Progress®



Datasets

OpenEdge®

Copyright

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

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

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

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

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

Overview of datasets....9
ProDataSets....9
ProDataSet concepts....10

Initial dataset setup....11
Define temp-tables for a dataset....12
Define a dataset....12
Define and attach data-sources....14

Populate a dataset using FILL....17
Use FILL-MODE to control how a dataset is populated....18
Example code to populate a dataset....19
Use FILL-WHERE-STRING to assign queries prior to FILL....20
Cleanup after using a dataset....22

Use datasets in a distributed environment....23
Pass a dataset within the same ABL session....24
Pass a dataset from a client to a server session....25
Minimize overhead when passing a dataset parameter....26
Write ABL client code to retrieve a dataset from the server....26

Update and sync dataset data....29
Update datasets in the client....30
Process dataset updates in the server....33
Merge changes back to the client....34
Complete example of client and server code to update a dataset....35

Handle events with datasets....41
Associate a dataset object with an event handler....42
Dataset FILL events....43
Temp-table buffer FILL events 45
Dataset change events....46
Other dataset events....48

Batch data with datasets....49

Client code to retrieve first batch of data....50
Server code to process batched data....50
Manage batched data in the client....52

Handle errors with datasets....55

Types of dataset errors....56
Handle system-generated errors....57
Handle data-related errors....59
Provide error information to the client....61

Preface

Purpose

This document describes a learning path for ProDataSets.

Audience

This document is intended for ABL developers who want to learn about ProDataSets.

Organization

This document consists of the following sections:

- ProDataSets on page 9
- Initial dataset setup on page 11
- Populate a dataset using FILL on page 17
- Use datasets in a distributed environment on page 23
- Update and sync dataset data on page 29
- Complete example of client and server code to update a dataset on page 35
- Handle events with datasets on page 41
- Batch data with datasets on page 49
- Handle errors with datasets on page 55

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

1

Overview of datasets

In this video you learn about datasets, a simple way to combine data from multiple data sources into a format that you can easily share across your application.

For details, see the following topics:

- ProDataSets
- ProDataSet concepts

ProDataSets

A Progress DataSet, or ProDataSet, is also commonly referred to as a dataset. The ProDataSet is a very powerful construct that widens and extends the functionality of temp-tables. A ProDataSet object is basically a collection of one or more member temp-tables. It also optionally contains a collection of data relationships among the member tables.

The benefits of using ProDataSets include:

- Ability to group multiple temp-tables into a single dataset object.
- Ability to define relationships between those tables.
- Ability to associate a data source with each dataset or member temp-table.
- Ability to track changes while manipulating the dataset data.
- Ability to save the changed data back to the data source.
- Ability to pass the dataset as a single parameter from one procedure to another, within a single ABL session or between sessions.
- Simplified management of complex transactions.

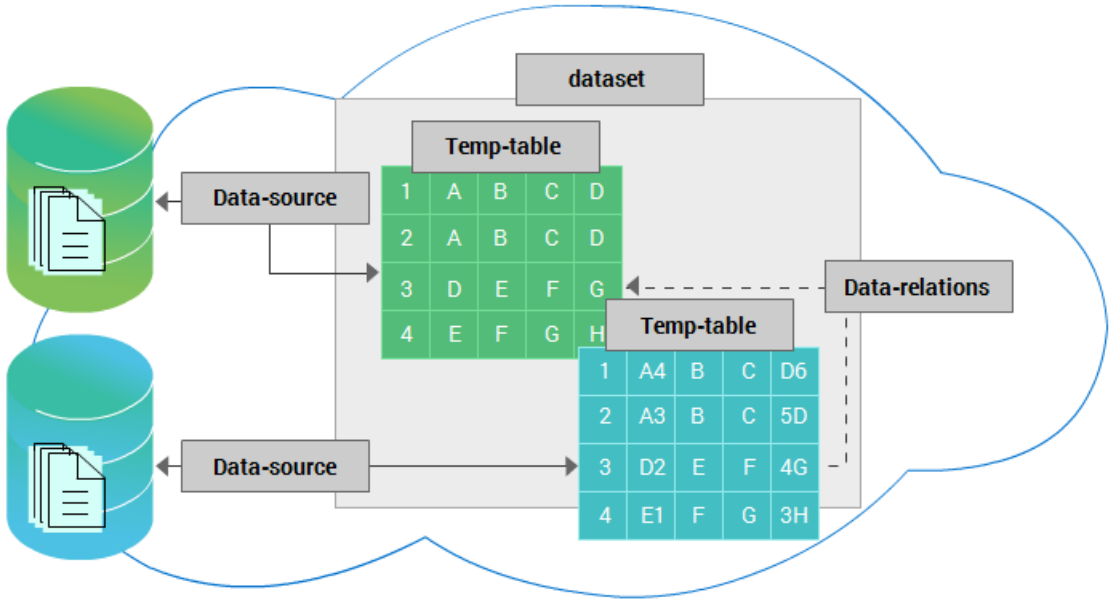
For more information, see Introduction to the OpenEdge DataSet in *Use ProDataSets*.

ProDataSet concepts

A dataset is a single, in-memory business object composed of ABL elements including temp-tables, data relationships, sources of data, and events.

A dataset often specifies the relationships between the component tables. In addition, it may be attached to data sources that can be used to populate the dataset with data. Changes to data in the dataset may then be stored back in the data sources. A dataset provides a mapping between a set of database tables or other data sources and their in-memory, possibly remote, representation.

Figure 1: ProDataSet



For more information, see ProDataSet goals and Architecture in *Use ProDataSets*

2

Initial dataset setup

The main steps for defining a dataset and mapping to data-sources are:

1. Define a temp-table for each table you want to use in the dataset.
2. Define the dataset by specifying the temp-tables and their relationships (if any).
3. Identify, define, and attach the data-sources that are used to populate the dataset.

For details, see the following topics:

- Define temp-tables for a dataset
- Define a dataset
- Define and attach data-sources

Define temp-tables for a dataset

Define temp-tables

To define a temp-table you use the 'DEFINE TEMP-TABLE statement' in *ABL Reference*. When defining a temp-table, you specify each field and index in the temp-table definition. You define each field as a specific data-type, as you would in a `DEFINE VARIABLE` statement. You use the `BEFORE-TABLE` keyword to specify a before-table. The before-table stores the original data from an updated record, when the data in the temp-table record is changed. To define a temp-table with specific fields and indexes, use the following syntax:

```
DEFINE TEMP-TABLE table-name [ NO-UNDO ] BEFORE-TABLE before-table-name
       { FIELD field-name  AS data-type }
       [ INDEX index-name [ IS [ UNIQUE ] [ PRIMARY ] ] { index-field } ].
```

The following is an example of the `DEFINE TEMP-TABLE` statement:

```
DEFINE TEMP-TABLE ttOrder NO-UNDO BEFORE-TABLE bttOrder
       FIELD OrderNum AS INTEGER
       FIELD OrderDate AS DATE
       FIELD ShipDate AS DATE
       FIELD PromiseDate AS DATE
       FIELD OrderTotal AS DECIMAL
       INDEX OrderNum IS UNIQUE PRIMARY OrderNum.
```

See also

'Static ProDataSet and its Data-Relations' in *Use ProDataSets*

Define a dataset

After you define the temp-tables, you use the 'DEFINE DATASET statement' in *ABL Reference* to define the dataset that is composed of those temp-tables. Similar to defining temp-tables in their own include files, you can define a dataset in its own include file. The `DEFINE DATASET` statement is a complex statement with multiple components. You use the statement to:

- Name the dataset and specify the temp-tables that comprise the dataset.
- Define any data-relations between those temp-tables.

Data-relations control how a dataset is populated. (Dataset population is typically done through the `FILL()` method, which is discussed later in this track). Data-relations in a dataset are different from OpenEdge database tables where there are no defined relationships. In a database the relationships are based on data values only and record retrieval of parent and children is based on a join query. In contrast in a dataset, any query used for filling the dataset is defined by each temp-table independently. So, the data-relation tells the AVM how the join is defined.

Syntax for the DEFINE DATASET statement

To define a dataset, you use the DEFINE DATASET statement. The following is the simplified syntax:

```
DEFINE DATASET dataset-name FOR temp-table-name [,temp-table-name ]...
      [ DATA-RELATION [data-relation-name] FOR parent-temp-table-name,
child-temp-table-name
      RELATION-FIELDS (parent-field1, child-field1 [, parent-fieldn, child-fieldn
 ]...) ]
```

dataset-name

Specifies the name of the dataset.

temp-table-name

Specifies the name of the temp-table(s) in the dataset.

data-relation-name

Specifies a data-relation. See 'ProDataSet relations' in *Use ProDataSets* for more detail.

parent-temp-table-name, child-temp-table-name

Identifies the parent and child temp-tables for the data relation.

parent-field*n*, child-field*n*

Define the relationship between fields in the temp-tables. For the most efficient joins, the RELATION-FIELDS fields are indexed.

Examples of the DEFINE DATASET statement

The following example defines a dataset called dsOrderOrderLine. It is comprised of two temp-tables called ttOrder and ttOrderLine. It states that the relationship between the two tables is based upon the ordernum fields. In this example the fields in the two tables have the same name, but the names don't need to be the same. You can specify relation-fields that have different names.

```
DEFINE DATASET dsOrderOrderLine FOR ttOrder, ttOrderline
      DATA-RELATION drOrderOrderLine FOR ttOrder, ttOrderline
      RELATION-FIELDS (ordernum,ordernum).
```

You can define more than one data-relation for a dataset and include more than two temp-tables. The following is an example:

```
DEFINE DATASET dsOrderOrderLineItem FOR ttOrder, ttOrderLine, ttItem
      DATA-RELATION drOrderOrderLine FOR ttOrder, ttOrderLine
      RELATION-FIELDS (Ordernum, Ordernum)
      DATA-RELATION drOrderLineItem FOR ttOrderline, ttItem
      RELATION-FIELDS (Itemnum, Itemnum).
```

Define and attach data-sources

After the dataset is defined, it does not contain any data records. To populate a dataset with data using the `FILL()` method, you need to associate each temp-table buffer in the dataset with a data-source object. After the data-sources are defined and attached, the temp-tables in the dataset are populated automatically or explicitly. As a best practice, define your data-sources in an include file and in your server-side procedure file, write code to attach the temp-tables of the dataset to their data-sources.

A data-source specifies where the data comes from to populate a dataset. Typically, a data-source identifies a set of OpenEdge database tables. The data-source can also identify another database accessed through an OpenEdge DataServer as the source of the data.

Each temp-table buffer in a dataset can be supplied with data from a different data-source. Make sure to define each data-source that you want to use.

Define a data-source

Before you attach data-sources to a dataset, define the data-sources. Data-sources are defined independently from the dataset and are not passed when the dataset is transported from procedure to procedure. This design allows you to switch the data-source at any time to retrieve information from multiple sources.

The following is the simplified syntax to define a data-source:

```
DEFINE DATA-SOURCE data-source-name FOR [QUERY query-name  ]
       [ source-table [KEYS(field...]...].
```

`data-source-name`

The name you give to the data-source.

`query-name`

The name of an optional query that you have previously defined. Recommended if you want to filter what data is retrieved.

`source-table`

Specifies one or more database tables for the data-source.

`field`

Specifies the fields that are primary keys.

The following example defines a data-source for the `OrderLine` table:

```
DEFINE DATA-SOURCE srcOrderLine FOR OrderLine.
```

The following example defines a data-source using a query for the `Order` table:

```
DEFINE QUERY qOrder FOR Order.
       QUERY qOrder:QUERY-PREPARE( "FOR EACH Order WHERE shipdate < TODAY" ).

       DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder.
```

Attach a data-source

After the data-source is defined, it must be attached to the dataset before it can be used to provide data to the dataset. The default mapping is to populate fields with matching names and ignore others. The ATTACH-DATA-SOURCE method associates temp-table buffers with their data-sources.

The following is the simplified syntax for attaching a data-source. Elements of the syntax allow you to modify this default behavior.

```
buffer-handle:ATTACH-DATA-SOURCE (data-source-handle).
```

buffer-handle

The handle of a temp-table buffer in the dataset.

data-source-handle

The handle to the data-source. You can also use DATA-SOURCE *data-source-name*:HANDLE to specify the handle.

The following example code shows how to attach data-sources.

```
BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).
       BUFFER ttOrderLine:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrderLine:HANDLE).
```

See also

'ATTACH-DATA-SOURCE( ) method' in *ABL Reference*

'DEFINE DATA-SOURCE statement' in *ABL Reference*

'Define a static Data-Source' in *Use ProDataSets*

'Attach Data-Sources' in *Use ProDataSets*

3

Populate a dataset using FILL

After you define a dataset and its data-sources, the next step is to populate the dataset with data. The simplest and most effective way to populate a dataset is with the 'FILL( ) method' in *ABL Reference*. The `FILL` method not only populates the dataset from the data-sources you attach, the method also coordinates the loading of data and maintains the integrity of your data-relations.

In a single operation the `FILL` method:

- Retrieves one or more top-level records from the defined data-sources (using `NO-LOCK`).
- Retrieves all the related records (based on the data-relations specified) from the defined data-sources (using `NO-LOCK`).
- Adds the retrieved records to the dataset.

When `FILL` executes, the method starts at a top-level table, retrieves a single parent record and then traverses through all the child records, filling each one with records related to the current parent record. Then, the method retrieves the next parent record and the process continues through all parent-child records. `FILL` uses the data-relations to identify and load the child records for each parent. The following is the syntax for the `FILL` statement:

```
DATASET dataset-name:FILL().
```

The following example shows how to call the `FILL` method on a dataset:

```
DATASET dsOrderOrderLine:FILL().
```

Record access after the FILL

After the FILL method executes, and the temp-tables are populated, you can access the data in the temp-tables using normal ABL syntax. In the following example, immediately after FILL executes, a `FIND FIRST` statement is called on the `ttOrder` temp-table which is part of the `dsOrderOrderLine` dataset.

```
DATASET dsOrderOrderLine:FILL().

      FIND FIRST ttOrder NO-ERROR.
```

I

See also

'Populate a ProDataSet' in *Use ProDataSets*

'Fill the entire ProDataSet' in *Use ProDataSets*

'Partially fill a ProDataSet' in *Use ProDataSets*

For details, see the following topics:

- Use FILL-MODE to control how a dataset is populated
- Example code to populate a dataset
- Use FILL-WHERE-STRING to assign queries prior to FILL
- Cleanup after using a dataset

Use FILL-MODE to control how a dataset is populated

While the default FILL behavior maintains relational integrity, there are cases in which you want to control the population of data records. You can set the 'FILL-MODE attribute' in *ABL Reference* for a temp-table buffer to specify how records are added to the dataset.

How FILL-MODE works

The 'FILL( ) method' in *ABL Reference* can be executed multiple times to add more data to a dataset temp-table. To control how the records are added, you choose a mode for the `FILL` method.

By default, each dataset temp-table is in "MERGE" mode. When you populate the temp-table using the `FILL` method in "MERGE" mode, duplicate records (determined by a unique temp-table index) are not added during the FILL operation. A unique primary temp-table index must exist so that `FILL` can identify duplicate records.

Note: Ensure `FILL-MODE` is set prior to executing a FILL operation.

The following are the modes that you can specify for each temp-table buffer in the dataset before you execute a `FILL`.

FILL-MODE	Description
MERGE (Default)	Adds records. Duplicate records (determined by a unique temp-table index) are not added during a FILL and no error condition is raised if duplicates are found.

FILL-MODE	Description
EMPTY	Empties the temp-table before a FILL begins; can be used to“refresh” a temp-table or dataset by emptying the temp-tables before populating them.
APPEND	Raises an error condition when duplicates are found and the FILL process is aborted. Use "APPEND" when you know there are no duplicates (for example, if you are adding batches of records).
REPLACE	If a row with the same unique primary key values is already present, its field values are replaced by the field values from the data-source. If the row is not already there, it is created in the temp-table.
NO-FILL	The temp-table is not filled. This mode is useful if you are filling a dataset that already has data in it and you want to suppress the FILL for a particular temp-table.

Set the FILL-MODE

The following is the syntax for setting `FILL-MODE` on a temp-table buffer:

```
buffer-handle:FILL-MODE = [ "MERGE" | "EMPTY" | "APPEND" | "REPLACE" | "NO-FILL" ].
```

The following code example shows how to set the `FILL-MODE`:

```
/* set FILL-MODE for each of the temp-tables */
       BUFFER ttOrder:FILL-MODE = "APPEND".
       BUFFER ttOrderLine:FILL-MODE = "APPEND".
```

Empty the table before the FILL

If you already loaded a table with data using `FILL`, and want to empty the table before adding more records to it, you set the `FILL-MODE` to "EMPTY":

```
BUFFER ttOrder:FILL-MODE = "EMPTY".
```

In this example, the next FILL operation empties a temp-table called `ttOrder` before populating it again.

See also

'Control the filling of each table' in *Use ProDataSets*

Example code to populate a dataset

The following example code shows the steps for populating a dataset from the initial definitions to the FILL operation.

```
/* temp-table definitions */
            DEFINE TEMP-TABLE ttOrder NO-UNDO BEFORE-TABLE bttOrder
```

```
            FIELD OrderNum  AS INTEGER
            FIELD CustNum AS INTEGER
            FIELD OrderDate  AS DATE
            FIELD ShipDate  AS DATE
            FIELD PromiseDate  AS DATE
            FIELD OrderTotal  AS DECIMAL
            INDEX OrderNum IS UNIQUE PRIMARY OrderNum.

            DEFINE TEMP-TABLE ttOrderLine NO-UNDO BEFORE-TABLE bttOrderLine
            FIELD OrderNum  AS INTEGER
            FIELD LineNum  AS INTEGER
            FIELD ItemNum  AS INTEGER
            FIELD Price      AS DECIMAL
            FIELD Qty      AS INTEGER
            FIELD Discount  AS INTEGER
            FIELD ExtendedPrice AS DECIMAL
            INDEX OrderNum_LineNum IS UNIQUE PRIMARY OrderNum LineNum.

            /* dataset definition */
            DEFINE DATASET dsOrderOrderLine FOR ttOrder, ttOrderLine
            DATA-RELATION drOrderOrderLine FOR ttOrder, ttOrderLine
            RELATION-FIELDS (OrderNum, OrderNum).

            /* query definition for the top-level temp-table */
            DEFINE QUERY qOrder FOR Order.

            /* data-source definitions */
            DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder. /* Points to query */
            DEFINE DATA-SOURCE srcOrderLine FOR OrderLine. /* Points to table */

            /* prepare query */
            QUERY qOrder:QUERY-PREPARE( "FOR EACH Order WHERE shipdate < TODAY" ).

            /* attach data sources */
            BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).
            BUFFER ttOrderLine:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrderLine:HANDLE).

            /* populate the dataset */
            DATASET dsOrderOrderLine:FILL().
```

Note the query is for the top-level temp-table. FILL populates the child temp-table based on the `DATA-RELATION` definition. For example, all `OrderLines` for the `Orders` that meet the criteria for `qOrder`.

See also

'Define a query on a child table' in *Use ProDataSets*

Use FILL-WHERE-STRING to assign queries prior to FILL

If you want to make a change dynamically to modify how FILL loads records into a dataset, you specify the 'FILL-WHERE-STRING attribute' in *ABL Reference*.

Work with the FILL-WHERE-STRING attribute on a data-source

The query for each buffer has a `FILL-WHERE-STRING` attribute. This attribute helps you determine the query that is in effect at any given time. It also can be used to customize the `WHERE` clause of the query prior to a FILL operation.

The AVM sets the FILL-WHERE-STRING value when it prepares the query. During FILL, the AVM creates a default WHERE clause based on the defined query. For example, the query might be on Order using a WHERE clause:

```
WHERE Order.ShipDate < TODAY.
```

Customize FILL-WHERE-STRING

You also might want to supply more criteria to the WHERE clause. You can modify the FILL-WHERE-STRING attribute to reflect new criteria; this attribute opens the query again to change the scope of records retrieved.

This technique is useful whenever you want to re-assign a new query before FILL. For example, you could create a number of different reports using the same dataset, but change the query each time to retrieve a different set of records.

For example, assume your query gets all Orders for a Customer. The FILL-WHERE-STRING value would be:

```
WHERE Order.CustNum = Customer.CustNum.
```

If you want to change the query to get only orders that are > $1,000, you can add to the FILL-WHERE-STRING as follows:

```
myString = DATA-SOURCE srcOrder:FILL-WHERE-STRING
  + " AND orderTotal > 1,000".
DATA-SOURCE srcOrder:FILL-WHERE-STRING = myString.
```

Steps for customizing the FILL-WHERE-STRING

To modify the FILL-WHERE-STRING, follow these steps:

1. Add to the FILL-WHERE-STRING.
2. Define a query to use the new criteria.
3. FILL the dataset.

Example using a dynamic query

```
VAR CHARACTER cMyString.
VAR HANDLE hOrderQuery.

{include\dsOrderOrderLine.i}

BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).
BUFFER ttOrderLine:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrderLine:HANDLE).
DATA-SOURCE srcOrder:FILL-WHERE-STRING = "WHERE ShipDate < TODAY".

/* Add to the FILL-WHERE-STRING */
cMyString = "For Each Order " + DATA-SOURCE srcOrder:FILL-WHERE-STRING
  + " AND OrderDate > 1/1/05".

MESSAGE cMyString VIEW-AS ALERT-BOX.

/* Define a dynamic query and prepare the query. */
/* Create a query for the Order */
CREATE QUERY hOrderQuery.
hOrderQuery:SET-BUFFERS(BUFFER Order:HANDLE).
hOrderQuery:QUERY-PREPARE(cMyString).
DATA-SOURCE srcOrder:QUERY = hOrderQuery.

/* Fill the dataset using the new FILL criteria. */
DATASET dsOrderOrderLine:FILL().

FIND FIRST ttOrder NO-ERROR.
```

```
IF AVAILABLE (ttOrder) THEN
  MESSAGE ttOrder.OrderNum ttOrder.ShipDate VIEW-AS ALERT-BOX.
```

Cleanup after using a dataset

There are two recommended practices for cleaning up after using datasets in your code:

- DETACH-DATA-SOURCE
- EMPTY-DATASET

DETACH-DATA-SOURCE

Generally, you should detach data-sources when you are done using them, unless you plan to populate or update the dataset later in the procedure.

The syntax to detach a data-source is:

```
buffer-handle:DETACH-DATA-SOURCE().
```

For example:

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

EMPTY-DATASET

If you plan to empty and populate a dataset again, or if you no longer need the data in your dataset in memory, use the EMPTY-DATASET method to empty a dataset object of all records in its associated temp-tables.

Once the dataset is emptied using this method, you may continue to use the dataset in your application.

The syntax to use the EMPTY-DATASET method is:

```
DATASET dataset-name:EMPTY-DATASET().
```

In the following example code, the EMPTY-DATASET method is called on the dsOrderOrderline dataset, removing all records. The structure of the dataset is left intact so it can be populated again.

```
DATASET dsOrderOrderLine:EMPTY-DATASET().
```

See also

'DETACH-DATA-SOURCE( ) method' in *ABL Reference*

'EMPTY-DATASET( ) method' in *ABL Reference*

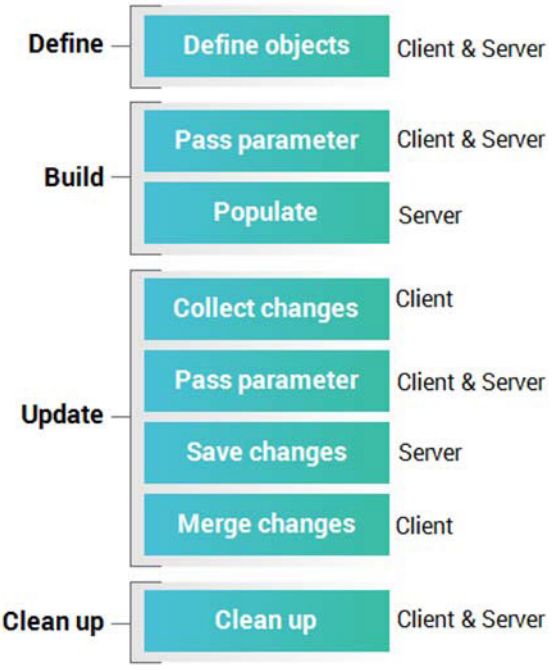
'Other ProDataSet methods' in *Use ProDataSets*

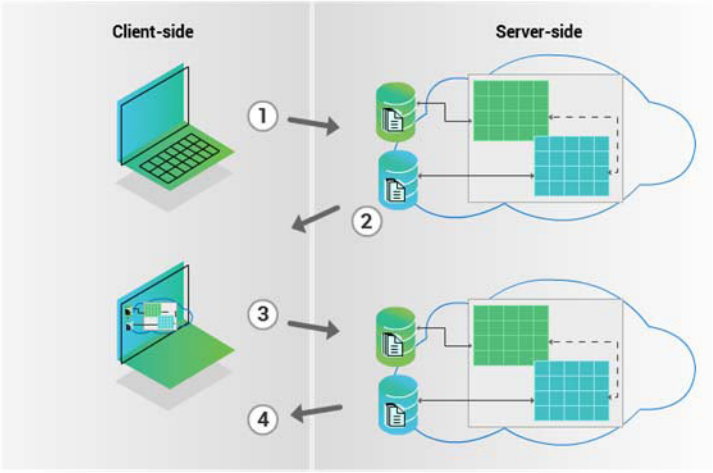
4

Use datasets in a distributed environment

To use datasets in a distributed environment, you pass a dataset as a parameter from one procedure to another or between a server and a client. The ability to pass a dataset as a parameter is the foundation for building distributed applications.

One of the most powerful features of a dataset is its ability to be transported or “marshaled” between client and server procedures. The following illustration shows steps on the client and server sides for defining, building, updating and cleaning up a dataset.





The most common reason for passing a dataset as a parameter is the “round-trip” scenario:

1. A client requests a dataset from the server by calling a procedure in the server, which populates the dataset and returns the dataset as a parameter to the client.
2. The client makes changes to the data in the retrieved dataset.
3. The client passes the changes to the server as a parameter by calling a procedure in the server, which applies the changes to the database and returns the result of the changes to the client.
4. The client reconciles the changes in its copy of the dataset.

Pass a dataset

When you develop your ABL application, plan for how a dataset is used within the ABL session and between ABL sessions. The server-side code may have multiple procedures that are used to process a client request for a dataset. For example, the client may call `GetData` which formats the query used to populate the dataset. `GetData` then calls `PopulateData`, which fills the dataset and returns the dataset to `GetData`. Finally, `GetData` returns the dataset to the client.

When passing a dataset between procedures in the same session, it is a good practice to ensure that the dataset is passed by reference. This practice ensures all routines are working on the exact same dataset, rather than copying the data from one instance of the dataset to another.

When the dataset is passed between the client and server (between different ABL sessions), the dataset must be passed by value.

See also

'Pass a ProDataSet as a parameter' in *Use ProDataSets*

For details, see the following topics:

- Pass a dataset within the same ABL session
- Pass a dataset from a client to a server session
- Minimize overhead when passing a dataset parameter
- Write ABL client code to retrieve a dataset from the server

Pass a dataset within the same ABL session

Pass a dataset within the same ABL session

When passing a dataset within the same ABL session it is common to pass the dataset by reference.

Provided a dataset was previously defined, you can define a dataset parameter by name as input, output, or input-output. The simplified syntax for defining a dataset parameter by name is:

```
DEFINE {INPUT|OUTPUT|INPUT-OUTPUT}
       PARAMETER DATASET FOR dataset-name.
```

For example:

```
/* PopulateData.p */
       ...

       DEFINE INPUT PARAMETER DATASET FOR dsOrderOrderLine.
```

This statement allows the dsOrderOrderLine dataset to be passed back to the calling procedure as an output parameter.

When a procedure that runs in the same ABL session calls another procedure, passing the dataset parameter, the calling procedure specifies the BY-REFERENCE keyword. In the following example, the GetData.p procedure calls the PopulateData.p procedure:

```
/* GetData.p */
       ...

      RUN PopulateData (INPUT pCriteria, INPUT DATASET dsOrderOrderLine BY-REFERENCE).
```

In this case, both procedures are using the same dataset object at runtime.

See also

'Pass a ProDataSet parameter by binding' in *Use ProDataSets*

'Pass a ProDataSet parameter by reference' in *Use ProDataSets*

Pass a dataset from a client to a server session

When a dataset is passed between a server and a client, it is passed by value.

Server-side

The following is the simplified syntax for defining the dataset parameter in server-side code:

```
DEFINE { INPUT | OUTPUT | INPUT-OUTPUT } PARAMETER
      {   DATASET FOR dataset-name
      | DATASET-HANDLE dataset-handle.
      }
```

The OUTPUT keyword is used for retrieving the dataset from the server.

The following is a server-side example of defining the dataset parameter and returning the dataset to the client. Notice an external procedure file is used, where all internal procedures, related to accessing the dsOrderOrderLine dataset, are placed.

```
/* Server-side code */

      {include\dsOrderOrderLine.i}

      PROCEDURE GetData:

      DEFINE INPUT PARAMETER pFilter AS CHARACTER NO-UNDO.
      DEFINE OUTPUT PARAMETER DATASET FOR dsOrderOrderLine.

      /* populate the dsOrderOrderLine dataset */
```

```
      RETURN.
      END PROCEDURE.
```

See also

'ProDataSet Parameters' in *Use ProDataSets*

Minimize overhead when passing a dataset parameter

When temp-tables that are part of a dataset are passed across a network, their schema can sometimes be substantial, leading to unnecessary overhead and slow processing times while transporting the dataset. If the client code has its own static dataset definition, you should use the 'SCHEMA-MARSHAL attribute' in *ABL Reference* to limit the amount of information that is transported across the network when the dataset is returned. By default, the `SCHEMA-MARSHAL` attribute for a temp-table is set to `"FULL"`, which means that all schema information is sent. The schema is used to make sure the schema on both sides match.

To limit the amount of information, you can set the `SCHEMA-MARSHAL` attribute for each temp-table in the dataset to `"NONE"`. You do not set the attribute on the dataset itself. If setting the attribute to `"NONE"` ensure that the schema on both sides match.

You set the `SCHEMA-MARSHAL` attribute on the server side that passes the dataset to the client. You do not need to specify the attribute on the client side.

Pass without sending schema information

The syntax to specify that a temp-table in a dataset is being passed without its schema definition is:

```
table-handle:SCHEMA-MARSHAL = "NONE".
```

In the following example, no schema information is passed with the dataset temp-tables. Ensure the calling procedure has a matching definition of the dataset and temp-tables to receive the passed dataset.

```
ASSIGN
      TEMP-TABLE ttOrder:SCHEMA-MARSHAL = "NONE"
      TEMP-TABLE ttOrderLine:SCHEMA-MARSHAL = "NONE".
```

See also

'Specify the session startup parameter for all temp-table parameters' in *Use ProDataSets*

'Set attributes for individual temp-table parameters' in *Use ProDataSets*

Write ABL client code to retrieve a dataset from the server

The server-side code runs in a Progress Application Server (PAS) for OpenEdge instance, so the client code needs to connect to this application server instance prior to accessing the server-side procedures.

Progress OpenEdge supports all types of clients, but in this set of topics an ABL client is used. The ABL client can be developed using a user interface (UI), such as the ABL GUI or GUI for .NET, or a batch procedure file that runs in the background.

Note: The client code you learn to write in this set of topics is only that part of the client code that retrieves the dataset. In a real client, you would add more code to present a UI or dynamically specify how filters are provided.

Call server-side code to retrieve a dataset

Recall that the server-side procedure that returns the dataset passes it by value. In the client, you call the procedure in the server providing an OUTPUT parameter with the DATASET specification.

The following is the simplified syntax for calling an internal procedure that runs in the server that returns a named dataset by value. In this syntax *procedure-name* is an internal procedure within a persistent procedure on the server.

```
RUN procedure-name [IN procedure-handle] ([{parameters},]
       [INPUT|OUTPUT|INPUT-OUTPUT] DATASET dataset-name).
```

The following is the simplified syntax to run an external procedure. In this syntax *procedure-name* is an external procedure ON SERVER *server-handle*.

```
RUN procedure-name ON [ SERVER ]{server-handle} ([{parameters},]
       [INPUT|OUTPUT|INPUT-OUTPUT] DATASET dataset-name).
```

See the 'RUN statement' in *ABL Reference* for more information.

Retrieve a dataset using APPEND

When the client receives the dataset from the server, you can specify whether you want the dataset to be replaced or appended to. By default, the data in the dataset is replaced. If you want to append the data in the dataset, you add the APPEND keyword to the RUN statement as follows:

```
RUN GetData IN hProc ("OrderNum = 8005", OUTPUT DATASET dsOrderOrderLine APPEND).
```

In this example, the procedure GetData is executed and the dataset dsOrderOrderLine is returned to the calling procedure. If the dataset has data in it for other Orders, the data in the dataset that is passed back is appended to the existing data.

When a dataset is returned to the client using the APPEND keyword, an error occurs if duplicate rows are found. For this reason, you need to write code to pro-actively prevent receiving duplicate rows.

For example, if your client-side dataset contains all order records for customer 1, the next call that you make to retrieve data from the server (using APPEND) should filter data so that no customer 1 records are included in the dataset that is returned.

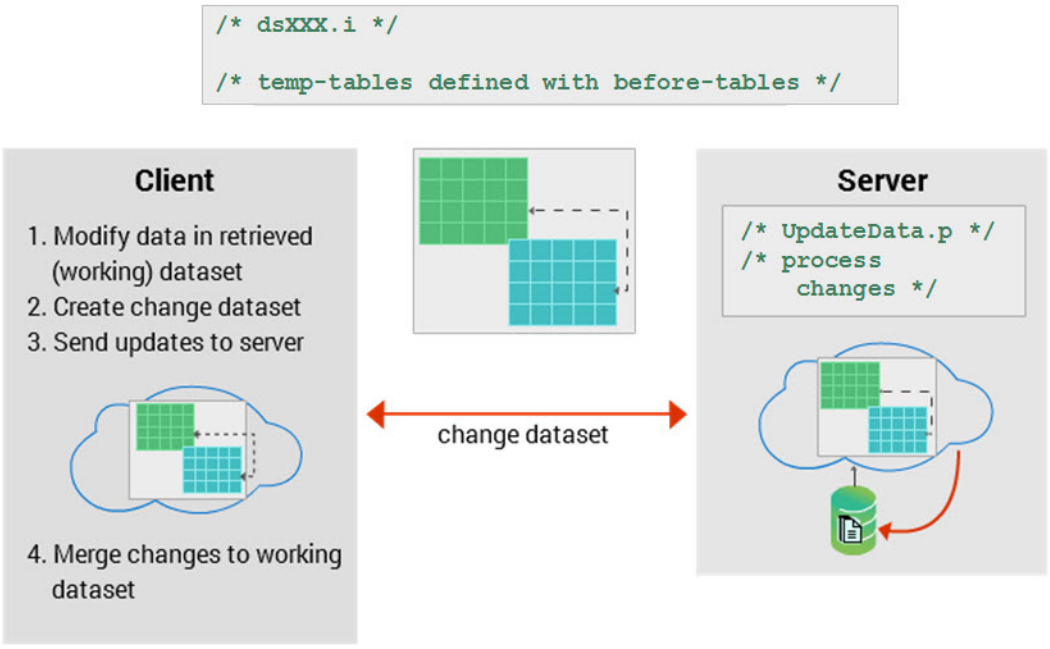
See also

'Pass a ProDataSet with APPEND' in *Use ProDataSets*

5

Update and sync dataset data

After a dataset is retrieved from the server, the client may want to modify the data in the dataset and submit the modifications to the server.



The steps for updating data with a dataset are as follows:

Client-side

1. Update the working dataset.

2. Prepare a change dataset to be sent to the server.

3. Call a procedure to pass updates to the server.

Server-side

4. Define procedure and parameters to work with the change dataset sent from the client.

5. Write changes back to the database.

6. Return the changed dataset to the client.

Client-side

7. Merge the returned changed dataset back to the working dataset.

8. Handle any errors.

For details, see the following topics:

- Update datasets in the client
- Process dataset updates in the server
- Merge changes back to the client
- Complete example of client and server code to update a dataset

Update datasets in the client

After a dataset is retrieved from the server, the client may want to modify the data in the dataset and submit the modifications to the server. Since a dataset often represents a set of records in multiple tables, the update process can be more complex than updating a single table.

When writing client code that updates dataset data, the two types of datasets you work with are:

- Working dataset — the dataset retrieved from the server and updated in the client.
- Change dataset — the dataset sent to the server which contains only changed information.

Before- and after- tables

When you define a dataset temp-table (see 'DEFINE TEMP-TABLE statement' in *ABL Reference*) for use in updating data on the server, you also define a `BEFORE-TABLE`. When changes are made to a record, the before-table stores the before version of the record. The temp-table of the working dataset contains the complete temp-table with updates and is referred to as an after-table.

Before-tables contain:

- Copies of modified and deleted records in the state they were in before they were modified or deleted.
- Placeholders for newly created records.

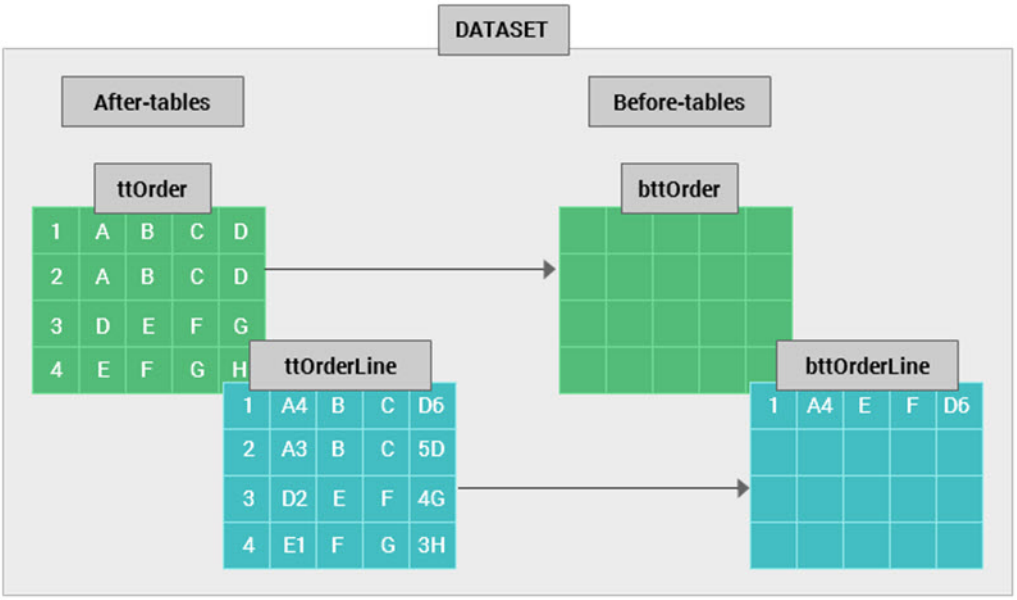
The 'ROW-STATE attribute' in *ABL Reference* contains the current change state of the ProDataSet temp-table row associated with the buffer object handle and has one of the following values: `ROW-UNMODIFIED`, `ROW-DELETED`, `ROW-MODIFIED`, or `ROW-CREATED`.

The 'MARK-ROW-STATE( ) method' in *ABL Reference* forces the creation of before-table records and assigns `ROW-STATE`. It is a way to apply changes without `FILL` or `TRACKING-CHANGES`.

Example of defining a before-table

```
DEFINE TEMP-TABLE ttOrder NO-UNDO BEFORE-TABLE bttOrder
           FIELD OrderNum AS INTEGER INITIAL 0
           FIELD CustNum AS INTEGER INITIAL 0
           . . .
```

Whenever the client creates, deletes, or updates records in their local copy of the working dataset, they must ensure that the changes they make to the dataset are marked as updates in the before-table of the dataset's temp-table. The change dataset that the client prepares contains all before-table records and the updated after-table records. The server uses the before-table records to apply changes to the database.



To ensure that updates to the working dataset are marked as changes in the before-table, you set TRACKING-CHANGES for the specific temp-table in the working dataset.

Use TRACKING-CHANGES

Each dataset temp-table has a 'TRACKING-CHANGES' in *ABL Reference* attribute that can be enabled to track changes. The TRACKING-CHANGES attribute on each temp-table is set to FALSE by default when the dataset is defined. You explicitly turn on TRACKING-CHANGES.

The following is the syntax for setting and turning TRACKING-CHANGES on or off for a temp-table. You set TRACKING-CHANGES to TRUE for any client-side temp-table that you update.

```
TEMP-TABLE table-name:TRACKING-CHANGES = TRUE | FALSE.
```

When updates to a temp-table are done, you turn TRACKING-CHANGES off by setting its value to FALSE. For example:

```
/* Client.p */

       /* set TRACKING-CHANGES for ttOrderLine */
       TEMP-TABLE ttOrderLine:TRACKING-CHANGES = TRUE.

      /* update ttOrderLine which adds a record to bttOrderLine with the original data
 */
       CREATE ttOrderLine.
```

```
       /* assign values to the ttOrderLine record */

       /* unset TRACKING-CHANGES when done with updates to ttOrderLine */
       TEMP-TABLE ttOrderLine:TRACKING-CHANGES = FALSE.
```

Prepare the change dataset to send to the server

To prepare the change dataset for sending to the server, you:

1. Create a change dataset ('CREATE DATASET statement' in *ABL Reference*).
2. Add schema information from the working dataset to the dynamic change dataset ('CREATE-LIKE( ) method' in *ABL Reference*).
3. Place before-table and related after-table records in the change dataset (GET-CHANGES( ) method' in *ABL Reference*).

You create a dynamic change dataset by assigning the dataset object to a handle variable. The following is the simplified syntax for creating a dynamic dataset:

```
CREATE DATASET change-dataset-handle.
```

After you create the dynamic change dataset, you copy the schema from the working dataset to the change dataset.

The following is the simplified syntax for copying schema from one dataset to a newly created dynamic dataset:

```
change-dataset-handle:CREATE-LIKE(working-dataset-handle).
```

After the change dataset is initialized with the schema information, you use the GET-CHANGES method to copy the before-table and related after-table records, in preparation for sending the change dataset to the server.

Note: Ensure TRACKING-CHANGES is set to FALSE for all temp-tables in the working dataset before you call GET-CHANGES.

The following is the simplified syntax for GET-CHANGES:

```
change-dataset-handle:GET-CHANGES (working-dataset-handle)
```

The following code shows the steps involved in preparing the change dataset.

```
VAR HANDLE hChangeDataSet.

       /* Create and prepare the change dataset */
       CREATE DATASET hChangeDataSet.
       hChangeDataSet:CREATE-LIKE(DATASET dsOrderOrderLine:HANDLE).
       hChangeDataSet:GET-CHANGES(DATASET dsOrderOrderLine:HANDLE).
```

Call the update procedure in the server

After the change dataset is prepared, the next step is to call the server-side procedure to update the database. The update procedure that runs in the server takes the change dataset as a parameter.

The following is the syntax for calling a server-side internal procedure using the handle to its external procedure that updates data based upon the change dataset:

```
RUN update-proc-name IN proc-handle (
       INPUT-OUTPUT DATASET-HANDLE change-dataset-handle).
```

Notice the parameter must be INPUT-OUTPUT because the server-side code returns the updated change dataset back to the client. The parameter type is DATASET-HANDLE, but as you learned, the dataset is passed by value between the client and server.

The following is an example of the update call from the client:

```
/* Update the data in the server */

       RUN datasetProcs.p ON hServer PERSISTENT SET hProc.
      RUN updateOrderOrderLineDS IN hProc (INPUT-OUTPUT DATASET-HANDLE hChangeDataSet).
```

See also

'Dynamic ProDataSet Basics' in *Use ProDataSets*

Process dataset updates in the server

On the server you apply the changes to the data source using the 'SAVE-ROW-CHANGES( ) method' in *ABL Reference* on the before buffer of the temp-table.

The following is the syntax for saving row changes for a temp-table buffer:

```
before-table-buffer-handle:SAVE-ROW-CHANGES().
```

The SAVE-ROW-CHANGES method validates the before-table record matches the current database record. If there is a match, it updates the database with corresponding after-table record values. It then repopulates the change dataset temp-table with data from the database. These changes could include automated changes made by database triggers or changes made by other clients.

If SAVE-ROW-CHANGES determines the before-table record does not match the current database record, it means that someone else has modified this record since it was read. Errors may be raised if the before-table record does not match or if the update does not succeed. You need to write code to handle these errors.

Transaction scope with SAVE-ROW-CHANGE

Since there is no method to save changes for an entire dataset or an entire temp-table, use SAVE-ROW-CHANGES at the row level. Since you are processing changes at the row level, you have complete control over the size and scope of each transaction. SAVE-ROW-CHANGES starts its own sub-transaction.

Define a large transaction

In some cases, it is desirable to have the transaction scoped at the individual record level. However, if your application requires a larger transaction scope (for example, updates all order data for an order in a single transaction), explicitly define the larger transaction using a TRANSACTION block.

Maintain database integrity

Depending on the relationships between your dataset tables, you may not be able to simply loop through each before-table in its entirety and perform SAVE-ROW-CHANGES on each row in turn. To maintain database integrity, you may have to process before-table records in a specific sequence across a group of tables (parent-child-child-child, next parent-child-child-child) and define a transaction scope that includes each logical grouping of tables. This processing enables you to roll back a transaction that spans multiple records in multiple tables if the update fails for any reason.

Manage change conflicts

While the save process can be very easy if you use the default behavior of the SAVE-ROW-CHANGES method, you may find that you need more control over the priority of changes when change conflicts are encountered. ABL datasets allow you to handle database change conflicts in a variety of ways.

Conflict resolution choices

Because the SAVE-ROW-CHANGES method supports an optimistic locking strategy, it is possible that another client may have modified a record since the dataset records were initially read. Depending on your business rules, you may want to do one of the following:

- Reject the dataset changes in favor of the changes made by other users. This option copies the database record back to the change dataset which is returned to the client. This behavior is the default.
- Reject the database changes made by others in favor of the dataset changes. This option is not recommended, so use with caution.

Detach the data-source

After the updates are processed, detach the data-sources using the 'DETACH-DATA-SOURCE( ) method' in *ABL Reference*. Detach the data-sources as soon as you are done with them. The following code example detaches the data-sources for `ttOrder` and `ttOrderLine`.

```
    BUFFER ttOrder:DETACH-DATA-SOURCE( ).
       BUFFER ttOrderLine:DETACH-DATA-SOURCE( ).
```

See also

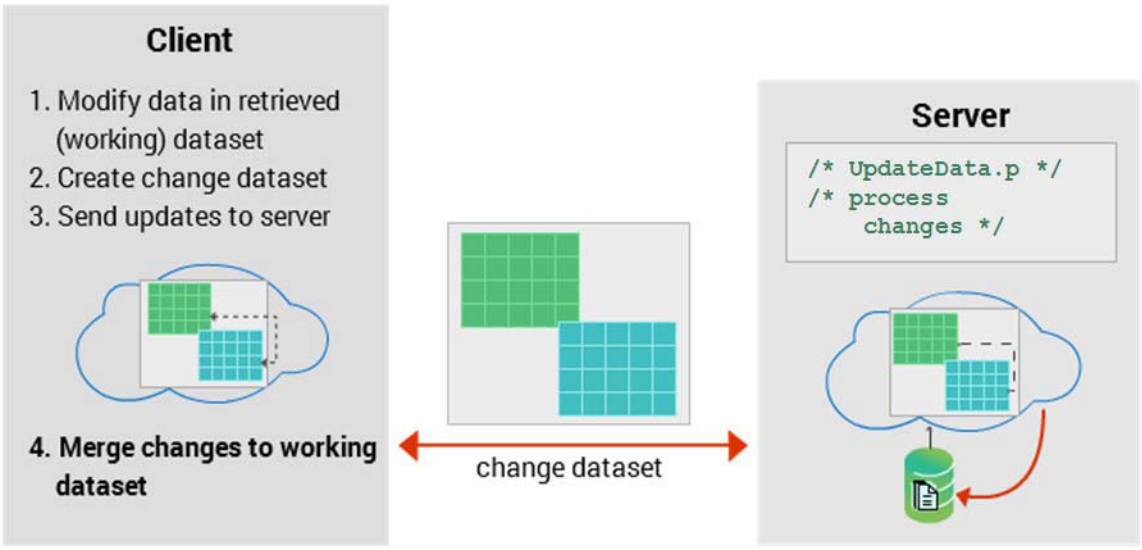
'Access a member buffer of a ProDataSet' in *Use ProDataSets*

Merge changes back to the client

When the server returns the change dataset back to the client, the returned dataset may not match the working dataset of the client because the dataset may contain changes made by database triggers or by other users. These server-side changes must be merged into the working dataset on the client.

Merging changes after an update is different from retrieving the data. When you retrieve the data, you do not know what changed. When you merge changes, you can examine every record to process the changes or errors that occurred as a result of the update call.

You write code to merge the changes returned from the server and clear the before-table in the working dataset. In some cases, the returned dataset may contain errors.



Merge changes back to the working dataset

You use the 'MERGE-CHANGES( ) method' in *ABL Reference* to merge any server-side changes back into the working data in the client. When you merge the changes, the before-table for the working dataset is cleared. Before you call `MERGE-CHANGES` for a returned change dataset, you should ensure that there are no errors in the returned change dataset, by checking the 'ERROR attribute' in *ABL Reference*.

In the following example code, these steps are performed on the client:

1. The client sends the change dataset to the server for update.
2. The change dataset is returned from the server.
3. If no errors are found, the changes are merged back into the working dataset and the before-table of the working dataset is cleared.

```
/* client.p */
       . . .
       /* update the data in the server */
      RUN updateOrderOrderLineDS IN hProc(INPUT-OUTPUT DATASET-HANDLE hChangeDataSet).

       IF hChangeDataset:ERROR = FALSE
       THEN DO:
       hChangeDataset:MERGE-CHANGES(DATASET dsOrderOrderLine:HANDLE).
       END.
```

See also

'MERGE-CHANGES and MERGE-ROW-CHANGES methods' in *Use ProDataSets*

'Update Data with ProDataSets' in *Use ProDataSets*

Complete example of client and server code to update a dataset

The following example demonstrates the client and server code for working with data in a dataset.

Include file for both client and server

The following include file code contains the temp-table, dataset, and data-source definitions. The file is used on both the client and the server.

```
/* dsOrderOrderLine.i */
DEFINE TEMP-TABLE ttOrder NO-UNDO BEFORE-TABLE bttOrder
FIELD OrderNum AS INTEGER
FIELD CustNum AS INTEGER
FIELD OrderDate AS DATE
FIELD ShipDate AS DATE
FIELD PromiseDate AS DATE
FIELD OrderTotal AS DECIMAL
INDEX OrderNum IS UNIQUE PRIMARY OrderNum.

DEFINE TEMP-TABLE ttOrderLine NO-UNDO BEFORE-TABLE bttOrderLine
FIELD OrderNum AS INTEGER
FIELD LineNum AS INTEGER
FIELD ItemNum AS INTEGER
FIELD Price AS DECIMAL
FIELD Qty AS INTEGER
FIELD Discount AS INTEGER
FIELD ExtendedPrice AS DECIMAL
INDEX OrderNum_LineNum IS UNIQUE PRIMARY OrderNum LineNum.


DEFINE DATASET dsOrderOrderLine FOR ttOrder, ttOrderline
DATA-RELATION drOrderOrderLine FOR ttOrder, ttOrderline
RELATION-FIELDS (OrderNum,OrderNum).

DEFINE QUERY qOrder FOR Order.
DEFINE DATA-SOURCE srcOrder FOR QUERY qOrder.

DEFINE DATA-SOURCE srcOrderLine FOR OrderLine.
```

Example code on client

`client.p` is the procedure code on the client-side. This procedure performs the following steps:

1. Define the temp-tables, dataset, and data-sources.

   The procedure uses an include file to define the temp-tables, dataset, and data-sources for the application. This same include file is also used on the server.

2. Connect to the server.

   The procedure connects to the server and runs `datasetProcs.p` persistently. `datasetProcs.p` contains two server procedures that the client calls.

3. Fill the dataset with data from the server.

   The empty, working dataset, `dsOrderOrderLine`, is passed as a parameter to the `fillOrderOrderLinesDS` procedure, which runs on the server. Upon returning to the client, the dataset is now populated with the data from the data-sources.

4. Update some data in the dataset.

   A `ttOrder` record is retrieved so that it can be updated. `TRACKING-CHANGES` is turned on for both the temp-tables (`ttOrder` and `ttOrderLine`) in the dataset. The `ttOrder` record is updated with a ship date and a new `ttOrderLine` record is created. After the update `TRACKING-CHANGES` is turned off.

5. Send the updated data to the server so it can be updated on the server.

A changes-only dataset is prepared to send to the server. This changes-only dataset is then passed as a parameter to the `updateOrderOrderLineDS` procedure, which runs on the server and makes the updates.

6. Merge the changes back into the working dataset on the client.

   After returning to the client, if there were no errors, `MERGE-CHANGES` is called to merge the changes back into the working dataset on the client. The client and server data are now in sync.

```
/* client.p */
{include\dsOrderOrderLine.i}

VAR HANDLE hServer.
VAR HANDLE hProc.
VAR HANDLE hChangeDataSet.
VAR INT saveCust.

/* This is where the connection to the PASOE server would be.
   for this sample code, using hServer = SESSION */
hServer = SESSION:HANDLE.

RUN datasetProcs.p ON hServer PERSISTENT SET hProc.

RUN fillOrderOrderLineDS IN hProc (OUTPUT DATASET dsOrderOrderLine).

FIND FIRST ttOrder NO-ERROR.

IF AVAILABLE (ttOrder) THEN DO:

    /* Set TRACKING-CHANGES for ttOrder and ttOrderLine */
    TEMP-TABLE ttOrder:TRACKING-CHANGES = TRUE.
    TEMP-TABLE ttOrderLine:TRACKING-CHANGES = TRUE.

    /* Change the shipDate for the ttOrder.
       This adds a record to bttOrder with the original data
       and ROW-STATE of ROW-MODIFIED */
    ttOrder.ShipDate = TODAY.

    /* Create a new ttOrderLine for the current ttOrder.
       This adds a record to the bttOrderLine temp-table
       with initial values and a ROW-STATE of ROW-CREATED */
    CREATE ttOrderLine.
    ttOrderLine.OrderNum = ttOrder.OrderNum.
    ttOrderLine.LineNum = 900.

    /* Turn TRACKING-CHANGES off */
    TEMP-TABLE ttOrder:TRACKING-CHANGES = FALSE.
    TEMP-TABLE ttOrderLine:TRACKING-CHANGES = FALSE.

END.

/* Get the changes-only dataset to send to the server */
CREATE DATASET hChangeDataSet.

hChangeDataSet:CREATE-LIKE(DATASET dsOrderOrderLine:HANDLE).

hChangeDataSet:GET-CHANGES(DATASET dsOrderOrderLine:HANDLE).


/* Update the data on the server */
RUN updateOrderOrderLineDS IN hProc (INPUT-OUTPUT DATASET-HANDLE hChangeDataSet).

IF hChangeDataSet:ERROR = TRUE THEN  DO:
    MESSAGE "errors during update " VIEW-AS ALERT-BOX.
END.
ELSE DO:
    hChangeDataset:MERGE-CHANGES(DATASET dsOrderOrderLine:HANDLE).
END.
```

```
DELETE PROCEDURE hProc.

END.
```

Example code on server

The server code, `datasetProcs.p`, uses the same include file to define the temp-tables, dataset, and data-sources. The code contains two internal procedures, `fillOrderOrderLineDS` and `updateOrderOrderLineDS`. The `fillOrderOrderLineDS` procedure performs the initial `FILL` of the dataset with records from the data-sources. The `updateOrderOrderLineDS` procedure updates the data-sources by calling `SAVE-ROW-CHANGES` to validate and then update the affected records.

```
/* datasetProcs.p */
{include\dsOrderOrderLine.i}

PROCEDURE fillOrderOrderLineDS:

    DEFINE OUTPUT PARAMETER DATASET FOR dsOrderOrderLine.

    QUERY qOrder:QUERY-PREPARE( "FOR EACH Order WHERE ShipDate < TODAY" ).

    BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).
    BUFFER ttOrderLine:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrderLine:HANDLE).

    DATASET dsOrderOrderLine:FILL().

    /* Detach the data sources */
    BUFFER ttOrder:DETACH-DATA-SOURCE( ).
    BUFFER ttOrderLine:DETACH-DATA-SOURCE( ).

    RETURN.
END PROCEDURE.

PROCEDURE updateOrderOrderLineDS:

    /* Dataset with just the changes made on the client */
    DEFINE INPUT-OUTPUT PARAMETER DATASET FOR dsOrderOrderLine.

    /* Attach the data sources in order to save changes back to the database
       This example passes the change dataset as a dataset-handle, but using
       the static dataset makes the code simpler */
    BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).
    BUFFER ttOrderLine:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrderLine:HANDLE).

    FOR EACH bttOrder:

       MESSAGE "saving order row-state = "  BUFFER bttOrder:ROW-STATE VIEW-AS ALERT-BOX.

        MESSAGE "saving " bttOrder.OrderNum bttOrder.ShipDate VIEW-AS ALERT-BOX.

        BUFFER bttOrder:SAVE-ROW-CHANGES().
    END.

    FOR EACH bttOrderLine:

        MESSAGE "saving orderline row-state = "  BUFFER bttOrderLine:ROW-STATE VIEW-AS
 ALERT-BOX.
       MESSAGE "saving " bttOrderLine.OrderNum bttOrderLine.LineNum VIEW-AS ALERT-BOX.


        BUFFER bttOrderLine:SAVE-ROW-CHANGES().
    END.

    /* Detach the data sources */
    BUFFER ttOrder:DETACH-DATA-SOURCE( ).
    BUFFER ttOrderLine:DETACH-DATA-SOURCE( ).
```

```
    RETURN.
END PROCEDURE.
```

6

Handle events with datasets

You can write application code that handles system-generated events related to datasets. For example a system-generated event is triggered when a row is created for a temp-table. You could use the event to execute a custom event handler to calculate initial values for other fields in the temp-table record or to modify data in a related temp-table.

In ABL, a system-generated event is associated with an ABL object such as a dataset or temp-table and has a pre-defined name. You can write a custom event handler (procedure) for a system-generated event. At runtime, when an operation occurs, such as the creation of a row, the AVM automatically triggers a call to your event handler procedure. The system-generated events used with datasets and their related objects can be categorized as:

- Fill events
- Change events
- Query events

Implement dataset event handling

Event handlers are internal procedures. You add event handler procedures in the calling procedure file or you add them as internal procedures in a separate external procedure file that is run persistently. Defining event handlers as internal procedures in an external procedure enables you to keep the code related to the event handlers for a dataset in a single file. You write code to associate the event handler procedure with a dataset event so that at runtime the AVM calls a specified procedure when a named event is triggered.

To implement dataset event handling, you typically:

1. Create the external procedure file to hold the event handler procedures for a dataset.
2. Write the event handler code for each required dataset event.

3. Run the external procedure file persistently.
4. In the code that fills or changes the dataset, associate the event handler procedure with a dataset or temp-table.

Example of event handlers in an external procedure file

To implement dataset handlers as internal procedures, you create an external procedure file and add the internal procedures for the event handlers to this file. For example:

```
/* dsOrderOrderLine_EventHandlers.p */
BLOCK-LEVEL ON ERROR UNDO, THROW.

{include/dsOrderOrderLine.i}

PROCEDURE AddTax:
  DEFINE INPUT PARAMETER DATASET FOR dsOrderOrderLine.
  /* event handler code */
END PROCEDURE.

PROCEDURE AdjustCarrier:
  DEFINE INPUT PARAMETER DATASET FOR dsOrderOrderLine.
  IF ttOrder.OrderStatus = "Ordered"
  THEN
    ttOrder.Carrier = "TBD".
END PROCEDURE.
```

The event handler takes the dataset as an input parameter by reference. In the event handler code, the current record is already selected in the table.

See also

'ProDataSet events' in *ABL Reference*

For details, see the following topics:

- Associate a dataset object with an event handler
- Dataset FILL events
- Temp-table buffer FILL events
- Dataset change events
- Other dataset events

Associate a dataset object with an event handler

After you define the internal procedures for dataset event handlers, you associate each handler and a named event with a particular dataset, temp-table buffer, or query object, in the code that fills or updates the dataset. In the code you first load the external procedure file that contains the internal procedures as a persistent procedure into the AVM. Then, you associate each internal procedure you want to use as a dataset event handler for a named event with its dataset object. The named event can be `BEFORE-FILL`, `AFTER-FILL`, `BEFORE-ROW-FILL`, `AFTER-ROW-FILL`, `ROW-CREATE`, `ROW-UPDATE`, `ROW-DELETE`, `FIND-FAILED`, `OFF-END`, or `SYNCHRONIZE`.

You use the 'SET-CALLBACK-PROCEDURE( ) method' in *ABL Reference* to establish this association.

The following is an example of server-side initialization code that fills the dataset at runtime. A handle for the procedure that holds the dataset event handlers is defined. The external procedure is run persistently and sets the handle to the procedure. The `ttOrder` temp-table is associated with the `AFTER-ROW-FILL` event handler, `AdjustCarrier`.

```
/* AppCode.p */

      /* define the procedure handle */
      VAR HANDLE hEventHandlers.
      . . .
      /* load the persistent procedure and set the procedure handle*/
      RUN dsOrderOrderLine_EventHandlers.p PERSISTENT SET hEventHandlers.
      . . .
      /* associate the event handler with after-row-fill event */
      BUFFER ttOrder:SET-CALLBACK-PROCEDURE(
      "AFTER-ROW-FILL","AdjustCarrier",hEventHandlers).
```

When you no longer need the persistent procedure, you should always delete it using the 'DELETE PROCEDURE statement' in *ABL Reference*.

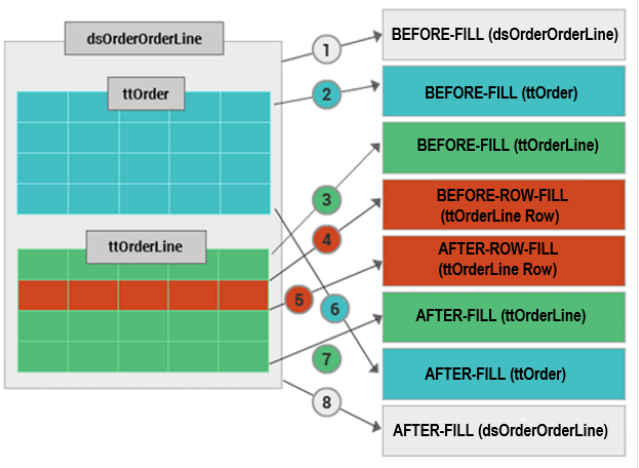
See also

'Event procedures for ProDataSets' in *Use ProDataSets*

Dataset FILL events

There are two types of FILL events. One event type is triggered at the dataset level. The other event type is triggered at the level of the temp-table buffer of a dataset. During the FILL operation for a dataset or a temp-table buffer, events are automatically triggered in the AVM. You can write an event handler to execute custom code when a particular FILL event triggers.

Fill events at the dataset level include `BEFORE-FILL` and `AFTER-FILL`. They are shown in gray in the following illustration. Fill events at the temp-table buffer of a dataset level include `BEFORE-FILL`, `AFTER-FILL`, `BEFORE-ROW-FILL` and `AFTER-ROW-FILL`. These events are shown in blue, green, and red in the illustration, and are discussed in the next topic.



The dataset level FILL events, `BEFORE-FILL` and `AFTER-FILL` are summarized in the table.

Table 1: Dataset FILL events

Event	When it occurs	Examples
BEFORE-FILL	Prior to the retrieval of the first record in a FILL operation; executes once per FILL operation on a dataset	• Load data from a non database source. • Attach data-sources.
AFTER-FILL	After the retrieval of the last record in a FILL operation; executes once per FILL operation on a dataset	• Reject the FILL operation. • Detach data-sources.

Alternatively, you do not need to use an event to perform these operations. The operations can be performed before or after the call to FILL in your code.

Note: Fill events are not triggered if the 'FILL-MODE attribute' in *ABL Reference* of a dataset temp-table is set to NO-FILL.

BEFORE-FILL on a dataset

In this example, the BEFORE-FILL event handler prepares the top-level query for the FILL operation.

```
PROCEDURE preDataSetFill:
      DEFINE INPUT PARAMETER DATASET FOR dsOrderOrderLine.

      /* Load initial data from a JSON file */
      DATASET dsOrderOrderLine:READ-JSON("FILE","Orders.json","EMPTY").

      END PROCEDURE.
```

In the internal procedure (preDataSetFill), the dataset is loaded with initial data from a JSON file called Orders.json.

AFTER-FILL on a dataset

In this example, after all the tables are filled the AFTER-FILL event handler detaches the data-sources for all temp-tables in the dataset.

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

This procedure uses the NUM-BUFFERS attribute and the GET-BUFFER-HANDLE method to detach all the data-sources in a single procedure call.

See also

'Define FILL events' in *Use ProDataSets*

'FILL events' in *ABL Reference*

Temp-table buffer FILL events

The following table summarizes the events that are triggered during a FILL operation for a temp-table buffer:

Event	When it occurs	Examples
BEFORE-FILL	For each parent temp-table, BEFORE-FILL executes once before the retrieval of the first record of the temp-table. For each child temp-table, BEFORE-FILL executes once after its parent record is retrieved, but before the child records are retrieved.	• Prepare a query for a parent table inside the FILL event. • Cancel a FILL of a child table. • Populate the table with data from a non-database source. • Attach a data-source.
AFTER-FILL	For each parent temp-table, AFTER-FILL executes once after the retrieval of the last record of the temp-table For each child temp-table, AFTER-FILL executes once after all child records for a parent are retrieved.	• Reject the FILL operation (Note: To abort a FILL operation you return error from the event handling procedure). • Detach a data-source. • Perform field calculations in the parent or perform other changes to data values.
BEFORE-ROW-FILL	BEFORE-ROW-FILL fires prior to the retrieval of each row during the population of the specified temp-table.	• Decide whether or not to create a record.
AFTER-ROW-FILL	AFTER-ROW-FILL fires after the retrieval of each row during the population of the specified temp-table.	• Calculate field values. • Reject a row by deleting it.

BEFORE-FILL on a temp-table buffer

In this example, a BEFORE-FILL event handler for the ttOrder parent temp-table buffer is used to attach data-sources to the dataset.

```
PROCEDURE preOrderFill:
      DEFINE INPUT PARAMETER DATASET FOR dsOrderOrderLine.
      BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).
      BUFFER ttOrderLine:ATTACH-DATA-SOURCE(DATA-SOURCE srcOline:HANDLE).
      END PROCEDURE. /* preOrderFill */
```

Note: If you want to pre-populate the temp-table from a non-database source (such as JSON, flat file, or spreadsheet) use the BEFORE-FILL event to execute code that populates the temp-table programmatically.

AFTER-FILL on a temp-table buffer

In this example, an AFTER-FILL event handler for the ttOrderLine temp-table buffer updates ttOrder.OrderTotal after all of the ttOrderLine records for the current ttOrder are added to the dataset.

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

BEFORE-ROW-FILL on a temp-table buffer

In this example, a BEFORE-ROW-FILL event handler for the ttOrderLine temp-table buffer is used to filter child records as they are populated during the FILL process.

```
PROCEDURE preOrderLineRowFill:
      DEFINE INPUT PARAMETER DATASET FOR dsOrderOrderLine.
      IF OrderLine.Discount > 10 THEN
      RETURN NO-APPLY.
      END PROCEDURE.
```

AFTER-ROW-FILL on a temp-table buffer

In this example, after each ttItem row is populated, an AFTER-ROW-FILL event handler for the ttItem temp-table buffer is used to capitalize the ItemName field in the record buffer.

```
PROCEDURE postItemRowFill:
      DEFINE INPUT PARAMETER DATASET FOR dsOrderOrderLine.
      ASSIGN ttItem.ItemName = CAPS(ttItem.ItemName).
      END PROCEDURE.
```

See also

'FILL events' in *ABL Reference*

'Define FILL events' in *Use ProDataSets*

Dataset change events

Dataset change events can occur in the server or in the client and are handled where they occur.

A change event is triggered when a record in a temp-table of a dataset is created, deleted, or updated. Change events are associated with a particular temp-table or temp-table buffer.

The following table lists the events that are triggered during a create, delete, or update operation:

Event	When it occurs	Examples
ROW-CREATE	Immediately after the record is created, a ROW-CREATE event occurs. The current buffer for the temp-table is available and contains	• Calculate initial values. • Make changes to other records based on the creation of the new record.

Event	When it occurs	Examples
	initial values as defined in the temp-table definition.	• Reject the creation by deleting the new temp-table record.
ROW-DELETE	Immediately before the temp-table record is deleted, a ROW-DELETE event occurs. Since the record has not yet been deleted, the record remains in the temp-table buffer and the code can access its values.	• Raise an error. • Cancel the delete. • Make adjustments to other records based on the delete.
ROW-UPDATE	Immediately before the temp-table record is updated, a ROW-UPDATE event occurs.	Log all changes to a table.

Use a change event handler in the client

In the following example, a ROW-UPDATE event handler for the ttOrderLine temp-table buffer is used to limit the maximum discount an order line can have to 20%. This event handler, named DiscountTooHigh, is an internal procedure of the external procedure library file that is used in the client.

```
/* external procedure for event handlers */
       BLOCK-LEVEL ON ERROR UNDO, THROW.

       {include/dsOrderOrderLine.i}

       PROCEDURE DiscountTooHigh:
       DEFINE INPUT PARAMETER DATASET FOR dsOrderOrderLine.

       IF ttOrderLine.Discount > 20
       THEN
       ttOrderLine.Discount = 20.
       END PROCEDURE.
```

The following is the client code that updates the dataset. To use the event handler, the client code first runs the external procedure as PERSISTENT and then associates the ROW-UPDATE event handler, DiscountTooHigh, with the ttOrderLine temp-table buffer. When the client updates data in the ttOrderLine record, it does so by calling the SetDiscount internal procedure. In the SetDiscount procedure, a transaction block is required around the code that updates the temp-table. As with database transactions, scope ABL transactions to be as small as possible for good performance. When SetDiscount updates the record, the DiscountTooHigh event handler runs.

```
/* client code to use the event handler */
       /* Define the procedure handle variable */
       VAR HANDLE hDSEventHandlers.
       . . .
       /* Run the persistent procedure and set the handle variable */
       RUN dsOrderOrderLine_EventHandlers.p PERSISTENT SET hDSEventHandlers.
       . . .
       /* Associate the event handler with the temp-table buffer */
       BUFFER
ttOrderLine:SET-CALLBACK-PROCEDURE("ROW-UPDATE","DiscountTooHigh",hDSEventHandlers).
       . . .
       /* Procedure to update the temp-table that calls SetDiscount */
       . . .
       DELETE PROCEDURE hDSEventHandlers.
       RETURN.


       PROCEDURE SetDiscount:
```

```
        DEFINE INPUT PARAMETER pDiscount AS INTEGER NO-UNDO.
        . . .
        TEMP-TABLE ttOrderLine:TRACKING-CHANGES = TRUE.

        ttOrderLine.Discount = pDiscount.

        TEMP-TABLE ttOrderLine:TRACKING-CHANGES = FALSE.
        . . .
        END PROCEDURE.
```

See also

'ProDataSet change events' in *Use ProDataSets*

'Row-level events' in *ABL Reference*

Other dataset events

There are two additional events that can be triggered for a dataset in the client or the server:

- 'FIND-FAILED event' in *ABL Reference*
- 'SYNCHRONIZE event' in *ABL Reference*

See also

'ProDataSet buffer FIND-FAILED event' in *Use ProDataSets*

'SYNCHRONIZE event for a ProDataSet buffer' in *Use ProDataSets*

7

Batch data with datasets

You can also use an event handler to implement batching of data from the server. Application performance is affected by the amount of data sent over the network. If too large a number (batch) of records are sent at once, performance may be impacted. If too few records are sent, more round trips are needed to retrieve the required data.

Batching data means you retrieve a subset of data from the server in separate server calls until you retrieved the required data. The batching capabilities of datasets allow you to tune for application performance. Depending on your application, user requirements, and your network, you can customize the dataset batch size to best meet your needs.

A recommended way to implement batching is to use the `OFF-END` query event in the client code. For example, you may want to view the first 10 records in a table that are retrieved from the server, and when **Next** is clicked, the next 10 records are retrieved from the server and displayed. Retrieving small batches of data allows you to see data immediately without having to wait for the entire table to be returned from the server. As your query iterates through the retrieved data, it detects the `QUERY-OFF-END` condition when it reaches the end of the last retrieved record. The `QUERY-OFF-END` condition triggers the `OFF-END` event. When the `OFF-END` query event is triggered the `OFF-END` event handler attempts to retrieves the next batch of data from the server. If there is no more data to retrieve the query ends, otherwise the query continues.

To implement data batching, you add code in the client and the server as follows:

- In the client, you write code to retrieve the initial batch of data, specifying the batch size, and the `ROWID` for starting the batch.
- In the server, you write code to set the batch size for data retrieval, set the starting `ROWID`, retrieve the data, and return the `ROWID` of the next record to be retrieved.
- In the client, you write code to associate the query with the `OFF-END` query event. Then, in the code, your query checks if it has reached the end of the iteration. If the `QUERY-OFF-END` condition is TRUE, the `OFF-END` event is triggered. You write the `OFF-END` handler procedure to retrieve the next batch of data, specifying the starting `ROWID`, and reset the condition.

See also

'Batch Data with ProDataSets' in *Use ProDataSets*

'OFF-END event' in *ABL Reference*

For details, see the following topics:

- Client code to retrieve first batch of data
- Server code to process batched data
- Manage batched data in the client

Client code to retrieve first batch of data

In this example, the client defines variables to hold the `ROWID` that specify where the last batch ended and an integer to specify the size of the batch to be retrieved from the server. The server-side procedure, `fetchOrders()`, is called to retrieve 10 order records that are added to the `dsOrder` dataset's temp-table, `ttOrder`. The first time this procedure is called, `rowRestart` is unknown. The server-side code uses this value to begin the FILL operation starting with the first record in the database table. The `fetchOrders()` procedure returns the `ROWID` of the next record to retrieve (where the batch ended). Note this code does not yet show the event handling in the client that is used to handle the `QUERY-OFF-END` condition that triggers another retrieval from the server.

```
/* client-side code */
{include/dsOrder.i}

VAR ROWID rowRestart.
VAR INT batchSize = 10.

DEFINE QUERY qryOrder FOR ttOrder.

/* Fetch the first batch of data, passing the rowid from which to
start fetching data. This comes from the NEXT-ROWID of the DATASOURCE
on the server after the previous fetch. Pass in ? and the
server-side code always start reading from the beginning of the
record-set which matches the query's criteria. */

rowRestart = ?.
RUN fetchOrders ON hServer (batchSize,INPUT-OUTPUT rowRestart,OUTPUT DATASET dsOrder).
OPEN QUERY qryOrder FOR EACH ttOrder BY ttOrder.OrderNum.
QUERY qryOrder:GET-NEXT().
DO WHILE NOT QUERY qryOrder:QUERY-OFF-END:
  MESSAGE "OrderNum:" ttOrder.OrderNum " Status: "
    ttOrder.OrderStatus.
 QUERY qryOrder:GET-NEXT().
END.
```

Server code to process batched data

The steps to implement batching in the server include:

1. Set the 'BATCH-SIZE attribute' in *ABL Reference* for the temp-table.
2. Set the 'RESTART-ROWID attribute' in *ABL Reference* for the data-source.

3. FILL the temp-table using the 'FILL( ) method' in *ABL Reference*.
4. Return the 'NEXT-ROWID attribute' in *ABL Reference* and the dataset to the client.

By default, the FILL operation on a dataset retrieves all available records for all temp-tables in a single batch. To change this default behavior, you assign a value to the BATCH-SIZE for one or more temp-tables. In addition, ensure that the server code uses the NEXT-ROWID and RESTART-ROWID of the data-source for a temp-table to specify where in the batch the next retrieval should begin.

The BATCH-SIZE attribute

BATCH-SIZE is the maximum number of temp-table rows to be retrieved in each FILL operation. The default value of the BATCH-SIZE attribute is 0 (zero). This instructs the dataset to retrieve all rows that satisfy the associated query.

To change the value of the BATCH-SIZE attribute, set the BATCH-SIZE attribute for one or more temp-tables.

How BATCH-SIZE works with table relations

Consider the situation where you have two temp-tables in a dataset and you defined one temp-table as a child of the other. If you specify a batch size for the child table, the batch size is used for each set of child records that are retrieved for each parent record. The AVM restarts the BATCH-SIZE counter for each parent record as you scroll through the parent records.

The NEXT-ROWID attribute

NEXT-ROWID is an attribute of the data-source of the temp-table. After a FILL operation, this attribute contains the ROWID value for the next record of a database table when the FILL does not retrieve all the records in the database table. The server-side code must obtain this value after a FILL operation and return it to the client.

The RESTART-ROWID attribute

RESTART-ROWID is an attribute of the data-source of a temp-table. Before a FILL operation, this attribute must be set so that the batch retrieval from the database table begins from a specific row.

Server-side code for batching of data

The following example is the fetchOrders.p code used to retrieve a batch of data. If rowRestart is unknown, the FILL begins with the beginning of the database table. Otherwise, the RESTART-ROWID attribute is set for the data-source of the temp-table using the rowRestart parameter. The BATCH-SIZE attribute is set for the temp-table using the batchSize parameter. Once these attributes are set, the FILL operation is called and the appropriate number of records are added to the temp-table. If the FILL operation encounters the end of the database table, then it sets the temp-table's LAST-BATCH attribute to TRUE. Finally, the code must update the rowRestart value to the NEXT-ROWID of the data-source of the temp-table, as this value must be returned to the client.

```
/* fetchOrders.p */

{include/dsOrder.i}

DEFINE DATA-SOURCE srcOrder FOR Order.
DEFINE INPUT PARAMETER batchSize AS INTEGER NO-UNDO.
DEFINE INPUT-OUTPUT PARAMETER rowRestart AS ROWID NO-UNDO.
DEFINE OUTPUT PARAMETER DATASET FOR dsOrder.

BUFFER ttOrder:ATTACH-DATA-SOURCE(DATA-SOURCE srcOrder:HANDLE).

/* Set the RESTART-ROWID using the rowid passed back from server. */
IF rowRestart NE ? THEN
 DATA-SOURCE srcOrder:RESTART-ROWID = rowRestart.

/* Set the batch size. */
BUFFER ttOrder:BATCH-SIZE = batchSize.
```

```
/* Set FILL-MODE to EMPTY. */
IF THIS-PROCEDURE:PERSISTENT THEN
 BUFFER ttOrder:FILL-MODE = "EMPTY".

/* Fill the dataset using the settings applied above. */
DATASET dsOrder:FILL().

/* Get the NEXT-ROWID from the DATA-SOURCE to be passed back here
   again for the fetch of a subsequent batch. */
rowRestart = DATA-SOURCE srcOrder:NEXT-ROWID.
BUFFER ttOrder:DETACH-DATA-SOURCE().

RETURN.
```

Manage batched data in the client

To implement data batching in the client code, the client calls the server-side procedure to retrieve an initial batch of records, specifying the batch size and the `rowRestart` value to indicate in which row the batch should resume. Initially, `rowRestart` is undefined, meaning that the batching begins with the first record in the database table.

```
RUN fetchOrders ON hServer (batchSize, INPUT-OUTPUT rowRestart, OUTPUT DATASET dsOrder).
```

To manage batching, the client iterates through the records retrieved in the dataset table and when the query reaches the 'QUERY-OFF-END condition' in *ABL Reference*, the client-side 'OFF-END event' in *ABL Reference* handler executes to determine if more data can be retrieved. If more data can be retrieved, the client again calls `fetchOrders.p` appending data to the existing `dsOrder` dataset. This situation is where `rowRestart` is important. This value is returned from a previous call to `fetchOrders.p` and must be used for the next call to the server. The event handler also checks the 'LAST-BATCH attribute' in *ABL Reference* of the temp-table to determine whether there is more data to retrieve from the server. If there are no more records to retrieve from the database table, the FILL operation that executes in the server automatically sets this attribute in the dataset returned to the client.

Complete client code for managing batching

The following is the complete client code for managing batching. Associate the `OFF-END` event handler for the query with the `offEnd` internal procedure. Notice the internal procedure is in the same file and an external procedure is not used. During the iteration, when the `QUERY-OFF-END` condition occurs, this event handler executes. Notice the input for this handler is the dataset.

In the event handler code, check the `LAST-BATCH` attribute of the `ttOrder` temp-table. If the attribute is not TRUE, call `fetchOrders()` again, passing the value of `rowRestart` that was returned from the previous call. Specify `APPEND` for the dataset so that the new batch of records is added to the existing dataset. After the call returns from `fetchOrders()` the code must return `NO-APPLY`. This ensures that the `QUERY-OFF-END` condition is set back to FALSE and the retrieval of records continues until all records are retrieved from the server.

```
{include/dsOrder.i}

      DEFINE QUERY qryOrder FOR ttOrder.
      DEFINE VARIABLE rowRestart AS ROWID NO-UNDO.
      DEFINE VARIABLE batchSize AS INTEGER NO-UNDO INITIAL 10.

      /* Set the internal event handler procedure to process the
      OFF-END event indicating the query needs a new batch of records. */
      QUERY qryOrder:SET-CALLBACK-PROCEDURE("OFF-END","offEnd").
```

```
      /* Fetch the first batch of data, passing the rowid from which to start
      fetching data. This comes from the NEXT-ROWID of the DATA-SOURCE on the
      server after the previous fetch. Pass in ? and the server-side code
      always starts reading from the beginning of the record-set which matches
      the query's criteria. */

      rowRestart = ?.
      RUN fetchOrders ON hServer (batchSize, INPUT-OUTPUT rowRestart, OUTPUT DATASET
dsOrder APPEND).
      OPEN QUERY qryOrder FOR EACH ttOrder BY ttOrder.OrderNum.
      QUERY qryOrder:GET-NEXT().

      DO WHILE NOT QUERY qryOrder:QUERY-OFF-END:
      MESSAGE "OrderNum:" ttOrder.OrderNum " Status: " ttOrder.OrderStatus.
      QUERY qryOrder:GET-NEXT().
      END.
```

```
PROCEDURE offEnd:

      DEFINE INPUT PARAMETER DATASET FOR dsOrder.

      /* Built into the buffer is whether or not it is already on
      the last batch of data that meets the query's criteria. */

      IF NOT BUFFER ttOrder:LAST-BATCH THEN DO:
      MESSAGE "Getting more data from the server.......".
     RUN fetchOrders (batchSize, INPUT-OUTPUT rowRestart, OUTPUT DATASET dsOrder APPEND
 ).

      /* Using RETURN NO-APPLY is required here to cancel the default effects
      of the OFF-END event on the query, which is to set the OFF-END attribute on the
      query to TRUE, which would prevent further batching. This allows the iteration
to
      proceed to process the OFF-END event and reopen the query, automatically
      repositioning to the next record that was just fetched from the server.*/
      RETURN NO-APPLY.
      END.
      ELSE
      MESSAGE "No more data.......".

      END PROCEDURE.
```

See also

'Buffer BATCH-SIZE and LAST-BATCH attributes' in *Use ProDataSets*

'Set up an event handler for the OFF-END query event' in *Use ProDataSets*

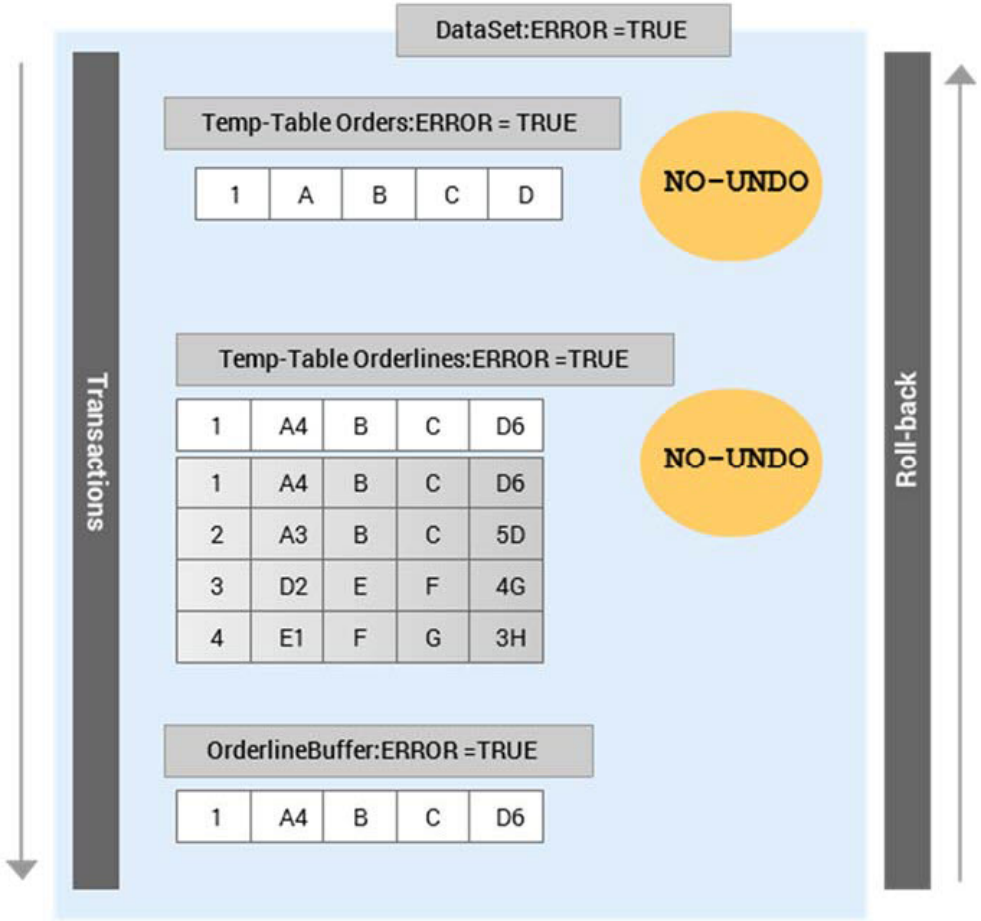
8

Handle errors with datasets

When you call the 'SAVE-ROW-CHANGES( ) method' in *ABL Reference*, by default, the transaction spans just one record at a time. If a rollback occurs, errors are returned to the client as expected.

However, if your transaction spans more than one execution of the `SAVE-ROW-CHANGES` method and you roll back the transaction, error information at the table level is undone. You cannot access the error information after the rollback.

To save error information on rollback, define the temp-tables in the dataset as `NO-UNDO`. Using `NO-UNDO` allows the error information on the temp-tables to not be lost.



For details, see the following topics:

- Types of dataset errors
- Handle system-generated errors
- Handle data-related errors
- Provide error information to the client

Types of dataset errors

Dataset objects contain attributes that enable you to detect and handle dataset-related errors. Dataset related errors occur in the server and are detected and handled in both the server and the client.

Errors could occur for a variety of reasons—some errors are system-generated, while others are in the data that you write application code to detect.

System-generated errors could include:

- Adding an `Order` record with an order number that already exists in the database.
- Modifying an `Order` record where another user has already modified it and the retrieved data in the client does not match what was in the database.

Data-related errors that you write application code to detect could include:

- Adding a `PurchaseOrder` with a supplier ID number that is not valid.
- Making a change to the `POLine`'s discount that is larger than the limit.

System-generated errors set the 'ERROR attribute' in *ABL Reference*. In addition, you can write application code to set and get the 'REJECTED attribute' in *ABL Reference* and ERROR attribute to detect and handle data-related errors. You can also write application code to set the 'ERROR-STRING attribute' in *ABL Reference* to provide information about errors that occurred.

See also

'Set and use ERROR, ERROR-STRING, and REJECTED' in *Use ProDataSets*

Handle system-generated errors

When you send a change dataset to the server for updating, system-errors could occur in the server. You need a way to locate and analyze system-generated errors when the dataset is returned to the client. The AVM sets the `ERROR` attribute when it detects an error during the call to `SAVE-ROW-CHANGES` or during a `FILL` operation.

When a system-generated error occurs, the `ERROR` attribute is set at three levels:

- For the particular temp-table buffer (row) where the error was detected; it is set in the before-table.
- For the particular temp-table where the error occurred; it is set for the before-table.
- For the dataset as a whole.

In the client, you write code to check the `ERROR` attribute at these levels to locate the row where the error occurred. First you check at the dataset level to determine if an error occurred. If an error did occur, then you typically check each before-table to determine which temp-table has an error. Then you iterate through the before-table rows to determine the particular row where the error occurred.

Enable system-generated ERROR setting in the server

If you want to allow the AVM to set the `ERROR` attribute in the server, use the `NO-ERROR` syntax when you call the `SAVE-ROW-CHANGES` or `FILL` methods. Otherwise if an error occurs, the server-side procedure fails and no `ERROR` attributes are set.

Detect system-generated errors on the server

The following is an example of what you need to add to your server-side code to handle system-generated errors. Notice `NO-ERROR` is specified when calling `SAVE-ROW-CHANGES()`. If an error occurs when applying changes to the database, the AVM automatically sets the `ERROR` attribute of the before-table buffer row to TRUE. In addition, the `ERROR` attributes of the before-table and dataset are also set to TRUE.

```
. . .
hQuery:GET-FIRST.
DO WHILE hBeforeBuffer:AVAILABLE:
  hBeforeBuffer:SAVE-ROW-CHANGES() NO-ERROR.
  IF hBeforeBuffer:ERROR = TRUE THEN
  DO:
    /* handle the error */
  END.
  hQuery:GET-NEXT().

END.
. . .
```

Detect the ERROR in the client

Assume SAVE-ROW-CHANGES is called in the server for rows of the Order before-table and its child OrderLine before-table rows. During the execution of SAVE-ROW-CHANGES, an error occurs and is set on the dataset, the relevant before-table, and the row that generated the error. For example, if the before-table value does not match what is in the database because another client has updated that particular row, SAVE-ROW-CHANGES fails and the ERROR attribute is set.

In the client, you examine the temp-table rows. Follow these steps to write client code to handle these errors in the returned change dataset:

1. Check if there are any errors in the dataset as a whole. If not, then you are done. It is not necessary to search the temp-tables for errors.
2. If errors exist, then check each temp-table for errors.
3. If you find any temp-tables with errors, examine each row to locate individual errors.

Detect errors at different levels of the dataset

To detect whether an error has occurred on the dataset as a whole, check the ERROR attribute on the dataset handle as follows:

```
IF hChangeDataSet:ERROR THEN DO:
  /* Perform ERROR handling */
END.
```

To detect whether an error has occurred in the temp-table of a dataset, check the ERROR attribute on the before-table handle as follows:

```
IF hBeforeBuffer:ERROR THEN DO:
   /* Perform ERROR handling */
END.
```

To detect whether an error has occurred in an individual row, use the ERROR attribute on the temp-table buffer handle as follows:

```
IF hBeforeBuffer:ERROR THEN DO:
  /* Perform ERROR handling */
END.
```

How rows with errors are merged back into the client's working dataset

Recall that in the client, you call MERGE-CHANGES( ) to merge the change dataset returned from the server to incorporate the changes back into the client's dataset. If any row has its ERROR attribute set to TRUE, the row is not merged back into the dataset of the client. MERGE-CHANGES behaves as follows for rows that have their ERROR attribute set:

- If the row was created in the dataset of the client, the row is deleted.
- If the row was deleted, the row is added back.
- If the row was modified, the modifications are rolled back.

Handle data-related errors

You may want to detect data-related errors in your server-side application code before an attempt is made to update the database. For example, if a field was changed in the client to a value that is outside the acceptable range of values, you can add code in the server-side application to detect the error. In such a case, you set the 'REJECTED attribute' in *ABL Reference* for a row and set the 'ERROR attribute' in *ABL Reference* at the temp-table and dataset levels. When you set the `REJECTED` attribute in the server for a row, you always set it for the before-table buffer.

Note: You can use either the `ERROR` or `REJECTED` attribute for a row when a data related error is detected. Using `REJECTED` makes it easier for the client part of the application to differentiate between a system-generated error and a data-related error.

Similar to how `MERGE-CHANGES` handles rows with the `ERROR` attribute set, `MERGE-CHANGES` does not merge rows with the `REJECTED` attribute set.

Set REJECTED for related records

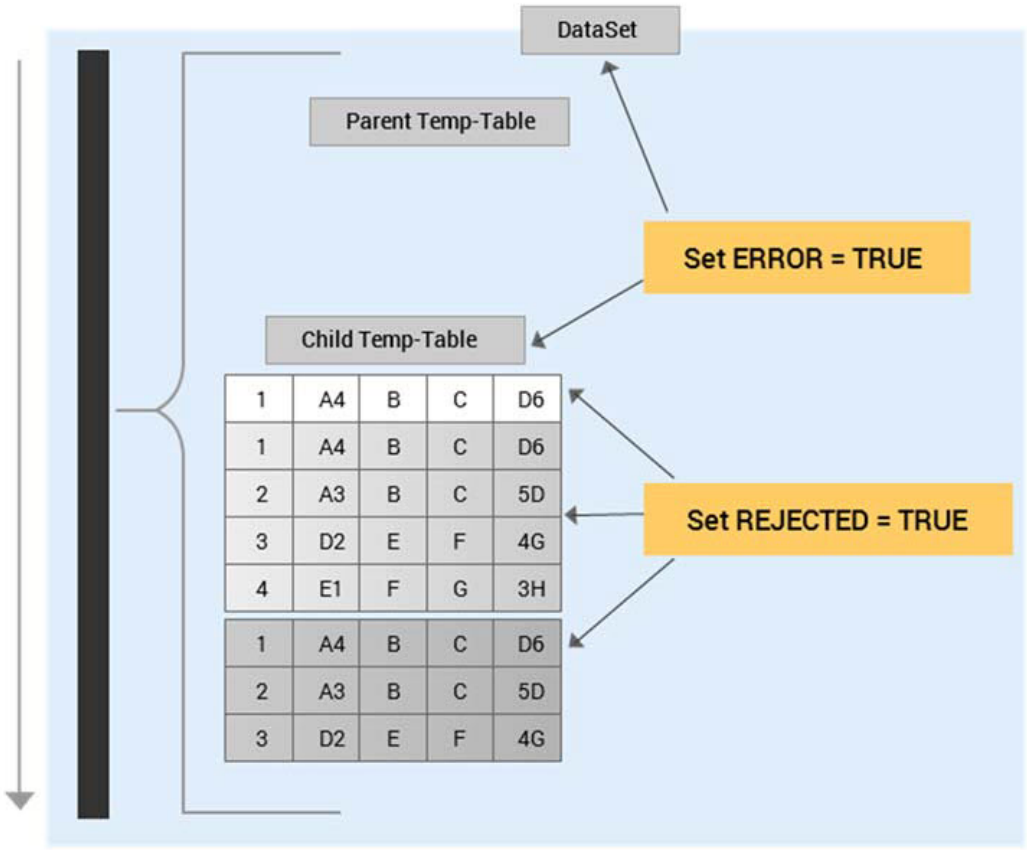
Depending on your business requirements, when you detect an error in a row you may also want to reject all related child rows and rollback all related changes. For example if an error occurs on a `ttOrder` row, you would:

- Set the `REJECTED` attribute to TRUE for all `ttOrderLine` rows in the `ttOrder`.
- Set the `REJECTED` attribute to TRUE for the `ttOrder` row.

In addition, you should set the `ERROR` attribute for the `ttOrderLine` and `ttOrder` temp-tables, and also the `dsOrderOrderLine` dataset.

The following illustration shows the `REJECTED` attribute set on child temp-table rows and the `ERROR` attribute set on the temp-table and dataset.

Set the ERROR and REJECTED attributes in the server



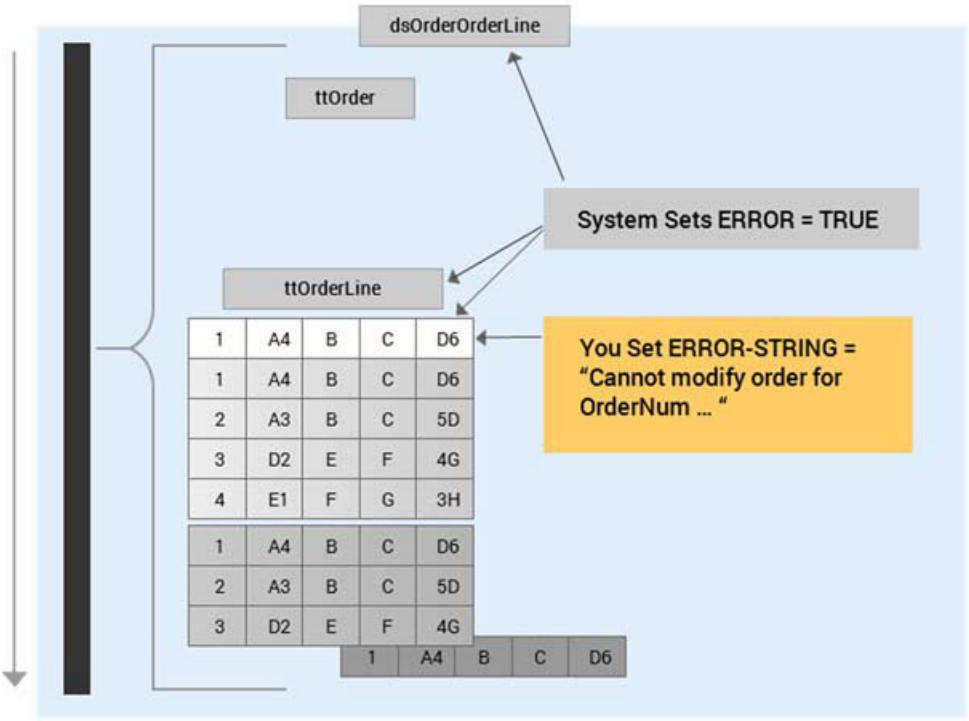
The following is an example in which the application sets the REJECTED attribute to TRUE for the before-table buffer row, if an error occurs when validating the data. The application also sets theERROR attribute to TRUE for the before-table and dataset.

```
. . .
hQuery:GET-FIRST.
DO WHILE hBeforeBuffer:AVAILABLE:
  /* procedure to validate the data */
  RUN ValidateData(hBuffer, OUTPUT lValid).
  IF lValid
  THEN
    hBeforeBuffer:SAVE-ROW-CHANGES() NO-ERROR.
  ELSE
    ASSIGN
      hBeforeBuffer:REJECTED = TRUE
      hBeforeTable:ERROR = TRUE
      hChangeDataset:ERROR = TRUE.

  hQuery:GET-NEXT.
END.
. . .
```

Provide error information to the client

When an error occurs on the server, you can return useful information to the client using the 'ERROR-STRING attribute' in *ABL Reference*. This attribute can be set at the row level for a system-related or data-related type of error. It can also be set at the temp-table level to provide error summary information for the temp-table.



Provide before-table and after-table row values for the error

The `ERROR-STRING` attribute is normally set when an error is encountered during `SAVE-ROW-CHANGES`. It is a best practice to provide contextual information about the error from the data in the corresponding after-table row. Format error information so that the client code can present the `ERROR-STRING` directly without modification.

ABL provides two attribute that can be used to match rows in the before-tables and after-tables:

- Before-table has an attribute called the 'AFTER-ROWID attribute' in *ABL Reference*.
- After-table has an attribute called the 'BEFORE-ROWID attribute' in *ABL Reference*.

These attributes can be used to locate a corresponding record in the other table using the 'FIND-BY-ROWID( ) method' in *ABL Reference*.

For example, if an error occurred in `SAVE-ROW-CHANGES` for a particular before-table buffer, you can obtain the row values in the corresponding after-table row by setting the buffer (`hBuffer`) as follows:

```
hBuffer:FIND-BY-ROWID(hBeforeBuffer:AFTER-ROWID).
```

The following example shows how to set the ERROR-STRING after encountering an error during SAVE-ROW-CHANGES.

- The ERROR attribute is set to TRUE by the system if an error occurs.
- Using the after-table, find the corresponding record using the FIND-BY-ROWID() method and the AFTER-ROWID for the before-table record.
- Set the ERROR-STRING attribute with information about the record in the after-table buffer.

```
hQuery:GET-FIRST.

       DO WHILE hQuery:AVAILABLE:
       hBeforeBuffer:SAVE-ROW-CHANGES() NO-ERROR.

       /* there was an error with save-row-changes;
       handle row-modified */
       IF hBeforeBuffer:ERROR AND hBeforeBuffer:ROW-STATE = ROW-MODIFIED
       THEN DO:
       hBuffer:FIND-BY-ROWID(hBeforeBuffer:AFTER-ROWID).
       hBuffer:ERROR-STRING = "Cannot modify order for OrderNum = " +
       STRING(hBuffer::OrderNum).
       END.
       hQuery:GET-NEXT.
       END.
```

Retrieve the ERROR-STRING value in the client

To retrieve ERROR-STRING values in your client code, you first check if an ERROR was set at the dataset level. If it was set, iterate through the change dataset's after-table to find specific errors. For each error found, you can display and/or process the error messages found in ERROR-STRING. Recall that attributes set in the server for the before-table and before-table-buffer are automatically set for the corresponding after-table and after-table buffer.

This code example shows the calling procedure's code on the client.

```
IF DATASET dsOrderOrderLine:ERROR = TRUE THEN
       DO:
       FOR EACH ttOrderLine:
       IF BUFFER ttOrderLine:ERROR = TRUE THEN
       DO:
       MESSAGE BUFFER ttOrderLine:ERROR-STRING.
       END.
       END.
       END.
```

Next steps

This completes the learning track for datasets. For more detailed information on datasets see the 'Use ProDataSets guide' in *Use ProDataSets*. For additional developer learning paths see the Developer landing page.