GET-BUFFER-HANDLE("ttItem"):ATTACH-DATA-SOURCE
    (DATA-SOURCE srcItem:HANDLE).
END FUNCTION. /* attachDataSet */
```

This could just as easily be an internal procedure, of course. A more thorough implementation of the function should check the return value for each SET-CALLBACK-PROCEDURE and ATTACH-DATA-SOURCE method and return an ERROR status to the caller. The SET-CALLBACK-PROCEDURE methods attach the needed FILL events handlers, and the ATTACH-DATA-SOURCE methods connect the ProDataSet instance to its database tables.

If a repository or other persistent store holds the field mapping and Data-Source mapping for the ATTACH-DATA-SOURCE methods, and the callback procedure names and locations for the SET-CALLBACK-PROCEDURE methods, then this function could become generic, and be part of a standard procedure that supports all Data Access objects (as a super procedure, for instance).

There should also be a function or procedure to detach all Data-Sources, as in this example:

```
FUNCTION detachDataSet RETURNS LOGICAL (INPUT phDataSet AS HANDLE):
  DEFINE VARIABLE iBuff AS INTEGER NO-UNDO.

  DO iBuff = 1 TO DATASET dsOrder:NUM-BUFFERS:
    phDataSet:GET-BUFFER-HANDLE(iBuff):DETACH-DATA-SOURCE().
  END.
END FUNCTION. /* detachDataSet */
```

As this sample shows, this function can easily be made generic, so it is written only once and resides in a Data Access support procedure.

It is important to note that once the Data-Sources are attached and any callback procedures established, the calling procedure that passed in the ProDataSet instance to attachDataSet can simply do a FILL if the Data-Source definitions and FILL event handlers fully determine what rows to populate the ProDataSet with. In other words, once the ProDataSet is returned to the caller, the Data-Sources and callback procedures remain associated with it through its handle, so these associations remain intact even in the ProDataSet instance is passed around within the session, so that other procedures and invoke a FILL or other methods on that handle from anywhere in the session. In many cases, however, a FILL will require that the ProDataSet's queries first be prepared to retrieve only a selected set of related rows before doing the FILL. An API for such calls is discussed in the next section.

Data retrieval API

There can be both standard and specialized API calls that populate a ProDataSet. Depending on the data in the ProDataSet, there will be different sets of data that are useful to the application. In the case of the sample `Order Entity`, there is a call to retrieve all data for a single `Order Number` and another call to retrieve summary data in just the `Order` table for some related set of `Orders`. What these calls have in common is that they require code that is "data source aware" to prepare the right queries or otherwise adjust the parameters of the `FILL`. For this reason, they belong in the Data Access object.

In general, it is a good idea not to allow anything except the Data Access object's Business Entity to use its API directly, so that if the user interface or some other part of the application needs to request data, it should use the API of the Business Entity, which can then turn around and use the API of the Data Access object to retrieve the right data.

In the sample procedures, there is a `fetchOrder` procedure that accepts an `Order Number` and returns a ProDataSet as output. This runs this procedure of the same name in the Data Access object `OrderSource.p`, passing in the `Order Number`, along with the Business Entity's ProDataSet instance as an `INPUT-OUTPUT` parameter:

```
PROCEDURE fetchOrder:
  DEFINE INPUT        PARAMETER piOrderNum AS INTEGER NO-UNDO.
  DEFINE INPUT-OUTPUT PARAMETER DATASET    FOR dsOrder.

  QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.OrderNum = " +
    STRING(piOrderNum) +
    ", FIRST Customer OF Order, FIRST SalesRep OF Order").
  /* Note that this reference to dsOrder is not using the local definition
     but rather the actual dataset instance being passed in. */
  IF VALID-HANDLE(DATASET dsOrder:GET-BUFFER-HANDLE(1):DATA-SOURCE) THEN
    DATASET dsOrder:FILL().
  ELSE DO:
    DATASET dsOrder:GET-BUFFER-HANDLE(1):TABLE-HANDLE:ERROR-STRING =
      "Data-Sources not attached".
    DATASET dsOrder:ERROR = TRUE.
  END.
  RETURN.
END PROCEDURE. /* fetchOrder */
```

The primary reason for this separation is to avoid having Data-Source-aware code, such as the `QUERY-PREPARE` method, in the Business Entity. If the nature of the data source ever changes, then all the references to it are captured in the Data Access object and can be changed together.

As you can see, `fetchOrder` requires that the caller previously run the `attachDataSet` method to attach Data-Sources and callback procedures. It could also be good practice to embed a check inside each API call in the Data Access object that assumes that the Data-Sources and any callback procedures are attached. The check can then run the attach function as needed, rather than depending on the caller to do this first.

For example, you could conditionally invoke `attachDataSet` from within `fetchOrder`. First you define a function prototype for `attachDataSet` at the top of the procedure, as shown:

```
FUNCTION attachDataSet RETURNS LOGICAL (INPUT phDataSet AS HANDLE) FORWARD.
```

Then `fetchOrder` can invoke the function if the Data-Sources are not attached, rather than raising an `ERROR`:

```
PROCEDURE fetchOrder:
  DEFINE INPUT        PARAMETER piOrderNum AS INTEGER NO-UNDO.
  DEFINE INPUT-OUTPUT PARAMETER DATASET    FOR dsOrder.

  DEFINE VARIABLE hDataSet AS HANDLE NO-UNDO.

  /* Note that this reference to dsOrder is not using the local definition
     but rather the actual dataset instance being passed in. */

  hDataSet = DATASET dsOrder:HANDLE.
  IF NOT VALID-HANDLE(DATASET dsOrder:GET-BUFFER-HANDLE(1):DATA-SOURCE) THEN
    attachDataSet(INPUT hDataSet).

  QUERY qOrder:QUERY-PREPARE("FOR EACH Order WHERE Order.OrderNum = " +
    STRING(piOrderNum) +
    ", FIRST Customer OF Order, FIRST SalesRep OF Order").

  DATASET dsOrder:FILL().
  /* The attach call makes the error message unnecessary.
  ELSE DO:
    DATASET dsOrder:GET-BUFFER-HANDLE(1):TABLE-HANDLE:ERROR-STRING =
      "Data-Sources not attached".
    DATASET dsOrder:ERROR = TRUE.
  END.
  */
  RETURN.
END PROCEDURE. /* fetchOrder */
```

Specialized update API

The ProDataSet update examples use a generic update procedure called `commitChanges.p` to return updates to the database. It illustrates how you can attach validation logic procedures in a standard way much as you would do for SmartDataObjects in the ADM2. In a large percentage of cases this can provide you with a general purpose update mechanism that applies to many different Business Entities.

There are however cases where a generic procedure will not be adequate. In particular, if there are no standard Data-Sources to attach to the ProDataSet, because the data source is unstructured data, then there can also be no built-in update support, and you will need to write your own update API to accept changed ProDataSets and do whatever is necessary to get those changes back to the data sources. As with the retrieval API, it is best if any requests from outside the Business Entity go through the Business Entity, which can then make the right request of the Data Access object to get the job done.

Data Access object template

There is at this point no specific Data Access object template, that is, a single procedure that can support most Business Entities. A template can be in the form of a static skeleton procedure that is filled in by a tool or wizard (or just by editing it by hand). Or it can be in the form of a dynamic procedure that is able to analyze a ProDataSet through its handle and operate on it in a generalized way.

A dynamic support procedure would not be able to do very much in a generalized way for most ProDataSets. As you have seen from the simple examples in these chapters, both the queries that prepare the ProDataSet for a `FILL` and the logic of `FILL` event procedures themselves are likely to be very specific to the structure of the ProDataSet. Even an `attachDataSet` function needs to know what event handlers to attach and what the field mapping is for attaching Data-Sources.

It certainly is possible to define a static template procedure to be filled in by a design tool, or by a wizard, or by the simple act of editing a copy of the procedure and replacing instructional comments with specific code to support an entity. Defining a specific such procedure is beyond the scope of this chapter, and will be done in conjunction with defining tools for designing and building Business Entities. The topics in this chapter are intended to provide some general guidelines of things to think about as you design your objects.

Business Entity object

The Business Entity "owns" the data that constitutes a business object. This means that each running instance of a Business Entity procedure represents and holds the data for a particular object. If the Business Entity is for an `Order` and uses temp-tables for the `Order` header and for its `OrderLines`, then typically an instance of the entity would hold the data for a single `Order` and its `OrderLines`. In some cases, of course, a Business Entity could be used for a more specialized purpose, such as retrieving all the `Order` headers for a `SalesRep`, as one of the earlier examples shows, and returning those to the client, so that the client can then request detail for an `Order` in a subsequent call. There can be a wide variety to these specialized types of requests, and they will vary a lot from entity to entity.

Elements of a Business Entity

In principle, a Business Entity as an object can map to an ABL procedure. This might in turn be supported by any number of other procedures that provide either specific validation logic for the entity or general support services used by all entities. Each Business Entity is normally paired with a Data Access object that manages the connection to the actual data sources. As discussed earlier, in many cases a single running instance of a Data Access object could provide data for any number of requests from different Business Entity instances.

In a distributed stateless environment, a server-side Business Entity will normally not live beyond a single request. It can fill its ProDataSet for a particular `Order`, for example, return that `Order` to the client, and then either terminate or remain in memory to be used by another independent request, depending on how the application manages its procedures. On the other hand, if one Business Entity is used by another Business Entity in the same session, as part of its validation logic, for example, then there could be a relationship between the two procedures that would last beyond a single request.

As with the Data Access object, the Business Entity will typically be a static object. That is, it will be based on one or more static ProDataSet definitions, and contain (directly or indirectly) mostly specific logic to support its use. This logic will include a specific API for the various kinds of requests that can be made of it, and validation logic to apply to update requests. A dynamic Business Entity designed to handle a certain class of similar data objects is also entirely possible, of course.

Given these basics, the following sections outline some of the common elements of the Business Entity.

ProDataSet and temp-table definitions

A Business Entity will typically contain the same temp-table and ProDataSet definitions as its Data Access object. The significant different is that it is the Business Entity's instance of the ProDataSet that is actually filled with data used to satisfy a request, when it is passed `BY-REFERENCE` to its Data Access object.

Relationship to the Data Access object

If the application model is that an instance of the Data Access object serves a single Business Entity instance, then the Business Entity can run that instance as a persistent procedure, as in this example:

```
RUN OrderSource.p PERSISTENT SET hSourceProc.
```

The Entity can then use the API of the Data Access object to run support methods in this procedure handle.

The Business Entity could also make the Data Access object a super procedure of the Business Entity, as shown:

```
THIS-PROCEDURE:ADD-SUPER-PROCEDURE(hSourceProc).
```

This allows the Entity to use the API of the Data Access object as if it were part of its own procedure.

Using the super procedure technique also changes the nature of calls from outside that are routed from the Business Entity to the Data Access object. This is discussed in the .

Attaching the Data-Sources and callbacks

The entity procedure can, as part of its main block startup code, request that the Data Access object attach Data-Sources and set callback procedures to its ProDataSet instance, as in this example:

```
lError = DYNAMIC-FUNCTION("attachDataSet" IN hSourceProc, hDataSet).
```

As was illustrated earlier, this can also be done in each of the methods of the Data Access object, rather than as a separate step.

Defining a data retrieval API

Any requests for data should be made to the Business Entity, never directly from outside to the Data Access object. This simply preserves the isolation of the different layers of the application. The `fetchOrder` procedure in `OrderEntity.p` is an example of this:

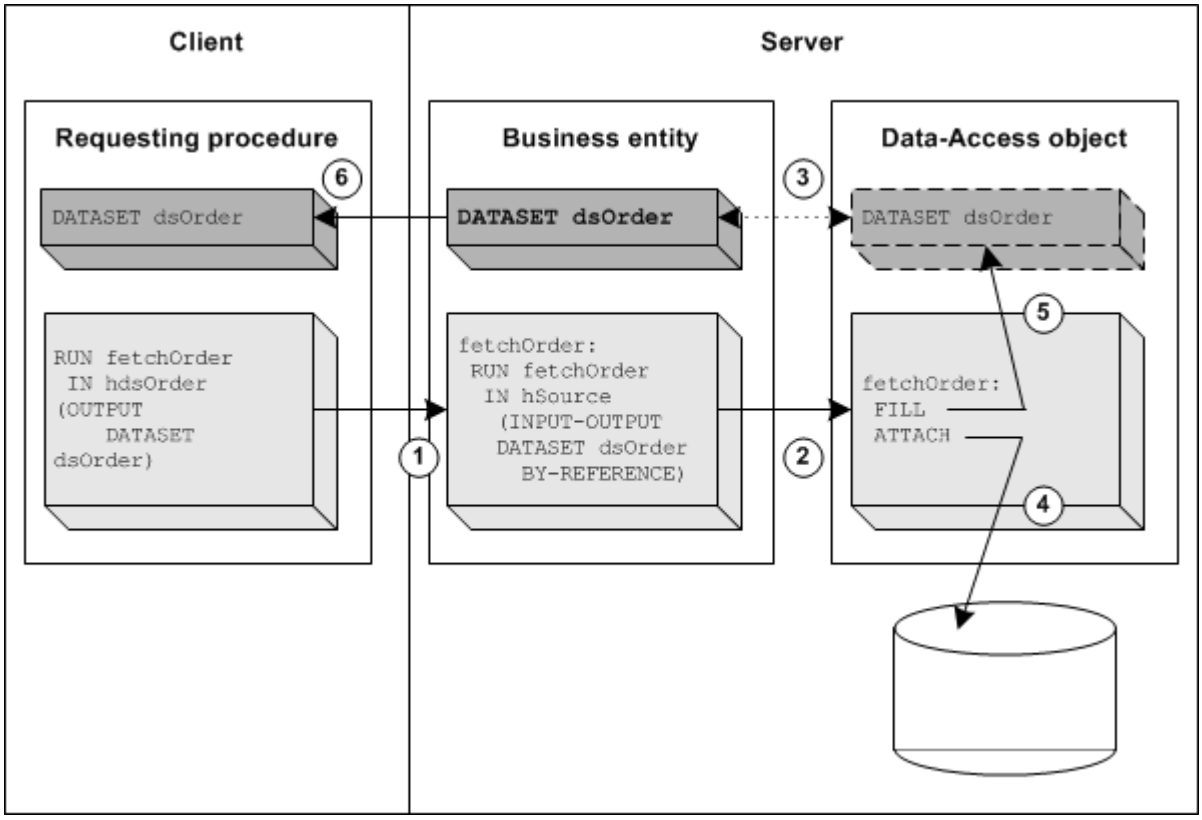
```
PROCEDURE fetchOrder:
  DEFINE INPUT  PARAMETER piOrderNum AS INTEGER NO-UNDO.
  DEFINE OUTPUT PARAMETER DATASET    FOR dsOrder BY-VALUE.

  /* This turns around and runs an equivalent procedure in the
     Data-Source procedure, passing in the static dataSet. */
  DYNAMIC-FUNCTION("attachDataSet" IN hSourceProc, hDataSet).
  RUN fetchOrder IN hSourceProc (INPUT piOrderNum,
    INPUT-OUTPUT DATASET dsOrder BY-REFERENCE).
  DYNAMIC-FUNCTION("detachDataSet" IN hSourceProc, INPUT hDataSet).
END PROCEDURE. /* fetchOrder */
```

This turns around and runs an equivalent procedure in the Data Access procedure. Significantly, the ProDataSet parameter is `INPUT-OUTPUT BY-REFERENCE` in the second-level call to the Data Access object. This uses the Business Entity's ProDataSet and avoids the expense of copying the ProDataSet back and forth. Having avoided this, the overhead of the second procedure call is not very significant.

The following figure shows how the ProDataSet is being used in this case.

Figure 19: ProDataSet flow



These are the steps illustrated in the above figure:

1. A requesting procedure on the client runs `fetchOrder` in the `Order` Business Entity on the server.
2. That `fetchOrder` procedure runs `fetchOrder` in the Data Access object in its session.
3. Because the ProDataSet is passed in `BY-REFERENCE`, it is actually the Business Entity's instance that is used (marked in bold). The dotted lines indicate that this instance is passed without being copied.
4. The `fetchOrder` procedure in the Data Access object attaches Data-Sources, which are tables in the database, and also sets any callback procedures for `FILL` events.
5. It then does a `FILL`, which actually fills the ProDataSet instance back in the Business Entity.
6. It returns the ProDataSet to the requesting procedure. It is copied there rather than being passed `BY-REFERENCE` because the presumption is that the requester is or may be in a different session.

If the Data Access object is a super procedure of the Business Entity, then in cases where the Business Entity version of a procedure like `fetchOrder` does not do any additional work of its own, it could be dispensed with, and a call to `fetchOrder` from another procedure would be handled automatically by the Data Access object. In this case, there are two things to consider:

- First, the Data Access object version of `fetchOrder` would need to make its ProDataSet parameter `OUTPUT` instead of `INPUT-OUTPUT`, because it would be passed back directly to the caller. The caller would not be passing in a ProDataSet of its own. In this case, the Data Access object's ProDataSet instance is the one

that is used to satisfy the request. It then becomes the Data Access object's responsibility to make sure the Data-Sources are attached, as illustrated in the second version of `fetchOrder` in the .

- Second, if there is any reason for the Business Entity procedure to run `fetchOrder` in the Data Access procedure, or to provide extended behavior for a call from outside to `fetchOrder`, then this arrangement becomes inefficient, because the ProDataSet would be copied from the Data Access procedure to the Business Entity procedure. For example, consider this alternative to `fetchOrder` in the Entity:

```
PROCEDURE fetchOrder:
  /* Alternative OrderEntity.p version for comparison only ! */
  DEFINE INPUT  PARAMETER piOrderNum AS INTEGER NO-UNDO.
  DEFINE OUTPUT PARAMETER DATASET    FOR dsOrder.

  /* fetchOrder does some prep work here, for example validating the
     INPUT data or the requester's privileges to make the request */

  /* Then it just does a RUN SUPER to invoke the standard behavior. */
  RUN SUPER (INPUT piOrderNum, OUTPUT DATASET dsOrder).

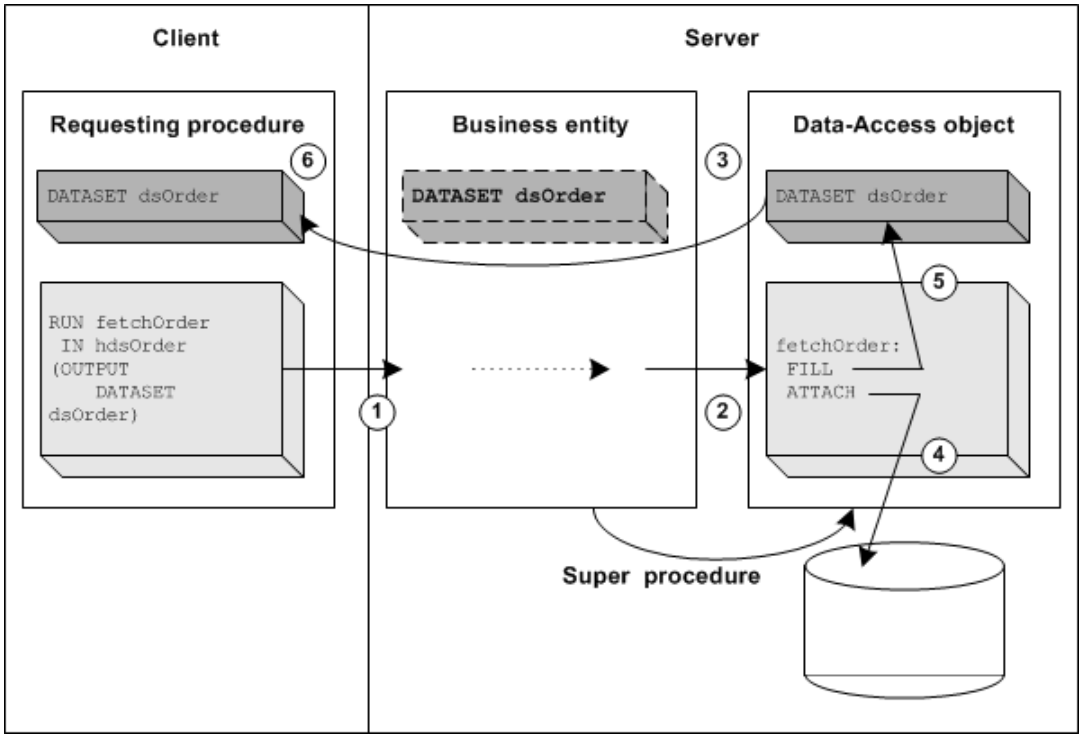
  /* The RUN SUPER is done instead of running it directly as in the
     original version:
  RUN fetchOrder IN hSourceProc (INPUT piOrderNum,
    INPUT-OUTPUT DATASET dsOrder BY-REFERENCE).
  */
END PROCEDURE. /* fetchOrder */
```

If the parameter definition for the ProDataSet in `fetchOrder` in the procedure `OrderSource.p` is changed to be `OUTPUT` instead of `INPUT-OUTPUT`, and `hSourceProc` is a super procedure of `OrderEntity.p`, and `fetchOrder` in `OrderSource.p` does the `attachDataSet`, then this arrangement works fine.

However, the ProDataSet is copied from `Order`Source to `Order`Entity before being copied back to the caller. This is not a good thing. Because of this, and because of the potential for confusion between when the Data Access ProDataSet instance is being used and when the Business Entity instance is being used, making the data access procedure a super procedure may not be a good practice.

The following figure illustrates what happens if the Data Access object is a super procedure of the Business Entity, and it has the only implementation of procedure `fetchOrder`.

Figure 20: Data Access object as super procedure



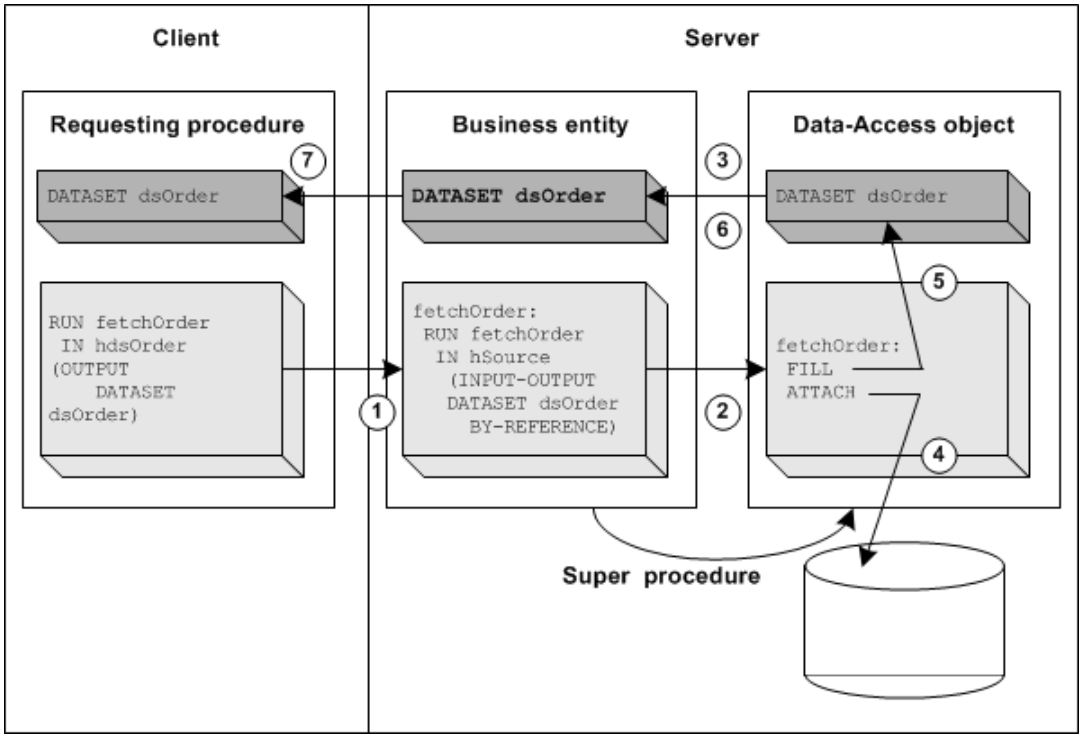
These are the steps illustrated in the above figure:

1. The requesting procedure runs `fetchOrder` in the Business Entity as before.
2. There is no `fetchOrder` procedure in the Business Entity. However, since the Data Access object is a super procedure, the AVM runs `fetchOrder` there.
3. This means that it is the Data Access object's ProDataSet instance that is used for the request.
4. The `fetchOrder` procedure attaches Data-Sources and callback procedures to this ProDataSet.
5. It then fills the ProDataSet.
6. The `fetchOrder` procedure then returns this ProDataSet to the original caller as an `OUTPUT` parameter. The ProDataSet instance in the Business Entity is not used.

This all works correctly in this case, but could be a source of confusion and errors because the ProDataSet instance being used is not consistent.

By contrast, the following figure shows the case where, once again, the Data Access object is a super procedure of the Business Entity, and `fetchOrder` in the Business Entity does a `RUN SUPER` to run the standard attach and fill behavior.

Figure 21: Data Access object as super procedure with RUN SUPER



These are the steps illustrated in the above figure:

1. The requesting procedure runs `fetchOrder` in the Business Entity as before. There is an implementation of `fetchOrder` there, so it is executed.
2. Procedure `fetchOrder` does a `RUN SUPER`, which runs `fetchOrder` in the Data Access object. Because the parameter definitions must be consistent in this case, the ProDataSet is simply an `OUTPUT` parameter.
3. Because of this, the Data Access object uses its own ProDataSet instance.
4. It attaches Data-Sources to its own ProDataSet instance.
5. It fills its own ProDataSet instance.
6. It returns its ProDataSet as an `OUTPUT` parameter to the Business Entity, copying it to the Business Entity's ProDataSet instance.
7. The Business Entity then returns it to the original caller as `OUTPUT`, again copying the ProDataSet.

Because of the extra copy operation, this is not a good configuration. This is something you need to keep in mind when you design your procedures and decide how they are related.

This discussion may seem complex, but the intention is to make you aware of some of the issues and how you should consider them when you are designing your application. As with every other aspect of design, once you have thought through an appropriate solution to a part of your design, if you keep to your solution consistently, then you will not have to worry about it anymore, and developers writing business logic do not need to be concerned about the details of the Data Access architecture supporting them.

Defining a generic update API

As the sample procedures in earlier chapters show, you can create a general-purpose update API that can take changes made to any ProDataSet and apply them to the database, even executing validation procedures for the specific ProDataSet if they conform to a standard naming convention. This approach is very much like how the SmartDataObject and SmartBusinessObject in the ADM2 handle their update logic.

The generic update procedure in the samples is `commitChanges.p`. The sample entity procedure, `OrderEntity.p`, wraps this in a call of its own to provide an API for other procedures to use, as shown:

```
PROCEDURE saveChanges:
  DEFINE INPUT-OUTPUT PARAMETER DATASET FOR dsOrder.

  DEFINE VARIABLE hDataSet AS HANDLE NO-UNDO.

  hDataSet = DATASET dsOrder:HANDLE.
  DYNAMIC-FUNCTION("attachDataSet" IN hSourceProc, hDataSet).
  RUN commitChanges.p (INPUT-OUTPUT DATASET dsOrder BY-REFERENCE).
  DYNAMIC-FUNCTION("detachDataSet" IN hSourceProc, hDataSet).
END PROCEDURE. /* saveChanges */
```

This wrapper procedure attaches the Data-Sources and event handlers, runs `commitChanges.p`, and then detaches the Data-Sources. This could as easily be incorporated directly into a procedure like `commitChanges` if it knows where to run the attach and detach methods. Beyond that, having a wrapper procedure gives the specific object an opportunity to add special commit logic before or after the standard code.

Validation procedures for the generic update API

For example, the `OrderEntity` procedure has this sample validation procedure, which based on its name, will be executed whenever a row in the `ttOline` table in the ProDataSet is modified:

```
PROCEDURE ttOlineModify:
  DEFINE INPUT PARAMETER DATASET FOR dsOrder.

  /* If customer doubled quantity ordered, then increase discount by 20%. */
  IF ttOline.Qty >= (ttOlineBefore.Qty * 2) AND
    ttOline.Discount = ttOlineBefore.Discount THEN
    ttOline.Discount = ttOlineBefore.Discount * 1.2.
  ELSE IF ttOline.Qty <= (ttOlineBefore.Qty * .5) THEN
    ASSIGN
      BUFFER ttOline:ERROR = TRUE
      BUFFER ttOline:ERROR-STRING =
        "Line " + STRING(ttOline.LineNum) +
        ": You can't drop the Qty that much!".
  RETURN.
END PROCEDURE. /* ttOlineModify */
```

The basic principles of writing procedures such as this one are:

- You can pass in the entire ProDataSet `BY-REFERENCE` from another procedure in the same session without the cost of copying it, so that the validation logic can examine any part of the ProDataSet that it needs to.
- The current rows in the buffers will be those that were current in the caller. In particular, the current row in the table that triggered the event is the one that was modified, so you can examine its values without having to `FIND` it.

Validation procedures can set the `ERROR` and/or `ERROR-STRING` attributes for the row to communicate status information. Setting `ERROR-STRING` without setting `ERROR` lets you return an informational or warning message without causing an actual error.

Defining a custom update API

In addition to using a standard update mechanism for general updates, there can of course be additional more specific update API calls in your Business Entity that handle particular situations where the default behavior is not sufficient.

Managing Business Entity instances

Managing Business Entities and their Data Access objects needs to be coordinated in some fashion, so that entities can locate each other, can be accessed from client requests, and can be started and stopped when appropriate.

In general, Business Entities and their APIs should be designed such that there are no dependencies between requests. This is especially necessary for a stateless distributed environment where a client cannot expect to be given access to the same procedure instance on an AppServer in successive calls. Thus, there is little reason to leave data in a Business Entity on the AppServer after a request completes.

This means that a Business Entity instance can be left running on an AppServer to service any number of unrelated requests. If the Business Entity procedures themselves are not enormous amount of r-code, it may be just as efficient to destroy each instance after its use and start a fresh instance when needed. It takes very little time to load a compiled procedure into memory. Any significant overhead is in startup code for the object, and this should be minimized. Keep in mind that while only one business instance is needed in an AppServer session to satisfy any number of successive requests from client sessions, the business logic within a session may need to use (and possibly start) other Business Entities to execute its own logic.

There should be a mapping between Business Entity names as used in business logic and actual procedure names, so that objects do not need to be aware of other procedure names to run them directly. This can be repository-based, and managed by a single session management utility within the session that accepts requests for entities and either locates or starts them and returns their handles. For example, an `Order` entity should be able to make a request of the `Customer` entity without having to know an actual procedure name to run. It should be the responsibility of the entity manager to handle this.

Support procedures that require only a single instance within a session, such as Data Access procedures, and most supporting business logic and update validation procedures, can be started by the Business Entities or started by the entity management utility the first time they are needed, and then left running for the duration of the session, or else shut down on some form of LRU basis if memory becomes a problem.

Business logic options

This section briefly outlines some of the issues and alternatives for managing Business Entity-specific logic.

Standard validation procedures

The previous section provided an example of a standard validation procedure that can be run automatically as part of the update process if there is a naming convention for them and a standard calling sequence and error handling mechanism. If the ProDataSet instance is always passed in as an `INPUT` or `INPUT-OUTPUT` parameter `BY-REFERENCE`, then the logic can freely access any data in both the before- and after-tables in the ProDataSet. Note that when a ProDataSet is passed locally `BY-REFERENCE`, `INPUT` and `INPUT-OUTPUT` amount to the same thing, because the procedures are sharing a single ProDataSet instance.

Accessing other entities in your business logic

Within one Business Entity, it may be necessary to read or write data managed by other Business Entities as part of the business logic. To preserve the separation of the application's internal data representation from the details of the physical storage, which the Data Access layer does for you, it is important to access other data through its Business Entities whenever possible, which basically means whenever performance considerations do not force you to compromise that approach. There is, of course, an overhead involved in accessing data through the Business Entities. This requires performing a `FILL` to retrieve data, which copies it into temp-tables, manipulating the data in the temp-tables, and then copying any changes back to the database or other data source. If you work to package your business logic in "chunks" that make this type of operation as efficient as possible, then this can work effectively for most situations. Remember that the ProDataSet can often move data between the database and temp-tables, and perform other operations, more efficiently than you can in your own ABL code, and more efficiently than ABL-based objects such as SDOs.

Some limited operations that either involve processing large batches of data or doing extremely data-intensive operations against the database may suffer unacceptably from the use of Business Entities as a strict access interface to the data. You must decide when the internal data model needs to be compromised for these reasons, and carefully isolate procedures that bypass the entities and access the database or other data source directly. If you do this only when really necessary and keep the instance of it isolated and identified, then you should be able to manage schema changes or other application life cycle events that change the nature of the data source without too much difficulty. It is very important, however, not to fall into the trap of making direct access to the database because it seems the simplest way to write a section of code.

If you construct sensible APIs for your business logic, you can develop a coding standard that uses Business Entities effectively without the need to access the database directly. These standards should in fact improve the modularity, maintainability, and reusability of your business logic by forcing you to identify, name, and encapsulate logic operations that can be modified or reused elsewhere, rather than simply writing line after line of business logic that is not reusable and does not clearly identify what the purpose of each piece of code is. If your business logic is modular, you can much more easily add rules processing and workflow layers to your application that run, sequence, extend, and customize as a series of identifiable organized steps rather than an unorganized mass of ABL code.

Trigger procedure logic

There will be other times when it may seem painful, and undesirably expensive, to reference all data through its Business Entity. In particular, some kinds of straightforward referential integrity checks, such as those which are typically done in traditional ABL through `CAN-FIND` functions and the like, will be significantly more complex and expensive if done through the data retrieval API of a Business Entity. Since the general recommendation in the past has been to use trigger procedures primarily for referential integrity enforcement, in order to make sure the database integrity is maintained no matter how the data is accessed, you should not hesitate to continue to do this level of basic validation check in database trigger procedures. It is important to keep in mind that to the extent that the internal Business Entity definition of the data differs in name or structure from the actual data source, a change in the nature of the data source will necessitate revisiting affected trigger procedures as well. If validation code that references database tables and fields in other entities directly is restricted to this layer, then this maintenance job can be kept under control.

In some cases, developers have used trigger procedures for substantial parts of their business logic, beyond basic integrity checks. This has not been the advised way of organizing business logic, partly because it can make the job of providing flexibility to that somewhat hidden logic more complicated, including customization of logic and changing parameter values for business logic calculations and decisions. Nonetheless, there can be advantages to this approach. In particular, since trigger procedures execute on the server-side of a distributed application, an older client-server-style application that has a large percentage of its logic encapsulated in trigger procedures already has done a lot of the separation of user interface from business logic that is required to make the application distributable between client and AppServer. Existing trigger procedures designed for client-server may still require changes, in particular to remove any `MESSAGE` statements or other user interface elements, and to make sure errors are properly returned in such a way that they can be transmitted back to the client. However, if you are going through an application transformation to provide a new user interface for an older OpenEdge application, you can consider leaving business logic in triggers at least for the first phase of your work, to simplify the amount of code rework that is required.

Again, the advantage is that at least you know where the code with the direct database table and field reference is. The disadvantage is that all the code that references those fields will need maintenance if the database schema changes, or if you need to substitute an entirely different data source type in the future. In addition, if the internal Business Entity definition differs from the database schema, there are then two different names you need to search for whenever you need to identify all the references to a particular data element.

To a large extent, the same applies to any existing application with business logic procedures that are already well adapted to a distributed environment. You will need to judge when it is most effective to leave complex logic procedures alone, providing some wrapper code for them as necessary to make them work in the new environment, and when it is better to rework them to use your Business Entity definitions. You do not need to make a hard and fast decision from the beginning of the process. It may be most practical to leave these procedures isolated while you are reworking the architecture of the rest of the application, and then address them at a later stage. Keep in mind that other aspects of a new application architecture should include standardized, flexible support for customization and personalization, language translation, application security, and other important features that can be supported in a standard way in a new architecture, but which are likely supported in a more ad hoc manner, intermixed with the real business logic, in older existing procedures.

Including context information in Business Entities

Generally the procedures and functions that make up the API of a Business Entity should be designed to be invoked independently of one another. In a stateless or state-free AppServer environment, there is really little choice, as you cannot maintain context between calls without binding an AppServer session, and you do not have the ability to get back to the same AppServer session you called before.

In many cases, however, a larger business task may require multiple successive calls where each AppServer procedure must be aware of the previous calls. There are at least two basic ways to approach handling this:

- The first is to pass context along with each call. This is practical if the amount of data is not excessive and if the client needs or actually produces some of the context information needed on the server. One way to do this is to include a context temp-table definition as part of the ProDataSet definition. In this way the context is identified as an essential part of the Business Entity, and always passed along with its other data. This temp-table could contain a single row with fields representing scalar context values, or it could contain multiple rows representing any sort of tabular data.

  If you use a standard naming convention for the context table, such as `ttContext`, then even a general purpose procedure on the client can routinely pass this table as input to any server call that requires context. For example, if one or more tables in the ProDataSet require batching, to avoid reading and passing all rows at once, then the context table could contain the key values for the last row retrieved by the previous batch, and perhaps also the selection criteria used for the query. The server batching procedure could receive the context table alone as input, repopulate an instance of the ProDataSet with the next batch of rows and return the ProDataSet, including the context table with the new identifier for the last row retrieved, to the client.

- The second method is to store the context persistently on the server, either in a database table or a file or in some other place where all AppServer sessions can read it. The `CONNECTION-ID` can be used as a key to differentiate one client's context from another's. This is the method used, for example, by the Progress Dynamics® context manager.

Summary

This chapter introduced you to some design issues related to Business Entities and Data Access objects using ProDataSets. Most fundamentally, design and implement your application in layers that provide the greatest flexibility:

- As your data sources change
- As you extend and specialize your business logic
- As you define the presentation layer of your application

If you define reusable templates and standard APIs for your business objects, you will be positioned to extend and maintain your application as your needs change.

Index

A

ACCEPT-CHANGES method *148*
ACCEPT-ROW-CHANGES method *148*
ACTIVE attribute *111*
ADD-RELATION method *94*
AFTER-ROW-FILL event *81*
ATTACH-DATA-SOURCE method *38*
attachDataSet function *220*
Attributes
  ACTIVE *111*
  BATCH-SIZE *178*
  DATA-SOURCE-ROWID *151*
  ERROR *167*
  ERROR-STRING *167*
  FILL-MODE *44*
  GET-ITERATION *81*
  LAST-BATCH *178*
  MAXIMUM-LEVEL *185*
  NEXT-ROWID *116*
  NUM-ITERATIONS *81*
  RECURSIVE *43*, *94*
  REJECTED *167*
  RESTART-ROW *116*, *180*
  RESTART-ROWID *179*
  ROW-STATE *137*
  SAVE-WHERE-STRING *155*
  TOP-NAV-QUERY *122*
  TRACKING-CHANGES *135*

B

BATCH-SIZE attribute *178*
Batching with ProDataSets
  include-field list *207*
  record sort order *205*
BEFORE-FILL event *81*
BEFORE-ROW-FILL event *81*
Before-tables
  creating *138*
  locating rows *140*
BIND parameter *53*
BUFFER-COMPARE method *40*
BUFFER-COPY method *40*
Business entity objects
  elements *263*
  managing instances *270*
Business entity procedure *238*
Business logic *270*
BY-REFERENCE qualifier, optimizing code *59*

C

Caching *223*
Callback procedures *173*
Change mode *135*
Child table queries *43*
COPY-DATASET method *193*, *197*, *199*
COPY-TEMP-TABLE method *193*
CREATE DATA-SOURCE statement *95*
CREATE DATASET statement *91*
CREATE-LIKE method *97*
CURRENT-ITERATION attribute *81*

D

Data access layer
  defined *216*
  granularity *216*
  internal representation *216*
  support procedure *219*
  top-level table *218*
Data access objects
  data retrieval API *261*
  elements *256*
  template *262*
Data access procedure *234*
Data-Relation objects
  accessing *111*
  ACTIVE attribute *111*
  ADD-RELATION method *94*
  creating *94*
  inactive *34*
  MAXIMUM-LEVEL attribute *185*
  NOT-ACTIVE option *34*
  recursive *43*, *94*
  RECURSIVE attribute *43*
Data-Source objects
  as object *37*
  attaching *36*
  attaching to buffers *38*
  creating dynamic *95*
  defined *24*, *35*
  defining a static *35*
  FILL-WHERE-STRING attribute *116*
  queries *37*
  RESTART-ROW attribute *116*, *180*
  RESTART-ROWID attribute *116*, *179*
  session attributes *122*
  static *35*
  using attributes and methods *116*
  when not to use *38*

DATA-SOURCE-ROWID attribute *151*
DEFINE DATA-SOURCE statement *35*
DEFINE DATASET statement
  REFERENCE-ONLY option *53*
  REPOSITION option *130*
DEFINE TEMP-TABLE statement
  LIKE option *29*
  LIKE-SEQUENTIAL option *29*
DELETE OBJECT statement *93*
detachDataSet function *220*
Dynamic ProDataSets
  creating *91*
  deleting *93*
  introduced *91*
  passing as a parameter *65*
Dynamic user interfaces *123*

E

ERROR attribute *167*
ERROR-STRING attribute *167*
Event procedures *79*
Events
  AFTER-ROW-FILL *81*
  BEFORE-FILL *81*
  BEFORE-ROW-FILL *81*
  FILL *25*
  FIND-FAILED *210*
  GET-CALLBACK-PROC-CONTENT method *79*
  GET-CALLBACK-PROC-NAME method *79*
  OFF-END *176*
  ProDataSet *26*
  ROW-CREATE *171*
  ROW-DELETE *171*
  ROW-LEAVE *141*
  ROW-UPDATE *171*
  SET-CALLBACK-PROCEDURE method *79*

F

Field mapping phrase *29*
FILL event *25*
FILL events
  defining *81*
FILL mode *135*
FILL-MODE attribute *44*
FIND-FAILED event *210*
Functions
  attachDataSet *220*
  detachDataSet *220*
  ROW-STATE *138*
  ROWID *38*

G

General update procedures *241*
GET-CALLBACK-PROC-CONTENT event *79*
GET-CALLBACK-PROC-NAME method *79*
GET-CHANGES method *144*
GET-ITERATION attribute *81*

I

INPUT BY-REFERENCE parameter *54*

L

LAST-BATCH attribute *178*
LIKE option *29*

M

MARK-NEW method *152*
MARK-ROW-STATE method *151*
MAXIMUM-LEVEL attribute *185*
MERGE-CHANGES method *146*, *166*
MERGE-ROW-CHANGES method *146*
Methods
  ACCEPT-CHANGES *148*
  ACCEPT-ROW-CHANGES *148*
  ADD-RELATION *94*
  ATTACH-DATA-SOURCE *38*
  BUFFER-COMPARE *40*
  BUFFER-COPY *40*
  COPY-DATASET *193*
  COPY-TEMP-TABLE *193*
  CREATE-LIKE *97*
  GET-CALLBACK-PROC-CONTENT *79*
  GET-CALLBACK-PROC-NAME *79*
  GET-CHANGES *144*
  MARK-NEW *152*
  MARK-ROW-STATE *151*
  MERGE-CHANGES *146*
  MERGE-ROW-CHANGES *146*
  REJECT-CHANGES *148*
  REJECT-ROW-CHANGES *148*
  SAVE-ROW-CHANGES *149*
  SET-CALLBACK-PROCEDURE *79*
  SYNCHRONIZE *129*
Microsoft .NET DataSets *20*, *143*
Microsoft DataSets *20*

N

Navigation mode *135*
NEXT-ROWID attribute *116*
NOT-ACTIVE option *34*, *111*
NUM-ITERATIONS attribute *81*

O

OFF-END query event
  event handler *208*
Options
  LIKE-SEQUENTIAL, DEFINE TEMP-TABLE statement *29*
  LIKE, DEFINE TEMP-TABLE statement *29*
  NOT-ACTIVE, DATA-RELATION object *34*, *111*
  REFERENCE-ONLY, DEFINE DATASET statement *53*
  REPOSITION, Data-Relation object *32*
  REPOSITION, DEFINE DATASET statement *130*
OUTPUT BY-REFERENCE parameter *54*

P

Parameters
  BIND *53*
  INPUT BY-REFERENCE *54*
  OUTPUT BY-REFERENCE *54*
Passing ProDataSets
  discussed *92*
  in called procedure *59*
  reducing data *66*
  using APPEND qualifier *68*
ProDataSet buffers
  accessing *106*
  enhanced query support *120*
  specifying *93*
ProDataSet passing
  as a parameter *51*
  in the caller *55*
  INPUT BY-REFERENCE *54*
  main block *56*
  OUTPUT BY-REFERENCE *54*
  with BIND *53*
  with BY-REFERENCE *53*
ProDataSets
  architecture *21*
  business logic *26*
  caching data in *220*
  capabilities *20*
  change conflicts *153*
  client/server round trips *29*
  compared to temp-tables *23*
  copying *97*
  defining static *29*
  development goals *21*
  events *26*, *171*
  filling entirely *42*
  filling partially *42*
  filling selectively *44*
  filtering *190*
  object life cycles *49*
  parameters *62*
ProDataSets *(continued)*
  partial FILL *185*
  populating *41*
  processing changes *144*
  retrieving detail *191*
  returning partial *190*
  session attributes *122*
  sharing definitions *218*
  standard attributes *34*
  static *29*
  static handles *34*
  static with Data-Relations *29*
  successive loading *182*
  temp-tables *22*
  updating *26*, *135*
  use cases *26*
  views *228*

Q

Queries
  child table *43*
  session attributes *122*

R

RECURSIVE attribute *43*, *94*
RECURSIVE option *43*
Reference architecture *254*
REFERENCE-ONLY option *53*
REJECT-CHANGES method *148*
REJECT-ROW-CHANGES method *148*
REJECTED attribute *167*
REPOSITION and navigation *33*
REPOSITION option *32*, *130*
RESTART-ROW attribute *116*, *180*
RESTART-ROWID attribute *179*
ROW-CREATE event *171*
ROW-DELETE event *171*
ROW-LEAVE event *141*
ROW-STATE attribute *137*
ROW-STATE function *138*
ROW-UPDATE event *171*
ROWID function *38*

S

SAVE-ROW-CHANGES method *149*, *166*
SAVE-WHERE-STRING attribute *155*
SET-CALLBACK-PROCEDURE method *79*, *176*
Standard attributes *34*
Statements
  CREATE DATA-SOURCE *95*
  CREATE DATASET *91*
  DEFINE DATA-SOURCE *35*

Statements *(continued)*
DEFINE DATASET *29*
DELETE OBJECT *93*
SYNCHRONIZE method *129*

T

Temp-tables
passing *23*
static versus dynamic *23*
Temp-tables *(continued)*
versus database tables *23*
TOP-NAV-QUERY attribute *122*
TRACKING-CHANGES attribute *135*, *138*
Trigger procedures *272*

V

Validation procedures *248*, *271*