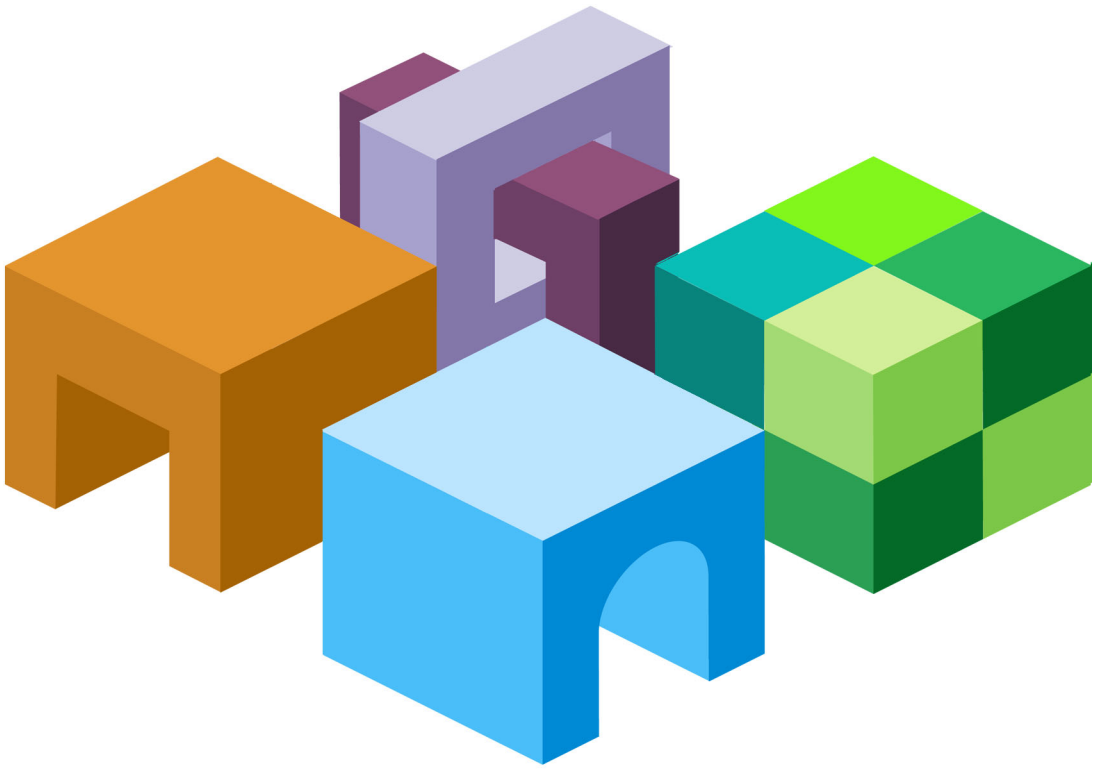


# HYPERION® SQR® PRODUCTION REPORTING

*RELEASE 11.1.1*

## DEVELOPER'S GUIDE

VOLUME 3: ACCESSING DATA WITH SQR PRODUCTION REPORTING DDO

ORACLE®
ENTERPRISE PERFORMANCE MANAGEMENT SYSTEM

Production Reporting Developer's Guide, 11.1.1



Authors: EPM Information Development Team

# Contents


**Part I. Understanding Production Reporting** . . . . . 9

**Chapter 1. Production Reporting Basics** . . . . . 11

About Production Reporting DDO . . . . . 11

Production Reporting DDO Software Development Kit . . . . . 11

Advantages of Production Reporting DDO Interfaces . . . . . 12

Production Reporting DDO Architecture . . . . . 12

Establishing a Data Source Connection . . . . . 13

Supporting Transactions . . . . . 14

Obtaining Metadata . . . . . 14

Obtaining Data . . . . . 15

Processing Results . . . . . 15

Implementing Production Reporting DDO Interfaces with Drivers . . . . . 16

Data Access Strategies . . . . . 17

**Chapter 2. Creating Production Reporting DDO Applications** . . . . . 19

About the Production Reporting DDO API . . . . . 19

A Simple Code Example . . . . . 19

Managing Data Sources . . . . . 20

Registry Files . . . . . 21

The DataSourceManagerAdmin Class . . . . . 21

Defining a New Data Source . . . . . 22

Establishing a Connection . . . . . 25

Processing Logon Properties . . . . . 25

Discovering Capabilities . . . . . 28

Checking for a Capability . . . . . 29

Obtaining Metadata . . . . . 30

Obtaining Schema Information . . . . . 31

Listing Data Objects in a Schema . . . . . 32

Obtaining Information about Data Objects . . . . . 33

Retrieving Column Metadata . . . . . 34

Obtaining Metadata About Procedures . . . . . 35

Determining the Return Value . . . 36
Requesting Data . . . 38
Retrieving Data with getData . . . 39
Executing Commands . . . 42
Calling Procedures and Processing Call Results . . . 44
Performing Transactions . . . 46
Obtaining Hierarchical and Multidimensional Data . . . 47
Retrieving Multidimensional Data Using a Regular Selector . . . 50
Retrieving Multidimensional Data Using MDSelector . . . 51

**Chapter 3. Writing a Production Reporting DDO Driver** . . . 55
About Writing a Driver . . . 55
Summary of Steps . . . 55
Step 1: Create the Properties, Capabilities, and Message Files . . . 56
Creating the Properties Files . . . 56
Creating the Capabilities Files . . . 56
Creating the Messages File . . . 57
The CSV Interface . . . 58
Step 2: Implement the DataSource Interface . . . 59
Step 3: Implement the Connection Interface . . . 60
Providing Column Metadata . . . 62
Implementing getData . . . 66
Implementing getData with Selector . . . 66
CSVFile Class . . . 67
Step 4: Implementing Rowset . . . 69

**Chapter 4. Programming Considerations** . . . 73
Production Reporting DDO Adapters . . . 73
Driver Organization Tips . . . 73
Registy Editor . . . 74
Messages and Exceptions . . . 75
CSV Example . . . 76
Message Forms . . . 76
Properties and Capabilities . . . 77
Descriptions . . . 78
Localization . . . 78
Hierarchical Structure . . . 78
Internationalization . . . 80
Message Editor . . . 80
Loading Messages . . . 81

Adding Messages . . . 82
Editing Message Contents . . . 82
Property Editor . . . 83
Loading Properties . . . 83
Testing and Debugging Drivers . . . 84
Using the Query Editor to Test and Debug Drivers . . . 84

**Chapter 5. Managing Data Sources** . . . 87
Data Source Specifications . . . 87
Adding a Data Source Specification in the Registry Editor . . . 88
Data Source Descriptions and Templates . . . 88
Hyperion Essbase . . . 89
SAP R/3 and SAP BW . . . 90
Microsoft OLEDB for OLAP . . . 94
Microsoft OLEDB . . . 94
JDBC . . . 95
XML . . . 102
Delimiter Separated Values . . . 103
OMG Corba Sample . . . 103
Microsoft DCOM Sample . . . 104
CSV Sample . . . 104

**Chapter 6. Utilities Package and Common Facilities** . . . 107
About the Utilities Package . . . 107
Common Components of the Utilities Package . . . 107
Naming Scheme . . . 107
Property Resource Bundles . . . 108
Instrumentation . . . 109
Message Facility . . . 110
Message Text . . . 110
Message Property Files . . . 110
Localization Example . . . 111
Services . . . 112
Property Facility . . . 112
DataSource Class . . . 113
Property Descriptions . . . 114
Property Sheets . . . 115
Retrieving Properties and Capabilities . . . 117
Secure Properties and Capabilities . . . 118
Property Auxiliary Services . . . 118

Property Validators . . . . . 119

**Part II. Using Production Reporting DDO to Access Data** . . . . . 121

**Chapter 7. Using Production Reporting DDO to Access SAP R/3 Data** . . . . . 123
Data Access Requirements . . . . . 123
Using the Registry Editor to Make an SAP R/3 Connection . . . . . 123
Using the Query Editor to See the SAP Tree Structure . . . . . 124
Using SQR Production Reporting Studio to Build a Report with a BAPI . . . . . 125
Understanding the Production Reporting Code for SAP R/3 . . . . . 126

**Chapter 8. Using Production Reporting DDO to Access an SAP BW Data Source** . . . . . 131
Accessing the SAP BW OLAP Server . . . . . 131
Supported Platforms . . . . . 131
Copying Files to the /lib Directory . . . . . 132
Adding the SAP BW Data Source to the Registry.properties File . . . . . 132
The Hierarchical Structure of Objects for an SAP BW Data Source . . . . . 133
SAP BW and the Production Reporting Language . . . . . 135
Accessing Dimension Properties . . . . . 135
Specifying Dimension Members . . . . . 136
Specifying the Order in Which to Return Dimension Members . . . . . 137
Restricting the Returned Result Set . . . . . 139
Limiting the Set of Values Used for a Dimension . . . . . 141
Using SAP BW Variables . . . . . 142
Returning a Set of Descendants . . . . . 143
Finding a Dimension's Ancestor . . . . . 144
Defining Calculated Key Figures, Restricted Key Figures, and Calculated Members . . 145
Accessing SAP BW Data from SQR Production Reporting Studio . . . . . 148

**Chapter 9. Using Production Reporting DDO to Access Essbase Cubes** . . . . . 151
Overview of Cubes . . . . . 151
Viewing Cubes . . . . . 152
Using Cube Commands in Production Reporting . . . . . 153
SET-MEMBERS . . . . . 154
SET-GENERATIONS . . . . . 156
SET-LEVELS . . . . . 157
Displaying Report Data . . . . . 159
Measures . . . . . 159
Aliases . . . . . 159
Column Order . . . . . 160

Accessing Cubes: An Example . . . . . 162
The Cube . . . . . 162
The Production Reporting Code Needed to Access the Cube . . . . . 163
An Explanation of the Code . . . . . 165

**Chapter 10. Using Production Reporting DDO to Access MSOLAP Cubes** . . . . . 167
Overview of Cubes . . . . . 167
Viewing Cubes . . . . . 168
Using Cube Commands in Production Reporting . . . . . 169
SET-MEMBERS . . . . . 170
SET-GENERATIONS . . . . . 171
SET-LEVELS . . . . . 173
Displaying Report Data . . . . . 175
Measures . . . . . 175
Aliases . . . . . 175
Column Order . . . . . 176
Accessing Cubes: An Example . . . . . 178
The Cube . . . . . 178
The Production Reporting Code Needed to Access the Cube . . . . . 179
An Explanation of the Code . . . . . 180

**Part III. Appendices** . . . . . 183

**Appendix A. Sample Resource Files** . . . . . 185
SAP DataSource Property Description . . . . . 185
SAP DataSource Messages . . . . . 190

**Appendix B. Using the HTTP-enabled XML DDO Driver** . . . . . 195
Usage . . . . . 195
Define a Registry Entry . . . . . 195
Declare a Connection . . . . . 196
Use Getdata= in the Begin-Execute Section . . . . . 196
An Alternate Method . . . . . 196
Specifying URLs at Runtime . . . . . 197
Accessing XML Files via HTTP Using the Production Reporting DDO Query Editor . . . . 197
Limitation . . . . . 197

**Index** . . . . . 199

Part I

# Understanding Production Reporting

In Understanding Production Reporting:


- Production Reporting Basics
- Creating Production Reporting DDO Applications
- Writing a Production Reporting DDO Driver
- Programming Considerations
- Managing Data Sources
- Utilities Package and Common Facilities

# 1 Production Reporting Basics

**In This Chapter**


About Production Reporting DDO....11
Production Reporting DDO Software Development Kit....11
Advantages of Production Reporting DDO Interfaces....12
Production Reporting DDO Architecture....12
Data Access Strategies....17


## About Production Reporting DDO

Oracle's Hyperion® SQR® Production Reporting DDO defines an open interface for data access, allowing applications to extract data from vastly different data sources. The Production Reporting DDO Software Development Kit provides the technical resources for building drivers with the special interface knowledge to access data sources, such as relational databases using JDBC, delimiter-separated values files (CSV), and XML data sources that can read XML directly from any Web input stream.

## Production Reporting DDO Software Development Kit

The Production Reporting DDO Software Development Kit provides classes that make it easier to write drivers for application business objects using COM or CORBA interfaces. Each driver provides an implementation of the Production Reporting DDO interface for a specific data source. The developer only needs to invoke functionality that is specific to a given data source.

While accessing a data source through an Production Reporting DDO driver, application developers use the Production Reporting DDO Software Development Kit to extend base Production Reporting DDO classes to rapidly implement access to their objects. The base classes provide the necessary infrastructure of describing properties and capabilities, providing localized messages, and default behavior.

To use the Production Reporting DDO Software Development Kit, you need:

- Basic Java programming skills
- Workstation with a Windows platform
- Java programming tool such as Symantec Visual Café, Visual J++, Visual Age for Java, or JBuilder

# Advantages of Production Reporting DDO Interfaces

Production Reporting DDO interfaces and their relationships are similar to those found in JDBC or ADO. Production Reporting DDO is an application layer interface to heterogeneous data sources.Production Reporting DDO drivers provide high-performance access to any data source, including relational and non-relational database systems, custom business objects, and more.

More importantly, unlike JDBC and ADO, Production Reporting DDO uses a rich bi-directional capabilities-and-properties model to enable transparency between applications and drivers. This allows a single application to access heterogeneous data sources without prior knowledge of the capabilities and properties of that data source.

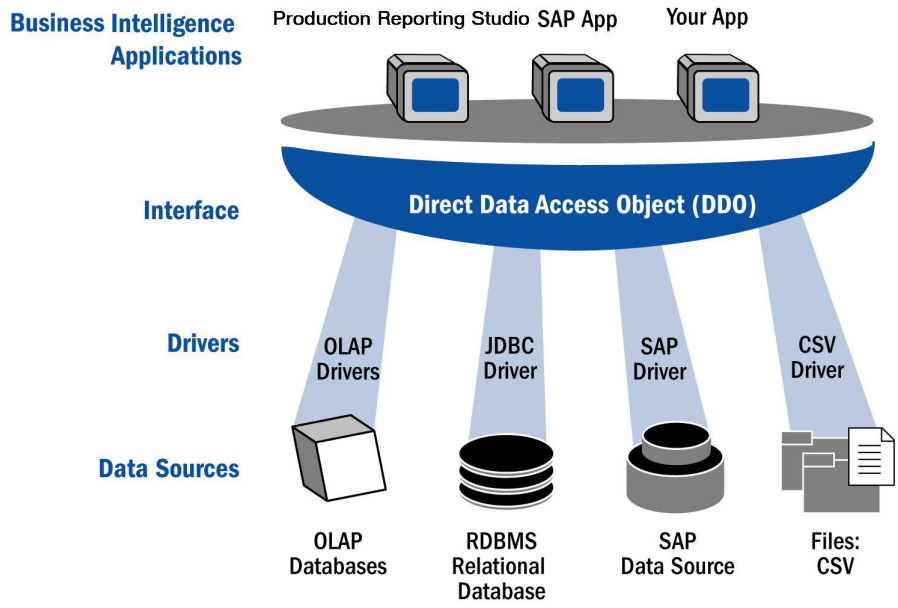
Production Reporting DDO provides these advantages:

- **Application Security**—Embraces Enterprise Resource Planning (ERP) and other multi-tier applications in which real objects in the business world, such as employees or purchase order requisitions, are modeled as business objects and data abstracted from these business objects. Information access through business objects enforces application security and encapsulates business rules.
- **Open System**—Makes no assumptions about applications and data sources. Provides an open interface for query or reporting tools to access both relational and non-relational data sources. Provides access to application business objects, multi-dimensional and hierarchical data, XML data, and arbitrarily complex data.
- **Metadata Access—Provides** rich access to metadata, allowing applications to interactively discover information objects and data source capabilities. Reporting applications can select data and build queries without intimate knowledge of the data source. They can make selections and build queries in a generic manner without having to separately support each data source.
- **SDK**—Makes it easy to add new data sources by implementing a driver. The driver declares the capabilities and properties of the data source. Most Production Reporting DDO capabilities are optional and the driver simply specifies which capabilities are supported.
- **Remote Data Access**—Sends data access requests to a Production Reporting DDO server and returning results back to the client. This allows the distribution of Production Reporting DDO drivers and clients.

# Production Reporting DDO Architecture

Production Reporting DDO provides a Java interface that conforms to the JavaBeans specification for reusable software components. You can access Production Reporting DDO from other programming languages using Component Object Model (COM) and Common Object Request Broker Architecture (CORBA) object-access methods. The interface keeps track of available data sources, connects to data sources to obtain metadata and data, and provides access to return results. Production Reporting DDO provides access to any object that holds data, not just databases.

Figure 1 Production Reporting DDO Architecture




## Establishing a Data Source Connection

Production Reporting DDO provides the necessary resources to establish a data source connection. These resources (also referred to as "interfaces") are components of the Production Reporting DDO interface and are described below:

- DataSourceManager—Maintains a list of available data sources in persistent registries from which the application locates and loads the data source.

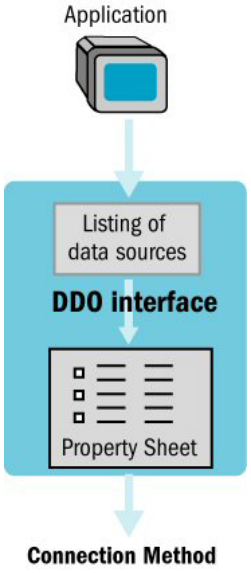
  Registering a new data source involves saving the name of the data source, the name of its implementation class (the driver), and a connection string. Multiple data sources of the same type can use the same driver.

- DataSource—Describes the data sources's capabilities and properties and allows the application to create a connection for the data source. Maintains a property sheet (PropertySheet) that describes the specific data source properties and capabilities, such as the properties required to open a data source connection.

  Data sources can implement different query languages. Data source support for a query language is optional. Data sources that do not support a query language may still support data filtering through a simple selector interface, thereby reducing the amount of data retrieved.

- Connection—The main Production Reporting DDO interface. Provides methods that obtain metadata and data. (Metadata describe the object hierarchy, the fields that the object provides, and the parameters.) Data can be obtained from a data source by executing a command, calling a procedure, or naming an object and requesting its data.

Figure 2 Establishing a Data Source Connection




## Supporting Transactions

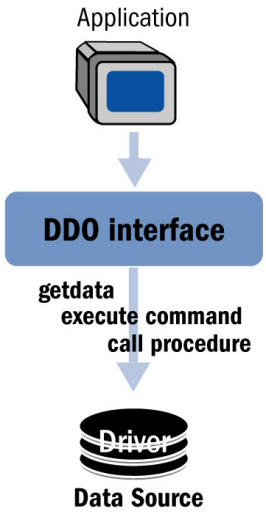
The transaction interface provides methods for beginning a transaction, commit, and rollback. A data source is not required to support transactions and will indicate whether it supports transactions. The scope of a transaction is one data source. Production Reporting DDO does not coordinate transactions between multiple data sources.

## Obtaining Metadata

A data source holds schemas that can contain data objects and procedures. Depending on the data source requirements, the application uses the appropriate method of the Production Reporting DDO connection interface to retrieve metadata.

- Data object—A generalized abstraction for any object that can provide its data on request. Includes tables, views, files, and business objects.This metadata describes all the data objects available through the established data source connection.
- Procedure—A generalized abstraction for any callable procedure or method that can be executed at the data source. Includes stored procedures and methods on business objects. A procedure can have parameters, a return value, and zero or more result sets.

Figure 3 Obtaining Metadata




## Obtaining Data

Production Reporting DDO can obtain data using three methods:

- **Executing a command**—A reporting application executes a command by passing command text and associated parameters. The command contains a statement that the data source can recognize along with any parameters. Results are returned via the Rowset interface.

  The result is a Rowset. The command can be anything that the data source understands. For example, a data source might specify that it can accept SQL and indicate the level of SQL that it understands. This allows reporting applications to send standard SQL to any data source that understands SQL as well as to pass-through data-source specific commands.

- **Using getData**—A data source that typically does not support a command language can return data from an information object. All that the application needs is the name of the object. The application names the information object from which data is requested and optionally specifies some selection criteria. The data source returns data as a Rowset interface.

- **Calling a Procedure**—The Procedure interface abstracts any parameterized object, remote procedure call, stored database procedure, or any object that behaves like a procedure. An application specifies the procedure it wants to execute and supplies any IN and IN/OUT parameters. The result object may contain a status and any number of Rowset objects representing multiple result sets. This interface allows the application to obtain output parameters, a return value, and any number or rowsets via the Rowset interface.

## Processing Results

The CallResults interface allows access to the results of a procedure call. This includes the return value, output parameters, and the Rowset objects that the procedure returns.

The Rowset interface is the main interface for processing results in Production Reporting DDO. It provides methods that retrieve the results a row at a time. Each Row contains Field objects. These are typed objects that are derived from the abstract Field class. The Rowset, Row, and Field objects are all self-describing. They provide information about the number, type, size, and name of the data items that they hold.

Rowset represents a forward-only read-only set of rows. It does not cache the entire set in memory. Once the next row is retrieved, the previous row may be discarded. A row may be retrieved from the database just in time for it to be retrieved. This allows the Rowset implementation to only keep a small number of rows in memory at any given time and therefore can handle very large results.

Field objects contain runtime metadata. This metadata is sufficient to process a field value without referencing the object metadata.

## Implementing Production Reporting DDO Interfaces with Drivers

In addition to defining an interface for applications, Production Reporting DDO provides a plug-and-play architecture for Production Reporting DDO drivers. Each driver provides an implementation of the Production Reporting DDO interface for its data source.

A major design objective of Production Reporting DDO is to make the task of writing an Production Reporting DDO driver as easy as possible:

- Production Reporting DDO only requires a driver to implement a minimal set of interfaces. Many of the interfaces are optional. In particular, an Production Reporting DDO data source is not required to support a query language or procedure calling, and it is not required to support transactions.
- An Production Reporting DDO driver declares its capabilities in a Capabilities resource file. No coding is required.
- Production Reporting DDO provides base classes that implement all the interfaces. These classes provide more than just default behavior. They provide all the functionality of managing Production Reporting DDO metadata. They also provide much of the common code that all drivers would otherwise have to implement.
- Production Reporting DDO provides a framework for exceptions and localized error messages. The messages are easily organized in resource files and can be managed using the tools that are included with the Production Reporting DDO SDK.
- Production Reporting DDO provides tools for logging messages and timing information. The driver can easily provide debugging information.

The driver developer need only write the code that is specific to the data source and the functionality that you wish to expose through the Production Reporting DDO API.

### Driver Manager

The Production Reporting DDO interface includes a driver manager that allows applications to list the available data sources and dynamically select them. Production Reporting DDO provides out-of-the-box support for common data sources by including a driver for relational databases using JDBC and drivers for delimiter-separated values files (CSV). In addition, Production Reporting DDO provides an XML driver, permitting access and interchange of intelligent data through XML. The driver can read XML directly from any web resource or an input stream.

An Production Reporting DDO driver developer only needs to implement functionality that is specific to the data source. The SDK provides classes that make it easier to write drivers for application business objects that use COM or CORBA interfaces.

## Data Access Strategies

To provide uniform data access for vastly different data sources, the Production Reporting DDO interface employs strategies that relieve the application from having to know each data source.

- **High Level of Abstraction**—TheProduction Reporting DDO interface reflects consistent patterns of data access. An application establishes a connection to the data source by specifying logon properties. The connection allows the application to obtain metadata, to retrieve data, and to perform transactions.

  The metadata describes information objects. These objects are grouped under schemas and organized as a hierarchy. Data from these objects can be obtained in one of three ways. First, the application can send commands and parameters to the data source. A command could use a query language to select data from one or more information objects. Second, the application can execute remote procedures. Third, the application can select an information object, select some or all of the fields that it provides, and specify selection criteria and ordering.

  The Production Reporting DDO interface hides implementation details. Performance optimizations such as pooling connections and commands, reading ahead, and caching are performed "under the cover" and are transparent to the application.

- **Descriptive Properties and Capabilities**—To provide uniform data access for vastly different data sources without resorting to a common denominator, Production Reporting DDO allows drivers to pick and choose which capabilities are relevant to their data source and which capabilities they will implement. For example, a driver can declare that it supports SQL commands and specify the version of SQL that it supports. The driver declares its capabilities in a resource file. There's no need to write code to describe capabilities. This simplifies the process of writing a driver. Applications can interrogate capabilities through the Production Reporting DDO data source interface.

  In addition to capabilities, drivers also declare properties. For example, the driver declares the properties that are used for logon, specify which are required, and which are secure. The reporting application can dynamically adapt to the capabilities of the data source.

- **Rich Metadata**—Production Reporting DDO allows applications to discover all the information objects that are accessible through a connection to a data source. It provides information about these objects, their types, and how they are organized. It describes

procedures, their input and output parameters, and the results they will provide once executed.

- **Mapping of Data Types**—Production Reporting DDO maps database data types to Production Reporting DDO Field objects. The Field object is an abstract object from which typed Field objects are derived. A Field object has a name, size, and value. The value of the field can be Null. Production Reporting DDO provides typed fields such as Text, Date, Number, Integer, Decimal, Long Binary, Object, Row, and Rowset.

  Production Reporting DDO offers automatic type conversions. For example, most of the basic types support a useful conversion of their value to and from text. All the numeric fields are special types of the Number field. This allows applications to treat all numeric data uniformly and, for example, convert their value to decimal. An application can choose which Production Reporting DDO types it recognizes. It does not need to know the data source's native data types.

- **Use of Hierarchies**—Production Reporting DDO draws a great degree of flexibility from supporting hierarchical structures. This comes into play in both metadata and data. Production Reporting DDO supports data sources that organize business intelligence (BI) objects in an arbitrarily deep hierarchy.

  The fields of a BI object can represent hierarchical dimensions. This is typical in multi-dimensional databases. You can traverse a dimension and obtain information about its levels and members. You can use members in making data selections from the object.

  Production Reporting DDO does not assume that data is flat or normalized. A Rowset can hold complex objects. In particular, a Rowset can hold additional Rowset objects.

- **Use of Procedures**—Production Reporting DDO supports the concept of a procedure, allowing data sources to expose their objects as procedures rather than tables, defining an interface for parameter passing without a query language, and enabling multiple Rowset retrieval.

- **Self-Describing Result Objects**—Production Reporting DDO result objects are self-describing. An application can determine the number of Rowset objects that are being returned as a result of executing a command or procedure. The Rowset object describes its fields. Fields describe their name, type, size, and precision.

- **Focus on Query and Reporting**—By focusing on query and reporting, the Data Access interface is simplified and, thus, is easier to implement. Rowset objects are read-only. A great degree of complexity involved in caching a Rowset, allowing bi-directional scrolling, allowing updates, and synchronization with the database has been eliminated. Updates to the databases are still supported through executing commands and procedures.

# 2 Creating Production Reporting DDO Applications

**In This Chapter**


About the Production Reporting DDO API....19
A Simple Code Example....19
Managing Data Sources....20
Establishing a Connection....25
Discovering Capabilities....28
Obtaining Metadata....30
Requesting Data....38


## About the Production Reporting DDO API

The Production Reporting DDO API defines Java interfaces that represent data sources, connections, information objects, result sets, and metadata. An application can enumerate the available data sources and create connections to them.TheProduction Reporting DDO API also allows applications to request data and process the results and includes a driver manager that can support multiple drivers that connect to data sources.

This chapter provides a code example that steps through locating, connecting and accessing a data source, and then processing data obtained. It then discusses how Production Reporting DDO supports and maintains these processes.

## A Simple Code Example

*Program ex1.sqr* demonstrates locating a data source, establishing a connection, retrieving data, and processing data. (The code omits some error checking.) The fields returned are implicitly converted to text. This program shows how to take advantage of the Rowset being self-describing to obtain the names of the columns returned.

The Java package `com.sqribe.access` implements the Production Reporting DDO API.

### Program ex1.sqr

```
import com.sqribe.access.*;
...
try {
   DataSource ds = DataSourceManager.getDataSource("accounting");
   ds.getPropertySheet().setProperty("user", "scott");
```

```
ds.getPropertySheet().setProperty("password", "tiger");
Connection c = ds.open();
Rowset rs = c.getData(c.getSchemaObject("Employee"));
System.out.println("<table><tr>");
// print headings
for (int i = 0; i < rs.getFieldCount(); i++)
   System.out.println("<td>" + rs.getField(i).getName() + "</td>");
while (rs.next()) {
   // print data
   System.out.println("</tr><tr>");
   for (int i = 0; i < rs.getFieldCount(); i++)
      System.out.println("<td>" + rs.getField(i) + "</td>");
}
System.out.println("</tr></table>");
c.close(); // close the connection to the data source
```

# Managing Data Sources

The DataSourceManager maintains a cache of available data sources called the Registry. The list identifies the data sources by name. Using the Registry, the DataSourceManager can locate and instantiate a data source and return a DataSource interface to the application.

To locate a data source using its name and obtain a DataSource interface, an application calls the `DataSourceManager.getDataSource()` method. For example:

```
DataSource ds;
ds = DataSourceManager.getDataSource("accounting");
if (ds == null) {
   System.out.println("The accounting data source is unavailable.");
```

In this code example, the DataSourceManager searches the Registry for a data source called *accounting*. If it locates the driver, DataSourceManager loads the driver for this data source and returns its DataSource interface. If it does not find the data source in the Registry, or if the data source entry in the Registry is improperly configured, the DataSourceManager returns a null reference.

**Note:**

Multiple calls to `getDataSource()` with the same name return the same object.

You can use the `DataSourceManage.getDataSources()` method to obtain a list of the available data sources in the current registry.

```
import com.sqribe.access.*;
import java.util.Enumeration;
...
Enumeration enum;
enum = DataSourceManager.getDataSources();
while (enum.hasMoreElements()) {
    DataSource ds = (DataSource)enum.nextElement();
    System.out.println(ds.getName());
```

By default, the registry contains entries that are stored in the file `Registry.properties`. This file can be found in the `properties` folder.

## Registry Files

This section shows how to manage multiple registries. If your application always uses the default registry (the file `Registry.properties`), then you can skip this section and proceed directly to "Defining a New Data Source" on page 22.

The Registry is a cache of data sources maintained by the DataSourceManager. The first time you use DataSourceManager, it loads the Registry from a file. The default Registry file is `Registry.properties`. This file resides in the Production Reporting DDO `properties` folder.

Your application can load the Registry from alternative sources using the DataSourceAdmin class. This class provides static methods for loading DataSource definitions from a file or InputStream.

A registry file contains a list of registered data sources along with a short description for the data source, the name of its implementation class, and a connection string.

You can use the same DataSource implementation class to implement multiple data sources. Each data source is registered separately.

The following code example loads the Registry from a file called `MyDataSources.properties`. This file is located in the `properties` folder.

```
try {
DataSourceManager.getDataSourcesAdmin().load("MyDataSources");
} catch (DataAccessException e) {
    e.printStackTrace();
```

Note that the `.properties` extension of the registry file is assumed and should not be specified in the `load()` call. Also note that like most of Production Reporting DDO calls, this call can fail with a DataAccessException.

In addition to loading registry entries from a properties file in the `properties` folder, the DataSourcesAdmin class allows you to load registry entries from any file or InputStream. You can perform multiple loads. The effect is cumulative. The DataSourcesAdmin class also allows you to save the current registry into a file or OutputStream.

## The DataSourceManagerAdmin Class

The DataSourceManager uses the DataSourceManagerAdmin class to maintain its properties in a file named:

`com_sqribe_access_DataSourceManager_Properties`

This file is located in the Production Reporting DDO `properties` folder.

Within this properties file, there is a property called `DataSources.files`:

```
DataSources.files = <list of data sources property files>
```

For example:

```
DataSources.files = Registry MyDataSources
```

The DataSourceManagerAdmin allows you to enumerate these names and subsequently use them in `DataSourcesAdmin.load()` calls. This allows an application to deal with multiple registries. For example, you have a restricted registry with only a few data sources and a power user's registry with all available data sources. The class also allows an administrative tool to obtain the list of usable registries.

# Defining a New Data Source

You define a data source in a registry file. You can either use the tool that is supplied with the Production Reporting DDO SDK or edit the registry file directly.

## Creating a New Data Source by Editing a Registry File

To understand how to add a new data source to a registry file, let's go through an example. This example adds a data source called HelpDesk to the default registry file, `Registry.properties`. The HelpDesk data source is an Oracle database. Access to this database is through the Production Reporting DDO JDBC Access driver using a JDBC driver from Oracle.

We will go through the following steps:

- Identifying the packages to add to the CLASSPATH
- Editing the `Registry.properties` file
- Specifying the name of the data source
- Specifying the Production Reporting DDO driver
- Specifying loading of the JDBC driver
- Specifying the JDBC URL

➤ To add a new data source to a registry file:

1 **Identify the packages to add to the CLASSPATH.**

We will access the database via JDBC using the JDBC Access Production Reporting DDO driver. This driver is implemented in the package `com.sqribe.jdbcacc` and distributed in the file `ddo11.zip`. If this file is not already part of your CLASSPATH environment variable, then add it to the CLASSPATH now.

The example uses a JDBC driver for connecting to Oracle. At the time of writing this book, Oracle is providing a JDBC driver in a file called "classes111.zip". This file includes the package `oracle.jdbc.driver` that implements the JDBC driver. Include this ZIP file as part of your CLASSPATH.

2 **Edit the `Registry.properties` file.**

Edit the `Registry.properties` file using a text editor such as Notepad. The following lines are the entry for our HelpDesk data source, which you can insert these lines at the end of the file.

```
HelpDesk.desc=Technical Support HelpDesk
HelpDesk.class=com.sqribe.jdbcacc.JDBCDataSource
HelpDesk.lib=oracle.jdbc.driver.OracleDriver
HelpDesk.load=
HelpDesk.conn=jdbc:oracle:oci7:@TechSupport.World
```

3 **Specify the name of the data source.**

In the example above, the name of the data source, HelpDesk, forms part of the property name.

**Table 1 Properties of the HelpDesk Data Source**

| Property | Purpose |
|---|---|
| HelpDesk.desc | Description of this data source. |
| HelpDesk.class | Class name of the Production Reporting DDO driver. This is the name of the Java class that implements the DataSource interface. |
| HelpDesk.lib | List of Java classes to load as part of the initialization of this data source. |
| HelpDesk.load | List of native libraries to load as part of the initialization of this data source. |
| HelpDesk.conn | Connection string that the Production Reporting DDO driver understands. In the case of the JDBC Access driver, this is a JDBC URL. |

4 **Specify the Production Reporting DDO Driver.**

An Production Reporting DDO driver is a Java class that implements the DataSource interface. For the JDBC Access driver, this class is `com.sqribe.jdbcacc.JDBCDataSource`. This class is packaged in the `ddo11.zip` file. Specifying the name of the class as the value of the `HelpDesk.class` property tells the DriverManager how to start this Production Reporting DDO driver.

5 **Specify loading of the JDBC driver.**

The Production Reporting DDO JDBC Access driver can use any JDBC driver to access a relational data source. A URL specifies the relational data source. The Java JDBC driver manager uses the URL to locate a suitable JDBC driver from among the drivers that are currently in memory (loaded by the JVM class loader).

Ensuring the availability of a suitable driver often requires explicitly loading the JDBC driver into memory. Production Reporting DDO allows specifying a list of Java classes to load in the `HelpDesk.lib` property. In our example, we specify the name of the Oracle JDBC driver's class, `oracle.jdbc.driver.OracleDriver`. By loading this driver explicitly, we make sure that the JDBC driver manager will successfully resolve the JDBC URL for the HelpDesk data source.

6 **Specify the JDBC URL.**

The `HelpDesk.conn` specifies the data source-specific connection string. In the case of the Production Reporting DDO JDBC Access driver, this is a JDBC URL. The URL always start with `jdbc:`. The rest of the URL selects Oracle and identifies the specific Oracle database and the

connectivity method. For more information about constructing the URL, refer to the JDBC driver documentation.

Following is another example using the Sun JDBC-ODBC bridge. The Registry in the example configures the HelpDesk data source to use the Sun JDBC-ODBC bridge. The example assumes an ODBC data source configured under the name `HelpDesk`.

```
HelpDesk.desc=Technical Support HelpDesk
HelpDesk.class=com.sqribe.jdbcacc.JDBCDataSource
HelpDesk.lib=sun.jdbc.odbc.JdbcOdbcDriver
HelpDesk.load=
HelpDesk.conn=jdbc:odbc:HelpDesk
```

Note that the `HelpDesk.class` property did not change. It is still the Production Reporting DDO JDBC Access driver. The `HelpDesk.conn` URL changed to specify `odbc` with the DSN (data source name) of `HelpDesk` (the name does not have to be `HelpDesk`. It can be any name that was given to this ODBC data source). The `HelpDesk.lib` loads the Sun JDBC-ODBC bridge driver.

## Creating a New Data Source at Run-Time

The Data Source Manager manages a registry of data sources to which an application can add a new data source using the `add()` method. You can obtain a reference to this object using the `getRegistry()` method of the DataSourceManager class.

The example code below defines the HelpDesk data source.

```
// Create a data source
DataSourceManager.getRegistry().add(
    "HelpDesk",                                    // name
    " Technical Support HelpDesk ",                // desc
    "com.sqribe.jdbcacc.JDBCDataSource",           // class
    "oracle.jdbc.driver.OracleDriver",             // lib
    "",                                            // load
    " jdbc:oracle:oci7:@TechSupport.World ");      // conn
```

After successfully registering the `HelpDesk` data source, an application can obtain the DataSource interface by calling

`DataSourceManager.getDataSource(HelpDesk).`

#### Tip:

Loading a data source using the `add()` method of the Registry class might prevent the automatic load of `Registry.properties`. This is because `Registry.properties` only loads if the registry is empty. To load `Registry.properties`, use one of the following options:

- Explicitly load it with the following: `DataSourceManager.getDataSourcesAdmin().load(Registry)`
- Have it automatically load before you add the new data source by calling the `getDataSource()` or `getDataSources()` method of the DataSourceManager class.

Calling these methods before adding the new data source (while the registry is empty) causes `Registry.properties` load.

# Establishing a Connection

Once your application locates a data source and obtains a DataSource interface, it can establish a connection to the data source using the `open()` call. This call returns a Connection interface.

We start by setting the logon properties required to connect to this data source. Typically, these are the `user` and `password` properties. However, Production Reporting DDO does not assume that this is always the case. Production Reporting DDO allows the data source to specify the logon properties and allows your application to discover these properties at run-time.

The code example below shows how Production Reporting DDO sets the `user` and `password` properties and how the `open()` call establishes a connection.

```
import com.sqribe.access.*;

try {
    DataSource ds = DataSourceManager.getDataSource("accounting");
    if (ds == null) ...
    ds.getPropertySheet().setProperty("user", "scott");
    ds.getPropertySheet().setProperty("password", "tiger");
    Connection c = ds.open();
} catch (Exception e) {
    e.printStackTrace();
```

After locating the data source and obtaining a DataSource interface, we set the `user` and `password` properties. The example obtains the DataSource PropertySheet and uses the `setProperty()` method of the PropertySheet class to set the `user` and `password` properties. Note that these calls would fail with a PropertyException if the data source does not recognize a `user` or `password` property. An application must be ready to handle this exception. The PropertySheet and PropertyException classes are defined in the common utilities package (`com.sqribe.comutil`). This package contains the general-purpose classes that are used by Production Reporting DDO.

Having set the required logon properties, our example uses an `open()` call to establish a connection. The `open()` call does not take any arguments. It returns a Connection interface. The application will use this interface to access the data source.

If the `open()` call fails, a DataAccessException is thrown.

An application cannot assume that `user` and `password` are the logon properties for any arbitrary data source.

## Processing Logon Properties

This section examines the general case in which the application checks for the logon properties for the given data source.

An Production Reporting DDO data source can be any application object that holds data—not necessarily a traditional DBMS. Therefore, one cannot assume that a logon requires only a user

name and password. In fact, a data source can require any number of items to establish a connection. These items can be anything that identifies the data to be accessed and the user who wants to access the data. These items can include an account number, folder name, identification code, certificate and digital signature, to name a few examples.

An interactive application can obtain the list of logon properties from a data source at run-time and prompt the user for these properties. For each property, the application can obtain a description of the property, type, valid values, whether the property is required and whether it is secure. Properties can be grouped. For example, the `logon` property groups all logon properties.

One can program a batch application to pass properties to a data source based on prior knowledge of the specific data source. You have seen an example of that earlier with the `user` and `password` properties.

The PropertySheet class provides access to PropertyDescription objects that describe its valid properties. PropertyDescription objects can be nested to represent grouped or nested properties.

Start by reviewing *Program ex2.sqr*.

## Program ex2.sqr

```
import com.sqribe.access.*;
import com.sqribe.comutil.*;
public class Example {
   public static void main(String[] args) {
      try {
        DataSource ds =          DataSourceManager.getDataSource("accntng");
        PropertySheet prop = ds.getPropertySheet();
        list(prop.getPropertyDescription("logon"));
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   static void list(PropertyDescription pd) {
     PropertyDescription l[] = pd.getIndexes();
      if (l != null)
        for (int x = 0; x < l.length; x++) list(l[x]);
      else
        System.out.println(pd.getName());
}
```

### Code

```
DataSource ds = DataSourceManager.getDataSource("accntng");
```

### Description

Locates the accounting data source and obtain a DataSource interface to it.

### Code

```
PropertySheet prop = ds.getPropertySheet();
```

### Description

Obtains the DataSource PropertySheet object.

### Code

```
list(prop.getPropertyDescription("logon"));
```

### Description

Calls the `getPropertyDescription()` method of PropertySheet to obtain a PropertyDescription object for `logon`, which groups the logon properties, then calls the `list()` method of our Example class to list the PropertyDescription. A PropertyDescription can be nested to group additional PropertyDescription objects.

### Code

```
PropertyDescription l[] = pd.getIndexes();
 if (l != null)
            for (int x = 0; x < l.length; x++) list(l[x]);
          else
            System.out.println(pd.getName());
```

### Description

The `list()` method checks to see if there are any nested PropertyDescription objects by calling `getIndexes()` and then checking to see if it's null. A null value means that there are no nested PropertyDescription objects for this PropertyDescription.

If there are nested objects, the example code calls `list()` recursively for each nested object.

Otherwise, the example code prints the property name. Recursive traversal eventually locates all the properties that are nested under `logon`.

The application can prompt the user for a value for each property. To do so, the application obtains additional attributes from the PropertyDescription objects. These attributes provide all the necessary information for the application to obtain logon attribute from the user and pass them to Production Reporting DDO. These attributes are summarized in Table 2.

**Table 2 Attributes Derived from Property Description Objects**

| Attribute | Purpose |
|---|---|
| **Label** | Prompts the user for this property. |
| **Description** | Description of the property offered to users as a help message. |
| **Required** | Whether users must supply a value for this property. |
| **Secured** | Whether the value of this property is secured. This is for properties such as password that should not be echoed on the screen and should be encrypted. Production Reporting DDO performs the encryption automatically. |
| **Validator** | Name of a class that implements the PropertyValidator interface. An application does not need to worry about this property since the Validator is typically supplied by either Production Reporting DDO or the DataSource driver. |

| Attribute | Purpose |
|---|---|
| **ValidationType** | Type of validation for the value of the property: This is a numeric value, 0 means no validation, 1 means that the value must fall inside a range (inclusive), and 2 means that the value must be picked from a list. The validation is performed by the validator class specified in the Validator attribute. |
| **ValidationValues** | This is a list. If the ValidationType is range then the first item on the list is the minimum value and the second is the maximum value. Otherwise the items represent the valid choices for the value for this property. An application can use these values to populate a list box. |

# Discovering Capabilities

Capabilities are read-only properties that describe capabilities and attributes of the data source. Like properties they can be grouped and nested. As with properties, access to capabilities is through the PropertySheet class and one describes them using the PropertyDescription class. Their main use is to describe the data-access interfaces that the data source supports.

If an application is familiar with the data source, it probably already knows its capabilities. However, if you are building an ad-hoc application that can connect to various data sources, it is important to be able to discover the capabilities of the data source at runtime. To support a great variety of data sources and to ease the task of writing an Production Reporting DDO driver, the Production Reporting DDO specification defines a minimal mandatory interface. Beyond the minimum, the driver is free to decide which interfaces to support—as long as it declares its capabilities.

Table 3 lists examples of data source capabilities. Production Reporting DDO allows your application to check if a data source supports each one of these capabilities.

**Table 3** Data Source Capabilities

| Capability | Description |
|---|---|
| **command.supported** | Indicates that the data source supports the execution of commands. A data source is not required to support any command language. |
| **call.supported** | Indicates that the data source supports the execution of procedures. A data source is not required to support procedures. |
| **selector.supported** | The data source has the option of supporting the getData(Selector) method that allows your application to pass a Selector object to the getData() method in order to qualify the data retrieval. |
| **transaction.supported** | Indicates that the data source supports the Transaction interface. If it does not support the Transaction interface then a commit and rollback calls will be silently ignored. |
| **md.supported** | Indicates that the data source is multidimensional. You can pass a MDSelector object on the getData(Selector) call. MDSelector is a special kind of selector for multidimensional data sources. You can also obtain hierarchical dimension metadata by fetching child members using the getChildren() method of SchemaObjectColumn. This is useful for multidimensional databases. If the data source does not support |

| Capability | Description |
|---|---|
| | this capability then a call to getChildren() on a SchemaObjectColumn would return null. The data source also supports the MDSchemaObject interface. |
| **concurrent.connection.supported** | Indicates whether or not the concurrent use of a connection is supported by the data source. When supported, multiple calls can be active through the same connection. The kind of concurrency supported is delineated by the specifications of the call, execute, selector, and transaction concurrency settings.<br>A call is said to be active if any of the Rowset objects that it returns are still active (still hold results pending). |
| **JDBC.Database.get-IdentifierQuoteString** | Example of a driver-specific capability. If your application is talking to the JDBC Access driver, it can obtain JDBC-specific capabilities.<br>The JDBC.Database.getIdentifierQuoteString capability tells your application what character to use for identifiers in SQL such as a column name that contains a space. Typically this is either single quote or double quote. |

# Checking for a Capability

*Program ex3.sqr* checks to see if the data source supports the Transaction interface.

## Program ex3.sqr

```
boolean checkTransactionSupport(DataSource ds) {
    PropertySheet prop = ds.getPropertySheet();
    Boolean supported =
        (Boolean)prop.getCapability("transaction.supported");
    if (supported == null) return false;  // capability is not defined
    return supported.booleanValue();
}
The following example checks for the "JDBC.Database.getIdentifierQuoteString" capability
and uses it to compose an SQL statement.
try {
    // connect to the HelpDesk database
    DataSource ds = DataSourceManager.getDataSource("HelpDesk");
    PropertySheet prop = ds.getPropertySheet();
    prop.setProperty("user", "scott");
    prop.setProperty("password", "tiger");
    Connection c = ds.open();
    // check for the JDBC.Database.getIdentifierQuoteString
    prop = c.getPropertySheet();
    String quote = (String)prop.getCapability(
                    "JDBC.Database.getIdentifierQuoteString");
    // use the quote string to construct the SQL
    String sql = "select " + quote + "Employee Name" +
                 quote + " from " + quote + "Employees" + quote;
    // Based on the data source, the following should display either
    // select 'Employee Name' from 'Employees'
    // or
    // select "Employee Name" from "Employees"
    System.out.println(sql);
```

```
    c.close();
} catch ...
```

# Obtaining Metadata

Using the Production Reporting DDO API, an application can obtain rich metadata about the data objects at the data source without any prior knowledge of the data objects. The metadata provides a complete description of how to access these objects, their parameters, and the data they provide.

To provide uniform metadata for very different data sources, Production Reporting DDO uses the following abstraction:

- A data source has one or more schemas. A schema is a grouping of data objects, procedures, or additional schemas.

  Some databases organize schemas within catalogs. Production Reporting DDO reflects these as schemas within schemas. Other data sources organize data objects and procedures in a hierarchy that can have any number of levels. Production Reporting DDO supports hierarchy levels with its recursive notion of schemas.
- Production Reporting DDO imposes no limit on the level of nesting of schemas within schemas.
- A data object is a generalized abstraction for any object that can provide its data on request. This includes tables, views, files, and business objects.
- A data object has a set of columns.
- A procedure is a generalized abstraction for any callable procedure or method that can be executed at the data source. This includes stored procedures and methods on business objects.
- A procedure can have parameters, a return value, and zero or more result sets.
- A procedure's result set has a set of columns.
- A column of a data object or a procedure's result set could represent a scalar item such as a date field, or a complex item such as a structure. The same is true for procedure parameters and return value.
- A column, parameter, or return value can have children that are themselves columns. This represents a hierarchy.
- A column hierarchy could represent a "dimension" in multidimensional (OLAP) terminology. It means that the column represents a hierarchy of members (for example departments in a hierarchical organization). By recursively enumerating the children of the column you can traverse the outline of a dimension.
- A column hierarchy could represent a structure or a table. In these cases by enumerating the children of this column you can list the fields of the structure or table. In the most general case this could also be a recursive process (nested structures and tables).

This is fairly abstract, but it will become clearer as we discuss the API and give example of its usage.

## Obtaining Schema Information

The Connection interface allows the listing of objects that are available at the data source. Production Reporting DDO groups these objects into schemas. A data source will always have at least one schema that groups the data objects and procedures that it provides. In the most general case, Production Reporting DDO organizes multiple schemas as a hierarchy. Each schema in the hierarchy can hold additional schemas as well as data objects and procedures.

The `getSchemas()` method of the Connection interface accesses the top of that hierarchy. This methods returns a `Schemas` object, which holds a list of the objects contained within the current schema. The list can contain data objects, procedures, or additional schemas.

*Program ex4.sqr* is a function that takes a Connection interface as a parameter and lists all the objects starting with the top of the hierarchy and recursively listing the schemas. *Program ex4.sqr* demonstrates the most general case in which the application has no knowledge of the data source and has to traverse the schemas to discover all the objects.

### Program ex4.sqr

```
static void listConnection(Connection c) throws DataAccessException
{
    list(c.getSchemas(), 0);
}
static void list(Schemas schemas, int level)
{
    Enumeration enum;
    enum = schemas.elements();
    while (enum.hasMoreElements()) {
        Schema schema = (Schema)enum.nextElement();
        if (level > 0) indent(level);
        if (schema instanceof SchemaObject)
            System.out.println("data object: " + schema.getName());
        else if (schema instanceof SchemaProcedure)
            System.out.println("procedure: " + schema.getName());
        else {
               Schemas children = schema.getChildren();
               if (children != null)
                   list(children, level + 1);
        }
    }
}
static void indent(int level)
{
    while (level-- > 0) System.out.print("    ");
}
```

**Code**

```
list(c.getSchemas(), 0);
}
    static void list(Schemas schemas, int level)
    {
        Enumeration enum;
        enum = schemas.elements();
        while (enum.hasMoreElements())
```

### Description

Recurses through the schema hierarchy. The `list()` function takes an object of type `Schemas` as an argument. The `Schemas` class is a collection of schema elements—objects of type `Schema`—data objects, procedures, and schemas. The other argument of the `list()` function is the level of recursion, which is used for indentation when listing objects.

`list()` starts at the top of the hierarchy by calling `getSchemas()` on the Connection interface. `getSchemas()` returns a `Schemas` object that holds all the top-level schemas and objects for this data source. We then call `list()` to list the schema recursively.

The `list()` function enumerates the items in the Schemas object. These items are of type Schema, which includes data objects and procedures that are abstracted in the SchemaObject and SchemaProcedure classes, respectively. The `SchemaObject` and `SchemaProcedure` classes are subclasses of the Schema class.

### Code

```
if (schema instanceof SchemaObject)

System.out.println("data object: " + schema.getName());
```

### Description

Checks if the Schema object is a data object by checking if it is an instance of the `SchemaObject` class. If the object is of type SchemaObject, then we list it as a data object.

### Code

```
else if (schema instanceof SchemaProcedure)
      System.out.println("procedure: " + schema.getName());
```

### Description

Checks the Schema object to see if it is a procedure by checking if it is an instance of the SchemaProcedure class. If the object is of type SchemaProcedure, then we list it as a procedure.

### Code

```
       else {
            System.out.println("schema: " + schema.getName());
               Schemas children = schema.getChildren();
               if (children != null)
                   list(children, level + 1);
```

### Description

Lists the SchemaObject as a schema and further lists its contents by making a recursive call to `list()` after having accounted for the cases of data objects or procedures.

## Listing Data Objects in a Schema

*Program ex5.sqr* connects to an Oracle database and lists the tables and views that are accessible under user SCOTT.

### Program ex5.sqr

```
try {
    DataSource ds = DataSourceManager.getDataSource("HelpDesk");
    PropertySheet prop = ds.getPropertySheet();
    prop.setProperty("user", "scott");
    prop.setProperty("password", "tiger");
    Connection c = ds.open();
    Schemas schemas = c.getSchemaObjects(c.getSchema("SCOTT"));
    Enumeration enum;
    enum = schemas.elements();
    while (enum.hasMoreElements()) {
        SchemaObject obj = (SchemaObject)enum.nextElement();
        System.out.println(obj.getType() + ": " + obj.getName());
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

The example creates a Connection to the HelpDesk database. It then calls the `getSchemaObjects()` method of the Connection interface to get a list of the objects in the schema named `SCOTT` and enumerates the elements in the schema. Each element is a SchemaObject. The example prints its type and its name. The type is a database specific name that represents the type of object, for example TABLE or VIEW.

In a similar manner, we list the stored procedure under the schema named `SCOTT` by making the following call:

```
Schemas schemas = c.getSchemaProcedures(c.getSchema("SCOTT"));
```

## Obtaining Information about Data Objects

Because Production Reporting DDO must deal with very different data sources, it must provide a generalized abstraction of what a data source holds. As you have seen in the previous section, a data source holds schemas that contain data objects and procedures.

This section focuses on these data objects. Here are a few examples of data objects:

- Tables and Views in a relational database. Basically any object from which you can "select" data.
- A "cube" in a multidimensional database.
- A data file in a data source that provides access to files.
- An XML document.
- An object representing a collection of objects in an object database or application. For example, a list of employee objects.

Production Reporting DDO provides methods that allow you to list data objects and obtain metadata for each object using a `SchemaObject` reference.

### Data Sources With No Query Language Support

If the data source doesn't support SQL or any other query language, an application can obtain data from it by using the `getData()` method of the Connection interface.

### Data Sources That Support a Query Language

If the data source supports a query language, an application can retrieve data from it by sending queries that name this object. It also accepts DML (data manipulation) commands. For example, an application can send an UPDATE statement to a data source that supports SQL.

Table 4 lists the attributes of a SchemaObject.

**Table 4 Attributes of a SchemaObject**

| Attribute | Description |
|---|---|
| Name | Object name. |
| Description | Description or comment associated with this object. |
| Type | Object type. This is a data source--specific string such as TABLE, or VIEW. |
| Parent | Schema that holds this object (if available). |

## Retrieving Column Metadata

An application can also obtain information about the columns of this data object. To do this, call the `getSchemaObjectColumns()` method of the Connection interface. This method takes a reference to a SchemaObject and returns a SchemaObjectColumns reference. Another way to get column metadata is to call the `getMetaData()` method of SchemaObject.

Here is an example.

```
void listColumns(SchemaObject obj) throws DataAccessException {
    SchemaObjectColumns cols = obj.getMetaData().getSchemaObjectColumns();
    for (int i = 0; i < cols.size(); i++) {
        SchemaObjectColumn col = (SchemaObjectColumn)cols.elementAt(i);
        System.out.println("    " + col.getName() +
                           ", " + col.getDBTypeName());
```

The function `listColumns` in the example takes a SchemaObject as an argument and lists its columns. We obtain a SchemaObjectColumns reference for this object using the `obj.getMetaData().getSchemaObjectColumns()` call, where SchemaObjectColumns object is a vector of SchemaObjectColumn objects. The `count()` method returns the number of columns in this object and the `elementAt()` method returns each column. Note that we cast the result of the `elementAt()` call to SchemaObjectColumn. For each column (`SchemaObjectColumn`), we print the name and database datatype name.

## Obtaining Metadata About Procedures

Procedures are very powerful objects in Production Reporting DDO. They provide an abstraction for information objects that are parameterized.

The following are examples of procedures.

- Stored database procedures and functions in a relational database.
- Remote procedure calls (RPCs).
- Method invocation on objects in object databases and applications.
- Method invocation of COM or CORBA interfaces to business objects.

You can pass input parameters and receive values back on output parameters. The same parameter applies to both input and output (INOUT parameter). Procedures can return a value. This is similar to a function call. Most of all, procedures can return data in multiple sets and Production Reporting DDO allows the retrieval of descriptions for the columns of each such result set.

Production Reporting DDO provides robust support for procedure calls by allowing parameters to be complex structures. Similarly, result sets are not limited to flat tables and can hold complex structures.

Table 5 lists attributes of a SchemaProcedure.

**Table 5 Attributes of a SchemaProcedure**

| Attribute | Description |
|---|---|
| Name | Procedure name. |
| Description | Description or comment associated with this object. |
| Type | Object type. This is a data source--specific string such as “Procedure”, or “Function.” |
| Parent | Schema that holds this object (if available). |

To get the metadata describing parameters, return value, and columns, you can call the `getProcedureMetaData()` method of the Connection interface. This method takes a reference to a SchemaProcedure and returns a ProcedureMetaData reference. You can also get metadata by calling `getMetaData().getProcedureMetaData()` on SchemaProcedure.

### Listing Procedure Parameters

*Program ex6.sqr* demonstrates listing the parameters of a procedure and displaying each parameter, whether it is input, output, or both.

#### Program ex6.sqr

```
void listParameters(ProcedureMetaData meta) {
   SchemaProcedureColumns params = meta.getParameters();
   if (params == null)
      System.out.println("  No parameters for this procedure.");
```

```
   else
      for (int i = 0; i < params.size(); i++) {
         SchemaProcedureColumn col = meta.getParameter(i);
         String kind = null;
         switch (col.getUse()) {
            case SchemaProcedureColumn.PARMIN:
               kind = "IN";
               break;
            case SchemaProcedureColumn.PARMOUT:
               kind = "OUT";
               break;
            case SchemaProcedureColumn.PARMINOUT:
               kind = "INOUT";
               break;
         }
         System.out.println("  parameter: " + col.getName()
                            +  " - " + kind);
      }
}
```

The `listParameters()` method takes a ProcedureMetaData object as an argument. Calling the `getParameters()` method on this object retrieves a SchemaProcedureColumns object representing the parameters list of this procedure. If the procedure has no parameters, `getParameters()` returns null. Otherwise, it lists each parameter. The `getUse()` method of SchemaProcedureColumn returns the kind of parameter—input, output, or both.

## Determining the Return Value

The code example below demonstrates how to check for the procedure's return value.

```
void listReturnValue(ProcedureMetaData meta) {
   SchemaProcedureColumn retValue = meta.getReturnValue();
   if (retValue == null)
      System.out.println("  No return value for this procedure.");
   else
      System.out.println("  return Value: " + retValue.getName()
                         +  " - " + retValue.getDBTypeName());
}
```

The `listReturnValue()` method takes a ProcedureMetaData object as an argument. Calling the `getReturnValue()` on this object retrieves a SchemaProcedureColumn object representing the return value of this procedure. The call returns a null when the procedure has no return value or it displays a return value along with its data source-specific type name.

### Listing Procedure Result Sets

*Program ex7.sqr* demonstrates listing the result sets of a procedure and listing the columns of each result set.

#### Program ex7.sqr

```
void listResultSets(ProcedureMetaData meta) {
```

```
   Vector resultSets = meta.getResultSets();
   if (resultSets == null)
      System.out.println("  No result sets for this procedure.");
   else
      for (int i = 0; i < resultSets.size(); i++) {
         System.out.println("  Result set " + i + ":");
         listResultSet(meta, i);
      }
}
void listResultSet(ProcedureMetaData meta, int i) {
   int columnCount = meta.getResultSet(i).size();
   for (int j = 0; j < columnCount; j++) {
      SchemaProcedureColumn col = meta.getResultSetColumn(i,j);
      System.out.println("    column: " + col.getName()
                         +  " - " + col.getDBTypeName());
   }
}
```

The `listResultSets()` method takes a ProcedureMetaData object as an argument. Calling the `getResultSets()` on this object retrieves a Vector (java.util.Vector) object representing an array of one or more result sets. The call returns a null when the procedure has no result sets. Otherwise, it calls `listResultSet()` to list the columns for each result set.

The `listResultSet()` method takes a ProcedureMetaData object and a result set number as arguments. Calling `getResultSet(i).size()` retrieves the number of columns in this result set, then displays metadata for each column using the `getResultSetColumn(i, j)` call. In this call, "`i`" represents the result set number and "`j`" represents the column within that result set.

## Columns

Columns describe data items in data objects and in procedure result sets. Columns also describe procedure parameters and return value. In previous sections, we saw how to obtain columns for data objects and procedures. These columns are abstracted in the SchemaObjectColumn and ProcedureObjectColumn classes. SchemaProcedureColumn is a subclass of SchemaObjectColumns. It shares all the attributes of SchemaObjectColumns. It also provides additional attributes.

Table 6 lists the attributes that are common to all columns.

**Table 6 Attributes Common to All Columns**

| Column | Description |
|---|---|
| **Name** | Name of the Column |
| **Size** | Width of the field. |
| **Precision** | Precision for numeric columns. |
| **Scale** | Scale for numeric columns. |
| **DBType** | Data source--specific number representing the data source--specific datatype. |

| Column | Description |
|---|---|
| **DBTypeName** | Data source--specific datatype name. |
| **FieldType** | Production Reporting DDO datatype for this column. When data is retrieved from the data source, this column will return a field of this type. The FieldType is an integer number that is one of the constants that are defined in the Field class. |
| **Field.Text** | Text string field. |
| **Field.Number** | Numeric field. This can be an integer, double, or Decimal. |
| **Field.Date** | A date or date and time field. |
| **Field.Boolean** | A true or false value. |
| **Field.Binary** | Binary raw data (stream of bytes). |
| **Field.Row** | A structure. |
| **Field.Rowset** | A table. |
| **Field.Object** | An arbitrary object. |

SchemaProcedureColumn, which has a dilatational attribute, describes procedure parameters and return values, as shown in Table 7.

**Table 7 Additional Attribute of SchemaProcedure Column**

| Attribute | Description | |
|---|---|---|
| Use | Use of this column. This is an integer number that is one of the constants that are defined in the SchemaProcedureColumn class. | |
| | Constant | Description |
| | PARMIN | Input parameter. |
| | PARMINOUT | Parameter used for both input and output. |
| | PARMOUT | Output parameter. |
| | RESULTCOLUMN | Column in a result set. |
| | RETURNVALUE | Procedure's return value. |
| | UNDEFINED | Undefined. |

# Requesting Data

There are three fundamental ways of requesting data in Production Reporting DDO:

- **getData**—Names an object and requests its data. Data sources that do not support any query language or procedure-call mechanism can use this method.

- **execute**—Executes a command such as a SQL SELECT statement. The command is a text string that is passed through to the data source driver. The command takes parameters using "?" in the command text.
- **call**—Calls a procedure. You can call procedures directly; pass parameters when calling procedures; and obtain output parameters, return values, and multiple result data sets.

The following section describes the `getData()` method and the associated Selector class. Using the selector class, you can specify the columns that should be returned.

## Retrieving Data with getData

The most basic method of data retrieval in Production Reporting DDO is `getData()`. This method takes the name of a data object as an argument, where the name can be a string or a SchemaObject reference. `getData()` retrieves all the data for the named object and returns a Rowset, as shown in *Program ex8.sqr*.

### Program ex8.sqr

```
try {
    DataSource ds = DataSourceManager.getDataSource("HelpDesk");
    PropertySheet prop = ds.getPropertySheet();
    prop.setProperty("user", "scott");
    prop.setProperty("password", "tiger");
    Connection c = ds.open();
    Rowset rowset =
        c.getData(c.getSchemaObject(new String[] { "SCOTT", "EMP"}));
    // print headings
    for (int i = 0; i < rowset.getFieldCount(); i++)
        System.out.print(rowset.getField(i).getName()+ ",");
    System.out.println("");
    while (rowset.next()) {
        // print data
        for (int i = 0; i < rowset.getFieldCount(); i++)
            System.out.print(rowset.getField(i) + ",");
        System.out.println("");
    }
    c.close(); // close the connection to the data source
} catch (Exception e) {
    e.printStackTrace();
}
```

The example uses `getData()` with a reference to the object. `getSchemaObject()` of connection `SCOTT`, followed by `EMP`, defines the path to the object. The interpretation of this path is specific to the data source. Assuming that the data source has a flat set of schemas containing tables, the path would be a schema `SCOTT` with a table `EMP`. The `getData()` method will return a Rowset interface with all the data in the EMP table. This is equivalent to executing the command `SELECT * FROM SCOTT.EMP`.

## Processing Results Using the Rowset Interface

Production Reporting DDO does not prescribe the implementation of a Rowset. This gives the data source driver great flexibility in handling data while providing the application a consistent interface.

In particular, the driver is free to implement just-in-time retrieval. This means that the Rowset does not actually hold all the data. The driver can fetch a record when the application calls the `next()` method. The benefit of this behavior is that the driver does not need to hold the entire result set in memory and can handle very large result sets.

The Rowset is self-describing. Using `getFieldCount()`, you can determine the number of fields in the Rowset. Each field is an instance of the abstract Field class. A Field object describes its type, size, and structure. In particular, a Field can be a complex structure, such as a Row or Rowset.

The benefit of Rowset being self-describing is that an application does not need to hardcode the expected type and sizes of each field. It can discover these attributes at run-time. *Program ex9.sqr* demonstrates how the datatype of the fields can be determined while processing a Rowset.

### Program ex9.sqr

```
static void printRowset(Rowset rs) throws DataAccessException {
   // write results in an HTML table
   System.out.println("<table><tr>");
   int fieldCount = rs.getFieldCount();
   // DateFormat object for printing dates
   DateFormat df = DateFormat.getDateInstance(DateFormat.LONG);
   // print headings
   for (int i = 0; i < fieldCount; i++)
       System.out.println("<td>" + rs.getField(i).getName() + "</td>");
   // print the data row by row
   while (rs.next()) {
      System.out.println("</tr><tr>");
      for (int i = 0; i < fieldCount; i++) {
         Field f = rs.getField(i);
         if (f.isNull()) {
            System.out.println("<td> </td>");
            continue;
         }
         switch (f.getType()) {
            case Field.Boolean:
            case Field.Text:
               System.out.println("<td align=left>" + f + "</td>");
               break;
            case Field.Number:
               if (f instanceof DecimalField &&
                    ((DecimalField)f).isCurrency())
                  System.out.println("<td align=right>$" + f + "</td>");
               else
                  System.out.println("<td align=right>" + f + "</td>");
               break;
            case Field.Date:
               System.out.println("<td align=left>"
                      + df.format(((DateField)f).dateValue()) + "</td>");
```

```
                break;
            case Field.Row:
            case Field.Rowset:
            case Field.Object:
            case Field.Binary:
                System.out.println("<td align=center>n/a</td>");
                break;
        }
    }
  }
  System.out.println("</tr></table>");
  rs.close(); // close the result set
}
```

`printRowset()` in the example takes a Rowset as an argument. It has no knowledge of how the Rowset was obtained. The Rowset can be the result of `getData()`, the result of executing a command such as an SQL statement, or it could be a result set from an execution of a stored procedure. In all cases, the Rowset is processed in the same manner.

The Rowset maintains a current row with a fixed number of fields. A call to the `next()` method of the Rowset will populate the current row with the next row of data. `next()` returns a boolean value of true for as long as records are available. Calling `next()` after the last row returns `false`.

For each row, process each field. Start by checking the value for null using `isNull()`. If the field is null print an empty cell and proceed to the next field.

Check the field type using the following statement:

```
switch (f.getType()) {
```

`getType()` returns an integer that matches one of the constants that are defined in the Field class. The example takes different actions depending on the type of the field, although you could simply print all fields as strings using the `toString()` method of the Field class.

Table 8 summarizes field types.

**Table 8** Field Types

| Type | Description |
|---|---|
| Boolean | True/false value. The field is an instance of BooleanField. |
| Binary | Array/stream of bytes. The field is an instance of BinaryField. It can also be an instance of LongBinaryField. |
| Date | java.util.Date value. The field is an instance of DateField. |
| Number | Numeric value. The field is an instance of a subclass of the abstract NumberField class. This means that the field is an instance of IntegerField, DoubleField, or DecimalField. |
| Object | Arbitrary Java Object. The field is an instance of ObjectField. |
| Row | Instance of RowField. |
| Rowset | Instance of RowsetField. |
| Text | String: instanc of TextField. |

## Selecting and Filtering

Using `getData()` to retrieve data from an object is very simple. Often, too simple. To allow for a more selective retrieval without requiring the data source to support a full-blown command language, Production Reporting DDO introduces the concept of a Selector.

A Selector specifies the columns that an application wishes to retrieve (rather than unconditionally retrieving all the columns) as well as simple selection criteria. In version 1 of Production Reporting DDO, only limited filtering capabilities are provided. These are discussed in "Obtaining Hierarchical and Multidimensional Data " on page 47.

For now, let's see how to use a Selector to pick the desired columns.

```
DataSource ds = DataSourceManager.getDataSource("CSVFiles");
Connection c = ds.open();
Selector selector = new Selector();
selector.setObject(c.getSchemaObject("Employee.csv"));
selector.includeColumn("Name");
selector.includeColumn("Salary");
selector.includeColumn("HireDate");
Rowset rs = c.getData(selector);
printRowset(rs);
```

In this example you use the Production Reporting DDO CSV Access driver. This driver allows access to CSV (comma separated values) files. While this driver does not support SQL, it does support the selector interface.

The code example above demonstrates how we use the Selector class to specify the data desired. Start by instantiating an empty selector. Then set the Object attribute to the desired CSV file. Pass a SchemaObject to `setObject()`. Obtain this object using the `getSchemaObject()` method of the Connection interface. Note that the Registry already contains information for this data source. The information includes a starting disk folder. This defined the default schema for this object. In other words, the code requests data from an object named `Employee.csv` that resides in the default schema.

Additionally, the example requests the columns that should be included with the `getData()` call and passes this selector. Note the requirement to specify the object before specifying the columns so that the Selector is able to look up the columns. The result is a Rowset containing the three fields requested.

The Selector class provides an SQL-like syntax for making the same request. This is demonstrated in the example code that follows. By passing the select statement on the Selector constructor, we have defined both the desired object name and the desired columns. We also pass the Connection interface to allow the Selector to look up objects. Note the use of double quotes around column or object names that can contain spaces or a dot.

```
Selector selector = new Selector(c,
   "select Name, Salary, HireDate from \"Employee.csv\"");
```

# Executing Commands

If the data source supports the command interface, then an application can send command statements to the data source for execution. This is most powerful for data sources that support

rich command language, such as SQL. Using a language, you can specify complex queries that include data selection, aggregation, and composition. For example, if your database supports SQL, you can perform joins and group and sort the results.

Remember that you can check that a data source supports the command interface by checking for that capability. Please refer to "Discovering Capabilities" on page 28 for how to check for a specific capability.

A command can be parameterized and your application can supply values for these parameters at run time. This is demonstrated in the following example.

```
Connection c = ds.open();
   Command command = new Command("select empno, ename, hiredate from " +
                  "emp where deptno = ?");
   command.setParameter(0, new IntegerField(20));
   Rowset rs = c.execute(command);
```

The example constructs a Command object with an SQL statement. Note the use of the ? symbol as a marker for a parameter in the statement. Passing a Field object to the `setParameter()` method of the Command class supplies a value to the parameter. Actually, the `setParameter()` method belongs to the ParameterList class from which Command is derived.

The `setParameter()` method takes two arguments. The first is the parameter number, zero being the first. The second argument is a Field with a value for this parameter. In this example we construct an IntegerField with a value of 20 for department 20. Using a Field object to supply the value is most useful when you bind the result of one command as a parameter to another command.

The `execute()` method of the Connection interface executes the command. The method returns a Rowset object. The Rowset is processed as described in the "Processing Results Using the Rowset Interface" on page 40.

A command can return a single row or even a single value. Production Reporting DDO still returns a Rowset; however, the Rowset can have a single row and a single field. For example, consider an SQL Update statement. The only information returned from an Update is the number of database rows that were effected by the update. Here is the code:

```
Connection c = ds.open();
   Command command = new Command("update emp set sal = sal * 1.1");
   Rowset rs = c.execute(command);
   rs.next();
   System.out.println(rs.getField("COUNT") + " records were updated.");
   c.close(); // close the connection to the data source
```

This example executes an Update SQL statement. This statement does not return data, but it does return a row count for the number of rows processed. The Production Reporting DDO JDBC Access driver will return the row count in a Rowset that has a single row and a single field. The field is named `COUNT`. The call to `getField(COUNT)` retrieves that field and then we print it. This variant of `getField()` locates a field by name. A faster way to get a field is by position number where zero is the first field. Alternatively, we could code the print statement as follows:

```
System.out.println(rs.getField(0) + " records were updated.");
```

### Note About Database Cursors

The Production Reporting DDO JDBC Access driver uses database cursors in a way that is completely transparent to the application. The driver maintains a pool of cursors (JDBC PreparedStatement objects). Executing a command causes the driver to check if a cursor already exists for this command. The cursor pool improves performance by avoiding repeated preparatory <<?>> operations of the same SQL statement.

Production Reporting DDO expects that drivers implement such performance optimizations in a manner that is transparent to the application. This is important for delivering excellent performance without cluttering the API with data source-specific methods such as cursor management methods.

## Calling Procedures and Processing Call Results

The `call()` method of the Connection interface executes a procedure. An application will pass an argument that identifies the procedure and can optionally pass a parameter list. The parameter list holds values for the input parameters of the procedure. Each parameter in the list is an object of type Field. This is useful when you want to pass a field that you just retrieved from the data source as an input parameter to the procedure call.

If the value to pass as a parameter is not a Field, you will need to construct a Field to hold that value. Production Reporting DDO provides several methods for constructing a field.

- An application can use the "new" operator to construct a field such as BooleanField, BinaryField, or DateField. Many of these methods can construct a field and set its value at once.
- An application can use the static methods of the ParameterFactory class to create Fields and supply them with the actual value.

Once you construct the parameter list, you can perform the call() method and obtain the results, as shown in *Program ex10.sqr*.

### Program ex10.sqr

```
public static void main(String[] args) {
   try {
      DataSource ds = DataSourceManager.getDataSource("Sales");
      Connection c = ds.open();
      // prepare the parameter list
      ParameterList params = new ParameterList(new Field[] {
                new DateField("1/1/98"), new DateField("12/31/98") });
      // call the procedure
      SchemaProcedure proc = c.getSchemaProcedure("Sales by Year");
      CallResults results = c.call(proc, params);
      // process all rowsets
      Rowset rs;
      while ((rs = results.getOutputRowset()) != null)
         printRowset(rs);
      // check for return value
      Row retval = results.getReturnValue();
      if (retval != null) printRow(retval);
```

```
      // check for output parameters
      Row outparams = results.getOutputParams();
      if (outparams != null) printRow(outparams);
      // close the call results
      results.close();
      c.close(); // close the connection to the data source
   } catch (Exception e) {
      e.printStackTrace();
   }
}
static void printRow(Row row) throws DataAccessException {
   for (int i = 0; i < row.getFieldCount(); i++) {
      Field f = row.getField(i);
      System.out.println("  " + f.getName() + ": " + f);
   }
}
```

**Code**

```
ParameterList params = new ParameterList(new Field[] {
                    new DateField("1/1/98"), new DateField("12/31/98") });
         // call the procedure
```

**Description**

Constructs the parameter list after connecting to the Sales data source. Assumes that the procedure to be called takes two date input parameters. Constructs a ParameterList object by passing an array of fields to the constructor. The fields hold the actual values for the parameters for this procedure call.

**Code**

```
SchemaProcedure proc = c.getSchemaProcedure("Sales by Year");
```

**Description**

Locates the procedure and obtains a SchemaProcedure object for it. Uses the `getSchemaProcedure()` method of Connection. If a procedure with this name does not exist, the method will throw a `DataAccessException`.

**Code**

```
CallResults results = c.call(proc, params);
// process all rowsets
Rowset rs;
```

**Description**

Makes the call and obtains a `CallResults` object. The `CallResults` allows the application to obtain all the data that is returned from the procedure. This includes multiple result sets, output parameters, and a return value. Note the processing of the result sets before obtaining the values for the output parameters and the return value.

**Code**

```
while ((rs = results.getOutputRowset()) != null)
printRowset(rs);
// check for return value
```

**Description**

Processes the result sets. In general, a procedure can return any number of result sets. It can return no result sets, it can return a single result set, or it can return multiple result sets. The example makes repeated calls to the `getOutputRowset()` method of the `CallResults` class until there are no more result sets.

**Code**

```
Row retval = results.getReturnValue();
if (retval != null) printRow(retval);
// check for output parameters
```

**Description**

Processes the return value. `CallResults` returns this value as a Row. This is useful when the value being returned is a structure. If you know that your procedure will return a scalar value (a single field), then you can use the following code instead:

```
Row retval = results.getReturnValue();
if (retval != null)
System.out.println("return value: " + retval.getField(0));
```

**Code**

```
Row outparams = results.getOutputParams();
```

**Description**

Processes output parameters. `CallResults` returns output parameters as a row in which each field represents a single output parameter. The ordering of the fields corresponds to the order of the output parameters of the procedure.

**Code**

```
results.close();
```

**Description**

Closes the `CallResults` object. This signals the driver that the execution context of this procedure call can be released.

## Performing Transactions

Business Intelligence applications do more than read data. They often update request tables, log tables, status fields, and more. Moreover, an application can stage data into intermediary storage to perform multiple passes and generate multiple reports.

Production Reporting DDO supports update activity to the database in several ways:

- Executing a command via the `execute()` method of Connection can perform any operation on the data source. In particular, it can create new objects and populate them with data.
- Calling a procedure via the `call()` method of Connection imposes no limit to what a procedure can do. The data source can allow the user to call procedures that update data and manipulate objects.
- To group data changes into transactions, Production Reporting DDO provides the transaction interface. If the data source supports transactions, the call to the `getTransaction()` method of the Connection interface returns a Transaction interface. Using this interface, an application can start a transaction and complete a transaction with either a commit or a rollback.

*Program ex11.sqr* is a simple example. In this example, "c" is a Connection reference for a valid connection to the HelpDesk Oracle database.

### Program ex11.sqr

```
try {
   c.getTransaction().beginTransaction();
   c.execute(
      new Command("update emp set sal = sal * 1.1 where job = 'CLERK'"));
   c.execute(
      new Command("update emp set sal = sal * 1.1 where job = 'MANAGER'"));
   c.getTransaction().commit();
   c.close();
   } catch (DataAccessException e) {
      try {
         if (c != null) c.getTransaction().rollback();
      } catch (Exception e2) {
         e2.printStackTrace();
      }
      e.printStackTrace();
}
```

The example starts by calling `beginTransaction()` on the transaction interface. This call is always required. It then executes two update statements. If an error occurs during the updates, then we catch an exception and rollback all the changes by calling the `rollback()` method of the Transaction interface. Otherwise, if the two updates are successful, we call the `commit()` method. In this example, we did not have to call `commit()` because the `close()` method on the connection will also commit any pending transaction.

## Obtaining Hierarchical and Multidimensional Data

Production Reporting DDO directly supports multidimensional databases (also called OLAP servers). These databases organize data to support multi-level aggregation and analysis. They define a data set—also called a hypercube—in terms of multiple dimensions. Each dimension represents a key aspect of the data. For example, in sales, data dimensions typically include product, territory, organization units, and time. These represent what was sold, where it was sold, who sold it, and when. Sales data can therefore be analyzed along these dimensions.

Each dimension typically defines a hierarchical structure. This is key for data aggregation. For example, territory can define a hierarchy or geographical regions. At the top of the hierarchy, data is summarized for all regions. Going one level down, for example, can divide the world into North America, Europe, and so on. North America can further be divided into countries and then states.

Table 9 shows the Production Reporting DDO mapping of multidimensional concepts to Production Reporting DDO objects.

**Table 9 Mapping Multidimensional Concepts to Production Reporting DDO Objects**

| Concept | Production Reporting DDO Objects |
|---|---|
| Hypercube | Data object (SchemaObject). |
| Dimension | Column (SchemaObjectColumn). |
| Dimension Hierarchy | Hierarchy of SchemaObjectColumn object. At the top of the hierarchy there is a SchemaObjectColumn object for each dimension. A getChildren() call on this column returns the first-generation members of that dimension hierarchy. A getChildren() call on a first-generation member returns second-generation members. You can continue down the hierarchy until getChildren() returns null.<br>You can also obtain information about levels and generations using the MDSchemaObject interface. |
| Measures | Numeric column (SchemaObjectColumn). |

Using Production Reporting DDO, your application can "walk" the dimension and discover all the members, generations, and levels, as shown in *Program ex12.sqr*.

## Program ex12.sqr

```
import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.Enumeration;
public class test19 {
 public static void main(String[] args) {
    try {
       DataSource ds = DataSourceManager.getDataSource("Essbase");
       PropertySheet prop = ds.getPropertySheet();
       prop.setProperty("user", args[0]);
       prop.setProperty("password", args[1]);
       Connection c = ds.open();
       Enumeration enum = c.getAllSchemasObjects().elements();
       while (enum.hasMoreElements()) {
          SchemaObject cube = (SchemaObject)enum.nextElement();
          listCube(cube);
       }
       c.close();
    } catch (Exception e) {
       e.printStackTrace();
    }
 }
 static void listCube(SchemaObject cube) throws DataAccessException {
    System.out.println("----------------------");
    System.out.println(cube.getName());
    System.out.println("----------------------");
```

```
    SchemaObjectColumns cols = cube.getMetaData().getSchemaObjectColumns();
    listColumns(cols, 1);
 }
 static void listColumns(SchemaObjectColumns cols, int level) {
    for (int i = 0; i < cols.size(); i++) {
       SchemaObjectColumn col = (SchemaObjectColumn)cols.elementAt(i);
       indent(level);
       printCol(col);
       SchemaObjectColumns children =
                                (SchemaObjectColumns)col.getChildren();
       if (children != null)
          listColumns(children, level + 1);
    }
 }
 static void printCol(SchemaObjectColumn col) {
    String name = col.getName();
    String desc = col.getDesc();
    if (desc != null && desc.length() > 0)
       System.out.println(name + " (" + desc + ")");
    else
       System.out.println(name);
 }
 static void indent(int level) {
    while (level-- > 0)
       System.out.print("    ");
 }
}
```

*Program ex12.sqr* begins by obtaining a list of all the data objects in this data source using the `getAllSchemasObjects()` method of the Connection interface. Then it goes through the list and call `listCube()` for each data object. Remember that each data object in a multidimensional database represents a hypercube.

The `listCube()` method prints the name of the hypercube and then calls listColumns() to lists the columns of the hypercube. Each column represents a dimension except for the last column that represents the numeric data.

The `listColumns()` method is a recursive method that recurse through the hierarchy of members within a dimension. The child members of a dimension or a member are obtained with a `getChildren()` call. Note that in the case of a SchemaObjectColumn, `getChildren()` will always return SchemaObjectColumns.

The `printCol()` method prints the name of each column. This is the name of the dimension or member of the dimension. Note that some multidimensional databases use the name of the member as a unique identifier. The member can also have an alias that is more suitable for display in a report and can use a localized language. If such an alias is available, you will find it in the description attribute of the column (see `col.getDesc()`). If a description is available, `printCol()` will display it along with the column name.

So far, you have seen how to get metadata for a multidimensional database. Next, you will see how to retrieve the data.

Production Reporting DDO applications do not have to be familiar with multidimensional concepts or even be aware that the data source is multidimensional. Therefore, Production Reporting DDO provides two modes for data retrieval:

- Regular Selector for an application that is not "multidimensional aware."
- Multidimensional Selector (MDSelector) for an application that is "multidimensional aware."

## Retrieving Multidimensional Data Using a Regular Selector

The previous section showed that listing the objects in a multidimensional data is exactly the same as listing the objects in any data source. Moreover, listing the dimensions and measures of a hypercube object is identical to listing columns of a table object.

Therefore, an application can retrieve data from a multidimensional data source by simply naming an object in a `getData()` call, or constructing a Selector object that names the hypercube object and includes selected dimensions using the `includeColumn()` method of Selector.

In both cases, Production Reporting DDO obtains the data as a rowset in which every dimension and measure is a field. The data is down to the lowest level (level 0) and there is a row for every intersection of the given dimensions (every cell in the hypercube). If you use a selector to pick specific dimensions, then the data is summarized across the other dimensions, as shown in *Program ex13.sqr*.

### Program ex13.sqr

```
import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.Enumeration;
public class test21 {
   public static void main(String[] args) {
      try {
         DataSource ds = DataSourceManager.getDataSource("Essbase");
         PropertySheet prop = ds.getPropertySheet();
         prop.setProperty("user", args[0]);
         prop.setProperty("password", args[1]);
         Connection c = ds.open();
         SchemaObject cube =
               c.getSchemaObject(new String [] { "Sample", "Basic" } );
           Selector selector = new Selector();
         selector.setObject(cube);
         selector.includeColumn("Year");
         Rowset rs = c.getData(selector);
         printRowset(rs);
         c.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   static void printRowset(Rowset rs) throws DataAccessException {
      int fieldCount = rs.getFieldCount();
      // print the data row by row
      for (int rowCount = 0; rs.next(); rowCount++) {
         StringBuffer line = new StringBuffer();
         for (int i = 0; i < fieldCount; i++) {
             line.append(rs.getField(i).toString());
             line.append('\t');
```

```
        }
        System.out.println(line);
    }
    rs.close();  // close the rowset
}
```

**Code**

```
Selector selector = new Selector();
```

**Description**

Creates a Selector object and set its object to the hypercube and pick one dimension called "Year."

And here is the output:

```
Jan     8024.0
Feb     8346.0
Mar     8333.0
Apr     8644.0
May     8929.0
Jun     9534.0
Jul     9878.0
Aug     9545.0
Sep     8489.0
Oct     8653.0
Nov     8367.0
Dec     8780.0
```

In this example, level 0 of the "Year" dimension is the month. Since this is the only dimension selected, the data is summarized along this dimension.

## Retrieving Multidimensional Data Using MDSelector

When retrieving data from a multidimensional database, there are several things to specify:

- Dimensions and measures to include.
- Members to include. This is the multidimensional way of restricting the data to a subset. For example, to get the data from a single month, you can select the "March, 1999" member of the time dimension.
- Level of aggregation. For example, you may want the data summarized into weekly number, monthly numbers, or quarterly numbers. The weeks, months, and quarters are levels in the time dimension hierarchy.

Production Reporting DDO supports these kind of selections using a specialized kind of a Selector object called a MDSelector. The MDSelector class is a subclass of Selector. It builds on the Selector class's capability to select an object and columns. It adds the ability to select members and level of aggregation.

Here is an example.

```
SchemaObject cube =
       c.getSchemaObject(new String [] { "Sample", "Basic" } );
MDSelector selector = new MDSelector();
```

```
selector.setObject(cube);
selector.includeColumn("Year");
selector.includeColumn("Product");
selector.setColumnLevel(0, 1);
selector.setColumnGeneration(1, 2);
Rowset rs = c.getData(selector);
printRowset(rs);
```

This example instantiates a MDSelector instead of a Selector. We `setObject()` to our hypercube (`Basic` in schema `Sample`) and select the `Year` and `Product` dimensions. So far, this is no different than selector. Now, we choose the level of aggregation. The statement `setColumnLevel(0, 1)` sets the level for the first column (column 0—Year) to be level 1. This means one level higher than the most detailed level. If you think of the dimension hierarchy as a tree, then level 0 is the leaves of the tree and level 1 is their immediate parents. The statement `setColumnGeneration(1, 2)` sets the level for the second column (column 1—Product) to be generation 1.

Specifying the aggregation in terms of generation is useful, especially if the hierarchy tree is not balanced with some leaves deeper than others. Generation 1 is the top, generation 2 is the immediate children of the root of the tree.

And here is the output:

```
Qtr1    100    7048.0
Qtr1    200    6721.0
Qtr1    300    5929.0
Qtr1    400    5005.0
Qtr1    Diet    7017.0
Qtr2    100    7872.0
Qtr2    200    7030.0
Qtr2    300    6769.0
Qtr2    400    5436.0
Qtr2    Diet    7336.0
Qtr3    100    8511.0
Qtr3    200    7005.0
Qtr3    300    6698.0
Qtr3    400    5698.0
Qtr3    Diet    7532.0
Qtr4    100    7037.0
Qtr4    200    7198.0
Qtr4    300    6403.0
Qtr4    400    5162.0
Qtr4    Diet    6941.0
```

Now suppose that we only wanted to see the first two quarters. Moreover, we only want to see product groups 100, 200, 300, and 400 ("Diet" is a grouping of products that are already counted under 100, 200, 300, or 400). To restrict a selection to specified members, Production Reporting DDO allows the use of the MDSelector `setColumnMembers()` method.

Here is an example:

```
SchemaObject cube =
     c.getSchemaObject(new String [] { "Sample", "Basic" } );
MDSelector selector = new MDSelector();
selector.setObject(cube);
selector.includeColumn("Year");
```

```
selector.includeColumn("Product");
selector.setColumnLevel(0, 1);
selector.setColumnGeneration(1, 2);
selector.setColumnMembers(0,
          new String[] { "Qtr1", "Qtr2" });
selector.setColumnMembers(1,
          new String[] { "100", "200", "300", "400" });
Rowset rs = c.getData(selector);
printRowset(rs);
c.close();
```

We used the version of the `setColumnMembers()` method that takes member names as strings. This assumes that member names are unique. Otherwise, the application must obtain the outline metadata for the dimension and pass members to the `setColumnMembers()` method as SchemaObjectColumn objects.

And here is the output:

```
Qtr1    100    7048.0
Qtr1    200    6721.0
Qtr1    300    5929.0
Qtr1    400    5005.0
Qtr2    100    7872.0
Qtr2    200    7030.0
Qtr2    300    6769.0
Qtr2    400    5436.0
```

# 3 Writing a Production Reporting DDO Driver

In This Chapter

About Writing a Driver ..... 55
Summary of Steps ..... 55
Step 1: Create the Properties, Capabilities, and Message Files ..... 56
Step 2: Implement the DataSource Interface ..... 59
Step 3: Implement the Connection Interface ..... 60
Step 4: Implementing Rowset ..... 69

## About Writing a Driver

This chapter shows how to write a driver that provides access to data in flat files stored in a folder. The data in the files, commonly-called *CSV* files, is comma-delimited and values are optionally enclosed in quotes. The first line in the file provides the names of the fields.

Any Production Reporting DDO application can use the Production Reporting DDO CSV driver that we develop. To the application, the CSV data source would look just like any other data source.

## Summary of Steps

➤ To write an Production Reporting DDO driver:

1. **Create the necessary properties, capabilities, and message files.**
2. **Implement the DataSource interface.**
3. **Implement the Connection interface.**
4. **Write code that implements the Rowset interface for data returned by the driver.**
5. **If the data source supports procedure calls, implement the call results interface.**
6. **If the data source supports transactions, implement the transaction interface.**

The example in this chapter shows the first four steps. The CSV driver will not implement procedure-calling or transactions.

# Step 1: Create the Properties, Capabilities, and Message Files

Begin by deciding on a package name for the driver implementation. This is important because Production Reporting DDO uses class names to locate its property files. The name of the sample package is `demo.csv`. The DataSource implementation class is `CSVDataSource`.

## Creating the Properties Files

Based on the package name and the DataSource class name, the properties files take the names shown in Table 10.

**Table 10** Names of the Properties Files

| File Type | File Name |
|---|---|
| Property values | demo_csv_CSVDataSource_Properties.properties |
| Property descriptions | demo_csv_CSVDataSource_PropertyDescriptions.properties |

Production Reporting DDO allows a driver to inherit properties and property descriptions from Production Reporting DDO by implementing the inheritance using the class and interface hierarchy. Specifically, inherited properties are from the com_sqribe_access_DataAccess_Properties.properties and com_sqribe_access_DataAccess_PropertyDescriptions.properties files. These files define the logon property and its user and password properties. These files reside in a folder called properties on the Java CLASSPATH.

We create an empty file for the property values because the driver has no special properties.

For the property description, we modify the logon property. The default logon property has two properties associated with it, `user` and `password`. Our example logon has none because it will access CSV files on a local drive. Here is the property descriptions file (`demo_csv_CSVDataSource_PropertyDescriptions.properties`):

```
# override the default logon property
logon.Name=Logon
logon.Description=Logon properties.
logon.Indices=
logon.Required=false
```

`Setting logon.Indices` to none eliminates the logon attributes.

## Creating the Capabilities Files

Based on the package name and the DataSource class name, the capabilities files take the names shown in Table 11.

**Table 11** Names of the Capabilities Files

| File Type | File Name |
|---|---|
| Capability values | demo_csv_CSVDataSource_Capabilities.properties |
| Capability descriptions | demo_csv_CSVDataSource_CapabilityDescriptions. properties |

The driver inherits the contents of the corresponding files in Production Reporting DDO:

```
com_sqribe_access_DataSourceAccess_Capabilities.properties
com_sqribe_access_DataSourceAccess_CapabilityDescriptions.properties
```

These files define common capabilities such as interfaces, command, call, selector, and transaction.

Since the driver inherits these capability descriptions, the capability descriptions file is empty. The driver also inherits default values for these capabilities. For now, we leave the capabilities file empty as well.

## Creating the Messages File

The driver can throw DataAccessExceptions with specific error messages and it can have other localizable text. Leveraging the facilities provided with Production Reporting DDO, we can let Production Reporting DDO ook up the message. We can reuse messages that are stored in the Production Reporting DDO essage files. Moreover, we can use the interactive tools that come with the Production Reporting DDO SDK to manage the message file.

The Production Reporting DDO message facility uses the class hierarchy to locate messages. It works by looking for a message file for the class that calls it. If the message file is not found, or if the specific message is not found in that file, then it goes on to look in the message files that correspond to super-classes of the specified class. It also looks at interfaces that the class implements and follows that hierarchy as well.

One file can store all the messages for all the classes in a driver when all the classes implement the same interface. This common interface can extend the Production Reporting DDO Access and Util interfaces, such that if a message is not found in the driver's file, then Production Reporting DDO will look for the message in its own files. In the section "The CSV Interface" on page 58 that follows, we create this common interface.

The message file is `demo_csv_CSV.properties` and is located in a `msgs` folder on the Java CLASSPATH. The message file contains entries for messages that are specific to the CSV driver. *Program ex14.sqr* shows an example of the file:

### Program ex14.sqr

```
# The number of column headings doesn't match the number of data columns
# 0 class name
# 1 method name
# 2 heading count
# 3 data count
HeadingCountMismatch.text={0}.{1}(): The number of heading columns, \
```

```
{2}, does not match the number of data columns, {3}.
# End of file encountered while looking for end quote
# 0 class name
# 1 method name
# 2 "file" name
# 3 line where quoted string started
MissingQuote.text={0}.{1}(): End of file encountered while \
searching for ending quote; file= {2}, line={3}.
# Unknown column name
# 0 class name
# 1 method name
# 2 column name
InvalidColumnName.text={0}.{1}(): Unknown column name, \
{2}, in selector specification.
```

The file has the same format as property files. The lines that begin with "#" are comments. The entries have the form `<message id>.text`. The Message ID is a unique key used in the code to identify a message. The Message ID is not displayed and does not need to be localized. The message text may contain special markers for variables. These markers are denoted by `{0}`, `{1}`, and so on. By convention, every message will have at least the first two markers. The markers will hold the value of the class name and method name where the message is generated. For a long message broken over multiple lines, place a backslash at the place where the line breaks.

The implementation code that we develop in the example in this chapter uses the messages in the file.

## The CSV Interface

Production Reporting DDO uses the class hierarchy to locate messages and properties. It works by walking up the class hierarchy, looking at the message or properties file that corresponds to each class and continuing up if the file or the entry is not found. It also walks up the interface hierarchy for the interfaces that your class implements.

The first Java file in our driver is the CSV interface, shown in *Program ex15.sqr*. This interface is empty, simply marking the classes that implement it in order to keep track of messages and properties. The CSV interface implements the Production Reporting DDO Access interface. This implementation directs the message and property facilities to look at the Production Reporting DDO files if these do not find a message or property in your driver's files.

### Program ex15.sqr

```
package demo.csv;
import com.sqribe.access.*;
/**
 * CSV is a holder interface, allowing properties, messages, etc,
 * common to the csvacc package to be specified once, e.g.,
 * demo_csv_CSV.properties.
 * @see com.sqribe.comutil.PropertySheet
 * @see com.sqribe.comutil.PropertyDescription
 * @see com.sqribe.comutil.Msg
 */
public interface CSV extends Access {
}
```

# Step 2: Implement the DataSource Interface

In the section "Managing Data Sources," you saw the attributes that define a data source: name, description, class, required Java and native libraries, and the connection string. For CSV, the class attribute will be `demo.csv.CSVDataSource`. This is the name of the DataSource implementation in our example. The lib and load attributes are empty, since we don't need to load any Java or native libraries for the CSV driver. Finally, we define the connection string attribute to point to a folder on the disk where the CSV files are found. Here is an example for a registry entry for a data source "DataFiles" that we will use:

```
DataFiles.desc=Current and historical sales data
DataFiles.class=demo.csv.CSVDataSource
DataFiles.lib=
DataFiles.load=
DataFiles.conn=c:\\data
```

*Program ex16.sqr* is the start of the implementation code:

### Program ex16.sqr

```
package demo.csv;
import com.sqribe.access.*;
import com.sqribe.comutil.*;
public class CSVDataSource extends DataSourceAdapter implements CSV {
   private static final String classname = CSVDataSource.class.getName();
   public CSVDataSource(String pName, String pDesc, String pConn) {
      super(pName, pDesc, pConn, classname);
   }
   public Connection open() throws DataAccessException {
      String folder = getConnectString();
      AccessIO io = AccessIO.createAccessIO("File", folder);
      return new CSVConnection(io, getPropertySheet());
   }
}
```

The CSVDataSource class implements the DataSource interface by extending the DataSourceAdapter class. The DataSourceAdapter class is an abstract class that provides implementation for most of the DataSource interface. We only need to implement the constructor and the `open()` method. The DataSourceAdapter provides the implementation for all the other methods of the DataSource interface.

The class defines a static data member called `classname`. All the classes in the driver use classname. For example, the message facility uses it to display the class name that generated the message. The constructor does nothing special but simply hands off its arguments to the constructor of its super class.

The `open()` method creates a new connection to our data source. It retrieves the folder name (for example, `c:\data`) from the registry (from the Data Source Manager) by calling the `getConnectionString()` method. This method is part of the DataSource interface and is already implemented for us in DataSourceAdapter.

To access files on the disk, we use the AccessIO class. This class is part of Production Reporting DDO. It provides IO access to the file system as well as other storage systems. In version 1 of

Production Reporting DDO , only file IO is supported, but the same mechanism can be extended to allow access to remote files over HTTP or other transport mechanisms.

To open a connection, we instantiate an AccessIO object using the `File` protocol and a folder name. The `AccessIO.createAccessIO()` method will create a file AccessIO object and validate that the folder is accessible. We pass this object on the constructor to the connection object.

## Step 3: Implement the Connection Interface

The connection interface is the main interface in Production Reporting DDO. We will need to implement several methods, starting with the methods contained in *Program ex17.sqr*.

### Program ex17.sqr

```
package demo.csv;
import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.*;
public class CSVConnection extends ConnectionAdapter implements CSV {
   private static final String classname = CSVConnection.class.getName();
   private AccessIO dir; // folder with CSV files
   private static final String fileTerm = "Table";
   public CSVConnection(AccessIO pDir,
                    PropertySheet pSheet) throws DataAccessException {
      super(pSheet);
      dir = pDir;
      if (dir.isLeaf())
         DataAccessException.rethrow("NotDirectory",
            new Object[] { classname, "CSVConnection", pDir.getName() });
   }
   private void createSchemas() throws DataAccessException {
      Schemas schemas = new Schemas();
      Vector names = dir.listDir();
      for (int idx=0; idx < names.size(); idx++) {
         String name = (String)names.elementAt(idx);
         if (dir.isLeaf(name))
            schemas.add(new SchemaObject(null, name, "", fileTerm, this));
      }
      setSchemas(schemas);
   }
   public Schemas getSchemas() throws DataAccessException {
      createSchemas(); // refresh the list of objects
      return getSchemasRoot();
   }
   public void close() {
      dir.close();
   }
}
```

The CSVConnection class implements the Connection interface by extending ConnectionAdapter. This abstract class provides default implementation for many of the methods of Connection. It also provides useful helpful functions that aid in the implementation of the CSVConnection class.

CSVConnection has a private data member, `dir`, that holds a reference to the AccessIO object that represents the folder on the disk containing the CSV files.

The CSVConnection constructor calls the constructor of ConnectionAdapter. That constructor takes the DataSource property sheet and copies its entries into the connection property sheet. It then validates the folder name where the CSV file exists. That is, it determines that it is not a regular file; thus, it must be a folder.

Here we see an example of throwing a DataAccessException using its static `rethrow()` method. This method re-throws exceptions in the driver (such as database, IO, and network exceptions) and turns them into a DataAccessException. The `rethrow()` method can also throw a new exception, as demonstrated here. The `rethrow()` method takes a message ID as its first argument. The `NotDirectory` message is already defined in Production Reporting DDO. We can use it here without having to define it in your message file.

The next argument to the `rethrow()` method of DataAccessException is an array of objects (typically String objects) that become part of the message. This example includes the name of the CSVConnection class and the `CSVConnection` constructor as well as the name of the folder.

The `createSchemas()` method generates the top-level (root) metadata (Schemas). In the driver, this is the list of the CSV files in the folder. Note that we do not assume that the CSV file has a `.csv` filename extension. Instead, we assume that all the files in the folder are CSV files. The code for a `createSchemas()` example follows.

```
private void createSchemas() throws DataAccessException {
      Schemas schemas = new Schemas();
      Vector names = dir.listDir();
      for (int idx=0; idx < names.size(); idx++) {
         String name = (String)names.elementAt(idx);
         if (dir.isLeaf(name))
            schemas.add(new SchemaObject(null, name, "", fileTerm, this));
      }
      setSchemas(schemas);
   }
```

The example allocates an empty Schemas object. Next, we obtain the list of CSV files in the folder using the `listDir()` method of AccessIO. For each file name in the list, we check that it is a file (and not a folder), then create a SchemaObject instance for it and add it to the Schemas object. When done, we call `setSchemas(schemas)`. The `setSchemas()` method is a method of the ConnectionAdapter class. It registers the top-level Schemas object. This is the object that is returned to the application when it calls `getSchemas()` on the Connection.

We override `getSchemas()` in the code to generate the root Schemas. As implemented, each time the application calls `getSchemas()`, we list the directory again. The effect is that the list of objects in our data source gets refreshed. If a CSV file was added to the folder, a call to `getSchemas()` adds it to the metadata.

The application calls the `close()` method. The ConnectionAdataper implementation of `close()` does nothing. In the case of a folder, `close()` actually does nothing, because there is no file to close. Nevertheless, the example demonstrates this cleanup as a good practice for `close()`. We override it here to close the AccessIO object.

## Providing Column Metadata

Information about columns, their name and type, is derived from the CSV file itself. For that purpose, we write a class called CSVFile. This class encapsulates the implementation of reading a CSV file. It deals with reading lines, dealing with delimiters, obtaining column names from the first line, sampling the data to guess the type of the column by looking at the next 5 lines, and finally reading the data. The source for CSVFile is listed later in this chapter. For now, we focus on the Production Reporting DDO part of things: creating the SchemaObjectColumns and implementing the `getSchemaObjectColumns()` method of Connection.

Before continuing, let's review the methods of the CSVFile class that you will use, as shown in Table 12.

**Table 12 CSVFile Class Methods**

| Method | Description |
|---|---|
| Constructor | Creates a new CSVFile object for this AccessIO. |
| getLineTokens | Peels off the delimiters and returns the items on a line. |
| getSampleData | Returns tokenized data for the first few lines in the file. |

To keep things simple, our driver only supports three data types: date, number, and text. *Program ex18.sqr* adds implementation for `getSchemaObjectColumns()` to the CSVConnection class.

### Program ex18.sqr

```
 public SchemaObjectColumns getSchemaObjectColumns(
       SchemaObject pSchema) throws DataAccessException {
CSVFile csvfile =
      new CSVFile(dir.createAccessIO(pSchema.getName()));
Vector headingsLine = csvfile.getLineTokens(); // Get the headings
Vector dataLine[] = csvfile.getSampleData(); // Sample data
csvfile.close();
int headingsCount = headingsLine.size();
int dataCount = dataLine[0].size();
if (headingsCount != dataCount) { // Must be the same number
   DataAccessException.rethrow("HeadingCountMismatch",
          new Object[] { classname, "createMetaData",
          new Integer(headingsCount), new Integer(dataCount) } );
}
SchemaObjectColumns columnMetaData = new SchemaObjectColumns();
for (int idx = 0; idx < headingsCount; idx++) {
   String name = (String) headingsLine.elementAt(idx);
   int prev = 0, curr = 0;  // type of previous and current value
   int size = 0;            // size of this coulum
   int scale = 0, prec = 0; // precision and scale (numeric)
   for (int i = 0; i < dataLine.length; i++) {
      if (dataLine[i] != null) {
         String sample = (String) dataLine[i].elementAt(idx);
         Field field = getField(sample);
         curr = field.getType();
         if (prev != 0 && prev != curr)
             curr = Field.Text;
```

```
                prev = curr;
                size = Math.max(size, sample.length());
                if (curr == Field.Decimal) {
                   scale = Math.max(scale,
                         ((DecimalField)field).decimalValue().scale());
                   prec = size;
                }
             }
          }
          SchemaObjectColumn soc = new SchemaObjectColumn(
                      pSchema, name, curr, curr,
                      getDBTypeName(curr),
                      size, prec, scale, "");
          columnMetaData.add(soc);
       }
       return columnMetaData;
    }
```

The Connection interface and the base implementation class define the ConnectionAdaptermethod signature for `getSchemaObjectColumns()`. We override the method to provide appropriate implementation for CSV files.

The purpose of the `getSchemaObjectColumns()` method is to describe the columns of a data object—a CSV file. This includes the number of columns, their name, type, and size.

**Code**

```
CSVFile csvfile =
      new CSVFile(dir.createAccessIO(pSchema.getName()));
```

**Description**

Constructs an AccessIO object for the object represented by the `pSchema` argument. The AccessIO object allows one to read the file. Next, we construct a CSVFile object for this file. The CSVFile object allows parsing of the CSV file. The implementation source-code for the CSVFile class is provided in the section "CSVFile Class" on page 67 later in this chapter.

Now that we have a CSVFile object for the CSV file, we read the heading line (the first line in the CSV file contains the column headings).

**Code**

```
Vector headingsLine = csvfile.getLineTokens(); // Get the headings
```

**Description**

These headings are the column names.

**Code**

```
Vector dataLine[] = csvfile.getSampleData(); // Sample data
        csvfile.close();
        int headingsCount = headingsLine.size();
        int dataCount = dataLine[0].size();
        if (headingsCount != dataCount) { // Must be the same number
```

```
        DataAccessException.rethrow("HeadingCountMismatch",
               new Object[] { classname, "createMetaData",
               new Integer(headingsCount), new Integer(dataCount) } );
    }
```

**Description**

Reads the sample data, which are the first 5 data lines in the file. We use this data to guess the data type of each column. CSV files really do not have type information. Looking at the first few values of each column, we can see if they are all valid dates or numbers. Otherwise, we take the column as text.

We validate that the number of headings matches the number of values in the data lines (we compare the number of headings to the number of items on the first line of data). If they don't match, we throw an exception. The Message ID used here `HeadingCountMismatch` refers to an entry in the message file `demo_csv_CSV.properties`.

**Code**

```
SchemaObjectColumns columnMetaData = new SchemaObjectColumns();
 for (int idx = 0; idx < headingsCount; idx++) {
       String name = (String) headingsLine.elementAt(idx);
       int prev = 0, curr = 0;//type of previous and current value
       int size = 0;            // size of this coulum
       int scale = 0, prec = 0; // precision and scale (numeric)
       for (int i = 0; i < dataLine.length; i++) {
          if (dataLine[i] != null) {
             String sample = (String) dataLine[i].elementAt(idx);
             Field field = getField(sample);
             curr = field.getType();
             if (prev != 0 && prev != curr)
                 curr = Field.Text;
             prev = curr;
             size = Math.max(size, sample.length());
             if (curr == Field.Decimal) {
                scale = Math.max(scale,
                      ((DecimalField)field).decimalValue().scale());
                prec = size;
```

**Description**

Constructs a new SchemaObjectColumns object. This object holds the collection of SchemaObjectColumn objects that describe the columns of our CSV file.

Next, we process the columns one by one, going over the sample data for each column to determine the type of the column. We do this using the `getField()` method. The `getField()` method returns a typed field (DateField, DecimalField, or TextField) using the `getType()` method of the Field class. We compare that type against the type of previous columns. If there is a conflict, then we resolve this column to be a text column. However, if all the values in the column are valid date values, we resolve this column to be a date column. We apply the same rule for columns if all values are valid numeric values.

We use the sample data to determine the size of the column as well as precision and scale for decimal columns.

### Code

```
            SchemaObjectColumn soc = new SchemaObjectColumn(
                        pSchema, name, curr, curr,
                        getDBTypeName(curr),
                        size, prec, scale, "");
            columnMetaData.add(soc);
        }
        return columnMetaData;
```

### Description

Constructs the SchemaObjectColumn for the column name, type, and size. We set the parent attribute of the column to point to the data object (the CSV file SchemaObject), then pass the name and the type. The driver uses the Production Reporting DDO field type as the database type as well. The `getDBTypeName()` method provides a descriptive name for the type. We add SchemaObjectColumn to the SchemaObjectColumns variable.

Next, let's look at the `getField()` and `getDBTypeName()` methods, shown in *Program ex19.sqr*.

## Program ex19.sqr

```
    private Field getField(String sample) {
        Field     field;
        field = new DateField(sample);  // see if valid date value
        if (field.isNull()) {
           field = new DecimalField(sample); // see if valid number
           if (field.isNull())
              field = new TextField(sample); // default is text
        }
        return field;
     }
     private String getDBTypeName(int pType) {
        String typeName;
        switch(pType) {
           case Field.Number:
              typeName = "NUMERIC";
              break;
           case Field.Date:
              typeName = "DATE";
              break;
           default:
              typeName = "VARCHAR";
              break;
        }
        return typeName;
     }
```

The `getField()` method returns a typed field for the given value. It starts by attempting to construct a DateField from the given value. If the value is not a valid date this would fail and return a value of the DateField as null. In such case, try a DecimalField. If that fails, take the value as a TextField. The function returns the typed field object.

The `getDBTypeName()` method returns a name for the type of the field. Production Reporting DDO does not define the DBTypeName. This is at the discretion of the data source. We return `NUMERIC` for a decimal number, `DATE` for a Date field, and `VARCHAR` for anything else.

## Implementing getData

To complete the implementation of the Connection interface in our CSVConnection class, we implement the `getData()` methods. We start by looking at the simpler `getData()` that takes a SchemaObject argument. The following section discusses the implementation of `getData()` with a Selector.

Rowset is an interface for processing a result set. The `getData()` method returns a Rowset interface. The driver must implement this interface. In this example, the CSVRowset implements the Rowset interface for processing a CSV file. Since most of the logic is in the CSVRowset, the `getData()` implementation is easy.

```
public Rowset getData(SchemaObject schema) throws DataAccessException {
   CSVFile csvfile = new CSVFile(dir.createAccessIO(schema.getName()));
   return new CSVRowset(csvfile, getSchemaObjectColumns(schema));
}
```

We construct a CSVFile object for reading the data from the file. The CSVRowset class uses the CSVFile object to read the data. The CSVRowset constructor takes two arguments.

- CSVFile object
- Column metadata (a SchemaObjectColumns) for this object

In "Step 4: Implementing Rowset" on page 69, you will see how the CSVRowset implements the Rowset interface for processing results of `getData()`.

## Implementing getData with Selector

A Selector provides a flexible means of retrieving data from an object. In the current version of Production Reporting DDO, the selector allows the application to specify the desired columns and their order. In future versions of Production Reporting DDO, the selector may specify filtering, sorting, and join criteria as well.

Production Reporting DDO provides a class RowsetFilter that applies a selector to a Rowset to yield a new Rowset. The new Rowset will have the fields specified in the Selector and in the order that is specified in the Selector.

The ConnectionAdapter class provides a default implementation for `getData(Selector)`. The implementation uses the RowsetFilter to apply the selector to the Rowset that the simpler `getData()` returns. The code in ConnectionAdapter looks like this:

```
public Rowset getData(Selector selector) throws DataAccessException {
   return new RowsetFilter(getData(selector.getObject()), selector);
}
```

For the CSV driver, this implementation is satisfactory. Therefore, we do not need to provide an implementation for `getData(Selector)`, as the base implementation will do.

Before proceeding to "Step 4: Implementing Rowset" on page 69 examine the code for the CSVFile class. This class encapsulates reading and parsing a CSV file.

## CSVFile Class

This implementation uses the CSVFile class. The source code is in *Program ex20.sqr*.

### Program ex20.sqr

```
package demo.csv;
import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.Vector;
import java.io.IOException;
public class CSVFile implements CSV {
   private static final String classname = CSVFile.class.getName();
   private AccessIO access;
   private String delimiters = "\",";
public CSVFile(AccessIO pAccess) {
      access = pAccess;
   }
   public Vector getLineTokens() throws DataAccessException {
      String line = readLine();
      Vector tokens = null;
      if (line != null) {
         int start, end;
         tokens = new Vector();
         for (start=0; start < line.length(); start=end+1) {
            end = getDelimiter(line, start);
            if (start != end) {
               String token = line.substring(start, end);
               if (token.length() != 0) tokens.addElement(token);
            }
            if (end < line.length() && line.charAt(end) == '\"') {
               line = line.substring(++end);
               start = 0;
               end = getQuotedString(line, start);
               if (start<=end) {
                  tokens.addElement(line.substring(start, end++));
               } else {
                  DataAccessException.rethrow("MissingQuote",
                        new Object[] { classname, "getLineTokens",
                                       access.getName(), line } );
               }
            } else if (start==end) {
               tokens.addElement(null);
            }
         }
      }
      return tokens;
   }
   public Vector[] getSampleData() throws DataAccessException {
      Vector[] dataLine = new Vector[5];
      int idx;
      for (idx=0; idx<dataLine.length; ++idx) {
```

```
            dataLine[idx] = getLineTokens();
            if (dataLine[idx]==null) break;
         }
         return dataLine;
      }
      private int getDelimiter(String pLine, int pIdx) {
         int idx = pIdx;
         for (; idx<pLine.length(); ++idx) {
            char c = pLine.charAt(idx);
            if (delimiters.indexOf(c) != -1) break;
         }
         return idx;
      }
      private int getQuotedString(String pLine, int pIdx) {
         int idx;
         if ((idx=pLine.indexOf('"', pIdx))==-1) {
            try {
               String line = readLine();
               if (line!=null) {
                  idx = pLine.length();
                  pLine.concat(line);
                  idx = getQuotedString(pLine, idx);
               }
            } catch (DataAccessException e) { }
         }
         return idx;
      }
      private String readLine() throws DataAccessException {
         StringBuffer line = new StringBuffer();
         while(true) {
            int token = read();
            if (token == -1) break;
            if (token=='\r') {
                token = read();
                break;
            }
            line.append((char)token);
         }
         String ret = line.toString();
         if (ret.length()==0) ret = null;
         return ret;
      }
      private int read() throws DataAccessException {
         int token = 0;
         try {
            token = (access.accessReader()).read();
            if (token == -1) access.closeReader();
         } catch (IOException e) {
            DataAccessException.rethrow("IOError",
               new Object[] { classname, "read", e.toString() } );
         }
         return token;
      }
   }
```

# Step 4: Implementing Rowset

The Rowset interface is a key interface in Production Reporting DDO. The way you implement this interface has great implications for the performance of your driver. Before implementing the CSVRowset class, let's review some of the theory behind the Rowset interface and its implications on performance.

- Rowset is an interface. You are expected to provide an implementation that optimizes performance for data retrieval from your data source.
- The Rowset interface supports very large result sets. You should not hold the entire result set in memory unless you can be sure that the result set is a single row or contains very few rows. Production Reporting DDO provides an implementation of Rowset, VectorRowset, that uses a Java Vector. You can use it for the case of a single row or very few rows. However, you should not use the VectorRowset class if you are retrieving a result set.
- An implementation of Rowset can hold off fetching the data until it is actually requested via the `next()` method. Take advantage of this to improve performance by fetching rows just in time. Of course, an application can perform some "fetch ahead" or buffering, but it should not retrieve all the data up front.
- The application using Rowset is not required to fetch all the data. The `close()` method signals that the application will not fetch any results that are still outstanding. Your implementation should respect the `close()` method. If your data source requires that all the data be processed, you can silently skip the data in your driver, rather than force the application to retrieve all the data.
- To minimize object creation, use the same record and the same fields when `next()` is called. Rather than allocate new fields, the driver can respond to `next()` by populating the same fields with new data. If the application wants to hold references to multiple records, it is the application's responsibility to make copies of rows. For most applications, this is not required, so why do all the extra work?

With this in mind, we are ready to review the CSVRowset implementation of the Rowset interface. The example provides methods that implement all the methods of the Rowset interface. We will hold a single row in memory and populate it with new values in the `next()` method.

### Program ex21.sqr

```
package demo.csv;
import com.sqribe.access.*;
import com.sqribe.comutil.*;
import java.util.*;
public class CSVRowset implements CSV, Rowset {
   private static final String classname = CSVRowset.class.getName();
   private CSVFile csvfile;
   private SchemaObjectColumns soc;
   private VectorRow currentRow;
   public CSVRowset(CSVFile pCSVFile, SchemaObjectColumns pSoc)
                                      throws DataAccessException {
      csvfile = pCSVFile;
      soc = pSoc;
      currentRow = new VectorRow();
      allocateFields();
```

```
      csvfile.getLineTokens();  // Position past the headings
}
   public Row getRow() {
      return currentRow;
   }
   public int getFieldCount() {
      return currentRow.getFieldCount();
   }
   public Field getField(int index) throws DataAccessException {
      return currentRow.getField(index);
   }
   public Field getField(String name) throws DataAccessException {
      return currentRow.getField(name);
   }
   public void close() {
      try {
         csvfile.close();
      } catch (Exception e) { }
   }
```

The constructor takes the two arguments shown in Table 13.

**Table 13 Constructor Arguments**

| Argument | Description |
|---|---|
| CSVFile pCSVFile | Provides an abstraction of a CSV file. Allows us to parse the file and obtain the data it holds. |
| SchemaObjectColumns pSoc | Column metadata for this CSV file. Provides column names, size, and type. |

The constructor creates a Row to hold one record using the VectorRow class in Production Reporting DDO. This class provides a simple implementation of a Row that uses a Java Vector to hold the fields. A driver can provide its own implementation of the Row interface.

Next, we allocate the fields based on the column metadata using the `allocateFields()` method. The example reads the first line from the CSV file. This line has the column headings. We don't need that, as we already have this information in the column metadata. We simply skip this first record.

After the constructor, note the `getRow()`, `getFieldCount()`, and `getField()` methods. The implementation of these methods is quite simple. The `close()` method will close the AccessIO object and close the file. The Rowset interface does not allow for errors during `close()` so we silently ignore any errors during the `close()` method.

Now let's see the `next()` method.

```
   public boolean next() throws DataAccessException {
      Vector tokens = csvfile.getLineTokens();
      if (tokens==null) return false;  // end of file reached
      for (int i=0; i < getFieldCount(); i++) {
         String value = (String)tokens.elementAt(i);
         Field f = getField(i);
         if (value.length()==0) f.setNull(true);
         else f.setValue(value);
      }
```

```
            return true;
        }
```

We start by reading a line from the CSV file. We get the line broken into fields according to the number of fields and their order in the file. We then go through the fields. We set the value of appropriate field to either null (if the file has no value for this field) or to the actual value using the `setValue()` method of Field. Note that Field may by a DecimalField, DateField, or TextField, the appropriate `setValue()` will be called (this is the essence of polymorphism).

The `next()` method returns true when you process a record and false when no more records are available and the end of the file has been reached.

Now let's see the `allocateFields()` method.

```
private void allocateFields() throws DataAccessException {
   Enumeration enum = soc.elements();
   SchemaObjectColumn col;
   while (enum.hasMoreElements()) {
      col = (SchemaObjectColumn)enum.nextElement();
      currentRow.addField(Field.createField(col.getName(),
                          col.getFieldType(), col.getSize(), true));
   }
}
```

`allocateFields()` goes through the column metadata and allocates a field for each column. The `createField()` method of the Field class creates a typed field based on the specified field type (second argument). In this case it returns one of DateField, DecimalField, or Text Field. Then add the field to our Row.

# 4 Programming Considerations

In This Chapter


Production Reporting DDO Adapters ..... 73
Driver Organization Tips ..... 73
Messages and Exceptions ..... 75
Properties and Capabilities ..... 77
Internationalization ..... 80
Message Editor ..... 80
Property Editor ..... 83
Testing and Debugging Drivers ..... 84


## Production Reporting DDO Adapters

Production Reporting DDO adapters provide default methods for features that drivers do not implement. They also provide an array of helper methods to perform common functions. For a list of the methods and a description of their functions, see the Production Reporting DDO API javadoc.

## Driver Organization Tips

In many respects, your driver represents a bi-directional gateway. On the one hand, it communicates with the data source. On the other, it presents metadata, results, and execution operations to the application, using the Production Reporting DDO interfaces. The operations translate to invocation sequences recognized by your data source.

- **DataSourceAdapter**—Handles common data source implementation methods.

  When writing your driver, you only need to implement the `open()` method, which creates a connection to the data source using the given properties. Use the methods of this adapter class to provide default behavior.

- **ConnectionAdapter**—Handles common connection implementation methods.

  For simple drivers, you can place most of the operational methods in your Connection class. For complex drivers, the Connection class can become too large and complex. In this case, delegating the functions to specialized worker objects will make the driver easier to understand and maintain. You may want to add specialized worker classes for object or

procedure processing. Or, you may want to handle the formation of the metadata hierarchy in a specific class.

## Registy Editor

Production Reporting DDO provides a number of tools to assist in application deployment. One such tool is the Registry Editor. The Registry Editor is a data-driven application. Driver writers should add connection-specification information for their drivers to enable configuration through the Registry Editor tool.

**Note:**

For detailed information on the Registry Editor, see Chapter 5, "Managing Data Sources."

➤ To add connection specification information:

1 **Add the common name and descriptive name of your driver to the DataSources.drivers property in properties/com_sqribe_access_DataSourceManager_Properties.properties.**

In the property file snippet that follows, we see six drivers identified. The templates for these drivers exist in the corresponding DataSourceManager message file.

- A semicolon (;) separates each pair.
- White space separates the name and description. If the description contains white space, it must be enclosed in quotes ("").
  - Strings to build the property descriptions are held in the message file *com_sqribe_access_DataSourceManager.properties*.
  - The message file contains resource strings (like those used to create labels for UI dialogs) and message text.
  - The resource strings that begin with the names listed in this property create property descriptions from the pseudo property descriptions for class, lib, load, and so on, provided in the *com_sqribe_access_DataSourceManager_PropertyDescriptions.properties* file, as shown in the following example.

```
# These are the names, with their descriptions, that we will use to build
# property descriptions and entries for the driver templates.
# Registry administration tools use these names to provide driver
# configuration information.
See the API documentation for com.sqribe.access.Registry for more information.
#
DataSources.drivers= \
csvacc "CSV driver"; \
essacc "Essbase driver"; \
jdbcacc "JDBC driver"; \
msmdacc "Microsoft ADO MD driver"; \
psacc "PeopleSoft driver"; \
sapr3acc "SAP R/3 driver"
```

2 **Add the template description for your driver to the message file msgs/com_sqribe_access_DataSourceManager.properties.**

The CSV driver template, shown in Code ex4b, mirrors the information required in the registry data source entries. The difference is in the connection string, which has bracketed (< >) entries for each substitution value in the connection string. The label in the brackets is used as the parameter label in the registry editor.

A driver can have multiple connection string templates. These can correspond to multiple lib entries. In the case of the Production Reporting DDO relational database driver, the corresponding entries represent related JDBC driver and connection strings.

```
# CSV driver template
csvacc.name.string=CSV DataSource Template
csvacc.class.string=com.sqribe.csvacc.CSVDataSource
csvacc.lib.string=
csvacc.load.string=
csvacc.conn.string="CSV:File:<Fully Qualified Directory Path Name>"
csvacc.desc.string=This data source represents a directory tree rooted in a file system.
\
Files in the tree having the file extension, ".csv", are interpreted as delimiter \
separated files. These files represent objects to this driver.
#
# JDBC datasource driver template
jdbcacc.name.string=JDBC DataSource Template
jdbcacc.class.string=com.sqribe.jdbcacc.JDBCDataSource
jdbcacc.lib.string=sun.jdbc.odbc.JdbcOdbcDriver \
oracle.jdbc.driver.OracleDriver \
com.sqribe.License.SQRIBE970801Ora \
com.sybase.jdbc.SybDriver \
com.sqribe.License.SQRIBE970801MSsqlSybase
jdbcacc.load.string=
jdbcacc.conn.string="JDBC:ODBC:<ODBC DSN>" \
"jdbc:oracle:oci7:@<Oracle TNS Name>" \
"jdbc:Weblogic:@<Oracle TNS Name>" \
"jdbc:sybase:Tds:<Host Name>:<Port Address>/<Database Name>" \
"jdbc:Weblogic:Tds:<Host Name>:<Port Address>/<Database Name>"
jdbcacc.desc.string=This driver provides data source access through JDBC. \
The JDBC driver name, e.g., com.sybase.jdbc.SybDriver, is given in the "lib" \
resource. Only a single name should be specified. This JDBC driver name must \
have a corresponding connection specification, given in the "conn" resource. \
The relative entries of the "lib" and "conn" resource lists are correlated, e.g., \
the "oracle.jdbc.driver.OracleDriver" corresponds to the \
"jdbc:oracle:oci7:@<Oracle TNS Name>" connection template. \
This Production Reporting DDO driver provides support to RDBMS tables and stored
procedures.
```

# Messages and Exceptions

To provide each class with access to messages, place the messages that your driver generates in a single file for the driver package.

## CSV Example

For example, if your driver was in the package, demo.csv, create an empty interface called CSV. Each class in your driver package would implement the CSV interface. As a result, each class would have access to the messages for CSV.

Further, the CSV interface would extend the com.sqribe.access.Access empty interface, making Access messages available to your driver. Since Access extends the com.sqribe.comutil.Util interface, your driver also has access to the common utility messages. The CSV empty interface would look like:

```
package demo.csv;
import com.sqribe.access.*;
/**
 * CSV is a holder interface, allowing properties, messages, etc, common
 * to the csv package to be specified once
 */
public interface CSV extends Access {
}
```

# Message Forms

The message facility supports user and log messages. The log messages provide more information than the user messages. While there is no requirement to have both, and in some situations it does not make sense to have both, it is a good idea to provide both message forms.

The message facility provides automatic logging of messages. Hence, when a message is written, the facility looks for a log message pattern and, if so, generates a log entry.

## Message Conventions

User and log messages follow specific conventions. While there is no requirement to follow these conventions, the Production Reporting DDO exceptions, for example, DataAccessException, provide convenience methods, such as *rethrow* that anticipate the use of these conventions:

- The first substitution argument is the name of the class requesting the message.
- The second substitution argument is the name of the method requesting the message.
- The remaining substitution arguments, if any, provide specific details for the message.

```
# There are no object names in the from clause of the select statement
# 0 The class reporting the error
# 1 The method reporting the error
# 2 The from clause
ObjectNameMissing.text=There are no object names in the FROM clause: "{2}".
ObjectNameMissing.logtext={0}.{1}(): There are no object names in the \
FROM clause: "{2}".
```

As you can see, the difference between the user and log message pattern is the prepending of "{0}.{1}(): " for the class and method name. It is not necessary to add message substitutions for a stack trace or exception string. These are available through the rethrow convenience methods, as shown in *Program ex22.sqr*.

### Program ex22.sqr

```
/**
 * Throw an exception after logging entry
 * @exception UtilException
 * A general exception for data source
 * operations such as
 * making a connection or performing a command.
 * @see ApplicationLog#logMsg
 */
public static void rethrow(String pMsgId, Object[] pArgs) throws
            UtilException {
throw new UtilException((String)pArgs[0], pMsgId, pArgs);
}
/**
 * Rethrow exception after logging entry. A stack trace is logged
 * with the message.
 * @param pMsgId The message identifier
 * @param pArgs The substitution arguments for the message,
 * pArgs[0] is the classname
 * @param pExc The exception to be recorded
 * @exception UtilException
 * A general exception for data source
 * operations such as
 * making a connection or performing a command.
 * @see ApplicationLog#logException
 */
public static void rethrow(String pMsgId, Object[] pArgs,
             Throwable pExc) throws UtilException {
if (pExc instanceof UtilException)
                  throw new UtilException((String)pArgs[0], pMsgId, pArgs);
else throw new UtilException((String)pArgs[0], pMsgId, pArgs,
                  pExc);
}
```

### Exceptions

Error notification is through exceptions. This allows application developers to organize error handling on the edges rather than in the main code path. Exceptions are often generated using the rethrow methods. Use of the getMsg method in the Message Facility, for example, is when all of the messages pertaining to an operation need to be gathered before raising an exception.

```
Object[] objs = new Object[]
   { classname, "bindParameters", new Integer(0), new Integer(parmcount) };
addErrorMsg(ApplicationLog.getMsg().getMsg(classname, "TooManyParameters", objs ));
```

## Properties and Capabilities

In Production Reporting DDO, properties and capabilities are in separate name spaces, though they are processed in much the same way. This means that a property and capability may have the same name but be treated as distinct entities. The rationale for a separate name space and form is purely pragmatic and is rooted in the perceived usage patterns. From an application perspective, capabilities are read-only object properties. From a driver perspective, they are

object properties. From a driver developer's perspective, an attribute should be stipulated as a capability, if and only if, the application should not change the value.

## Descriptions

All properties and capabilities must have associated descriptions. If you feel that a description is not warranted for a capability that is a string and is not computed at runtime, then you should make it a message string type.

## Localization

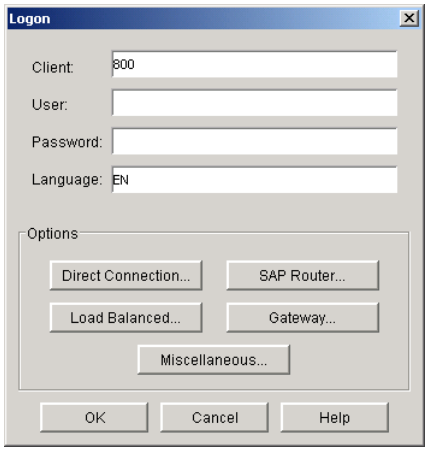
Properties, capabilities, their values, and their descriptions may be localized. You can choose to localize at the granularity of an attribute; that is, a single property attribute within a property resource bundle may be translated, placing it in a localized property resource bundle. You could translate the property description names, and nothing else. Or, you may choose to translate the name and descriptive name.

## Hierarchical Structure

Property and capability descriptions can be hierarchical. The application can use this hierarchical structure to create property pages. The logon property is a hierarchical structure. The Registy Editor tool uses this structure to create a logon panel.

In Figure 4, Client, User, Password, and Language are first-level leaf attributes. Gateway, Load Balancing, No Load Balancing, Advanced, and Miscellaneous are first-level indices. This logon property description is in the property resources *bundle com_sqribe_sapr3ace_DataSourcePropertyDescription.properties*. The dialog box in Figure 4 is part of the *com.sqribe.accessui* package. While the Registry Editor knows that the property, `logon`, is for logon attributes, it has no knowledge of any driver specific logon semantics.

**Figure 4 Logon Dialog Box**



*Program ex23.sqr* shows that the default values for Client and Language are part of the connection string, while the default values for logon.type and logon.trace are in the com_sqribe_sapr3ace_DataSourcePropertyDescription.properties file. By convention the connection string attributes are merged into the default values.

**Example**

Connect string in the registry.properties file:

```
SAPR3.conn=SAPR3:JNI:ASHOST=sundance SYSNR=00 CLIENT=800 LANG=EN
```

**Example**

Property description in:

```
com_sqribe_sapr3acc_SAPR3DataSource_PropertyDescriptions.properties
```

## Program ex23.sqr

```
logon.Name=Logon
logon.Description=Logon is required to establish a connection with an SAP R3 data
source. \
The SAP R3 data source takes a number of parameters as part of the connection string. \
Included among these are the operator (or user) identification and the operator (or
user) \
password. The user identification and password are included as logon information. The
remaining \
connection attributes are part of the connection string provided with the data source \
description (see the SAP R3 DataSource interface documentation for a complete
description). \
To summarize, the SAP R3 connection string may have a form similar to: \
SAPR3:JNI:ASHOST=hs0311 SYSNR=53. \
Where the ASHOST would be replaced by the data source host identifier; the
SYSNR would be replaced by \
the system number for the host. \
The client (CLIENT=), user identification (USER=), password (PASSWD=) and
language (LANG=) \
are appended to the connection string. These are usually retained as
properties. A user \
interface will normally prompt for these values.
logon.Indices=logon.client \
user \
password \
logon.language \
logon.gateway \
......
logon.client.Name=Client
logon.client.Description=SAP logon client identifier.
logon.advanced.Indices=logon.type \
logon.check \
logon.trace
logon.type.Name=RFC Server Type
logon.type.Description=The RFC server type. This parameter should be left
empty or set to 3, the default.
.....
```

**Example**

Property values in:

```
com_sqribe_sapr3acc_SAPR3DataSource_Properties.properties
logon.client=
logon.language=
logon.type=3
logon.check=1
logon.trace=0
logon.dest=
logon.gwhost=
logon.gwserv=
logon.mshost=
logon.r3name=
logon.group=
logon.ashost=
logon.sysnr=
```

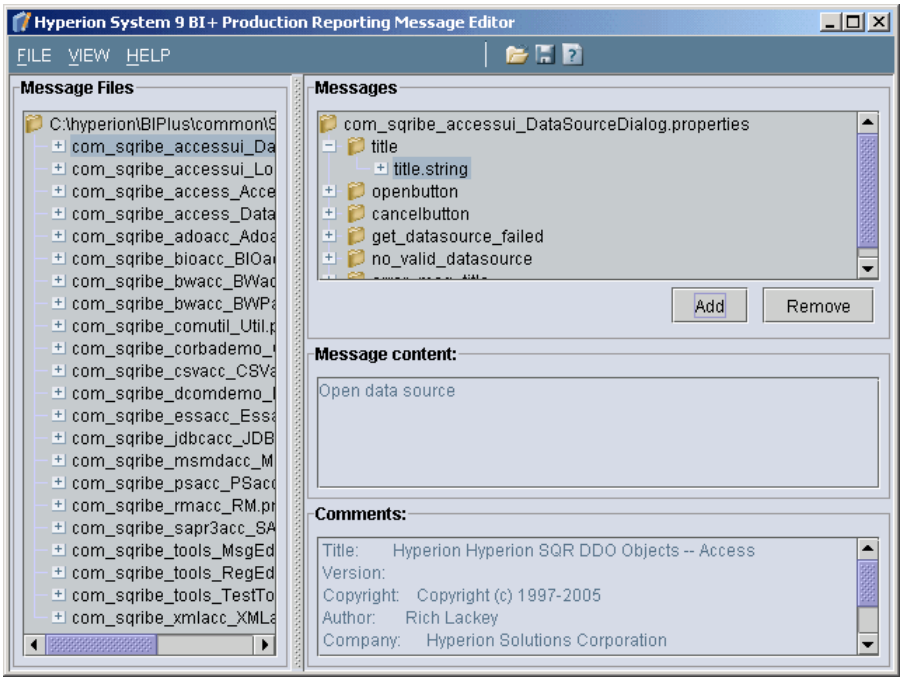
# Internationalization

Production Reporting DDO is fully internationalized. The message and property facilities use property resource bundles as the vehicle to support localization. Applications and driver developers are encouraged to use these facilities as the foundation for internationalizing their packages. The message and property facility are described further in the next chapter.

# Message Editor

Use the Message Editor to create message properties files for Production Reporting DDO drivers.

Figure 5 Message Editor Main Window




The Message Editor contains:

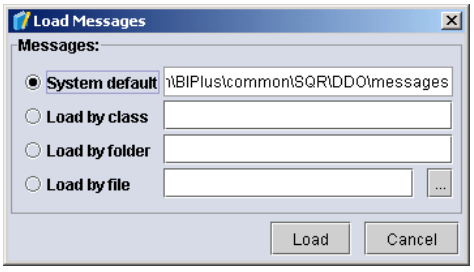
- **Message Files**—All the message files. The root node is the message file name. Each leaf node represents a message file, which takes its name from the message ID. Each message ID can include messages of different types.
- **Messages**—Information about the selected message file. If the selected message contains other messages, an expanded tree appears.
- **Message Content**—The contents of the selected message. You can add header information or edit file contents.
- **Comments**—Comments about the message. You can add one comment for all messages with the same Message ID.

## Loading Messages

➤ To load messages:

1 Select **File > Load Message Files**.

2 Enter information in the Load Messages dialog box and click **Load**.



**Note:**

If there is no *msgs* directory in the Java class path, an error message appears, prompting you to create such a directory.

**Table 14** Ways to Load Message Files

| Option | Description |
|---|---|
| **System Default** | Load the messages files stored in the *msgs* directory included in the classpath. |
| **Load by Class** | Enter theProduction Reporting DDO driver class name to load the corresponding message file. For example, entering **com.sqribe.xmlacc** loads the **com_sqribe_xmlacc_XMLacc.properties** from the *msgs* directory. |
| **Load by Folder** | Enter the path to a folder that you provide. |
| **Load by File** | Load a message property file.<br>a. Click the small button beside the text field to display a file dialog box.<br>b. Select a message file from the *msgs* directory. (The file dialog box points to the *msgs* directory included in the class path.)<br>c. Enter a message file name and click **Open**.<br>This copies the new message file name to the text field.<br>d. Click **Load** to create the message file.<br>The file creation time is saved to the newly created message file's header section. |

## Adding Messages

➤ To add a new message ID:

1 **Select the root node under Messages.**

The name of the root node is the message file name.

2 **Click Add to display the *Add New Message* dialog box.**

3 **Enter the new name and select a message type.**

## Editing Message Contents

➤ To change the content of a message:

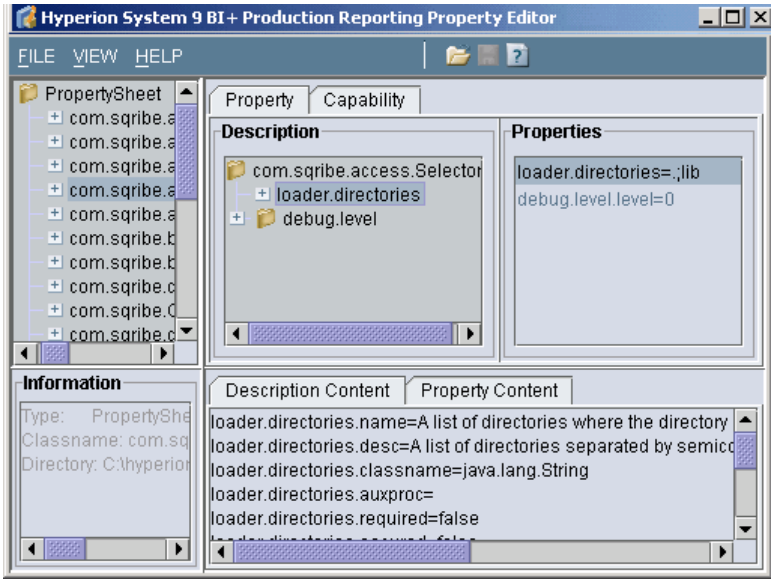
1 **Select a message.**

2 **Edit the text under Message Content and select File > Save.**

# Property Editor

Use the Property Editor to create and edit property, capability, property description, and capability description files for Production Reporting DDO drivers.

Figure 6 Property Editor Main Window




The Property Editor contains:

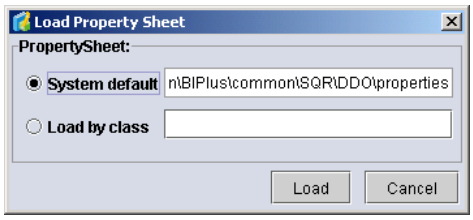
- **PropertySheet**—The DataSource class names of the Production Reporting DDO driver. To edit properties and capabilities, you must select the Production Reporting DDO driver's class name first.
- **Information—The classname and directory for the selected property sheet.**
- **Description**—The property names in the description files. The root node is the name of the driver's DataSource class. Non-leaf nodes are properties whose indices are not equal to null. Leaf nodes are properties whose indices equals null.
- **Properties/Capabilities—Depending on the tab selected (Property or Capability)**, the properties or capabilities in the driver's property or capability file.
- **Description Content**—The attributes of the property or capability selected under Description.
- **Property Content—The name and value of the** property or capability selected under Description. Click Edit, to edit the value.

## Loading Properties

➤ To load properties:

1 Select **File > Open.**

2 Enter information in the Load PropertySheet dialog box and click **Load**.




**Note:**

If there is no *properties* directory in the Java class path, an error message appears, prompting you to create such a directory.

**Table 15** Ways to Load Property Files

| Option | Description |
|---|---|
| **System Default** | Load the property files from the *properties* directory included in the classpath. |
| **Load by Class** | Provide the Production Reporting DDO driver's DataSource class name.<br>For example, entering `com.sqribe.xmlacc.XMLDataSource` loads the `DDO xml` driver property files from the *properties* directory. |

# Testing and Debugging Drivers

Three command line sample programs and a graphical query editor are available for driver development. The command line sample programs are located in: `\hyperion\products\biplus\docs\Server\DDO\Test`

- **AccessMeta**—Used to test metadata hierarchy driver functionality. Takes a schema name as an argument and produces an HTML report displaying the metadata for the subtree whose root is the named schema.
- **AccessObjects**—Used for drivers supporting the Production Reporting DDO object retrieval interface. Given a qualified object name, produces an HTML report displaying the object's result set.
- **AccessProcs**—Used for drivers supporting the Production Reporting DDO call retrieval interface. Given a qualified procedure name and a parameter list, produces an HTML report displaying the in/out and output parameters, the return value, and the result sets.

## Using the Query Editor to Test and Debug Drivers

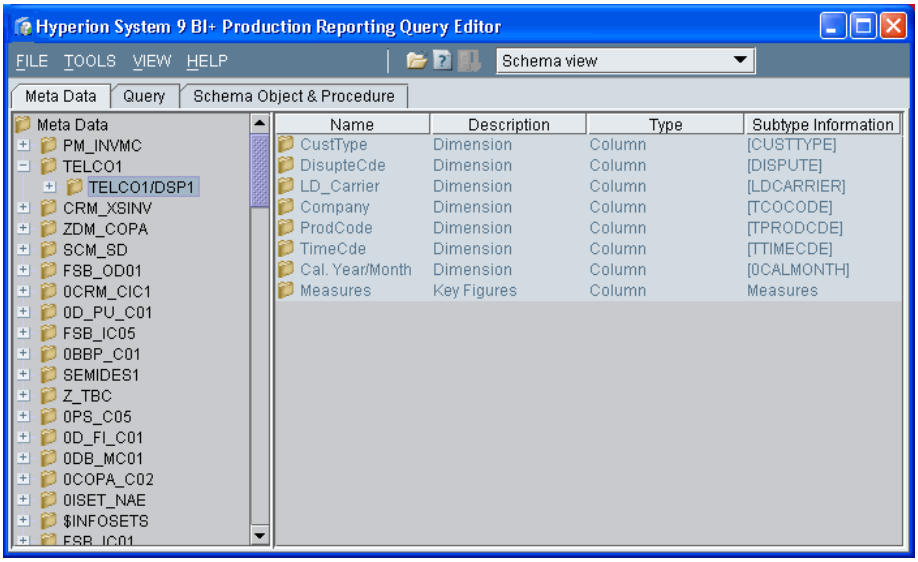
Use the Production Reporting DDO Query Editor to debug your drivers. The Query Editor works the same way as a database query editor, allowing you to look at tables, run procedures to see what they return, test commands or SQL statements, and retrieve row sets.

➤ To use the Query Editor for testing and debugging:

1. Select **Start > Programs > Oracle EPM System> Reporting and Analysis > Production Reporting for DDO > Query Editor**.
2. Select **File > Open** to display the Open Data Source window.
3. Select and log onto a data source.
4. Select the **Meta Data** tab to display the contents.
5. Select the desired node.

Figure 7 shows the Query Editor displaying objects in an SAPBW data source.

**Figure 7 Displaying a Schema Object View in the Query Editor**



# 5 Managing Data Sources

In This Chapter

Data Source Specifications ..... 87
Adding a Data Source Specification in the Registry Editor ..... 88
Data Source Descriptions and Templates ..... 88

## Data Source Specifications

A data source specification consists of:

- **Data source name** (required)—Logical name for the data source. Production Reporting DDO applications associate this name with the connection.
- **Description (optional)**—Used to assist the user in selecting a data source. Production Reporting DDO applications can display the descriptive name along with the data source name. Alternatively, an application can display the descriptive name as a *tooltip*.
- **Class** (required)—Class name loaded by the Production Reporting DDO data source manager for a selected data source.
- **Variants**—An Production Reporting DDO driver supports one or more data source variants. The JDBC driver, for example, supports a number of database vendors. Each database is reflected as a variant.
- **Java libraries (may be required by the driver)**—The driver may require additional classes to be loaded prior to the instantiation of the data source driver class. The driver provides the classes and they appear when you select the driver in the registry editor. You should not have to provide additional information here.
- **Native libraries** (may be required by the driver)—The driver may require platform-specific Java Native Interface (JNI) libraries to be loaded, prior to the instantiation of the data source driver class. The driver provides these classes and they appear when you select the driver in the registry editor. You should not have to provide additional information here.
- **Connection string** (optional)—All drivers require connection information, for some drivers, you can supply this information with the connection request. Other drivers require some information in advance. The drivers supply templates for connection information that the Registry Editor uses in a dialog box for the connection string.

# Adding a Data Source Specification in the Registry Editor

The Production Reporting DDO Registry Editor provides a graphical interface for managing data sources. Production Reporting DDO allows multiple registries, with each registry containing multiple data sources. Each data source defines the objects and connection specifications for the source.

You can quickly create and update registries using the Production Reporting DDO Registry Editor. In addition, the editor's test connection function helps you ensure that the data source specification is correct before trying it in your application.

In the following example, we show the use of the Production Reporting DDO Registry Editor to add a specification for an Oracle Data Source

➤ To add a specification in the registry:

1. Start the Registry Editor by selecting **Start > Programs > Oracle EPM System > Reporting and Analysis > Production Reporting > DDO > Registry Editor**.
2. Select **File** > **Open**.
3. Select **Registry** in the Open Registry window and click **Open**.

   Registry is the default registry and should always appear.
4. Click **Add** in the Registry Editor main window.
5. Select a driver name in the Create New Data Source dialog box and click **OK**.

   This is the Production Reporting DDO driver needed to connect to the data source.
6. Enter the data source information in the Setup Data Source dialog box that appears.
7. Click **Build** to display the Build Connection String window; and enter the requested information.
8. Click **OK** to return to the Setup Data Source dialog box; then, click **Test** to ensure that the data source specification is correct.

   If the data source specification is correct, a message appears indicating that the connection test succeeded and the Logon dialog box appears. (In our example, you would enter an Oracle user name and password in the Logon dialog box and click OK.)

   If the data source specification is *not* correct, an error message appears and you must re-enter the data source specification.
9. Click **OK** to return to the main window; then, save the information by clicking the Save Registry toolbar button.

   After you save data source information, the Registry Editor remains open to make additional data source specifications or changes. If you do not wish to make further changes, select **File** > **Exit** to close the Registry Editor.

# Data Source Descriptions and Templates

The following sections provide data source and parameter descriptions for each supported driver connection. The following data source drivers are available:

- Hyperion Essbase
- SAP R/3 and SAP BW
- Microsoft OLEDB for OLAP
- Microsoft OLEDB
- JDBC
- XML
- Delimiter Separated Values
- OMG Corba Sample
- Microsoft DCOM Sample
- CSV Sample

# Hyperion Essbase

This data source represents an Essbase database. Essbase is a multidimensional database, where objects represent cubes. You must have an Essbase client library installed to use this driver.

The following Hyperion Essbase connections are supported:

- Host Name
- IP Address

**Note:**

When you connect to an Essbase data source with Production Reporting DDO, you must add the ARBORPATH variable to the Essbase client, and add the location of the Essbase shared library files to the path.

## Host Name

### Template

```
<Host Name>
```

### Parameters

```
Host Name
```

Default host name of the Essbase server.

### Example

```
Essbase.acme.com
```

### IP Address

**Template**

```
<IP Address>
```

**Parameters**

```
IP Address
```

Default IP address of the Essbase server.

**Example**

```
129.3.4.50
```

## SAP R/3 and SAP BW

This driver provides access to SAP R/3 and SAP BW data sources. The following SAP connections are supported:

- Unbalanced/Direct Connection
- Balanced Connection
- External Gateway Connection
- Single Hop SAPRouter Connection

The connection string can contain any of the SAP logon parameters, including:

**CLIENT**—Client number.

**LANG**—User language.

**TYPE**—RFC server type. Should be left empty or set to the default value of 3.

**CHECK**—SAP logon check during Open. The default is 1 and should always be used.

**TRACE**—Establishes an RFC trace log. The default is 0, do not establish a trace log.

**DEST**—Destination in saprfc.ini, when using saprfc.ini.

**GWHOST**—Host name of the SAP gateway, when the server is R/2 or External.

**GWSERV**—Service of the SAP gateway, when the server is R/2 or External.

**MSHOST**—Host name of the SAP message server, when using load balancing.

**R3NAME**—Name of the R/3 system, when using load balancing.

**GROUP**—Name of the group of application servers, when using load balancing. A name containing embedded blanks must be enclosed in quotes, for example, “Finance Group”.

**ASHOST**—Host name of a specific application server, when using R/3 without load balancing.

**SYSNR**—SAP system number of a specific application server, when using R/3 without load balancing.

**TPHOST**—Host name of a specific external RFC program server.

**TPNAME**—Path and name of the external RFC server program or Program ID of a registered RFC server program.

## Unbalanced/Direct Connection

### Template

```
SAPR3:JNI:ASHOST=<Host Name> SYSNR=<SAP System Number> CLIENT=<Client
Number> LANG=<Language>
```

```
SAPBW:JNI:ASHOST=<Host Name> SYSNR=<SAP System Number> CLIENT=<Client
Number> LANG=<Language>
```

### Parameters

`Host Name`

Default host name of a specific application server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`SAP System Number`

Default system number of the R/3 or SAP BW application server. This is typically a two or three digit value.

`Client Number`

Default client number for users of this connection. This is typically a three digit value.

`Language`

Default language for users of this connection. This is either the SAP language code or the ISO language code.

### Example

```
SAPR3:JNI:ASHOST=R3Server SYSNR=01 CLIENT=850 LANG=EN
```

```
SAPBW:JNI:ASHOST=BWServer SYSNR=03 CLIENT=800 LANG=EN
```

## Balanced Connection

### Template

```
SAPR3:JNI:MSHOST=<Message Server> R3NAME=<R/3 System Name>
GROUP=<Application Group> CLIENT=<Client Number> LANG=<Language>
```

```
SAPBW:JNI:MSHOST=<Message Server> BWNAME=<SAPBW System Name>
GROUP=<Application Group> CLIENT=<Client Number> LANG=<Language>
```

### Parameters

`Message Server`

Default host name of a specific message server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`R/3 System Name or SAP BW System Name`

Default name of the R/3 or SAP BW application server, when using load balancing. This parameter value should only be specified when the server is R/3 or SAP BW and load balancing is used; otherwise, it should be left empty.

`Application Group`

Default name of the group of application servers, when using load balancing. A name containing embedded blanks must be enclosed in quotes. For example, “Finance Group.” This parameter value should only be specified when the server is R/3 or SAP BW and load balancing is used; otherwise, it should be left empty.

`Client Number`

Default client number for users of this connection. This is typically a three digit value.

`Language`

Default language for users of this connection. This is either the SAP language code or the ISO language code.

### Example

```
SAPR3:JNI:MSHOST=R3MSGSVR R3NAME=FINANCE GROUP=PUBLIC CLIENT=850 LANG=EN
```

```
SAPBW:JNI:MSHOST=BWMSGSVR R3NAME=FINANCE GROUP=PUBLIC CLIENT=850 LANG=EN
```

## External Gateway Connection

### Template

```
SAPR3:JNI:TYPE=E TPHOST=<Server Host Name> TPNAME=<Server Program Name>
GWHOST=<Gateway Host> GWSERV=<Gateway Server> CLIENT=<Client Number>
LANG=<Language>
```

```
SAPBW:JNI:TYPE=E TPHOST=<Server Host Name> TPNAME=<Server Program Name>
GWHOST=<Gateway Host> GWSERV=<Gateway Server> CLIENT=<Client Number>
LANG=<Language>
```

### Parameters

`Server Host Name`

Default host name of a specific external RFC program server. This may be specified as a host name (for example, ws1.acme.com,)or an IP address (for example, 129.3.4.50).

`Server Program Name`

Default path and name of the external RFC server program or Program ID of a registered RFC server program. This parameter value should only be specified when the server is External; otherwise, it should be left empty.

`Gateway Host`

Host name of IP address (most likely an IP address) of the gateway host that is the partner of the client for this connection.

`Gateway Server`

Host name of IP address (most likely an IP address) of the gateway server that is the partner of the gateway host for this connection.

`Client Number`

Default client number for users of this connection. This is typically a three digit value.

`Language`

Default language for users of this connection. This is either the SAP language code or the ISO language code.

### Example

```
SAPR3:JNI:TPHOST=R3EXTSVR TPNAME=R3EXTPGM CLIENT=850 LANG=EN GWHOST=204.79.
199.5 GWSERV=207.213.200.19
```

```
SAPBW:JNI:TPHOST=BWEXTSVR TPNAME=BWEXTPGM CLIENT=850 LANG=EN GWHOST=204.79.
199.5 GWSERV=207.213.200.19
```

## Single Hop SAPRouter Connection

### Template

```
SAPR3:JNI:ASHOST=/H/<SAP Router Host>/H/<Gateway Host>/H/<Host Name>
SYSNR=<SAP System Number> CLIENT=<Client Number> LANG=<Language>
```

```
SAPBW:JNI:ASHOST=/H/<SAP Router Host>/H/<Gateway Host>/H/<Host Name>
SYSNR=<SAP System Number> CLIENT=<Client Number> LANG=<Language>
```

### Parameters

`SAP Router Host`

Host name or IP address for the SAPRouter application.

`Gateway Host`

Host name or IP address (most likely an IP address) of the gateway host that is the partner of the SAPRouter for this connection.

`Host Name`

Default host name of a specific application server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`SAP System Number`

Default system number of the R/3 application server. This is typically a two or three digit value.

`Client Number`

Default client number for users of this connection. This is typically a three digit value.

`Language`

Default language for users of this connection. This is either the SAP language code or the ISO language code.

**Example**

```
SAPR3:JNI:ASHOST=/H/10.213.33.238/H/204.79.199.5/H/

207.213.200.19 SYSNR=01 CLIENT=850 LANG=EN

SAPBW:JNI:ASHOST=/H/10.213.33.238/H/204.79.199.5/H/

207.213.200.19 SYSNR=01 CLIENT=850 LANG=EN
```

# Microsoft OLEDB for OLAP

This data source represents a Microsoft OLAP server, where objects represent cubes. You must have an Microsoft SQLServer OLAP client library installed to use this driver. You can connect to Microsoft OLEDB for OLAP through the provider.

## Provider

**Template**

```
Data Source=<Server Name>;Provider=<Driver Provider Name>{msolap};
Initial Catalog=<Catalog Name>
```

**Parameters**

`Server Name`

ODBC logical data source name.

`Driver Provider Name`

Name of the driver provider. The default is msolap. There is no need to change the default.

`Catalog Name`

Name of the catalog containing the cubes to be analyzed.

**Example**

```
Data Source=JAWS;Provider=msolap;Initial Catalog=Food Mart
```

# Microsoft OLEDB

This driver provides data source access through Microsoft ADO. There is an OLE DB provider with ADO MSDASQL for ODBC databases, MSIDXS for Microsoft Index Server, ADSDSOObject for Microsoft Active Directory Service, Microsoft.Jet.OLEDB.3.51 for Microsoft Jet databases, SQLOLEDB for Microsoft SQL Server, MSDAORA for Oracle databases.

You can connect to Microsoft OLEDB through the provider.

## Provider

### Template

```
Data Source=<Data Source Name>;Provider=<Driver Provider Name>{MSDASQL
SQLOLEDB MSDAORA Microsoft.Jet.OLEDB.3.51 MSIDXS ADSDSOObject };Initial
Catalog=<Database Name>
```

### Parameters

`Data Source Name`

ODBC logical data source name.

`Driver Provider Name`

Name of the driver provider. The choices are: MSDASQL, SQLOLEDB, MSDAORA, Microsoft.Jet.OLEDB.3.51, MSIDXS, and ADSDSOObject.

`Database Name`

Name of the database within the given data source.

### Example

```
Data Source=JAWS;Provider=MSDASQL;Initial Catalog=Food
```

# JDBC

This driver provides data source access through JDBC. The JDBC driver name, for example, `com.sybase.jdbc.SybDriver`, is given in the `lib` resource. Only a single name should be specified. This JDBC driver name must have a corresponding connection specification, given in the `conn` resource. This driver provides support for RDBMS tables and stored procedures.

The following JDBC connections are supported:

- JDBCODBC
- Oracle Thin Client
- Oracle OCI Client
- Sybase Thin Client
- DB/2 Thin Client
- DB/2 Local Client
- Informix Thin Client
- Hyperion DB2 Client
- Hyperion Informix Client
- Hyperion Oracle Client
- Hyperion SQL Server Client
- Hyperion Sybase Client

## JDBCODBC

### Template

JDBC:ODBC:<*ODBC DSN*>

### Parameters

ODBC DSN

ODBC logical data source name.

### Example

JDBC:ODBC:Northwind

## Oracle Thin Client

### Environment Variables

CLASSPATH=<*ORAHOME*>/jdbc/lib/classes111.zip;

### Template

jdbc:oracle:thin:@<Host Name>:<Port Address>:<Oracle TNS Name>

### Parameters

Host Name

Host name of the Oracle database server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

Port Address

Port number at the host name of the Oracle database server. This is a value between 0 and 64565, inclusive.

Oracle TNS Name

Oracle logical network name for the connection.

### Example

jdbc:oracle:thin:@swhale:1521:Oracle.World

## Oracle OCI Client

### Environment Variables

CLASSPATH=<*ORAHOME*>/jdbc/lib/classes111.zip;

### Template

jdbc:oracle:oci<Oracle Version>{7 8}:@<Oracle TNS Name>

### Parameters

`Oracle Version`

Oracle version number of the client library for the database server. Valid values are: 7 and 8.

`Oracle TNS Name`

Oracle logical network name for the connection.

### Example

```
jdbc:oracle:oci8:@Oracle.World
```

## Sybase Thin Client

### Environment Variables

`CLASSPATH=<`*SYBHOME*`>/jConnect-4_2/classes;`

### Template

```
jdbc:sybase:Tds:<Host Name>:<Port Address>/<Database Name>
```

### Parameters

`Host Name`

Host name of the Sybase database server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`Port Address`

Port number at the host name of the Sybase database server. This is a value between 0 and 64565, inclusive.

`Database Name`

Sybase database name.

### Example

```
jdbc:sybase:Tds:swhale:2545/SWHALE11
```

## DB/2 Thin Client

### Environment Variables

`CLASSPATH=<`*DB2HOME*`>/java/db2java.zip;`

`PATH=<`*DB2HOME*`>/bin;%PATH%`

### Template

```
jdbc:db2:<Host Name>:<Port Address>/<Database Name>
```

### Parameters

`Host Name`

Host name of the DB/2 database server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`Port Address`

Port number at the host name of the DB/2 database server. This is a value between 0 and 64565, inclusive.

`Database Name`

DB/2 database name.

### Example

```
jdbc:db2:mooneye:50000/db2inst
```

## DB/2 Local Client

### Environment Variables

`CLASSPATH=<`*`DB2HOME`*`>/java/db2java.zip;`

`PATH=<`*`DB2HOME`*`>/bin;%PATH%`

### Template

```
jdbc:db2:<Database Name>
```

### Parameters

`Database Name`

DB/2 database name.

### Example

```
jdbc:db2:db2inst
```

## Informix Thin Client

### Environment Variables

`CLASSPATH=<`*`IFXHOME`*`>/1.4/lib/ifxjdbc.jar;`

### Template

```
jdbc:informix-sqli://<Host Name>:<Port Address>/<Database
Name>:INFORMIXSERVER=<Server Name>
```

### Parameters

`Host Name`

Host name of the Informix database server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`Port Address`

Port number at the host name of the Informix database server. This is a value between 0 and 64565, inclusive.

`Database Name`

Informix database name.

`Server Name`

Informix server name.

#### Example

```
jdbc:informix-sqli://swhale:1521/nickldb:INFORMIXSERVER=server01;
```

## Hyperion DB2 Client

#### Template

```
jdbc:hyperion:db2://<Host Name>:<Port Address>; databaseName=<Database
Name>
```

#### Parameters

`Host Name`

Host name of the DB2 database server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`Port Address`

Port number for the named DB2 database server. This is a value between 0 and 64565, inclusive.

`Database Name`

DB/2 database name.

#### Example

```
jdbc:hyperion:db2://bass:50002;databaseName=JH71DB
```

## Hyperion Informix Client

#### Template

```
jdbc:hyperion:informix://<Host Name>:<Port Address>;
informixserver=<server name>;databaseName=<Database Name>; DBDate=<Date
Format>{MDY4/MDY4-MDY4.MDY2/MDY2-MDY2.YMD4/YMD4- YMD4.YMD2/YMD2-YMD2.}
```

**Parameters**

`Host Name`

Host name of the Informix database server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`Port Address`

Port number for the named Informix database server. This is a value between 0 and 64565, inclusive.

`Server Name`

Informix server name.

`Database Name`

Informix database name.

`DBDate`

Informix environment variable used to format the output when `DATE` values are displayed. With standard formats, you can specify the following attributes:

- The order of the month, day, and year in a date format.
- Whether the year is printed with two digits (Y2) or four digits (Y4).
- The separator between the month, day, and year.

The format string can include the following characters:

- - (hyphen), . (dot), / (slash)—Separator characters used in a date format.

  The separator always goes at the end of the format string (for example, `Y4MD-`). If no separator or an invalid character is specified, the slash ( / ) character is the default.
- `0`—Indicates that no separator is displayed.
- `D,M`—Characters that represent the day and the month.
- `Y2,Y4`—Characters that represent the year and the number of digits in the year.

Valid `DBDATE` formats include:

- `DMY2`
- `DMY4`
- `MDY4`
- `MDY2`
- `Y4MD`
- `Y4DM`
- `Y2MD`
- `Y2DM`

**Note:**

For the U.S. ASCII English locale, the default setting for DBDATE is Y4MD-, where Y4 represents a four-digit year, M represents the month, D represents the day, and hyphen ( - ) is the separator. For example, 1998-10-08.

### Example

```
jdbc:hyperion:informix://sandtiger:4111/jerryh_db;
INFORMIXSERVER=sandtiger920;databaseName=jerryh_db;DBDate=MDY4/
```

## Hyperion Oracle Client

### Template

```
jdbc:hyperion:oracle://<Host Name>:<Port Address>; SID=<Oracle SID>
```

### Parameters

`Host Name`

Host name of the Oracle database server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`Port Address`

Port number for the named Oracle database server. This is a value between 0 and 64565, inclusive.

`Oracle SID`

Oracle logical network name for the connection.

### Example

```
jdbc:hyperion:oracle://swhale:1521;SID=swhale817
```

## Hyperion SQL Server Client

### Template

```
jdbc:hyperion:sqlserver://<Host Name>:<Port Address>
```

### Parameters

`Host Name`

Host name of the SQL Server database server. This may be specified as a host name, for example, ws1.acme.com, or an IP address, for example, 129.3.4.50.

`Port Address`

Port number for the named SQL Server database server. This is a value between 0 and 64565, inclusive.

### Example

```
jdbc:hyperion:sqlserver://Glass:1433
```

## Hyperion Sybase Client

### Template

```
jdbc:hyperion:sybase://<Host Name>:<Port Address>
```

### Parameters

```
Host Name
```

Host name of the Sybase database server. This may be specified as a host name, for example, ws1.acme.com, or an IP address, for example, 129.3.4.50.

```
Port Address
```

Port number for the named Sybase database server. This is a value between 0 and 64565, inclusive.

### Example

```
jdbc:hyperion:sybase://Perch:4110
```

# XML

This data source represents a directory tree rooted in a file system. Files in the tree having the file extension `.xml` are interpreted as delimiter separated files. These files represent objects to this driver. Be sure that ddo11.zip and `xml4j.jar` are in your classpath.

You can connect to XML through the directory path name.

## Directory Path Name

### Template

```
<Fully Qualified Directory Path Name>
```

### Parameters

```
Fully Qualified Directory Path Name
```

Fully qualified path name for a directory forms the root of a tree containing XML files. There is no defined limit to the number of subdirectories beneath the root. XML files may appear anywhere within the tree. The directories may contain other kinds of files. These will be ignored by the driver.

### Example

```
D:\\Projects\\Data Access\\Test\\XmlTest
```

# Delimiter Separated Values

This data source represents a directory tree rooted in a file system. Files in the tree having the file extension `.csv` are interpreted as delimiter separated files. These files represent objects to this driver.

You can connect to delimiter separated values through the directory path name.

## Directory Path Name

### Template

```
CSV:File:<Fully Qualified Directory Path Name>
```

### Parameters

```
Fully Qualified Directory Path Name
```

Fully qualified path name for a directory forms the root of a tree containing CSV files. There is no defined limit to the number of subdirectories beneath the root. CSV files may appear anywhere within the tree. The directories may contain other kinds of files. These will be ignored by the driver.

### Example

```
CSV:File:d:\\Projects\\Data Access\\Test
```

# OMG Corba Sample

This sample data source represents a CORBA server that resides on a remote host. The CORBA server can be a wrapper for a legacy system on a remote host. To use this driver the northwind.jar has to be in the classpath path. The java runtime environment should have support for the JavaIDL.

You can connect to the OMG Cobra Sample through the sample data source.

## Sample Data Source

### Environment Variables

```
CLASSPATH=<AVALANCHE_HOME>/Documentation/SQR/Server/DDO/demo/CORBA
/CORBADemoSvr.jar;%CLASSPATH%
```

### Template

```
HOST=<Host Name> PORT=<Port Number>
```

### Parameters

```
Host Name
```

Host name of the Corba server. This may be specified as a host name, for example, ws1.acme.com, or an IP address, for example, 129.3.4.50.

`Port Number`

Port number at the host name of the Corba server. This is a value between 0 and 64565, inclusive.

**Example**

```
HOST=JAGUAR;PORT=1080
```

# Microsoft DCOM Sample

This sample data source represents a DCOM server that resides on a remote host. The DCOM server can be a wrapper for a legacy system on a remote host. To use this driver the northwind.jar has to be in the classpath path.

You can connect to the Microsoft DCOM Sample through the sample data source.

## Sample Data Source

**Template**

```
HOST=<Host Name> PORT=<Port number>
```

**Parameters**

`Host Name`

Host name of the DCOM server. This may be specified as a host name (for example, ws1.acme.com) or an IP address (for example, 129.3.4.50).

`Port Address`

Port number at the host name of the DCOM server. This is a value between 0 and 64565, inclusive.

**Example**

```
HOST=JAGUAR;PORT=1080
```

# CSV Sample

This data source represents a directory tree rooted in a file system. Files in the tree with the file extension `.csv` are interpreted as delimiter separated files. These files represent objects to this driver.

You can connect to the CSV Sample through the directory path name.

## Directory Path Name

### Template

```
CSV:File:<Fully Qualified Directory Path Name>
```

### Parameters

```
Fully Qualified Directory Path Name
```

Fully qualified path name for a directory forms the root of a tree containing CSV files. There is no defined limit to the number of subdirectories beneath the root. CSV files may appear anywhere within the tree. The directories may contain other kinds of files. These will be ignored by the driver.

### Example

```
CSV:File:d:\\Projects\\Data Access\\Test
```

# 6 Utilities Package and Common Facilities

**In This Chapter**


About the Utilities Package ..... 107
Common Components of the Utilities Package ..... 107
Message Facility ..... 110
Property Facility ..... 112


## About the Utilities Package

Production Reporting DDO includes a common utilities package (com.sqribe.comutil). Components of this utilities package provide an infrastructure that supports properties, messages, logs, and exceptions. This chapter provides information on the common components of the utilities package, the message facility, and the property facility.

## Common Components of the Utilities Package

The message and property facilities use a similar naming scheme, use Java property resource bundles, and use common instrumentations.

### Naming Scheme

The message and property facilities use naming schemes that ensure that class name collisions do not occur. The naming schemes implement the following:

- Placing property resource bundles in a separate directory:
  - Message facility: *msgs* directory
  - Properties and capabilities facility: *properties* directory
- Including in the CLASSPATH the directory that contains msgs and properties directories.
- Change the separators (‘.’) of the fully qualified class name (<package name>.<class name>) to underscores (_) to produce a file name. For the properties facility, append the **type** suffix.
- Complete the file name with the extension **.properties**.

**Table 16** Message and Property Facilities File Names

| File Name | Comment |
|---|---|
| com_sqribe_access_Access.properties | Message facility file name: resides in the msgs directory |
| | Property facility file names: reside in the properties directory |
| com_sqribe_access_DataSource_Properties.properties | Contains property attribute values |
| com_sqribe_access_DataSource_PropertyDescriptions. properties | Contains property attribute metadata |
| com_sqribe_access_DataSource_Capabilities.properties | Contains capability attribute values |
| com_sqribe_access_DataSource_CapabilityDescriptions.properties | Contains capability attribute metadata |

# Property Resource Bundles

A property resource bundle is a special type of Java resource bundle that stores its values in a property file. The message facility stores messages in the bundle. The property facility stores properties and capabilities values and metadata in the bundle.

## Property Resource Bundle Processing

Java encourages the use of packages, which contain autonomous components that form closely knit set of relationships. Classes within packages could share the same messages and the same properties and capabilities. In order to keep together messages or attributes and values that might be shared by classes, component users can extend classes or implement interfaces prescribed in a package.

The message and property facilities recognize class relationships and use them to locate properties. Here is a message facility example:

- The Production Reporting DDO relational database driver is in the package com.sqribe.jdbcacc.
- Within this package, com.sqribe.jdbcacc.JDBCacc is an empty interface, extending the interface com.sqribe.access.Access.
- Other classes in the com.sqribe.jdbcacc package implement the JDBCacc interface.
- The com.sqribe.access.Access interface extends the com.sqribe.comutil.Util interface. Like the JDBCacc interface, these are also empty interfaces.

## Locating Property Resource Bundles

To retrieve a property resource bundle, the common utility facility needs two pieces of information: a fully-qualified base name and a locale identifier

Production Reporting DDO uses a combination of introspection and Java ResourceBundle.getBundle invocation to retrieve resource bundles. The Java ResourceBundle class concatenates the two strings, separating them by an underscore to form a class name. It

then attempts to load a class with that name using the default system loader. If a class with the name cannot be loaded, then the name is successively shortened until a resource bundle class is successfully loaded and instantiated. The `getBundle()` method will look for a property resource bundle whenever a class name fails to produce a resource bundle object. In particular, the class name is appended with the string ".properties". If such a file exists, a PropertyResourceBundle object is created for that properties file.

# Instrumentation

The Production Reporting DDO common utilities packages includes common mechanisms, or instrumentations, that are available to applications.

## Diagnostic Exits

The message facility provides a mechanism for invoking diagnostic classes.that may be associated with a message. Requesting the message instantiates the diagnostic class. The common utilities package provides an empty interface, Diagnostic, and instructions for constructor signatures.

Driver developers are encouraged to use the Debug class in conjunction with the toString method for the classes comprising their driver package. The Debug class is a specialization of the StringBuffer class, focused upon dumping class state information. Production Reporting DDO makes extensive use of these classes for runtime diagnostic information.

## Timing

There is a set of monitor properties that provide timing instrumentation. These are based on a simple common utility stopwatch class called Monitor, which provides start and stop timing methods. The Access package includes a DataSourceMonitor class that uses this mechanism for medium level timing statistics. Data Source Monitor provides a convenience mechanism for presenting the settings for various monitoring states.

The monitoring states are set from properties when this singleton class is instantiated by the DriverSourceManager. The DataSourceMonitorAuxProc class registered for the monitor properties updates property descriptions for these states.

It is possible to override monitoring states associated with a specific driver through the use of virtual properties.

For more information on the individual monitors refer to the monitor property descriptions.

While the monitors are setup as a hierarchy, querying the monitors is flat. Here are the relevant retrieval/loading monitoring points:

- **retrievalExecute**—Execute monitoring is active.
- **retrievalCall**—Call monitoring is active.
- **retrievalGetData**—GetData monitoring is active.
- **metadataSchema**—Schema names monitoring is active.
- **metadataSchemaObjects**—Schema objects monitoring is active.

- **metadataSchemaObjectColumns**—Schema object columns monitoring is active.
- **metadataSchemaProcedures**—Schema procedures monitoring is active.
- **metadataSchemaProceduresMeta**—Schema procedures metadata monitoring is active.
- **propertysheetLoad**—Propertysheet load monitoring is active.
- **drivermanagerLoad**—Data source manager load monitoring is active.

# Message Facility

The message facility uses `java.text.MessageFormat` to produce language-specific user messages that contain number, currency, percentages, date, time, and string variables. As a result, it provides full Java message formatting facilities. In addition, the message facility:

- Obtains message patterns from `java.util.PropertyResourceBundle` instances.
- Uses introspection to obtain inheritance and implementation graphs to locate a message pattern, thus minimizing message duplication.
- Provides message-triggered diagnostic aids.

## Message Text

The message facility provides full Java message formatting facilities, supporting three kinds of message patterns.

- **String**—A string is a message text having no substitution elements. A string may be used as a substitution element or may appear in a user interface as a resource. The Java message format class is not used to process message strings. These are likely candidates for localization.
- **Text**—A text message is a message pattern used for a user message or the string returned from java.lang.Throwable *getMessage()*. The Java message format class is used to process substitution elements appearing in the message pattern. These are likely candidates for localization.
- **LogText**—A logtext message is a message pattern used for application log messages. These message patterns contain detailed information that may be used for security, diagnostics, or support. The Java message format class is used to process substitution elements appearing in the message pattern. These are not likely candidates for localization.

## Message Property Files

A property consists of a property name and a property value. Message property files, shown in Table 17, are Java property resource bundles. The message facility interprets a property name as a message identifier plus a message type extension. In the following table, the message identifier is indicated by *msg*. The identifier can constitute a hierarchical name.

**Table 17** Message Property Files

| Name | Value | Description |
|---|---|---|
| *Msg*.string | Localized message label | *String* indicates a localized string constant. |
| *Msg*.text | Localized message format text | *Text* indicates a message requiring formatting, while *string* indicates a localized string constant (that may be used as a substitution in a formatted message). |
| *Msg*.logtext | Localized log message format text | *Logtext* indicates a log message requiring formatting. |
| *Msg*.diagnostic | Fully qualified class name | *Diagnostic* is used to instantiate a class to perform diagnostics as a consequence of the given message. |
| *Msg*.exception | Fully qualified class name | *Exception* is used to create an Exception object to be thrown by the message facility. |

### Message Property Search Example

➤ To search for the message property, `SchemaException.text`.

1 **Look in** `com_sqribe_jdbcacc_JDBCConnection*.properties`.

2 **Look in** `com_sqribe_jdbcacc_JDBCacc*.properties`.

The asterisk ('*') in the name indicates where the resource bundle search appends the localization suffix. Since message property files only exist for the packages, only one message property file is searched:

```
com_sqribe_jdbcacc_JDBCacc*.properties
```

Had the message property been "DescriptionNotDefined.text," then the search would have continued and these message property files would have been included in the search:

```
com_sqribe_access_Access*.properties.
```

```
com_sqribe_comutil_Util*.properties.
```

## Localization Example

Let's look at the effect of adding a French localization (without the country identifier) to the message property search. For this localization, we have translated the user message properties, for example, `SchemaException.text`, but not the log message properties, for example, `SchemaException.logtext`. A search for `SchemaException.text` results in a search of the message property file:

```
com_sqribe_jdbcacc_JDBCacc_fr.properties
```

On the other hand, a search for `SchemaException.logtext` results in a search of the following message property files:

```
com_sqribe_jdbcacc_JDBCacc_fr.properties
```

```
com_sqribe_jdbcacc_JDBCacc.properties
```

The French message property file extends the default message property file, supplying localized user message patterns.

## Services

In addition to message processing, the message facility offers diagnostic support. The writing of a message triggers support. The support tags are:

- **Diagnostics**—Instantiates a class to perform diagnostics related to the cause of the message. This may be used in a production or debugging mode to capture information affiliated with a message, where insufficient log information would be available.

  The value of this property is the fully qualified class name of the diagnostic object. The class must have a public constructor of the form: classname(String pMsgid, Object[] pSubs). The constructor should perform the diagnostics and clean up its resources. The message facility does not retain a reference to the instantiated diagnostic class.

  You would use this feature to report supplemental diagnostic information as the result of a message and continue the normal processing flow for the message. This kind of support is useful when the condition causing the message is intermittent, environment specific, or timing related. In such cases it may be impractical to run an application trace or change the executable.

- **Exception**—Throws an exception for the given class name. This may be used to drive a specific form of error recovery. The compiler will not be able to detect the throw class. The class must have a constructor of the form: classname(String pMsgid, Object[] pSubs). You can use this tag to override the application behavior associated with a message. For example, to obtain a stack trace and terminate the application: nest a throw inside the exception constructor, catching the resulting exception, and use the `printStackTrace()` method to send the trace to System.err. Subsequently, the message facility will throw this exception.

# Property Facility

The property facility is part of the common utility component. Properties and capabilities act like java.util.Properties and java.util.PropertyResourceBundles with these notable additions. The property facility

- Retains capability values as objects, rather than strings.
- Can secure property and capability values.
- Has associated properties and capabilities metadata, providing a mechanism for applications to use and manipulate them without prior knowledge.
- Can be hierarchically structured properties and capabilities; for example, the logon property is a structured property whose default attributes are user and password.
- Can compute properties and capabilities on the fly; for example, retrieve the value from the data source or represent a virtual or class attribute.
- Can associate properties and capabilities with user dialogs.

Properties and capabilities are value and metadata components stored in property resource bundles. Both the values and the metadata component attributes can be localized. The property facility provides for get and set accessor methods. Drivers use these methods to reflect its capabilities. However, applications should not use the capabilities set accessor methods.

Capabilities are read-only properties, including:

- Static attributes of the underlying data source; for example, the maximum length of a command or the maximum number of columns in a select statement.
- Drive attributes; for example, a driver supports the procedure interface or a driver supports concurrent operations within a connection. Locating Properties and Capabilities

## DataSource Class

Scanning through the *properties* directory, you will notice that many of the properties and capabilities refer to the DataSource class. The DataSourceAdapter provides default processing for data sources. Each driver extends the DataSourceAdapter and may have additional or overriding properties and capabilities. The DataSourceAdapter creates a property sheet for the data source. To create the property sheet, it generates a copy of the basic properties and capabilities and then augments the copy with the driver specific properties and capabilities. We say the driver data source inherits the basic data source properties and capabilities.

A driver connection uses a copy of the properties and capabilities for the data source. This is done in an analogous manner to data source property sheet creation. There is a Connection interface, a ConnectionAdapter, and each driver implements a connection object that extends the ConnectionAdapter. The driver connection objects may provide specific properties and capabilities. This pattern is followed throughout Production Reporting DDO.

While the driver usage pattern is different than that used for the message facility, the property resource bundle processing for the property facility is the same. Given a class, say com.sqribe.jdbcacc.JDBCDataSource, in a driver, the properties facility will fabricate a property resource bundle file name from this fully qualified class name. The fabricated name is given to the resource bundle to locate a property resource bundle. If the property resource bundle was located, then a property adapter is used to merge the contents of the bundle into the given property container. The StringPropertyAdapter is used to merge properties. The ObjectPropertyAdapter is used to merge capabilities. These adapters implement the PropertyAdapter interface. The ObjectPropertyAdapter creates objects for attribute values, while the StringPropertyAdapter uses strings for attribute values.

Given a class, say com.sqribe.jdbcacc.JDBCDataSource, in a driver, the properties facility will fabricate a property resource bundle file name from this fully qualified class name. The fabricated name is given to the resource bundle to locate a property resource bundle. If the property resource bundle was located, then a property adapter is used to merge the contents of the bundle into the given property container. The StringPropertyAdapter is used to merge properties. The ObjectPropertyAdapter is used to merge capabilities. These adapters implement the PropertyAdapter interface. The ObjectPropertyAdapter creates objects for attribute values, while the StringPropertyAdapter uses strings for attribute values.

## Property Descriptions

A PropertyDescription describes the attributes of a property. Property descriptions are common to both property and capability metadata. A property may be required or optional. The property has a name, which is its key. A property has a descriptive name. This is a short explanation of the property that may be appropriate for a tooltip. A property takes a value or value set. A property value may be secured; for example, a passphrase. This means that the value is stored in memory and transferred in an encrypted form. These values are always passed as Strings. Values are expressed as type specific Java objects, for example, java.lang.Boolean. A value may have domain requirements in the form of a range or list. The value may be indexed, that is, it may have multiple values, or be transferred as an array of objects. A value may have an associated description. Hence, indexed values may have indexed descriptions.

The design goal for the property facility is to eliminate the need for a client to have prior knowledge of a data source driver to establish driver properties. That is, the client may present the property descriptions to the user to complete. The description content should contain sufficient information to present and validate a property.

**Table 18** Attribute Descriptions and Use

| Name | Description | Required | Explanation |
|---|---|---|---|
| **Name** | The fully qualified name of the property | Yes | |
| **Description** | The descriptive name of the property (for example, help, tooltip) | No | |
| **ClassName** | The class name of the value type, for example, Integer, Boolean | No | Must agree with indices |
| **AuxProc** | The class name of the class providing virtual fetch and post processing of a property or capability. This class implements the PropertyAuxProc interface. | No | Must agree with indices |
| **Required** | Whether a value is required for this property | No | Default: not required |
| **Secured** | Whether the value for this property must be encrypted | No | Default: not secured |
| **ValidationType** | The type of validation: none, range, or list | No | Default: no validation |
| **ValidationValues** | The validation values: none:null; range: 0(min), 1(max); list: discrete values. Note: values are specified as strings and are converted by the validator to the proper type | No | Must agree with validation type |
| **Validator** | The class name of the validation implementation of the interface, PropertyValidator. **See Also** PropertyValidator | No | Default: no validator |
| **Indices** | The property descriptions of the indexed values. When there are indexed values, then classname is ignored; otherwise classname must be specified | No | Must be consistent with validation values |
| **Dialog** | The class name of a custom property dialog to display (and, optionally, validate) this property and its property subtree. | No | Default: no custom dialog |

The attribute keys are scoped names whose final component represents one of the names in the table. The only required attribute is the key with the Name extension. Other attributes are context

sensitive. When the Indices extension is specified, indicating a structured property non-leaf node, then these attributes should not be specified: ClassName, AuxProc, Secured, ValidationType, ValidationValues, and Validator. For a leaf node, the ClassName is required. This indicates the type of object the value should represent.

For example, ClassName with a value of java.lang.Integer, indicates that integer values (or a numeric string) will be acceptable. This provides syntactic validation of the value. Adding a ValidationType and ValidationValues, provides some simple semantic checks. A Validator may be associated with the property to perform the checks specified in the ValidationType and ValidationValues. Validators are provided for Java primitive types and selected objects.

### Property Description Merge

Let's follow the property description merge for the class, `com.sqribe.jdbcacc.JDBCDataSource`.

1. Obtain the class name.
2. Descend the implementation graph for the class:
   - Look in the class, com.sqribe.jdbcacc.JDBCacc
   - Look in the class, com.sqribe.access.Access
   - Look in the class, com.sqribe.comutil.Util
   - Merge property descriptions while ascending this graph.
3. Descend the inheritance graph for the class:
   - Look in the class, com.sqribe.access.DataSourceAdapter
   - Look in the class, com.sqribe.access.DataSource
   - Look in the class, com.sqribe.access.Access
   - Look in the class, com.sqribe.comutil.Util
   - Merge property descriptions while ascending this graph.

Each time a class is considered for a property merge, the class name is converted to a property resource bundle name. In this example, com.sqribe.jdbcacc.JDBCacc would be converted to properties.com_sqribe_jdbcacc_JDBCacc_PropertyDescriptions*. The asterisk ('*') in the name indicates where the resource bundle search appends the localization suffix.

## Property Sheets

So, far we have discussed properties, capabilities, and property sheets, but have not described their usage. Let's focus on the how to access property sheets. A PropertySheet encapsulates the behaviors for properties, capabilities and their descriptions. A PropertySheet is a composite object. The secure property and capability descriptions and values are accessible through a property sheet.

Properties, capabilities, and their descriptions are stored in property resource bundles. These files are relative to the directory containing the class whose properties or capabilities were

requested. Specifically, they are located in the directory "properties" which is in the directory where the class (or jar) was loaded. The resource bundle path name is of the form:

```
properties/<package name with '.' replaced by '_'>_<class
name>_<type>.properties
```

For example, the property file for com.sqribe.access.DataSource would be:

```
properties/com_sqribe_access_DataSource_Properties.properties
```

```
Where:
```

```
com_sqribe_access is the package name,
```

DataSource is the class name, and Properties is the type.

The types are:

- **Properties**—Property values for the given class. Properties are configurable attributes of a feature or facility.
- **PropertyDescriptions**—Property descriptions for the given class. Property descriptions define the characteristics of a property. There is sufficient information in a description to formulate and syntactically validate a property. That is, the description provides information and classes for explaining the use of a property and methods to create and validate the property value.
- **Capabilities**—Capability values for the given class. Capabilities are (read only) configured attributes of a feature or facility.
- **CapabilityDescriptions**—Capability descriptions for the given class. Capability descriptions define the characteristics of a capability. There is sufficient information in a description to formulate and syntactically validate a capability. That is, the description provides information and classes for explaining the use of a capability and methods to create and validate the capability value.

## Property Sheet Methods

Property sheet instance methods available to applications include:

- **copy**—Merge the given property sheet into the current property sheet.
- **getProperty**—Retrieve the named property value.
- **setProperty**—Change or add a property value. While getProperty takes the property name as the key, this polymorphic method takes either the property description object or the property name as the key. This allows property descriptions to be added on the fly.
- **getPropertyNames**—Retrieve an enumeration of the property names. This list is derived from the property values, rather than the property descriptions, container.
- **getPropertyDescription**—Retrieve the named property description.
- **setPropertyDescription**—Add or replace a property description with the given property description.
- **getPropertyDescriptionNames**—Retrieve an enumeration of the property description names.

- **getCapability**—Retrieve the named capability. While retrieving a property value will return a string or null, retrieving a capability will return an object or null.
- **getCapabilityNames**—Retrieve an enumeration of the capability names. This list is derived from the capability values, rather than the capabilities descriptions, container.
- **getCapabilityDescription**—Retrieve the named capability description.
- **getCapabilityDescriptionName**— Retrieve an enumeration of the capability description names.

## Retrieving Properties and Capabilities

Let's look at the flow for retrieving the properties and capabilities for a connection to an Oracle database through the Production Reporting DDO relational database driver. The processing for four resource bundle types, which are properties, property descriptions, capabilities, and capability descriptions. The result is a new property sheet for the data source.

1. Process a class interface merge:
    - An application uses the data source class name of the Production Reporting DDO relational database driver, com.sqribe.jdbcacc.JDBCDataSource as the basis for locating properties and capabilities.
    - The property facility descends the interface graph for this class.
    - Once at the leaf, the facility descends the inheritance graph for the leaf class.
    - Once at the inheritance leaf, the facility obtains the property resource bundle for the leaf class.
    - The facility ascends the inheritance graph, merging attributes from the property resource bundles.
    - Once at the root of the inheritance graph for the leaf in, the facility ascends to the interface graph, performing the inheritance graph merge for each interface.
2. Process an attribute merge:
    - The facility moves to the inheritance graph.
    - Make a complete descent and perform a bottom-up attribute merge. Note, an inherited class may implement interfaces; hence, interface merge processing occurs while ascending the inheritance graph.
    - Merge the attributes contained in the com.sqribe.jdbcacc.JDBCDataSource.
3. At this point the driver may merge data source specific attributes.

    In our example, the driver would be merging attributes specific to an Oracle data source. This is not done for the Production Reporting DDO relational database driver.
4. The application may alter or augment the properties contained in the new property sheet.

    The modifications are scoped to the given property sheet. In this example, the data source is associated with a specific Oracle database instance. It may, for example, be reasonable to use a single user name and password for all connections to this data source for the life of this DataSource object. In this case these logon properties would be set once and used for

all connections; that is, all connections would be opened using the same user name and password properties.

On the other hand, the application may need a different user name and password for each connection. This is done by setting the user name and password properties prior to opening a specific connection.

What makes both of these scenarios possible, is the copy of the property sheet made for the connection instance. This way property sheet changes made through the connection object do not affect the data source object and vice versa.

The Production Reporting DDO relational driver supports concurrent operations on a given connection, when the underlying relational database supports this level of concurrence. The property sheet for the connection is shared by all threads. Hence, changes to the connection property sheet will be observed by all threads.

### Localization Example

Let's look at the effect of adding a French localization (without the country identifier) to the property search. For this localization, we have translated the logon property descriptions. When we instantiate the Production Reporting DDO relational database driver, these property descriptions are merged into the property sheet.

```
com_sqribe_access_DataSource_PropertyDescriptions

com_sqribe_access_DataSource_PropertyDescriptions_fr

com_sqribe_jdbcacc_JDBCDataSource_PropertyDescriptions

com_sqribe_jdbcacc_JDBCDataSource_PropertyDescriptions_fr
```

The French property description files extend the default property description files, supplying in this case localized description text. This algorithm is different from simple java.util.ResourceBundle processing, where a localized resource bundle represents a complete replacement of the less specific resource bundle. Here, the localized resource bundle extends the less specific bundle.

## Secure Properties and Capabilities

A distinguishing characteristic of the property facility is the encryption of secret values. One of the attributes given in a property (or capability description) is whether the associated value should be secured. If so, the value is retained in an encrypted form. When a get accessor method requests the attribute, a decrypted copy is made. The decrypted copy should be a temporary (local) variable, discarded when the containing block goes out of scope.

## Property Auxiliary Services

The PropertyAuxProc interface is used to implement specialized runtime processing. These methods are used to obtain driver attributes (either properties or capabilities) that are not part of the property/capability resource bundles, per se; rather, these properties or capabilities are

supported through some other mechanism; for example, they may be attributes of the underlying data source that cannot (or should not) be persistent in a property/capability resource bundle.

The get and set methods of this interface are independent. The use of the get method to obtain a virtual property does not imply the use of the set method to post process the property. Likewise, employing the set method to update class attributes does not imply that the get method is used to create those attributes.

While the get method, typically, acquires the information, it should arrange for caching the value in the hash table. This implies that the get method should have some first time logic. Further, properties (and capabilities) utilizing this interface are not saved, that is, made persistent. This implies that the get method could test for the presence of the property in the hash table as the first time logic, suffering the cost of a hash table lookup for each test.

## Property Validators

A PropertyValidator provides an interface to property validation classes. In addition, the AbstractPropertyValidator class implements major portions of the interface, reducing the effort to two methods: createValue and compare. *Program ex24.sqr* shows the BigDecimalValidator to provide a feel for validators.

### Program ex24.sqr

```
public class BigDecimalValidator extends AbstractPropertyValidator {
    /**
     * Create the value from a string
     * @param pValue        The value to be converted
     * @return              The converted value
     * @exception PropertyException
     *                      Generic exception indicating that the value
     *                      could not be converted
     */
    public Object createValue(String pValue) throws PropertyException {
        try {
            return new BigDecimal(pValue);
        } catch (Throwable e) {
            throw new PropertyException(e.getMessage());
        }
    }

    /**
     * Compare to values
     * @param pValue1       First comparand
     * @param pValue2       Second comparand
     * @return              First<Second==-1;
 *                      First==Second==0;
 *                        First>Second==1
     * @exception PropertyException
     *                      Generic exception indicating that the value
     *                      was not the correct data type
     */
    public int compare(Object pValue1, Object pValue2) throws
                    PropertyException {
```

```
        return (((BigDecimal)pValue1).compareTo((BigDecimal)pValue2));
    }
}
```

The common utilities package provides implementations for Java primitives and Java SQL types. These validators extend AbstractPropertyValidator:

- BigDecimalValidator
- BooleanValidator
- ByteValidator
- CharacterValidator
- DateValidator
- DoubleValidator
- FloatValidator
- IntegerValidator
- LongValidator
- ShortValidator
- StringValidator

Part II

# Using Production Reporting DDO to Access Data

In Using Production Reporting DDO to Access Data:


- Using Production Reporting DDO to Access SAP R/3 Data
- Using Production Reporting DDO to Access an SAP BW Data Source
- Using Production Reporting DDO to Access Essbase Cubes
- Using Production Reporting DDO to Access MSOLAP Cubes

# 7 Using Production Reporting DDO to Access SAP R/3 Data

**In This Chapter**


Data Access Requirements ..... 123
Using the Registry Editor to Make an SAP R/3 Connection ..... 123
Using the Query Editor to See the SAP Tree Structure ..... 124
Using SQR Production Reporting Studio to Build a Report with a BAPI ..... 125
Understanding the Production Reporting Code for SAP R/3 ..... 126


## Data Access Requirements

Before accessing SAP R/3 data, ensure that you have:

- Connectivity values
- BAPI parameters

  For a typical BAPI, you need:

  - Host Name (IP Address)
  - System Number
  - Client Name
  - Language
  - User Name
  - Password

    In addition, BAPIs usually have their own individual parameters. Contact your SAP R/3 Administrator for parameter information and values.
- Software
  - Production Reporting DDO
  - Oracle's Hyperion® SQR® Production Reporting Studio

## Using the Registry Editor to Make an SAP R/3 Connection

The Production Reporting DDO Registry Editor allows you to make your SAP R/3 connection work like an ODBC connection.

➤ To use the Production Reporting DDO Registry Editor:

1. Select **Start > Programs > Oracle EPM System > Reporting and Analysis > Production Reporting for DDO > Registry Editor**.
2. Select **File > Open** or click the **Open** button on the toolbar.
3. Highlight **Registry** and click **Open**.
4. Click **Add**.
5. Highlight **sapr3acc** and click **OK**.
6. Do the following in the Setup Data Source dialog box that appears.
   a. Enter a name and description for the data source.
   b. Click **Build**, enter the following parameters, and click **OK**.
      - **Host**—IP address where SAP R/3 system data resides
      - **System #**—Get from SAP R/3 Administrator
      - **Client**—Get from SAP R/3 Administrator
      - **Language**—Get from SAP R/3 Administrator
7. Click **Test**.
8. Enter your username and password in the Logon dialog box.

   If you do not know your username and password, contact your SAP Administrator.

# Using the Query Editor to See the SAP Tree Structure

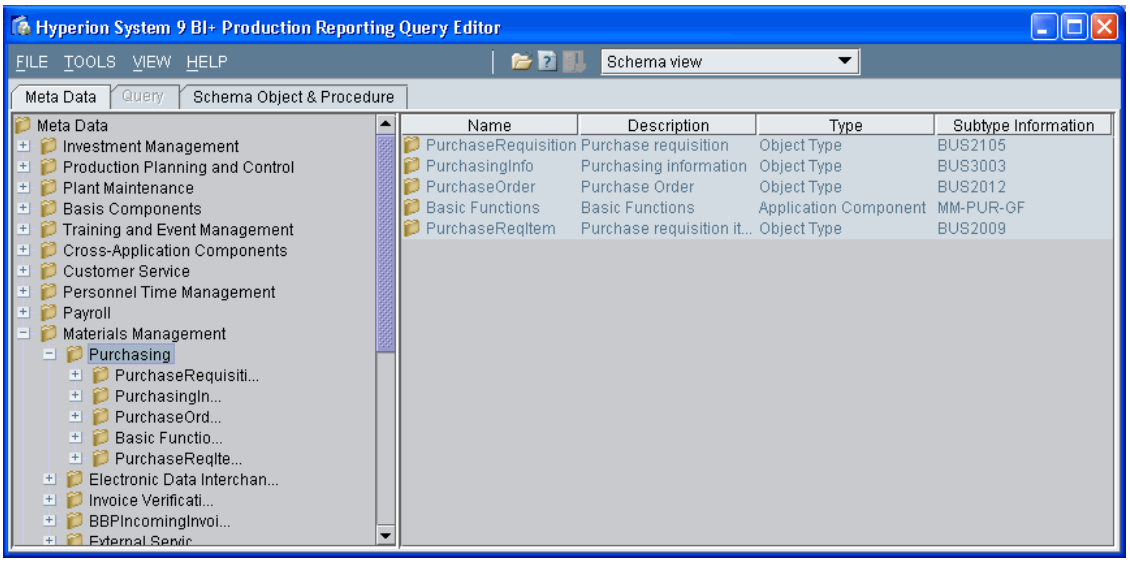
The Production Reporting DDO Query Editor allows you to see the SAP tree structure as well as the parameters and return variables SAP requires/passes.

➤ To use the Production Reporting DDO Query Editor:

1. Select **Start > Programs > Oracle EPM System > Reporting and Analysis > Production Reporting for DDO > Query Editor**.
2. Select **File > Open** or click the **Open** button on the toolbar.
3. Highlight the data source under the Registry folder and click **Open**.
4. Enter your username and password.

   You may have to double click the Meta Data folder, but it should automatically display the SAP tree structure

   By clicking the desired folders, you can see how BAPIs and their variables relate.




# Using SQR Production Reporting Studio to Build a Report with a BAPI

➤ To create a report using a BAPI:

1. Select **Start > Programs > Oracle EPM System > Reporting and Analysis > Production Reporting Studio.**
2. Select a report type to access the Query Builder..
3. Click **New** on the Connection tab and enter the data source name.
4. Click **DDO.**
5. In the Registry folder, find your Production Reporting DDO source.

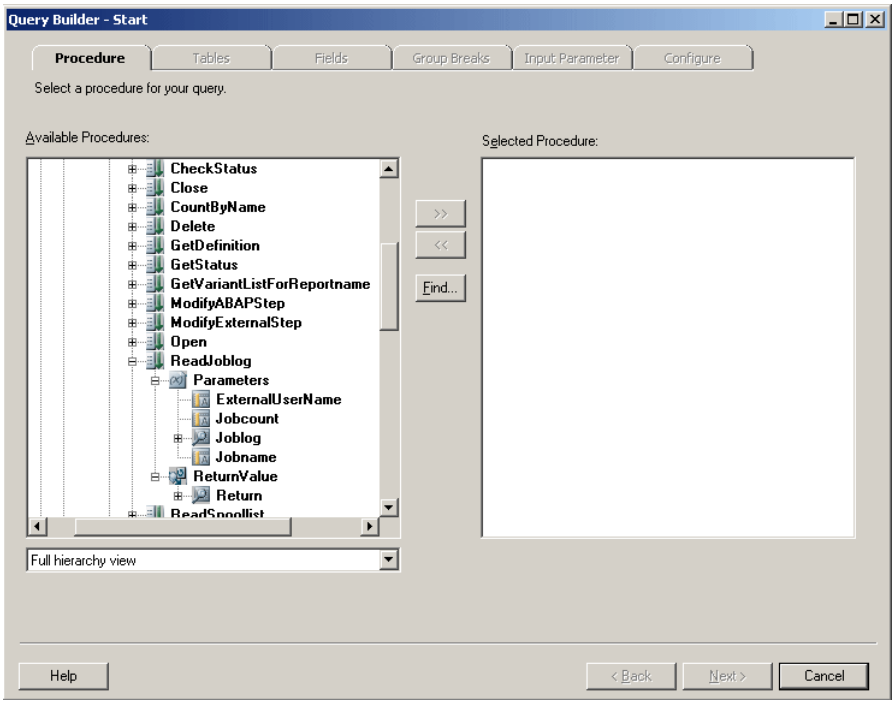
   If it is not there, you will have to re-enter some parameters based on your creation of the Registry Editor.
6. Enter your username and password in the DDO Logon dialog box

   You may need to get this from your SAP administrator.
7. Build a query on the remaining tabs in the Query Builder.

   As you open folders in the Query Builder, you will be able to see inside BAPIs. BAPIs usually require parameters, RETURN variables and TABLES of variables.

   The 'ReadJobLog' BAPI, shown in the following figure, includes:

   - Parameters: Jobname, Jobcount, ExternalUserName
   - Parameters: TABLE: JobLog
   - Return Values: TABLE: Return




➤ To select the 'ReadJobLog' BAPI:

1 **Highlight ReadJobLog, click Add, and click Next.**

You can only add one BAPI at a time.

2 **Advance through the remaining Query Builder pages until you get to the Layout window.**

3 **Process and save the report.**

**Note:**

For more information on using BAPIs to create reports in SQR Production Reporting Studio, see "Creating SAP R/3 Reports" in the *Hyperion SQR Production Reporting Studio User's Guide*.

# Understanding the Production Reporting Code for SAP R/3

Following is the code built with SQR Production Reporting Studio in the earlier example.

```
!-------------------------------------------------------------------------------
! Generated on Sun Mar 03 23:52:30 2002 by SQR Production Reporting Studio 9.0
!
! Filename: C:\Documents and Settings\nmoscaritolo.NICOLAS2K\Desktop\Untitled.sqr
! Format  : Tabular
! Username: Oracle
!-------------------------------------------------------------------------------
Begin-Setup
 Declare-Layout Default
  Orientation = Portrait
  Paper-Size = (Letter)
```

```
  Top-Margin    = 0.500
  Bottom-Margin = 0.500
  Left-Margin   = 0.500
  Right-Margin  = 0.500
  Line-Height = 1
  Char-Width  = 1
 End-Declare
! procedure parameters
!***** These are ALL of the parameters and return variables within the selected BAPI.
Declare-Variable
   date $job_protocol_entertime    !type=date(Time),width=8,size=6,precision=0
   date $job_protocol_enterdate    !type=date(Date),width=10,size=8,precision=0
   text $job_protocol_msgid    !type=char(Char),width=20,size=20,precision=0
   decimal #job_protocol_msgno    !type=decimal(Number),width=3,size=3,precision=0
   text $job_protocol_text    !type=char(Char),width=200,size=200,precision=0
   text $job_protocol_rabaxkey    !type=char(Char),width=64,size=64,precision=0
   decimal #job_protocol_rabaxkeyln    !type=decimal
(Integer),width=10,size=4,precision=0
   text $job_protocol_msgv1    !type=char(Char),width=50,size=50,precision=0
   text $job_protocol_msgv2    !type=char(Char),width=50,size=50,precision=0
   text $job_protocol_msgv3    !type=char(Char),width=50,size=50,precision=0
   text $job_protocol_msgv4    !type=char(Char),width=50,size=50,precision=0
   text $job_protocol_program    !type=char(Char),width=40,size=40,precision=0
   text $job_protocol_pfkey    !type=char(Char),width=20,size=20,precision=0
   text $job_protocol_dynpro    !type=char(Char),width=4,size=4,precision=0
   text $return_type    !type=char(Char),width=1,size=1,precision=0
   text $return_id    !type=char(Char),width=20,size=20,precision=0
   decimal #return_number    !type=decimal(Number),width=3,size=3,precision=0
   text $return_message    !type=char(Char),width=220,size=220,precision=0
   text $return_log_no    !type=char(Char),width=20,size=20,precision=0
   decimal #return_log_msg_no    !type=decimal(Number),width=6,size=6,precision=0
   text $return_message_v1    !type=char(Char),width=50,size=50,precision=0
   text $return_message_v2    !type=char(Char),width=50,size=50,precision=0
   text $return_message_v3    !type=char(Char),width=50,size=50,precision=0
   text $return_message_v4    !type=char(Char),width=50,size=50,precision=0
   text $return_parameter    !type=char(Char),width=32,size=32,precision=0
   decimal #return_row    !type=decimal(Integer),width=10,size=4,precision=0
   text $return_field    !type=char(Char),width=30,size=30,precision=0
   text $return_system    !type=char(Char),width=10,size=10,precision=0
 End-Declare
End-Setup

Begin-Program
 Position (1,1)
 Do Master_Query
End-Program

Begin-Procedure Master_Query
! ***** This is not needed but it demonstrates how you could ALTER your DDO connection
Alter-connection
    Name=default

Parameters=logon.trace=0;logon.client=850;logon.check=1;logon.type=3;logon.sysnr=00;logo
n.language=EN;logon.ashost=/H/10.215.22.227/H/204.79.199.244/H/172.20.11.6;
! ***** These three parameters are REQUIRED from you and your SAP Administrator
! list and user assigned input parameters
 let $jobname = '123ewqsa'    !type=char(Char),width=32,size=32,precision=0
```

```
 let $jobcount = '123123'    !type=char(Char),width=8,size=8,precision=0
 let $external_user_name = 'Oracle'    !type=char(Char),width=16,size=16,precision=0
! ***** This new '%' variables collect all of the returned data into a LIST variable for
later use
let %job_protocol = list( $job_protocol_entertime, $job_protocol_enterdate,
$job_protocol_msgid,
                          #job_protocol_msgno, $job_protocol_text,
$job_protocol_rabaxkey,
                          #job_protocol_rabaxkeyln, $job_protocol_msgv1,
$job_protocol_msgv2,
                          $job_protocol_msgv3, $job_protocol_msgv4,
$job_protocol_program,
                          $job_protocol_pfkey, $job_protocol_dynpro )
 let %return = list( $return_type, $return_id, #return_number, $return_message,
$return_log_no,
                          #return_log_msg_no, $return_message_v1, $return_message_v2,
                          $return_message_v3, $return_message_v4, $return_parameter,
#return_row,
                          $return_field, $return_system )

! ***** BEGIN-EXECUTE is required for DDO
 Begin-Execute
! ***** SCHEMA can be retrieved from your SAP Administrator. SAP calls it Business
Object Type
    SCHEMA='XBPJOB' ! Basis Components(BC);Computing Center Management System(BC-
CCM);Application Programming Interfaces(BC-CCM-API);Complementary Software Interfaces
(BC-CCM-API-CSI);BackgroundJob(XBPJOB)
! ***** PROCEDURE is the BAPI name
    PROCEDURE='READJOBLOG' ! ReadJoblog[READJOBLOG]
! ***** PARAMETERS are found in the parameter section of the BAPI
    PARAMETERS=($jobname IN, $jobcount IN, $external_user_name IN, NULL)
! ***** STATUS return the SQR return LIST variables which contains valuable debugging
values and messages
    STATUS=%return
   Print-Direct printer=html '%%ResetColor'
   Print-Direct printer=html '%%ResetBorder'
! ***** BEGIN-SELECT is used to select the SAP TABLE structure where the FROM PARAMETER
command uses the TABLE !NAME within the BAPI
Begin-Select
 Alter-Printer Font=901 Point-Size=10    ! [SQR.INI] 901=MS Shell Dlg,proportional
ENTERTIME &Master_Query_ENTERTIME=date  (12,1) Edit MM/DD/YYYY   !type=date
(Time),width=8,size=6,precision=0
ENTERDATE &Master_Query_ENTERDATE=date  (12,129) Edit MM/DD/YYYY   !type=date
(Date),width=10,size=8,precision=0
   Print-Direct printer=html '%%ResetColor'
 Next-Listing  Need=12
From  Parameter = 'JOB_PROTOCOL'
End-Select
 End-Execute
 Next-Listing
   Print-Direct printer=html '%%ResetColor'
   Print-Direct printer=html '%%ResetBorder'
End-Procedure
Begin-Heading 48
   Print-Direct printer=html '%%ResetColor'
   Print-Direct printer=html '%%ResetBorder'
 Alter-Printer Font=901 Point-Size=10    ! [SQR.INI] 901=MS Shell Dlg,proportional
```

```
 Print $current-date (12,1) edit 'MM/DD/YYYY'
 Page-Number (12,517)
 Print 'Entertime' (42,1,9) Underline Bold
 Print 'Enterdate' (42,129,9) Underline Bold
 Alter-Printer Font=901 Point-Size=10
End-Heading
```

# 8 Using Production Reporting DDO to Access an SAP BW Data Source

In This Chapter


Accessing the SAP BW OLAP Server ..... 131
Supported Platforms ..... 131
Copying Files to the /lib Directory ..... 132
Adding the SAP BW Data Source to the Registry.properties File ..... 132
The Hierarchical Structure of Objects for an SAP BW Data Source ..... 133
SAP BW and the Production Reporting Language ..... 135
Accessing SAP BW Data from SQR Production Reporting Studio ..... 148


## Accessing the SAP BW OLAP Server

The DDO SAP BW driver is built using the SAP Java Connector JAR file and shared libraries, and the SAP RFC shared libraries. These components, necessary to access the SAP BW OLAP server, are not distributed with Production Reporting DDO. To obtain copies of these files, go to the SAP official connector's site at `https://websmp104.sap-ag.de/connectors`.

## Supported Platforms

The DDO SAP BW driver is supported on the following platforms:

**Table 19 Supported Platforms**

| Platform | Operating System |
|---|---|
| Windows | Windows 2000, Windows 2000 Server, Windows XP, Windows 2003 Server |
| Solaris | 5.8 and 5.9 |
| HP-UX | B.11.11 |
| HP/Itanium | B.11.23 |
| IBM-AIX | 5.1 and 5.2 |
| Linux | AS 3.0 and 4.0 |

# Copying Files to the /lib Directory

Before connecting to SAP BW, you need to copy some files to the `/lib` directory of your installation.

- For Windows platforms:

  After you install Production Reporting DDO, copy the following files to the `%HYPERION_BIPLUS_HOME%/lib` directory:

  - `sapjco.jar`
  - `sapjocrfc.dll`
  - `librfc32.dll` (version 6.20.6 or greater of the `librfc32.dll` is required)

- For UNIX platforms:

  After you install Production Reporting DDO, copy the following files to the `$HYPERION_BIPLUS_HOME/lib` directory:

  - `sapjco.jar`
  - `libsapjcorfc.<so, sl>`
  - `librfccm.<so, sl, o>` (version 6.20.6 or greater of the `librfccm` shared library is required)

    For UNIX platforms, you must update the following environment variable to include the `/lib` directory:

    `LD_LIBRARY_PATH/SHLIB_PATH/LIBPATH`

# Adding the SAP BW Data Source to the Registry.properties File

To use the DDO SAP BW driver, you must first add a SAP BW data source to the `Registry.properties` file in the DDO Registry Editor.

**Note:**

The DDO Registry Editor provides a graphical interface for managing data sources. For more information, see Chapter 5, "Managing Data Sources."

➤ To add the SAP BW data source to the `Registry.properties` file:

1. Start the Registry Editor by selecting **Start > Programs > Oracle EPM System > Reporting and Analysis > Production Reporting for DDO > Registry Editor** to display the editor's main window; then, select **File > Open** or click the **Open** button on the toolbar.
2. Select **Registry** in the Open Registry window and click **Open**.
3. Click **Add** in the Registry Editor main window.
4. Click the **bwacc** (SAP BW OLAP) driver name in the Create New Data Source window and click **OK**.
5. Enter the SAP BW data source information in the Setup Data Source dialog box and click **Build**.

6. Enter the information in the Build Connection String dialog box and click **OK**.
7. The connection string appears in the Setup Data Source dialog box. Click **Test** to test the connection.
8. Enter a valid username and password in the Logon dialog box and click **OK**.
9. If the connection succeeded, a Test Data Source dialog box appears. Click **OK** to view the SAPBW data source in the Registry Editor
10. Select **File > Save** or click the Save button to save the data source.

# The Hierarchical Structure of Objects for an SAP BW Data Source

Figure 8 shows the hierarchical structure of the objects for an SAP BW Data Source.

**Figure 8 Hierarchical Structure of Objects for an SAP BW Data Source**

Information Providers

-----InfoCubes

-----QueryCubes

-----ODS Objects

-----InfoSets

---------Characteristics

----------CharacteristicLevels

----------CharacteristicLevelMembers

----------CharacteristicChildMembers

----------CharacteristicMembers

----------CharacteristicChildMembers

----------Properties (Optional)

----------Property

----------PropertyMembers

----------MandatoryProperty

---------NavigationalAttribute

----------NavigationalAttributeLevels

----------NavigationalAttributeLevelMembers

----------NavigationalAttributeMember

----------NavigationalAttributeMembers

----------NavigationalAttributeMember

----------Hierarchy

----------HierarchyLevels

----------HierarchyLevelMembers

----------HierarchyChildMembers

----------HierarchyMembers

----------HierarchyChildMembers

----------Properties (Optional)

----------Mandatory Properties

---------Key Figures

----------Key Figure Attributes

---------SAP Variables (QueryCubes only)

----------SAP Variable—Single

----------SAP Variable—Interval

----------SAP Variable—Single Mandatory

----------SAP Variable—Interval Mandatory

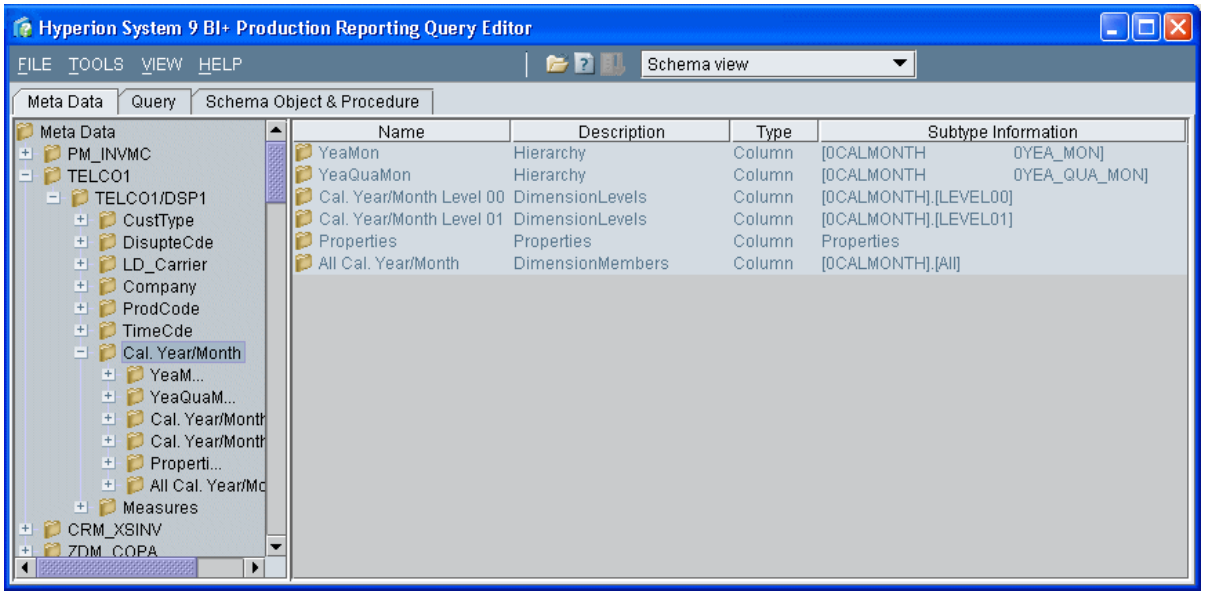
----------SAP Variable—Single MandatoryNol

----------SAP Variable—Interval MandatoryNol

----------SAPVariableMembers

Figure 9 shows the hierarchies for a single characteristic.

**Figure 9 Hierarchies for a Single Characteristic**



# SAP BW and the Production Reporting Language

The Production Reporting language includes the following functionality to enable SAP BW processing.

- Accessing Dimension Properties
- Specifying Dimension Members
- Specifying the Order in Which to Return Dimension Members
- Restricting the Returned Result Set
- Limiting the Set of Values Used for a Dimension
- Using SAP BW Variables
- Returning a Set of Descendants
- Finding a Dimension's Ancestor
- Defining Calculated Key Figures, Restricted Key Figures, and Calculated Members

**Note:**

The SET-GENERATIONS and SET-LEVELS arguments in the DECLARE-CONNECTION and ALTER-CONNECTION commands are *not* supported when using the DDO SAP BW driver. Production Reporting ignores these arguments when it is connected to a SAP BW data source.

**Note:**

The Production Reporting language syntax in this section refers to characteristics as dimensions and key figures as measures.

## Accessing Dimension Properties

Use the following syntax in `BEGIN-SELECT` to select a dimensions' property values.

**Note:**

See BEGIN-SELECT in Volume 2 of the *Production Reporting Developer's Guide*.

### Syntax

```
{[$Dimension.Propertyname] $synonym = (char|number|date)}
```

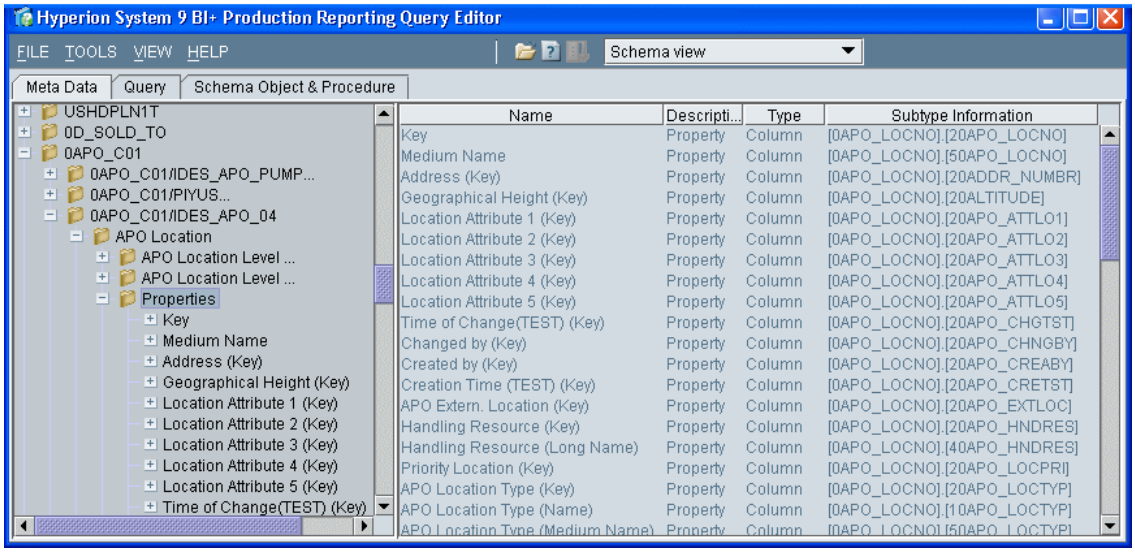
### Arguments

*`Dimension.Propertyname`*

The dimension's' property value. You cannot request a dimension's property value without first requesting the dimension.

### Example

This example illustrates what is returned from the select of a dimension property. The screen below shows a small set of the available properties for the 0APO_LOCNO dimension.




The following returns the 0APO_LOCNO, the 0APO_LOCNO address key, and the selected measures.

```
begin-select
    0APO_LOCNO
    0APO_LOCNO.20ADDR_NUMBER
    Measures.D5XK1R2CAGKTXDWNCTIDVK8RI
    from SAP BW
end-select
```

The result set from this request is:

| 0APO-LOCNO | 20ADDR_NUMBER | Measures.D5XK1R2CAGKTXDWNCTIDVK8RI |
|---|---|---|
| All APO Locations | | 210.720 |
| New York | 10294 | 210.720 |

## Specifying Dimension Members

Use the `WHERE` clause to specify the dimension members to use in the query.

### Syntax

```
WHERE VAR [VAR]=(txt_var|_col|_lit)
```

### Arguments

`VAR`

The string variable, column, or literal that represents a legal dimension or measure.

### Description

- WHERE must immediately follow FROM.
- You can only have one WHERE in each select statement.
- If VAR names contain a space, surround the name with double quotes.
- The contents of WHERE must specify a valid dimension and one of its members.
- WHERE can specify a dimension previously declared by SET-MEMBERS to further restrict the selection.
- Only expressions that resolve to a single column and row intersect are allowed.
- WHERE implicitly supports logical AND operations involving members across different dimensions. To support logical AND operations involving members within a *single dimension*, declare SET-MEMBERS.
- You can use standard Production Reporting variable references ($, #, &) where appropriate.
- Standard Production Reporting variable replacement ([$xxx]) is supported.

### Example

The following returns the selected 0APO_LOCNO and measures for the 38th week of 2001.

```
begin-select
    0APO_LOCNO
    Measures.D5XK1R2CAGKTXDWNCTIDVK8RI
    From SAP BW
    WHERE 0CALWEEK.200138
end-select
```

The result set from the above request *without* the WHERE clause is:

| 0APO_LOCNO | Measures.D5XK1R2CAGKTXDWNCTIDVK8RI |
|---|---|
| All APO Locations | 210.720 |
| New York | 210.720 |

The result set from the above request *with* the WHERE clause is:

| 0APO_LOCNO | Measures.D5XK1R2CAGKTXDWNCTIDVK8RI |
|---|---|
| All APO Locations | 17.861 |
| New York | 17.861 |

## Specifying the Order in Which to Return Dimension Members

Use the ORDER BY clause to specify the order the selected Measure's dimension members are returned for the requested dimension members. You can sort by a dimension property or by a selected measure.

### Syntax

```
ORDER BY [(DIM, EXPRESSION)] {SO} [(DIM, EXPRESSION) {SO}]…
```

### Arguments

DIM

A string variable, column, or literal that represents a legal dimension or hierarchy.

EXPRESSION is of the form:

```
(MEASURE[.VALUE])
```

or

```
(Dimension[.Dimension Property])
```

SO

A literal value of one of the following:

- ASC—Sort in ascending order and preserve the hierarchy.
- BASC—Sort in ascending order and break the hierarchy.
- DESC—Sort in descending order and preserve the hierarchy.
- BDESC—Sort in descending order and break the hierarchy.

SO cannot be a bind variable.

### Description

- ORDER BY must follow FROM.
- You can only have one ORDER BY within each select statement.
- ORDER BY must include a value indicating the sort order and mode.
- You can sort data in ascending or descending order, and either break or preserve the hierarchy.

### Example

The following sorts the selected measure by a specific dimension property (0APO_RTYPE, 0APO_RTYPE.20APO_RTYPE) in ascending order and breaking the hierarchy (BASC).

```
begin-select
    0APO_RTYPE
    Measures.D5XK1R2CAGKTXDWNCTIDVK8RI
    From SAP BW
    ORDER BY (0APO_RTYPE, 0APO_RTYPE.20APO_RTYPE) BASC
end-select
```

### Example

The following sorts the selected measure by two dimension properties (ZN_STATE, ZN_STATE.1ZN_STATE) and (ZN_SKU, ZN_SKU.1ZN_SKU) in ascending order and breaking the hierarchy (BASC).

```
begin-select
    ZN_STATE
    ZN_SKU
    Measures.ZN_ADDTNS
    Measures.ZN_CLINV
    Measures.ZN_COGS
    Measures.ZN_MISC
    Measures.ZN_MRKTNG
    Measures.ZN_OPINV
    Measures.ZN_PROLL
    Measures.ZN_SALES
    From SAP BW
    ORDER BY (ZN_STATE, ZN_STATE .1ZN_STATE) BASC
             (ZN_SKU, ZN_SKU.1ZN_SKU) BASC
end-select
```

### Example

The following sorts the selected measure by a specific dimension property (`ZN_STATE, Measures.ZN_ADDTNS`) in ascending order without breaking the hierarchy (`ASC`).

```
begin-select
    ZN_STATE
    Measures.ZN_ADDTNS
    Measures.ZN_CLINV
    Measures.ZN_COGS
    Measures.ZN_MISC
    Measures.ZN_MRKTNG
    Measures.ZN_OPINV
    Measures.ZN_PROLL
    Measures.ZN_SALES
    From SAP BW
    ORDER BY (ZN_STATE, Measures.ZN_ADDTNS) ASC
end-select
```

## Restricting the Returned Result Set

Use the `FILTER` clause to restrict the returned result set.

### Syntax

`FILTER (DIM, {`*`Boolean Expression`*`} [{`*`Relation`*`} {`*`Boolean Expression`*`}]...)`

### Arguments

`DIM`

A string variable, column, or literal that represents a legal dimension.

*`Boolean Expression`*

The Boolean Expression is of the form:

`(MEASURE[.VALUE] {Operator} VALUE [SLICED BY DIM[.VALUE] [DIM[.VALUE]...])`

or

```
(Dimension[.Dimension Property] {Operator} "VALUE" [SLICED BY DIM[.VALUE]
[DIM[.VALUE]...])
```

Where operator is one of the following:

< > <= >= = !=

**BottomCount**

**BottomSum**

**BottomPercent**

**TopCount**

**TopSum**

**TopPercent**

*Relation*

The keyword AND or OR.

### Description

- `FILTER` can have any number of boolean expressions.
- You can have multiple `FILTER` clauses; however, you can only have one `FILTER` clause for a single Dimension.
- If dimension or variable names contain a space, surround the name with double quotes.
- You can use standard Production Reporting variable references ($, #, &) where appropriate.
- Standard Production Reporting variable replacement ([$xxx]) is supported.

### Example

The following returns the result set where the returned value is greater than 60 and less than 200.

```
declare-connection
    set-members('APO_RTYPE','0APO_RTYPE'.'LEVEL01')
    end-connection
    begin-select
        0APO_RTYPE
        Measures.D5XK1R2CAGKTXDWNCTIDVK8RI
        From SAP BW
        FILTER (0APO_RTYPE,
        (Measures.D5XK1R2CAGKTXDWNCTIDVK8RI > 60.00) AND
        (Measures.D5XK1R2CAGKTXDWNCTIDVK8RI < 200.00))
    end-select
end-declare
```

The result set from the above request *without* the `FILTER` clause is:

| 0APO_RTYPE | Measures.D5XK1R2CAGKTXDWNCTIDVK8RI |
|---|---|
| 4 | 57.759 |

| 0APO_RTYPE | Measures.D5XK1R2CAGKTXDWNCTIDVK8RI |
|---|---|
| 5 | 152.961 |
| 8 | |
| 9 | |
| Not Assigned | |

The result set from the above request *with* the `FILTER` clause is:

| 0APO_RTYPE | Measures.D5XK1R2CAGKTXDWNCTIDVK8RI |
|---|---|
| 5 | 152.961 |
| All Resource types | 210.720 |

## Limiting the Set of Values Used for a Dimension

Use the `EXCEPT` clause to provide additional filtering capability by limiting the set of values used for a selected dimension.

### Syntax

```
EXCEPT (DIM [,DIM]...)
```

### Arguments

`DIM`

A string variable, column, or literal that represents a legal dimension.

### Description

- You can have multiple `EXCEPT` clauses.
- If dimension names contain a space, surround the name with double quotes.
- You can use standard Production Reporting variable references($, #, &) where appropriate.
- Standard Production Reporting variable replacement ([$xxx]) is supported.

### Example

The following removes Resource Type 5 from the types returned by `SET-MEMBERS`.

```
declare-connection
    set-members=('0APO_RTYPE','0APO_RTYPE.LEVEL01')
    end-connection
    begin-select
        0APO_RTYPE
        Measures.D5XK1R2CAGKTXDWNCTIDVK8R
        From SAP BW
        Except (0APO_RTYPE.5)
```

```
    end-select
end-declare
```

The result set from the above request *without* the EXCEPT clause is:

| 0APO_RTYPE | Measures.D5XK1R2CAGKTXDWNCTIDVK8RI |
|---|---|
| 4 | 57.759 |
| 5 | 152.961 |
| 8 | |
| 9 | |
| Not Assigned | |

The result set from the above request *with* the EXCEPT clause is:

| 0APO_RTYPE | Measures.D5XK1R2CAGKTXDWNCTIDVK8RI |
|---|---|
| 4 | 57.759 |
| 8 | |
| 9 | |
| Not Assigned | |

## Using SAP BW Variables

Use the SAPVARIABLES clause to create an SAP BW QueryCube.

### Syntax

```
SAPVARIABLES
(Variable_Name {Inclusion/Exclusion Expression}
Single_Variable_Value)...|
(Variable_Name {Inclusion/Exclusion Expression}
Low_Variable_Value...)
(Variable_Name {Inclusion/Exclusion Expression}
High_Variable_Value...)
```

### Arguments

*Variable_Name*

A string variable, column, or literal that represents a legal SAP variable name.

*Inclusion/Exclusion Expression*

An expression in the form of INCLUDING or EXCLUDING.

*Single_Variable Value, Low_Variable_Value, High_Variable_Value*

A string variable, column, or literal that represents a legal SAP variable value.

### Description

- Multiple SAPVARIABLES clauses are permitted but are not required.
- You can only use SAPVARIABLES clauses with SAP BW QueryCubes.

### Example

The following processes the selected QueryCube including Resource Type 5 and Resource Types 8 -15. It excludes Resource Type 10.

```
declare-connection
        set-members=('0APO_RTYPE','0APO_RTYPE.LEVEL01')
        end-connection
begin-select
        0APO_RTYPE
        Measures.D5XK1R2CAGKTXDWNCTIDVK8R
    From SAP BW
SAPVARIABLES
(0APORTYPE INCLUDING 0APO_RTYPE.5)
(0APORTYPE INCLUDING 0APO_RTYPE.8 0APO_RTYPE.15)
(0APORTYPE EXCLUDING 0APO_RTYPE.10)
end-select
```

## Returning a Set of Descendants

Use the DESCENDANTS clause to return a set of descendants for a dimension using a level as a reference point. If no level or descendant flag is provided, the dimension and all of its descendants are returned.

### Syntax

DESCENDANTS (DIM *DIM_LEVEL DESCENDANT_FLAG*)

### Arguments

DIM

A string variable, column, or literal that represents a legal SAP dimension name.

*DIM_LEVEL*

A string variable, column, or literal that represents a legal SAP dimension level name.

*DESCENDANT_FLAG*

One of the following values:

- Self
- After

- Before
- Before_and_After
- Self_and_After
- Self_and_Before
- Self_Before_After
- Levels

### Description

- You can have multiple DESCENDANTS clauses, but only a single parameter for each dimension.
- The DESCENDANTS and ANCESTORS clauses are mutually exclusive.

### Example

The following returns only the members "AFTER" the "State" level (All Cities).

```
begin-select
        0APO_LOCATION
        Measures.D5XK1R2CAGKTXDWNCTIDVK8R
    From SAP BW
DESCENDANTS("0APO_LOCATION" "0APO_LOCATION.STATE" AFTER)
end-select
```

### Example

The following returns the "State" members (All States) and all members "AFTER" the "State" level (All Cities).

```
begin-select
        0APO_LOCATION
        Measures.D5XK1R2CAGKTXDWNCTIDVK8R
    From SAP BW
DESCENDANTS("0APO_LOCATION" "0APO_LOCATION.STATE" SELF_and_AFTER)
end-select
```

## Finding a Dimension's Ancestor

Use the ANCESTOR clause to find a source dimensions' ancestor at the target level. If no level is provided, the source dimension is returned.

### Syntax

```
ANCESTORS (DIM DIM_LEVEL)
```

### Arguments

DIM

A string variable, column, or literal that represents a legal SAP dimension name.

*DIM_LEVEL*

A string variable, column, or literal that represents a legal SAP dimension level name.

### Description

- You can have multiple ANCESTORS clauses, but only a single parameter for each dimension.
- The ANCESTORS and DESCENDANTS clauses are mutually exclusive.

### Example

The following returns the State member for the city of Dayton (Ohio).

```
begin-select
        0APO_LOCATION
        Measures.D5XK1R2CAGKTXDWNCTIDVK8R
    From SAP BW
ANCESTORS ("0APO_LOCATION.DAYTON" "0APO_LOCATION.STATE")
end-select
```

## Defining Calculated Key Figures, Restricted Key Figures, and Calculated Members

Use the WITH MEMBER clause to define calculated key figures, restricted key figures, and calculated members.

### Syntax

```
WITH MEMBER  "MemberName.CalculationName" AS ("member-formula")
```

### Arguments

MemberName

Any valid dimension or hierarchy or the key “measures”.

CalculationName

A unique name to identify the calculation.

member-formula

Any valid MDX expression.

### Description

- WITH MEMBER must immediately follow FROM.
- More than one calculated member can be defined in a WITH section.
- Accepted functions are: Aggregate(), Avg(), Max(), Median(), Min(), Sum(), Count() for standard dimensions, including TIME. Additional functions for properly constructed TIME dimensions are Ytd(), Qtd(), Mtd(), Wtd(), OpeningPeriod(), ClosingPeriod(), Periodstodate(), Parallelperiod().

- Member functions include: Ancestor(), Cousin(), Item(), Lag(), Lead (), .CurrentMember, .FirstChild, .LastChild, .FirstSibling, .LastSibling, .NextMember, .Parent, and .PrevMember
- Set functions include: Ancestors(), Ascendants(), BottomCount(), BottomPercent(), BottomSum(), CrossJoin(), Descendants(), Except(), Filter(), TopCount(), TopPercent(), TopSum(), Union(), Head(), Tail(), Distinct(), .AllMembers, .Children, .Members, and .Siblings
- To specify left and right curly brackets, use of ^| and |^ in place of the actual brackets. For example:

```
MEMBER 0APO_RTYPE.NEWAGGR AS
AGGREGATE ( ^| "0APO_RTYPE.05","0APO_RTYPE.04" |^ )
MEMBER 0D_VENDOR.Aggr AS
     'AVG( ^| 0D_VENDOR.0000001000 ,
     0D_VENDOR.0000001001,
     0D_VENDOR.0000001050 |^)'
MEMBER 0CALMONTH.Agg AS
'AGGREGATE(
^|
0CALMONTH 0YEA_MON.1999  0CALYEAR.Children 0CALMONTH
0YEA_MON.2001 0CALYEAR.Children
|^ )'
```

- All literal values (both numeric and string) must be wrapped around single quotes. For example:

```
WITH
MEMBER Measures.EMPLOYEECHANGE AS
"(Measures.BCU2WEMLJL5ZK9N1F3ZRDUXS7 -
(Measures.BCU2WEMLJL5ZK9N1F3ZRDUXS7, 0CALMONTH.LAG('12')))"
```

**Example**

In the following example, at runtime, the calculated key figure `PROFIT CHANGE` is calculated by subtracting the profit for the previous month from the profit for the current month.

```
begin-select
   Measures.CKF_SI_PROFIT
   Measures.Profit Change
   FROM 0D_SD_C03/SAP_DEMO_ODBO
   WITH MEMBER Measures.Profit Change AS
   '((Measures.CKF_SI_PROFIT)-(Measures.CKF_SI_PROFIT,
   0CALMONTH.PREVMEMBER) )'
end-select
```

The result set from the above request shows both the profit and the profit change against the previous month for all months in the year 2001.

| | Profit | Profit Change |
|---|---|---|
| JAN 2001 | 11.324.466.00* | 11.324.466.00* |
| FEB 2001 | 7.767.949.00* | -3.556.517.00* |

| | | |
|---|---|---|
| MARCH 2001 | 9.598.544.00* | 1.830.595.00* |
| APRIL 2001 | 7.225.499.00* | -2.373.045.00* |
| MAY 2001 | 9.216.444.00* | 1.990.945.00* |
| JUNE 2001 | 12.050.631.00* | 2.834.187.00* |
| JULY 2001 | 14,757,033.00* | 2.706.402.00* |
| AUG 2001 | 558.144.00 MIX | -14198889.00 |
| SEP 2001 | 14,834.377.00* | 14276233.00 |
| OCT 2001 | 13,158.103.00* | -1.676.274.00* |
| NOV 2001 | 17.673.019.00* | 4.514.916.00* |
| DEC 2001 | 13.833.383.00* | -3.839.636.00* |

**Example**

In the following example, at runtime, the calculated key figure EMPLOYEECHANGE is calculated by subtracting the number of employees for the previous month from the number of employees for the current month using the PrevMember function.

```
begin-select
    Measures.BCU2WEMLJL5ZK9N1F3ZRDUXS7,
    Measures.EMPLOYEECHANGE
  0CALMONTH.MEMBERS
  FROM 0PA_C01/0PA_C01_Q024
  WITH MEMBER Measures.EMPLOYEECHANGE AS
  '(Measures.BCU2WEMLJL5ZK9N1F3ZRDUXS7 -
  (Measures.BCU2WEMLJL5ZK9N1F3ZRDUXS7, 0CALMONTH.PREVMEMBER))'
end-select
```

The result set from the above request is:

| All Calendar Year/Month | Number of Employees | Employee Change |
|---|---|---|
| JAN 2003 | 1,232 | 1,232 |
| FEB 2003 | 1,232 | 0 |
| MARCH 2003 | 1,238 | 6 |
| APRIL 2003 | 1,239 | 1 |
| MAY 2003 | 1,239 | 0 |
| JUNE 2003 | 1,239 | 0 |
| JULY 2003 | 1,239 | 0 |
| AUG 2003 | 1,239 | 0 |

| | | |
|---|---|---|
| SEP 2003 | 1,239 | 0 |
| OCT 2003 | 1,239 | 0 |
| NOV 2003 | 1,239 | 0 |
| DEC 2003 | 1,240 | 1 |
| JAN 2004 | 1,243 | 3 |
| FEB 2004 | 1,243 | 0 |
| MARCH 2004 | 1,243 | 0 |
| APRIL 2004 | 1,243 | 0 |
| MAY 2004 | 1,247 | 4 |
| JUNE 2004 | 1,247 | 0 |
| JULY 2004 | 1,247 | 0 |
| AUG 2004 | 1,247 | 0 |

# Accessing SAP BW Data from SQR Production Reporting Studio

You can use SQR Production Reporting Studio to retrieve data and create reports using data from an SAP BW OLAP data warehouse.

➤ To create a report using data from an SAP BW OLAP data warehouse:

1. Open SQR Production Reporting Studio by selecting **Start > Programs > Oracle EPM System > Reporting and Analysis > Production Reporting Studio**.
2. Select the desired type of report and click **New** to create a new data connection.
3. On the first page of the Data Connection wizard, enter a name for the SAP BW data source.
4. On the second page of the wizard, select **DDO** to identify the data source provider.
5. On the third page of the wizard, select an SAP BW data source.
6. On the fourth page of the wizard, enter the requested login parameters and select **Enable Properties Attributes**.

   This option appears *only* if you selected a valid SAP BW data source (see step 5).
7. Click **Finish** to exit from the Data Connection wizard.
8. Build the query using the SQR Production Reporting Studio Query Builder.
9. Create the report in the SQR Production Reporting Studio Layout window.
10. Process and save the report.

**Note:**

For detailed information on how to use Oracle's Hyperion® SQR® Production Reporting Studio to create reports using SAP BW data, see "Creating SAP BW Reports" in the *Hyperion SQR Production Reporting Studio User's Guide*.

# 9 Using Production Reporting DDO to Access Essbase Cubes

**In This Chapter**


Overview of Cubes ..... 151
Viewing Cubes ..... 152
Using Cube Commands in Production Reporting ..... 153
Displaying Report Data ..... 159
Accessing Cubes: An Example ..... 162


## Overview of Cubes

Cubes contain multidimensional database components supporting multiple data views. The components, arranged in a "hierarchical tree" (outline) structure, include:

- **Dimensions—Information** categories, such as Location, Products, Stores, and Time.

  Essbase dimensions include:

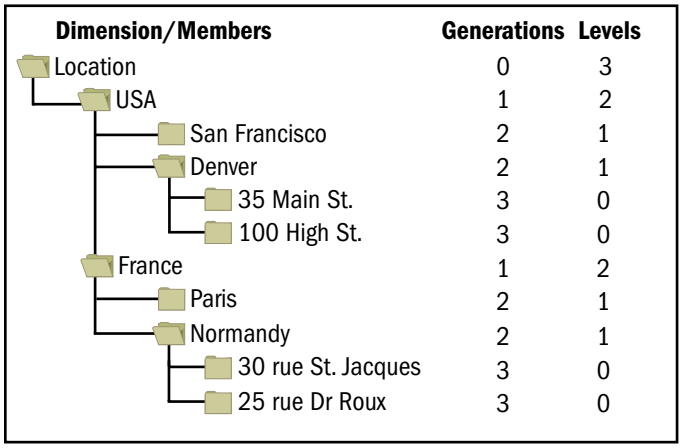
  Standard—Core dimensions often relating to departmental functions such as product line or division.

  Attribute dimensions—Further group and analyze standard dimension members. For example, you could compare a certain aspect of a product line with another aspect of that same product line.

- **Members—Dimension** content values. A Location dimension for example, could contain members such as USA, France, San Francisco, Paris, and 35 Main Street.
- **Generations**—Consolidation of dimension levels. Starting at Generation 1, the generations count down toward each dimension member.
- **Levels**—Groups of similar member types. For example, USA and France could belong to the Country level, San Francisco and Paris could belong to the City level, and 35 Main Street could belong to the Address level. Levels are counted in reverse order of generations and start at zero.
- **Aliases**—Optional, descriptive names given to members and stored in alias tables. In report output, aliases can be used instead of member names when member names are non-descriptive.
- **Measures**—Aggregations stored in columns in fact tables for quick retrieval by users querying cubes. Measures are numeric data displayed in reports.

Figure 10 illustrates a folder tree containing the Location dimension members in a cube. In this example, Location is the dimension, and USA, France, and all other branches are its members. Location is generation 1, USA and France are generation 2, San Francisco and Paris are generation 3, and 35 Main St. and 30 rue St. Jacques are generation 4. Levels refer to dimension branches and are in reverse order of generations.

**Figure 10 Location Dimension Hierarchy**




# Viewing Cubes

To write Production Reporting programs to access Essbase cubes, you must specify the correct dimension, member names, and hierarchies used by the database. The Production Reporting DDO Query Editor displays Essbase tree structures as well as member names and aliases. Member names appear under the *Subtype Information* column, and member aliases appear under the *Name* column.

**Caution!**

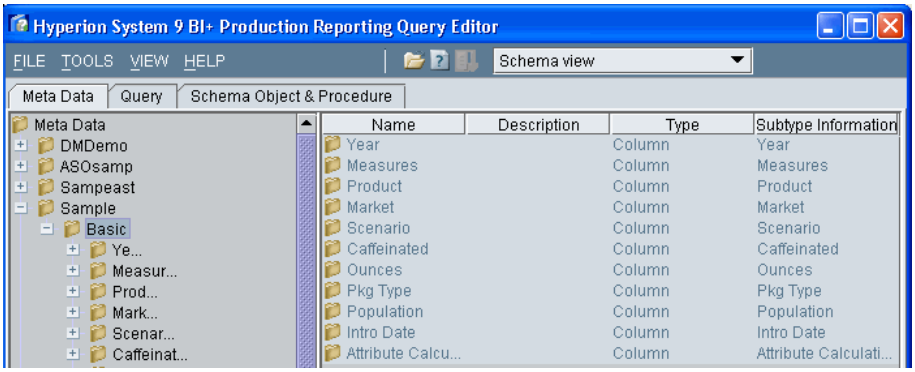
When referencing members in Production Reporting DDO code, use the member name in the DDO Query Editor's *Subtype Information* column.

Figure 11 displays member names and aliases in the Production Reporting DDO Query Editor. Notice that the name of the Period member is Year, but the alias is Period. In the Production Reporting DDO code, for example, you would use Year (member name) under `DECLARE-CONNECTION`, but the word Period (member alias) would display in your report if you had the alias parameter enabled. (See "Aliases" on page 159.)

**Note:**

If no alias is specified for a member, the member name is used in both columns. This is the case for the Market member in Figure 11. The member name "Market" is used in both columns because no alias is specified in the Essbase alias table.

Figure 11 Production Reporting DDO Query Editor




**Note:**

To view database content with the DDO Query Editor, you must first add a data-source specification in the Registry Editor. See "Data Source Specifications" on page 87.

➤ To use the Production Reporting DDO Query Editor to view dimensions and members:

1. Select **Start > Programs > Oracle EPM System > Reporting and Analysis > SQR Production Reporting > DDO > Query Editor**.
2. Select **File > Open** or click the **Open** button on the toolbar.
3. Highlight the data source under the Registry folder and click **Open**.
4. Enter your username and password.

   You may have to double click the Meta Data folder to view the hierarchy.
5. Use the name supplied under **Subtype Information** in the Production Reporting program.

# Using Cube Commands in Production Reporting

The following Production Reporting commands access cubes:

- SET-MEMBERS
- SET-GENERATIONS
- SET-LEVELS

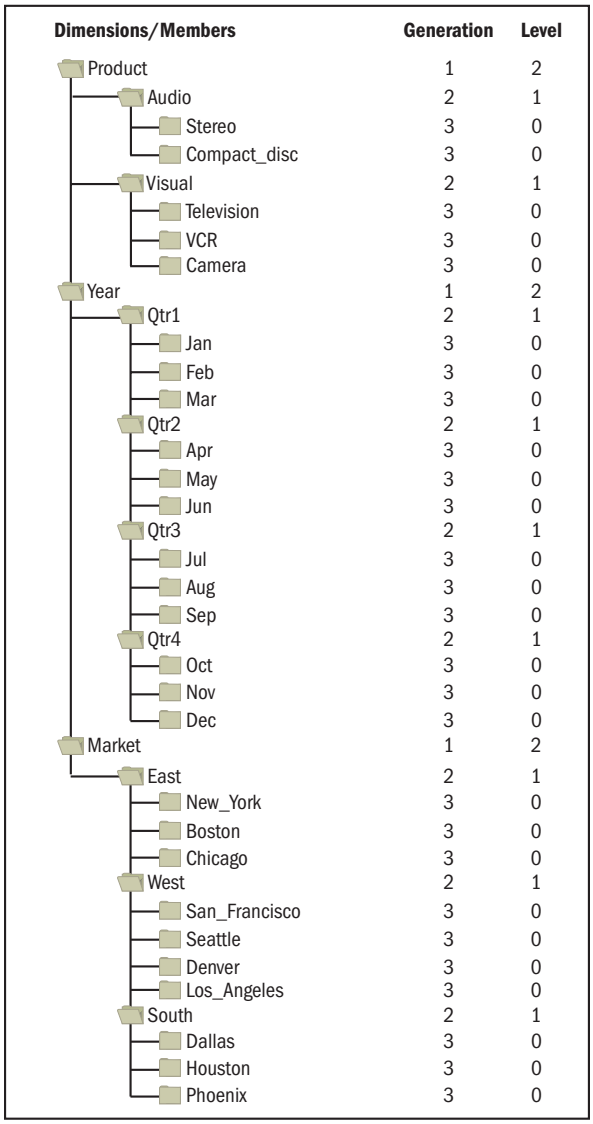
Use these commands under `DECLARE-CONNECTION` or `ALTER-CONNECTION` in your Production Reporting program.

Review the following sections for information on each of these commands. The information is based on the same sample code used to access the Product, Year, and Market Dimension Folder Tree in Figure 12, which displays a hierarchy of members in a multidimensional database.

**Note:**

The sample data used in this section is from the Demo application included with the Essbase OLAP Server. Refer to the Hyperion Essbase Installation Notes for installation instructions.

Figure 12 Product, Year, and Market Dimension Folder Tree



| Dimensions/Members | Generation | Level |
|---|---|---|
| Product | 1 | 2 |
| Audio | 2 | 1 |
| Stereo | 3 | 0 |
| Compact_disc | 3 | 0 |
| Visual | 2 | 1 |
| Television | 3 | 0 |
| VCR | 3 | 0 |
| Camera | 3 | 0 |
| Year | 1 | 2 |
| Qtr1 | 2 | 1 |
| Jan | 3 | 0 |
| Feb | 3 | 0 |
| Mar | 3 | 0 |
| Qtr2 | 2 | 1 |
| Apr | 3 | 0 |
| May | 3 | 0 |
| Jun | 3 | 0 |
| Qtr3 | 2 | 1 |
| Jul | 3 | 0 |
| Aug | 3 | 0 |
| Sep | 3 | 0 |
| Qtr4 | 2 | 1 |
| Oct | 3 | 0 |
| Nov | 3 | 0 |
| Dec | 3 | 0 |
| Market | 1 | 2 |
| East | 2 | 1 |
| New_York | 3 | 0 |
| Boston | 3 | 0 |
| Chicago | 3 | 0 |
| West | 2 | 1 |
| San_Francisco | 3 | 0 |
| Seattle | 3 | 0 |
| Denver | 3 | 0 |
| Los_Angeles | 3 | 0 |
| South | 2 | 1 |
| Dallas | 3 | 0 |
| Houston | 3 | 0 |
| Phoenix | 3 | 0 |

## SET-MEMBERS

Selects a specific dimension hierarchy. Using dot notation with the alpha name of dimensions and members, you can use `SET-MEMBERS` to retrieve specific information for one or more data items.

**Note:**

A program that does not use SET-MEMBERS returns all of the dimensions within a declared column.

*Program ex25.sqr* uses SET-MEMBERS to access information in the Product, Year, and Market Dimension Folder Tree in Figure 12.

In this example, SET-MEMBERS:

- returns the set of members in the dimension Product at the specific hierarchy of Audio.
- returns the set of members in the dimension Year at the specific hierarchy of Qtr1.
- returns the set of members in the dimension Market at the specific hierarchy of West.

## Program ex25.sqr

```
Begin-Setup
  Declare-Connection ESSConnection
    DSN=Essbase6
    User='Oracle User'
    Password=dbadmin1
    Parameters=member.alias=true;
    set-members=('Year','Qtr1','Product','Audio','Market','West')
  End-Declare
End-Setup

Begin-Program
  print 'Quarter Product Region' (+2,1)
  print 'Total Expenses Cost of Goods' (0,40)
  do Read_Cube
end-program

Begin-Procedure Read_Cube
  Begin-Execute
    Connection=ESSConnection
    schema='Demo'
    getdata='Basic'
Begin-select loops=5000
Year     &year (+1,1)
Product  &prod (0,10)
Market   &mkt  (0,20)
Measures.Profit.Total_Expenses.Marketing   &exp (0,40) edit   $99,999,999.
99
Measures.Profit.Margin.Cost_of_Goods_Sold   &cogs (0,60) edit   $99,999,999.
99
From Essbase
End-select
  End-Execute
End-Procedure
```

In the output for *Program ex25.sqr*, the alias "Q1" appears instead of the name "Qtr1". (See"Aliases" on page 159 for information on using aliases.)

### Output for *Program ex25.sqr*

| Quarter | Product | Region | Total Expenses | Cost of Goods |
|---|---|---|---|---|
| Q1 | Audio | West | $ 1,465.00 | $ 12,259.00 |

## SET-GENERATIONS

SET-GENERATIONS selects a specific point within the hierarchy for the previously-declared dimension. You can use SET-GENERATIONS to set an exact location within the dimension declared by SET-MEMBERS. The dimension and hierarchy defined with SET-GENERATIONS can be a *literal* value only.

*Program ex26.sqr* uses SET-GENERATIONS to access the information in the Product, Year, and Market Dimension Folder Tree in Figure 12. The example builds on *Program ex25.sqr*, and uses SET-GENERATIONS to retrieve more specific information.

In this example, SET-GENERATIONS:

- requests a specific generation for Product.

  SET-MEMBERS narrows the range to Audio, and SET-GENERATIONS requests all members at the third generation within Audio (Stereo and Compact_Disc).

- requests a specific generation for Market.

  SET-MEMBERS narrows the range to West, and SET-GENERATIONS requests all members at the third generation within West (San_Francisco, Seattle, Denver, and Los_Angeles).

### Program ex26.sqr

```
Begin-Setup
  Declare-Connection ESSConnection
    DSN=Essbase6
    User='Oracle User'
    Password=dbadmin1
    Parameters=member.alias=true;
    set-members=       ('Year','Qtr1','Product','Audio','Market','West')
    set-generations=('Product',3,'Market',3)
  End-Declare
End-Setup

Begin-Program
  print 'Quarter Product Region' (+2,1)
  print 'Total Expenses Cost of Goods' (0,40)
  do Read_Cube
end-program

Begin-Procedure Read_Cube
  Begin-Execute
    Connection=ESSConnection
    schema='Demo'
    getdata='Basic'
Begin-select loops=5000
Year     &year (+1,1)
Product  &prod (0,10)
Market   &mkt  (0,20)
```

```
Measures.Profit.Total_Expenses.Marketing &exp (0,40) edit
    $99,999,999.99
Measures.Profit.Margin.Cost_of_Goods_Sold &cogs (0,60) edit
    $99,999,999.99
From Essbase
End-select
  End-Execute
End-Procedure
```

As shown in "*Output for Program ex26.sqr*", adding SET-GENERATIONS produces a much larger report than using SET-MEMBERS alone, but it excludes the levels in *Output for Program ex25.sqr*.

### Output for *Program ex26.sqr*

| Quarter | Product | Region | Total Expenses | Cost of Goods |
|---|---|---|---|---|
| Q1 | Stereo | San_Francisco | $ 304.00 | $ 2,246.00 |
| Q1 | Stereo | Seattle | $ 124.00 | $ 1,356.00 |
| Q1 | Stereo | Denver | $ 130.00 | $ 1,461.00 |
| Q1 | Stereo | Los_Angeles | $ 225.00 | $ 1,747.00 |
| Q1 | Compact_Disc | San_Francisco | $ 155.00 | $ 1,558.00 |
| Q1 | Compact_Disc | Seattle | $ 56.00 | $ 1,258.00 |
| Q1 | Compact_Disc | Denver | $ 208.00 | $ 1,249.00 |
| Q1 | Compact_Disc | Los_Angeles | $ 263.00 | $ 1,384.00 |

## SET-LEVELS

SET-LEVELS extends the dimension hierarchy for the previously-declared dimension. After declaring the starting point with SET-MEMBERS, use SET-LEVELS to specify how many levels to move down the hierarchy. The dimension and hierarchy defined with SET-LEVELS can be a *literal* value only.

*Program ex27.sqr* uses SET-LEVELS with SET-MEMBERS to access the information in the Product, Year, and Market Dimension Folder Tree in Figure 12.

In this example, SET-LEVELS:

- requests additional levels for Product.

  SET-MEMBERS sets the starting point to Audio, and SET-LEVELS requests Audio (level 1) plus the next level below (Stereo and Compact_Disc).

- requests additional levels for Market.

  SET-MEMBERS sets the starting point to West (level 1), and SET-LEVELS requests West plus the next level below (San_Francisco, Seattle, Denver, and Los_Angeles).

### Program ex27.sqr

```
Begin-Setup
  Declare-Connection ESSConnection
    DSN=Essbase6
    User='Oracle User'
    Password=dbadmin1
    Parameters=member.alias=true;
```

```
    set-members=       ('Year','Qtr1','Product','Audio','Market','West')
    set-levels= ('product', 1,'market', 1)
  End-Declare
End-Setup

Begin-Program
  print 'Quarter Product Region' (+2,1)
  print 'Total Expenses Cost of Goods' (0,40)
  do Read_Cube
end-program

Begin-Procedure Read_Cube
  Begin-Execute
    Connection=ESSConnection
    schema='Demo'
    getdata='Basic'
Begin-select loops=5000
Year     &year (+1,1)
Product  &prod (0,10)
Market   &mkt  (0,20)
Measures.Profit.Total_Expenses.Marketing &exp (0,40) edit
    $99,999,999.99
Measures.Profit.Margin.Cost_of_Goods_Sold &cogs (0,60) edit
    $99,999,999.99
From Essbase
End-select
  End-Execute
End-Procedure
```

As shown in Output for *Program ex27.sqr*, adding SET-LEVELS produces a larger report than SET-GENERATIONS, and it includes the levels requested by SET-MEMBERS.

## Output for *Program ex27.sqr*

| Quarter | Product | Region | Total Expenses | Cost of Goods |
|---|---|---|---|---|
| Q1 | Audio | West | $ 1,465.00 | $ 12,259.00 |
| Q1 | Audio | San_Francisco | $ 459.00 | $ 3,804.00 |
| Q1 | Audio | Seattle | $ 180.00 | $ 2,614.00 |
| Q1 | Audio | Denver | $ 338.00 | $ 2,710.00 |
| Q1 | Audio | Los_Angeles | $ 488.00 | $ 3,131.00 |
| Q1 | Stereo | West | $ 783.00 | $ 6,810.00 |
| Q1 | Stereo | San_Francisco | $ 304.00 | $ 2,246.00 |
| Q1 | Stereo | Seattle | $ 124.00 | $ 1,356.00 |
| Q1 | Stereo | Denver | $ 130.00 | $ 1,461.00 |
| Q1 | Stereo | Los_Angeles | $ 225.00 | $ 1,747.00 |
| Q1 | Compact_Disc | West | $ 682.00 | $ 5,449.00 |
| Q1 | Compact_Disc | San_Francisco | $ 155.00 | $ 1,558.00 |
| Q1 | Compact_Disc | Seattle | $ 56.00 | $ 1,258.00 |
| Q1 | Compact_Disc | Denver | $ 208.00 | $ 1,249.00 |
| Q1 | Compact_Disc | Los_Angeles | $ 263.00 | $ 1,384.00 |

# Displaying Report Data

Once you decide what data to access and which commands to use, you can display the information in a report. Review the following sections to understand some aspects of report layout.

- Measures
- Aliases
- Column Order

## Measures

Measures are the numeric data displayed in reports. Some common measures are sales, cost, expenditures, and production count. Measures are aggregations stored for quick retrieval by users querying cubes. Each measure is stored in a column in a fact table in a cube. Measures can contain multiple columns combined in expressions. For example, the Profit measure is the difference of two numeric columns: Sales and Cost.

The format for measure columns is *'measures', dot, 'measure name'* (for example, *measures.profit*). Use this format regardless of the name used by the data source to declare measures.

*Program ex30.sqr* displays a measure, which cannot be included in `SET-MEMBERS`, `SET-GENERATIONS`, or `SET-LEVELS` but is written under `BEGIN-SELECT`.

## Aliases

Aliases are optional, descriptive names given to members and stored in alias tables. In report output, you can use aliases instead of member names when member names are non-descriptive. For example, a member name could be a number while its alias could be a noun such as Sales. Use the parameters property in the setup section of your Production Reporting program to specify whether to use names or aliases in your output.

➤ To use *aliases* in Production Reporting output, write the following line in the setup section of your Production Reporting program:

```
Parameters=member.alias=true;
```

Including this statement displays the aliases in the *Name* column of the Production Reporting DDO Query Editor. (See "Viewing Cubes" on page 152 for more information.)

➤ To use *names* in your Production Reporting DDO output, do not include the following statement in your Production Reporting program:

```
Parameters=member.alias=true;
```

If you do not include this statement, the output displays the member names under the *Subtype Information* column of the Production Reporting DDO Query Editor. (See "Viewing Cubes" on page 152 for more information.)

**Note:**

Regardless of whether you display aliases in report output, you must use member names when referencing the members in the `SET-UP` section of your Production Reporting program.

## Column Order

The order in which dimension columns display in the `BEGIN-SELECT` section determines the order the data displays in multiple rowset queries. *Program ex28.sqr* shows `Total_Expenses` and `Cost_of_Goods_Sold` for the selected Year, Market, and Product.

### Program ex28.sqr

```
Begin-Setup
    Declare-Connection ESSConnection
        DSN=Essbase6
        User='Oracle User'
        Password=dbadmin1
        Parameters=member.alias=true;
        set-members=('Year','Qtr1','Market','West')
        set-generations=('Year', 2,'Product',1)
        set-levels= ('Year',1,'Product', 3)
    End-Declare
End-Setup

Begin-Program
    print 'Quarter Product Region' (+2,1)
    print 'Total Expenses Cost of Goods' (0,40)
    do Read_Cube
end-program

Begin-Procedure Read_Cube
    Begin-Execute
        Connection=ESSConnection
        schema='Demo'
        getdata='Basic'
Begin-select loops=5000
Year     &year (+1,1)
Market   &mkt  (0,25)
Product  &prod (0,10)
Measures.Profit.Total_Expenses.Marketing &exp (0,45) edit $99,999,999.99
Measures.Profit.Margin.Cost_of_Goods_Sold &cogs (0,65) edit $99,999,999.99
From Essbase
End-select
    End-Execute
End-Procedure
```

In Output for *Program ex29.sqr*, the rows are sorted first by Year: All the Q1 rows are first, followed by the Jan, Feb, and Mar rows. The Quarter column sets the order of the rows because Year is listed first under `BEGIN-SELECT`.

### Output for *Program ex29.sqr*

```
Quarter     Product          Region      Total Expenses         Cost of Goods
```

| | | | | |
|---|---|---|---|---|
| Q1 | Stereo | West | $ 783.00 | $ 6,810.00 |
| Q1 | Compact_Disc | West | $ 682.00 | $ 5,449.00 |
| Q1 | Television | West | $ 1,669.00 | $ 6,676.00 |
| Q1 | VCR | West | $ 438.00 | $ 5,408.00 |
| Q1 | Camera | West | $ 1,051.00 | $ 3,160.00 |
| Jan | Stereo | West | $ 278.00 | $ 2,385.00 |
| Jan | Compact_Disc | West | $ 231.00 | $ 1,847.00 |
| Jan | Television | West | $ 527.00 | $ 2,451.00 |
| Jan | VCR | West | $ 145.00 | $ 1,833.00 |
| Jan | Camera | West | $ 354.00 | $ 1,146.00 |
| Feb | Stereo | West | $ 260.00 | $ 2,181.00 |
| Feb | Compact_Disc | West | $ 220.00 | $ 1,768.00 |
| Feb | Television | West | $ 628.00 | $ 2,102.00 |
| Feb | VCR | West | $ 146.00 | $ 1,812.00 |
| Feb | Camera | West | $ 360.00 | $ 999.00 |
| Mar | Stereo | West | $ 245.00 | $ 2,244.00 |
| Mar | Compact_Disc | West | $ 231.00 | $ 1,834.00 |
| Mar | Television | West | $ 514.00 | $ 2,123.00 |
| Mar | VCR | West | $ 147.00 | $ 1,763.00 |
| Mar | Camera | West | $ 337.00 | $ 1,015.00 |

To change the look of the output, modify the dimension order under BEGIN-SELECT. If the order of the Product, Year, and Market dimensions is reversed (as in *Program ex29.sqr*) then you get a report with columns displayed in a different sorting order.

### Program ex29.sqr

```
Begin-select loops=5000
Product   &prod    (+1,10)
Market    &mkt     (0,25)
Year      &year    (0,1)
Measures.Profit.Total_Expenses.Marketing &exp (0,45) edit $99,999,999.99
Measures.Profit.Margin.Cost_of_Goods_Sold &cogs (0,65) edit $99,999,999.99
From Essbase
End-select
```

In the *Output for Program ex29.sqr*, the rows are sorted first by Product: All the Stereo rows are first, followed by the Compact_Disc, Television, VCR, and Camera rows. The Product column sets the row order because Product is listed first under BEGIN-SELECT.

### Output for *Program ex29.sqr*

| Quarter | Product | Region | Total Expenses | Cost of Goods |
|---|---|---|---|---|
| Q1 | Stereo | West | $ 783.00 | $ 6,810.00 |
| Jan | Stereo | West | $ 278.00 | $ 2,385.00 |
| Feb | Stereo | West | $ 260.00 | $ 2,181.00 |
| Mar | Stereo | West | $ 245.00 | $ 2,244.00 |
| Q1 | Compact_Disc | West | $ 682.00 | $ 5,449.00 |
| Jan | Compact_Disc | West | $ 231.00 | $ 1,847.00 |
| Feb | Compact_Disc | West | $ 220.00 | $ 1,768.00 |
| Mar | Compact_Disc | West | $ 231.00 | $ 1,834.00 |
| Q1 | Television | West | $ 1,669.00 | $ 6,676.00 |
| Jan | Television | West | $ 527.00 | $ 2,451.00 |
| Feb | Television | West | $ 628.00 | $ 2,102.00 |
| Mar | Television | West | $ 514.00 | $ 2,123.00 |
| Q1 | VCR | West | $ 438.00 | $ 5,408.00 |

| | | | | | | |
|---|---|---|---|---|---|---|
| Jan | VCR | West | $ | 145.00 | $ | 1,833.00 |
| Feb | VCR | West | $ | 146.00 | $ | 1,812.00 |
| Mar | VCR | West | $ | 147.00 | $ | 1,763.00 |
| Q1 | Camera | West | $ | 1,051.00 | $ | 3,160.00 |
| Jan | Camera | West | $ | 354.00 | $ | 1,146.00 |
| Feb | Camera | West | $ | 360.00 | $ | 999.00 |
| Mar | Camera | West | $ | 337.00 | $ | 1,015.00 |

# Accessing Cubes: An Example

This section gives an example of a cube and discusses the Production Reporting code necessary to access the cube. Review this section for information on:

- The Cube
- The Production Reporting Code Needed to Access the Cube
- An Explanation of the Code

**Note:**

Essbase includes the DECIMAL function within the Report Script generated by DDO to access data from Essbase cubes. The default number of decimals to the right of the decimal point is 6.

To change the default value:

1. Open the com_sqribe_essacc_EssDataSource_Properties file.
2. Change the `decimal.points=6` entry to the desired number of decimals.
3. Save your changes.

## The Cube

Figure 13 shows a cube with three dimensions: Product (Y-axis), Year (X-axis), and Market (Z-axis). This example illustrates only a small part of an entire cube. For example, the Year dimension could have four quarters while only the first three quarters are illustrated here. In addition, dimensions such as Location or Salesperson, could also be added to this cube.

Each cell of the cube represents a data value in a database. For example, one cell might hold a value of Audio products for the Western market region in Qtr1. In addition to being divided by cells, the cube can also be sliced into small segments, such as all products for the Western region in the first quarter.

Figure 13 Three-Dimensional Data Cube

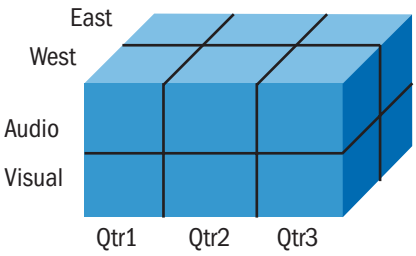



Figure 14 reveals the dimensions and levels of the cube in Figure 13. Audio and Visual are members of the Product dimension, Qtr1, Qtr2, Qtr3, and Qtr4 are members of the Year dimension, and East, West, and South are members of the Market dimension. Each dimension corresponds to a cube axis.

Figure 14 Hierarchy of a Data Cube

| Dimensions/Members | Generation | Level | Axis |
|---|---|---|---|
| Product | 1 | 2 | Y-axis |
| Audio | 2 | 1 | |
| Stereo | 3 | 0 | |
| Compact_disc | 3 | 0 | |
| Visual | 2 | 1 | |
| Television | 3 | 0 | |
| VCR | 3 | 0 | |
| Camera | 3 | 0 | |
| Year | 1 | 2 | X-axis |
| Qtr1 | 2 | 1 | |
| Jan | 3 | 0 | |
| Feb | 3 | 0 | |
| Mar | 3 | 0 | |
| Qtr2 | 2 | 1 | |
| Qtr3 | 2 | 1 | |
| Qtr4 | 2 | 1 | |
| Market | 1 | 2 | Z-axis |
| East | 2 | 1 | |
| West | 2 | 1 | |
| South | 2 | 1 | |

## The Production Reporting Code Needed to Access the Cube

The Production Reporting code in *Program ex30.sqr* accesses the Essbase cube discussed in the previous section. The code uses the three commands for accessing a cube: SET-MEMBERS, SET-GENERATIONS, and SET-LEVELS.

### Program ex30.sqr

```
Begin-Setup
  Declare-Connection ESSConnection
    DSN=Essbase6
    User='Oracle User'
    Password=dbadmin1
    Parameters=member.alias=true;
```

```
    set-members=('Year','Qtr1','Market','West')
    set-generations=('Year', 2,'Product',1)
    set-levels= ('Year',1,'Product', 3)
  End-Declare
End-Setup

Begin-Program
  print 'Quarter Product Region' (+2,1)
  print 'Total Expenses Cost of Goods' (0,40)
  do Read_Cube
end-program

Begin-Procedure Read_Cube
  Begin-Execute
    Connection=ESSConnection
    schema='Demo'
    getdata='Basic'
Begin-select loops=5000
Product  &prod   (+1,10)
Market   &mkt    (0,25)
Year     &year   (0,1)
Measures.Profit.Total_Expenses.Marketing &exp (0,45) edit $99,999,999.99
Measures.Profit.Margin.Cost_of_Goods_Sold &cogs (0,65) edit $99,999,999.99
From Essbase
End-select
  End-Execute
End-Procedure
```

**Note:**

The order in which the columns and rows in your output appear depends on the order in which the dimensions are listed under `BEGIN-SELECT`. The columns in *Program ex30.sqr* are listed in Product-by Market-by Year order. To change the look of the output, modify the dimension order under `BEGIN-SELECT`.

## Output for *Program ex30.sqr*

| Quarter | Product | Region | Total Expenses | Cost of Goods |
|---|---|---|---|---|
| Q1 | Stereo | West | $ 783.00 | $ 6,810.00 |
| Jan | Stereo | West | $ 278.00 | $ 2,385.00 |
| Feb | Stereo | West | $ 260.00 | $ 2,181.00 |
| Mar | Stereo | West | $ 245.00 | $ 2,244.00 |
| Q1 | Compact_Disc | West | $ 682.00 | $ 5,449.00 |
| Jan | Compact_Disc | West | $ 231.00 | $ 1,847.00 |
| Feb | Compact_Disc | West | $ 220.00 | $ 1,768.00 |
| Mar | Compact_Disc | West | $ 231.00 | $ 1,834.00 |
| Q1 | Television | West | $ 1,669.00 | $ 6,676.00 |
| Jan | Television | West | $ 527.00 | $ 2,451.00 |
| Feb | Television | West | $ 628.00 | $ 2,102.00 |
| Mar | Television | West | $ 514.00 | $ 2,123.00 |
| Q1 | VCR | West | $ 438.00 | $ 5,408.00 |
| Jan | VCR | West | $ 145.00 | $ 1,833.00 |
| Feb | VCR | West | $ 146.00 | $ 1,812.00 |
| Mar | VCR | West | $ 147.00 | $ 1,763.00 |
| Q1 | Camera | West | $ 1,051.00 | $ 3,160.00 |
| Jan | Camera | West | $ 354.00 | $ 1,146.00 |

```
Feb         Camera          West              $       360.00          $       999.00
Mar         Camera          West              $       337.00          $     1,015.00
```

## An Explanation of the Code

Table 20 explains the code necessary to access the example Essbase cube.

**Table 20 Code Explanation**

| Code | Explanation |
|---|---|
| `Begin-Setup`<br><br>`  Declare-Connection ESSConnection`<br><br>`    DSN=Essbase6`<br><br>`    User='Oracle User'`<br><br>`    Password=dbadmin1`<br><br>`    Parameters=member.alias=true;`<br><br>`    set-members=       ('Year','Qtr1','Market','West')`<br><br>`    set-generations=       ('Year', 2,'Product',1)`<br><br>`    set-levels= ('Year',1,'Product', 3)`<br><br>`  End-Declare`<br><br>`End-Setup` | The setup section of the Production Reporting program.<br><br>● `Declare-Connection`—User-defined name for describing a cube connection. In this case, we used `EssConnection`.<br>● `DSN`—Essbase connection. Enter the DDO registry name in the DDO registry.<br><br>For Essbase, enter the name of your database as the DSN.<br><br>● `User`—User name for the Essbase connection.<br>● `Password`—Password for the user.<br><br>You can also connect from the command line using the following syntax:<br><br>`SQR [`*program*`] DSN/[`*username*`]/[`*password*`]`<br><br>● `Parameters`—Optional parameters. Declares whether to use an alias table.<br><br>If you include this statement, aliases instead of names are displayed in the output.<br><br>● `set-members`—The name of the dimension at the specific hierarchical level.<br>● `set-levels`—Extends the dimension hierarchy for the previously-declared dimension.<br>● `set-generations`—Overrides the specific hierarchical level declared by `set-members`. |
| `Begin-Program`<br><br>`  print 'Quarter Product Region' (+2,1)`<br><br>`  print 'Total Expenses Cost of Goods' (0,40)`<br><br>`  do Read_Cube`<br><br>`end-program`<br><br>`end-program` | The program section of the Production Reporting program.<br><br>● `print`—Prints column header text.<br><br>In this example, the program prints the headings Quarter, Product, Region, Total Expenses, and Cost of Goods.<br><br>● `do Read_Cube`—Directs Production Reporting to read the cube. |
| `Begin-Procedure Read_Cube`<br><br>`  Begin-Execute`<br><br>`    Connection=ESSConnection` | Begins the new query or procedure execution.<br><br>● `Connection`—Points to the information in `Declare-Connection`.<br>● `schema`—Name of the cube. |

| Code | Explanation |
|---|---|
| schema='Demo'<br><br>getdata='Basic' | ● getdata—Name of the table in the cube.<br><br>You must include Connection, schema, and getdata to successfully connect. |
| Begin-select loops=5000<br><br>Product &prod (+1,10)<br><br>Market &mkt (0,25)<br><br>Year &year (0,1) | Begins a SELECT paragraph.<br><br>● loops=5000—(Optional) Specifies the number of rows to retrieve. After the specified number is processed, the SELECT loop exits.<br><br>● &prod,&mkt, and &year—Read-only column variables. You can use their existing value, but you cannot assign a new value to a column variable. (See "Column Variables" in Volume 1 of the *Production Reporting Developer's Guide*) |
| Measures.Profit.Total_Expenses.Marketing &exp (0,45) edit $99,999,999.99<br><br>Measures.Profit.Margin.Cost_of_Goods_Sold &cogs (0,65) edit $99,999,999.99 | Required to retrieve the numeric data.<br><br>Measures are hierarchical in design; each dot defines another level or member.<br><br>The measures in this example are Profit, Total_Expenses, Marketing, Margin, and Cost_of_Goods_Sold. |
| From Essbase | From —Required for Production Reporting.<br><br>Essbase —(Optional) Documents the data source of a select statement. |
| End-Select | Completes Begin-Select. |
| End-Execute | Completes Begin-Execute. |
| End-Procedure | Completes Begin-Procedure. |

# 10 Using Production Reporting DDO to Access MSOLAP Cubes

**In This Chapter**


Overview of Cubes ..... 167
Viewing Cubes ..... 168
Using Cube Commands in Production Reporting ..... 169
Displaying Report Data ..... 175
Accessing Cubes: An Example ..... 178


**Note:**

For information on how to set up MSOLAP drivers, see Chapter 3, "Writing a Production Reporting DDO Driver." You can use your driver against the sample cubes provided with MSOLAP.

## Overview of Cubes

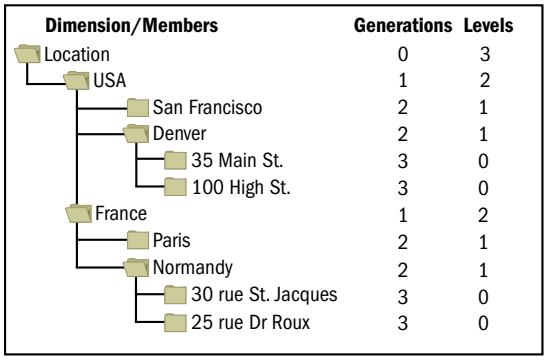
Cubes contain multidimensional database components that support multiple data views. The components, arranged in a "hierarchical tree" (outline) structure, include:

- **Dimensions—Information** categories, such as Location, Products, Stores, and Time.
- **Members—Dimension** content values. A Location dimension for example, could contain members such as USA, France, San Francisco, Paris, and 35 Main Street.
- **Generations**—Consolidation of dimension levels. Starting at Generation 0, the generations count down toward each dimension member.
- **Levels**—Groups of similar member types. For example, USA and France could belong to the Country level, San Francisco and Paris could belong to the City level, and 35 Main Street could belong to the Address level.
- **Aliases**—Optional, descriptive names given to members and stored in alias tables. In report output, aliases can be used instead of member names when member names are non-descriptive.
- **Measures**—Aggregations stored in columns in fact tables for quick retrieval by users querying cubes. Measures are numeric data displayed in reports.

Figure 15 illustrates a folder tree containing the Location dimension members in a cube. In this example, Location is the dimension, and USA, France, and all other branches are its members. Location is generation 0, USA and France are generation 1, San Francisco and Paris are

generation 2, and 35 Main St. and 30 rue St. Jacques are generation 3. Levels refer to the dimension branches and are in reverse order of generations.

Figure 15 Location Dimension Hierarchy




# Viewing Cubes

To write Production Reporting programs to access MS OLAP cubes, you must specify the correct dimensions, member names, and hierarchies used by the database. The Production Reporting DDO Query Editor displays MSOLAP tree structures as well as member names and aliases. Member names appear under the *Subtype Information* column, and member aliases appear under the *Name* column.

**Caution!**

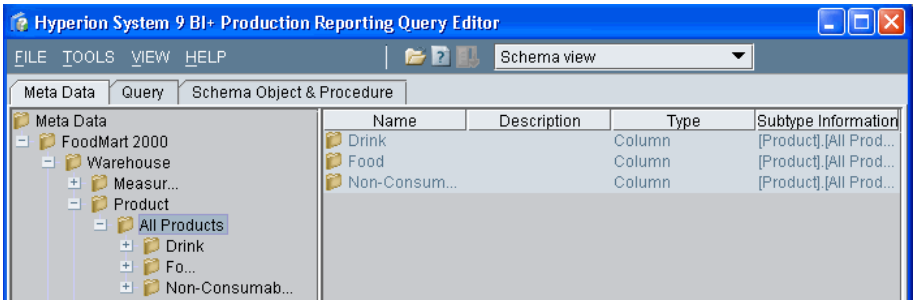
When referencing members in Production Reporting code, use the member name in the DDO Query Editor's *Subtype Information* column.

Figure 16 displays member names and aliases in the Production Reporting DDO Query Editor. Notice that the name of the Drink member is Product.All Product.Drink, but the alias is Drink. In the Production Reporting code, for example, you would use Product.All Product.Drink (member name) under `DECLARE-CONNECTION`, but the word Drink (member alias) would display in your report if you had the alias parameter enabled. (See "Aliases" on page 175.)

**Note:**

If no alias is specified for a member, the member name is used in both columns because no alias is specified in the MSOLAP alias table.

Figure 16 Production Reporting DDO Query Editor




**Note:**

To view database content with the DDO Query Editor, you must first add a data-source specification in the Registry Editor. See "Data Source Specifications" on page 87.

➤ To use the Production Reporting DDO Query Editor to view dimensions and members:

1. Select **Start > Programs > Oracle EPM System > Reporting and Analysis > Production Reporting DDO > Query Editor**.
2. Select **File > Open** or click the **Open** button on the toolbar.
3. Highlight the data source under the Registry folder and click **Open**.
4. Enter your username and password.

   You may have to double click the Meta Data folder to view the hierarchy.
5. Use the name supplied under **Subtype Information** in the Production Reporting program.

# Using Cube Commands in Production Reporting

The following Production Reporting commands access cubes:

- SET-MEMBERS
- SET-GENERATIONS
- SET-LEVELS

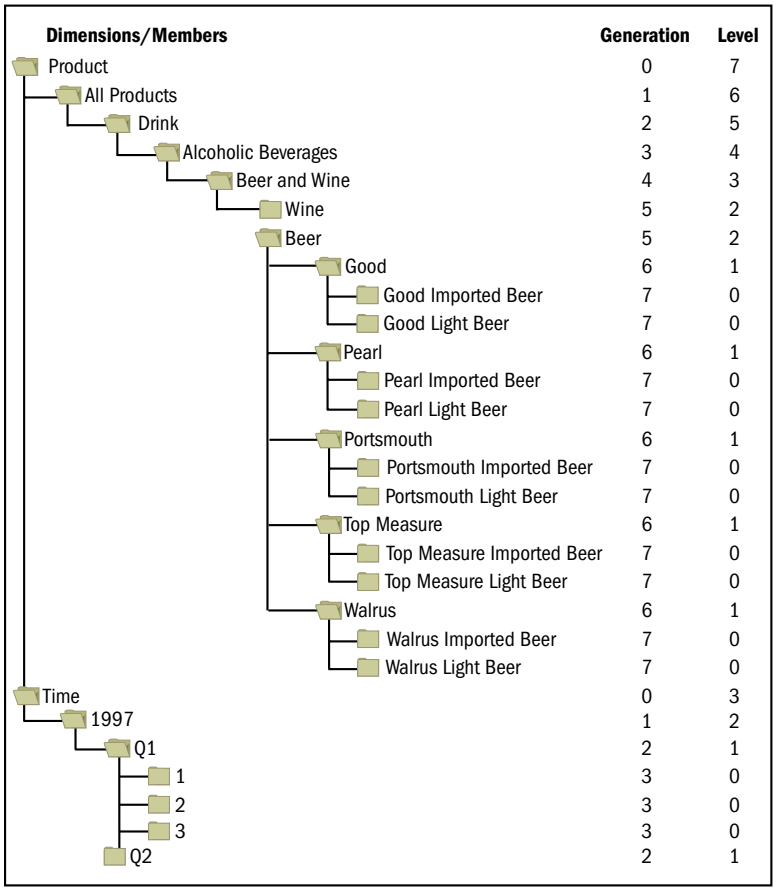
You can use these commands under `DECLARE-CONNECTION` or `ALTER-CONNECTION` in your Production Reporting program.

Review the following sections for information on each of these commands. The information is based on the same sample code used to access the Product Dimension Folder Tree in Figure 17, which displays a hierarchy of members in a multidimensional database.

**Note:**

The sample data used in this section is from the Demo application included with the MSOLAP Server.

Figure 17 Product Dimension Folder Tree




# SET-MEMBERS

Selects a specific dimension hierarchy. Using dot notation with the alpha name of dimensions and members, you can use `SET-MEMBERS` to retrieve specific information for one or more data items.

**Note:**

A program that does not use `SET-MEMBERS`, returns all of the dimensions within a declared column.

*Program ex30.sqr* uses `SET-MEMBERS` to access the information in the Production Dimension Folder Tree in Figure 17.

In this example, `SET-MEMBERS`:

- returns the set of members in the dimension Product at the specific hierarchy of All Products, at the specific level of Drink, at the specific level of Alcoholic Beverages, at the specific level of 'Beer and Wine'.
- returns the set of members in the dimension Time at the specific hierarchy of 1997, at the specific level of Q1.

### Program ex30.sqr

```
Begin-Setup
        Declare-Connection MSOLAP
                dsn=MSOLAP
                set-members=('product','all products.drink.alcoholic
beverages.beer and wine','time','1997.Q1' )
        End-Declare
End-Setup
Begin-Program
print 'Quarter' (+2,1)
print 'Product' (0,15)
print 'Profit'  (0,48)
print 'Sales'   (0,64)
print 'Cost'    (0,75)

do Read_Cube
end-program

Begin-Procedure Read_Cube
Begin-Execute
        Connection=MSOLAP
        Schema='FoodMart 2000'
                GetData='Sales'
Begin-select loops=5000
Time            &time        (+2,1)
Product         &prod        (0,15)
Measures.Profit      &prof   (0,45)  edit 999999.99
"Measures.Store Cost"  &cog  (0,70) edit 999999.99
"Measures.Store Sales" &sale (0,60) edit 999999.99
From Rowsets=(1)
End-select
End-Execute
End-Procedure
```

*Program ex30.sqr* produces the following output.

### Output for *Program ex30.sqr*

| Quarter | Product | Profit | Sales | Cost |
|---|---|---|---|---|
| Q1 | Beer and Wine | 1858.19 | 3082.00 | 1223.81 |

## SET-GENERATIONS

SET-GENERATIONS selects a specific point within the hierarchy for the previously-declared dimension. You can use SET-GENERATIONS to set an exact location within the dimension declared by SET-MEMBERS. The dimension and hierarchy defined with SET-GENERATIONS can be a *literal* value only.

*Program ex31.sqr* uses SET-GENERATIONS to access the information in the Production Dimension Folder Tree in Figure 17. This example builds on *Program ex30.sqr*, and uses SET-GENERATIONS to broaden the range of report output from 'Beer and Wine' listed together to 'Beer' and 'Wine' listed separately.

In this example, SET-GENERATIONS:

- requests a specific generation for Product.

  SET-MEMBERS narrows the range to the Product hierarchy of 'all products.drink.alcoholic beverages.beer and wine', and SET-GENERATIONS requests all members at the fifth generation within Beer and Wine (Beer and Wine–listed separately).

- returns the set of members in the Time dimension that are at the second generation within the dimension. That is, it returns all Q1 members (generation 2) under the Time hierarchy of '1997.Q1.'

## Program ex31.sqr

```
Begin-Setup
        Declare-Connection MSOLAP
                dsn=MSOLAP
                set-members=('product','all products.drink.alcoholic
beverages.beer and wine','time','1997.Q1' )
                set-generations= ('time',2,'product', 5)
        End-Declare
End-Setup
Begin-Program
print 'Quarter' (+2,1)
print 'Product' (0,15)
print 'Profit'  (0,48)
print 'Sales'   (0,64)
print 'Cost'    (0,75)

do Read_Cube
end-program

Begin-Procedure Read_Cube
Begin-Execute
        Connection=MSOLAP
        Schema='FoodMart 2000'
                GetData='Sales'
Begin-select loops=5000
Time            &time         (+2,1)
Product         &prod         (,15)
Measures.Profit      &prof    (0,45)  edit 999999.99
"Measures.Store Cost"  &cog   (0,70) edit 999999.99
"Measures.Store Sales" &sale (0,60) edit 999999.99
From Rowsets=(1)
End-select
End-Execute
End-Procedure
```

The SET-GENERATIONS command in *Program ex31.sqr* produces the following output.

## Output for *Program ex31.sqr*

| Quarter | Product | Profit | Sales | Cost |
|---|---|---|---|---|
| Q1 | Beer | 439.67 | 722.99 | 283.32 |
| Q1 | Wine | 1418.53 | 2359.01 | 940.48 |

## SET-LEVELS

SET-LEVELS extends the dimension hierarchy for the previously-declared dimension. After declaring the starting point with SET-MEMBERS, use SET-LEVELS to specify how many levels to move down the hierarchy. The dimension and hierarchy defined with SET-LEVELS can be a *literal* value only.

*Program ex32.sqr* uses SET-LEVELS with SET-MEMBERS to access the information in the Production Dimension Folder Tree in Figure 17.

In this example, SET-LEVELS:

- requests additional levels for Product.

  SET-MEMBERS sets the starting point to Beer and Wine, and SET-LEVELS requests Beer and Wine list separately (level 2) plus the next level below (Good, Pearl, Portermouth, Top Measure, and Walrus).

- requests additional levels for Time.

  SET-MEMBERS sets the starting point to Q1 (level 1), and SET-LEVELS requests Q1 plus the next level below (1,2, and 3).

### Program ex32.sqr

```
Begin-Setup
        Declare-Connection MSOLAP
                dsn=MSOLAP
                set-members=('product','all products.drink.alcoholic
beverages.beer and wine','time','1997.Q1' )
                set-levels=('time', 1,'product', 2)
        End-Declare
End-Setup

Begin-Program
print 'Quarter' (+2,1)
print 'Product' (0,15)
print 'Profit'  (0,48)
print 'Sales'   (0,64)
print 'Cost'    (0,75)
do Read_Cube
end-program

Begin-Procedure Read_Cube
Begin-Execute
        Connection=MSOLAP
        Schema='FoodMart 2000'
                GetData='Sales'
Begin-select loops=5000
Time           &time         (+2,1)
Product        &prod         (0,15)
Measures.Profit      &prof   (0,45)  edit 999999.99
"Measures.Store Cost"  &cog  (0,70) edit 999999.99
"Measures.Store Sales" &sale (0,60) edit 999999.99
From Rowsets=(1)
End-select
End-Execute
```

```
End-Procedure
```

As shown in Output for *Program ex32.sqr*, adding SET-LEVELS produces a larger report than using SET-MEMBERS alone.

## Output for *Program ex32.sqr*

| Quarter | Product | Profit | Sales | Cost |
|---|---|---|---|---|
| Q1 | Beer and Wine | 1858.19 | 3082.00 | 1223.81 |
| 1 | Beer and Wine | 592.31 | 981.88 | 389.57 |
| 2 | Beer and Wine | 553.40 | 919.27 | 365.87 |
| 3 | Beer and Wine | 712.48 | 1180.85 | 468.37 |
| Q1 | Beer | 439.67 | 722.99 | 283.32 |
| 1 | Beer | 132.59 | 219.11 | 86.52 |
| 2 | Beer | 138.25 | 229.08 | 90.83 |
| 3 | Beer | 168.82 | 274.80 | 105.98 |
| Q1 | Wine | 1418.53 | 2359.01 | 940.48 |
| 1 | Wine | 459.72 | 762.77 | 303.05 |
| 2 | Wine | 415.14 | 690.19 | 275.05 |
| 3 | Wine | 543.66 | 906.05 | 362.39 |
| Q1 | Good | 54.51 | 89.08 | 34.57 |
| 1 | Good | 21.38 | 33.64 | 12.26 |
| 2 | Good | 15.61 | 27.16 | 11.55 |
| 3 | Good | 17.51 | 28.28 | 10.77 |
| Q1 | Pearl | 78.40 | 130.39 | 51.99 |
| 1 | Pearl | 25.55 | 44.44 | 18.89 |
| 2 | Pearl | 26.88 | 43.45 | 16.57 |
| 3 | Pearl | 25.97 | 42.50 | 16.53 |
| Q1 | Portsmouth | 114.59 | 182.82 | 68.23 |
| 1 | Portsmouth | 35.02 | 56.85 | 21.83 |
| 2 | Portsmouth | 32.50 | 52.98 | 20.48 |
| 3 | Portsmouth | 47.07 | 72.99 | 25.92 |
| Q1 | Top Measure | 39.89 | 68.42 | 28.53 |
| 1 | Top Measure | 12.43 | 21.66 | 9.23 |
| 2 | Top Measure | 12.74 | 22.28 | 9.54 |
| 3 | Top Measure | 14.73 | 24.48 | 9.75 |
| Q1 | Walrus | 152.29 | 252.28 | 100.00 |
| 1 | Walrus | 38.22 | 62.52 | 24.30 |
| 2 | Walrus | 50.52 | 83.21 | 32.69 |
| 3 | Walrus | 63.54 | 106.55 | 43.01 |
| Q1 | Good | 277.60 | 456.28 | 178.68 |
| 1 | Good | 95.17 | 156.23 | 61.06 |
| 2 | Good | 70.06 | 116.43 | 46.37 |
| 3 | Good | 112.36 | 183.62 | 71.26 |
| Q1 | Pearl | 251.82 | 422.27 | 170.45 |
| 1 | Pearl | 84.69 | 143.66 | 58.98 |
| 2 | Pearl | 105.90 | 176.38 | 70.48 |
| 3 | Pearl | 61.23 | 102.23 | 41.00 |
| Q1 | Portsmouth | 342.06 | 571.62 | 229.56 |
| 1 | Portsmouth | 94.47 | 155.15 | 60.68 |
| 2 | Portsmouth | 85.34 | 144.29 | 58.95 |
| 3 | Portsmouth | 162.25 | 272.18 | 109.93 |
| Q1 | Top Measure | 235.53 | 391.27 | 155.74 |
| 1 | Top Measure | 93.79 | 155.60 | 61.81 |
| 2 | Top Measure | 66.78 | 109.40 | 42.62 |
| 3 | Top Measure | 74.95 | 126.27 | 51.32 |
| Q1 | Walrus | 311.52 | 517.57 | 206.05 |

```
1               Walrus                  91.60         152.13         60.53
2               Walrus                  87.06         143.69         56.63
3               Walrus                 132.86         221.75         88.89
```

# Displaying Report Data

Once you decide what data to access and which commands to use, you can display the information in a report. Review the following sections to understand some aspects of report layout.

- Measures
- Aliases
- Column Order

## Measures

Measures are the numeric data displayed in reports. Some common measures are sales, cost, expenditures, and production count. Measures are aggregations stored for quick retrieval by users querying cubes. Each measure is stored in a column in a fact table in a cube. Measures can contain multiple columns combined in expressions. For example, the Profit measure is the difference of two numeric columns: Sales and Cost.

The format for measure columns is *'measures', dot, 'measure name'* (for example, *measures.profit*). Use this format regardless of the name used by the data source to declare measures.

*Program ex30.sqr* displays a measure, which cannot be included in `SET-MEMBERS`, `SET-GENERATIONS`, or `SET-LEVELS` but is written under `BEGIN-SELECT`.

## Aliases

Aliases are optional, descriptive names given to members and stored in alias tables. In report output, you can use aliases instead of member names when member names are non-descriptive. For example, a member name could be a number while its alias could be a noun such as Sales. Use the parameters property in the setup section of your Production Reporting program to specify whether to use names or aliases in your output.

➤ To use *aliases* in your Production Reporting output, write the following line in the setup section of your Production Reporting program:

```
Parameters=member.alias=true;
```

Including this statement displays the aliases in the *Name* column of the Production Reporting DDO Query Editor. (See "Viewing Cubes" on page 168 for more information).

- To use *names* in your Production Reporting output, do not include the following statement in your Production Reporting program:

  ```
  Parameters=member.alias=true;
  ```

  If you do not include this statement, the output displays the member names under the *Subtype Information* column of the Production Reporting DDO Query Editor. (See "Viewing Cubes" on page 168 for more information.)

**Note:**

Regardless of whether you display aliases in report output, you must use member names when referencing the members in the `SET-UP` section of your Production Reporting program.

## Column Order

The order in which dimension columns display in the `BEGIN-SELECT` section determines the order the data displays in multiple rowset queries. *Program ex33.sqr* shows `Profit`, `Sales`, and `Cost` reports for the selected Quarter and Product.

### Program ex33.sqr

```
Begin-Setup
        Declare-Connection MSOLAP
                dsn=MSOLAP
                set-members=('product','all products.drink.alcoholic
beverages','time','1997.Q1' )
                set-levels= ('time', 1,'product', 2)
                set-generations= ('time',2,'product', 3)
        End-Declare
End-Setup

Begin-Program
print 'Quarter' (+2,1)
print 'Product' (0,15)
print 'Profit'  (0,48)
print 'Sales'   (0,64)
print 'Cost'    (0,75)
print '' (+1,1)
do Read_Cube
end-program

Begin-Procedure Read_Cube
Begin-Execute
        Connection=MSOLAP
                Schema='FoodMart 2000'
                        GetData='Sales'
Begin-select loops=5000
Product         &prod          (+1,15)
Time            &time          (0,1)
Measures.Profit       &prof    (0,45) edit 999999.99
"Measures.Store Cost"  &cog    (0,70) edit 999999.99
"Measures.Store Sales" &sale   (0,60) edit 999999.99
From Rowsets=(1)
```

```
End-select
End-Execute
End-Procedure
```

*Program ex33.sqr* produces the following output.

### Output for *Program ex33.sqr*

| Quarter | Product | Profit | Sales | Cost |
|---|---|---|---|---|
| Q1 | Alcoholic Beverages | 1858.19 | 3082.00 | 1223.81 |
| 1 | Alcoholic Beverages | 592.31 | 981.88 | 389.57 |
| 2 | Alcoholic Beverages | 553.40 | 919.27 | 365.87 |
| 3 | Alcoholic Beverages | 712.48 | 1180.85 | 468.37 |
| Q1 | Beer and Wine | 1858.19 | 3082.00 | 1223.81 |
| 1 | Beer and Wine | 592.31 | 981.88 | 389.57 |
| 2 | Beer and Wine | 553.40 | 919.27 | 365.87 |
| 3 | Beer and Wine | 712.48 | 1180.85 | 468.37 |
| Q1 | Beer | 439.67 | 722.99 | 283.32 |
| 1 | Beer | 132.59 | 219.11 | 86.52 |
| 2 | Beer | 138.25 | 229.08 | 90.83 |
| 3 | Beer | 168.82 | 274.80 | 105.98 |
| Q1 | Wine | 1418.53 | 2359.01 | 940.48 |
| 1 | Wine | 459.72 | 762.77 | 303.05 |
| 2 | Wine | 415.14 | 690.19 | 275.05 |
| 3 | Wine | 543.66 | 906.05 | 362.39 |

To change the look of the output, modify the dimension order under BEGIN-SELECT. If the order of the Time and Product dimensions is reversed (as in *Program ex34.sqr*) then you get a report with columns displayed in a different sorting order.

### Program ex34.sqr

```
Begin-select loops=5000
Product         &prod          (+1,15)
Time            &time          (0,1)
Measures.Profit     &prof      (0,45) edit 999999.99
"Measures.Store Cost"  &cog    (0,70) edit 999999.99
"Measures.Store Sales" &sale   (0,60) edit 999999.99
From Rowsets=(1)
End-select
End-Execute
End-Procedure
```

In Output for *Program ex34.sqr*, the rows are sorted first by Quarter: All the Q1 rows are first, followed by rows 1, 2, and 3. The Quarter column sets the row order because Time is listed first under BEGIN-SELECT.

### Output for *Program ex34.sqr*

| Quarter | Product | Profit | Sales | Cost |
|---|---|---|---|---|
| Q1 | Alcoholic Beverages | 1858.19 | 3082.00 | 1223.81 |
| Q1 | Beer and Wine | 1858.19 | 3082.00 | 1223.81 |
| Q1 | Beer | 439.67 | 722.99 | 283.32 |
| Q1 | Wine | 1418.53 | 2359.01 | 940.48 |
| 1 | Alcoholic Beverages | 592.31 | 981.88 | 389.57 |

| | | | | |
|---|---|---|---|---|
| 1 | Beer and Wine | 592.31 | 981.88 | 389.57 |
| 1 | Beer | 132.59 | 219.11 | 86.52 |
| 1 | Wine | 459.72 | 762.77 | 303.05 |
| 2 | Alcoholic Beverages | 553.40 | 919.27 | 365.87 |
| 2 | Beer and Wine | 553.40 | 919.27 | 365.87 |
| 2 | Beer | 138.25 | 229.08 | 90.83 |
| 2 | Wine | 415.14 | 690.19 | 275.05 |
| 3 | Alcoholic Beverages | 712.48 | 1180.85 | 468.37 |
| 3 | Beer and Wine | 712.48 | 1180.85 | 468.37 |
| 3 | Beer | 168.82 | 274.80 | 105.98 |
| 3 | Wine | 543.66 | 906.05 | 362.39 |

# Accessing Cubes: An Example

This section gives an example of a cube and discusses the Production Reporting code necessary to access the cube. Review this section for information on:

- The Cube
- The Production Reporting Code Needed to Access the Cube
- An Explanation of the Code

## The Cube

Figure 18 shows a cube with three dimensions: Product (Y-axis), Time (X-axis), and Accounts (Z-axis). This example illustrates only a small part of an entire cube. For example, the Time dimension could have four quarters while only the first quarter is illustrated here. In addition, dimensions such as Location or Salesperson, could also be added to this cube.

Each cell of the cube represents a data value in a database. For example, one cell might hold a value of Beer Sales in January. In addition to being divided by cells, the cube can also be sliced into small segments, such as, Sales for Beer across the entire quarter or Cost in February (2) for all products (Beer and Wine).

**Figure 18 Three-Dimensional Data Cube**

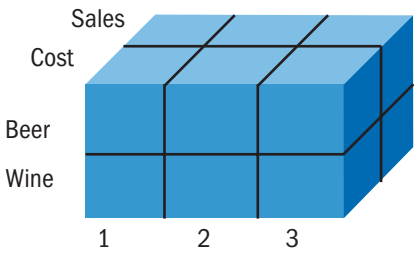
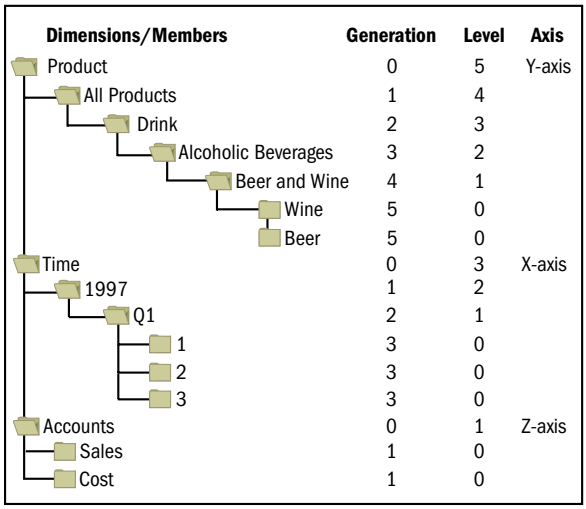



Figure 19 reveals the dimensions and levels of the cube in Figure 18 above. Drinks is a member of the Product dimension, 1997 is a member of the Time dimension, and Sales and Cost are members of the Accounts dimension. Each dimension corresponds to a cube axis.

Figure 19 Hierarchy of a Data Cube




## The Production Reporting Code Needed to Access the Cube

The Production Reporting code in *Program ex35.sqr* accesses the OLAP cube discussed in the previous section. The code uses the three commands for accessing a cube: SET-MEMBERS, SET-GENERATIONS, and SET-LEVELS.

### Program ex35.sqr

```
Begin-Setup
        Declare-Connection MSOLAP
                dsn=MSOLAP
                set-members=('product','all products.drink.alcoholic
beverages','time','1997.Q1' )
                set-levels= ('time', 1,'product', 2)
                set-generations= ('time',2,'product', 3)
        End-Declare
End-Setup

Begin-Program
print 'Quarter' (+2,1)
print 'Product' (0,15)
print 'Profit'  (0,48)
print 'Sales'   (0,64)
print 'Cost'    (0,75)
print '' (+1,1)
do Read_Cube
end-program

Begin-Proced Read_Cube
Begin-Execute
        Connection=MSOLAP
                Schema='FoodMart 2000'
                        GetData='Sales'
Begin-select loops=5000
Time                    &time   (+1,1)
Product                 &prod   (0,15)
```

```
Measures.Profit         &prof   (0,45) edit 999999.99
"Measures.Store Cost"   &cog    (0,70) edit 999999.99
"Measures.Store Sales" &sale    (0,60) edit 999999.99
From Rowsets=(1)
End-select
End-Execute
End-Procedure
```

**Note:**

The order in which the columns and rows in your output appear depends on the order in which the dimensions are listed under `BEGIN-SELECT`. The columns in Output for Program ex34.sqr are listed in Product-by Market-by Year order. To change the look of the output, modify the dimension order under `BEGIN-SELECT`.

### Output for *Program ex35.sqr*

| Quarter | Product | Profit | Sales | Cost |
|---|---|---|---|---|
| Q1 | Alcoholic Beverages | 1858.19 | 3082.00 | 1223.81 |
| 1 | Alcoholic Beverages | 592.31 | 981.88 | 389.57 |
| 2 | Alcoholic Beverages | 553.40 | 919.27 | 365.87 |
| 3 | Alcoholic Beverages | 712.48 | 1180.85 | 468.37 |
| Q1 | Beer and Wine | 1858.19 | 3082.00 | 1223.81 |
| 1 | Beer and Wine | 592.31 | 981.88 | 389.57 |
| 2 | Beer and Wine | 553.40 | 919.27 | 365.87 |
| 3 | Beer and Wine | 712.48 | 1180.85 | 468.37 |
| Q1 | Beer | 439.67 | 722.99 | 283.32 |
| 1 | Beer | 132.59 | 219.11 | 86.52 |
| 2 | Beer | 138.25 | 229.08 | 90.83 |
| 3 | Beer | 168.82 | 274.80 | 105.98 |
| Q1 | Wine | 1418.53 | 2359.01 | 940.48 |
| 1 | Wine | 459.72 | 762.77 | 303.05 |
| 2 | Wine | 415.14 | 690.19 | 275.05 |
| 3 | Wine | 543.66 | 906.05 | 362.39 |

## An Explanation of the Code

Table 21 explains the code necessary to access the example OLAP cube.

**Table 21** Code Explanation

| Code | Explanation |
|---|---|
| `Begin-Setup`<br>`  Declare-Connection` MSOLAP<br>`    dsn=MSOLAP`<br>`    User=`<br>`     Password=`<br>`    set-members=`<br>`('product','all  products.drink.alcoh`<br>`olic  beverages','time','1997.Q1' )` | The setup section of the Production Reporting program.<br>● `Declare-Connection`—User-defined name for describing a cube connection. In this case, we used `OLAP`.<br>● `DSN`—Essbase connection. Enter the DDO registry name in the DDO registry.<br>● `USER`—User name for the MSOLAP connection.<br>● `PASSWORD`—Password for the user. |

| Code | Explanation |
|---|---|
| `set-levels= ('time', 1, 'product', 2)`<br>`set-generations= ('time', 2, 'product', 3)`<br>`End-Declare`<br>`End-Setup` | You can also connect from the command line using the following syntax:<br>`SQR [program] DSN/[username]/[password]`<br>● `Parameters`—Optional parameter. Declares whether to use an alias table.<br>If you include this statement, aliases instead of names are displayed in the output.<br>● `set-members`—The name of the dimension at the specific hierarchical level.<br>● `set-levels`—Expends the dimension hierarchy for the previously-declared dimension.<br>● `set-generations`—Overrides the specific hierarchical level declared by `set-members`. |
| `Begin-Program`<br>`print 'Quarter' (+2,1)`<br>`print 'Product' (0,15)`<br>`print 'Profit' (0,48)`<br>`print 'Sales' (0,64)`<br>`print 'Cost' (0,75)`<br>`do Read_Cube`<br>`end-program` | The program section of the Production Reporting program.<br>● `print`—Prints column header text.<br>In this example, the program prints the headings Quarter, Product, Profit, Sales, and Cost.<br>● `do Read_Cube`—Directs Production Reporting to read the cube. |
| `Begin-Procedure Read_Cube`<br>`Begin-Execute`<br>`Connection=MSOLAP`<br>`Schema='FoodMart 2000'`<br>`GetData='Sales'` | Begins the new query or procedure execution.<br>● `Connection`—Points to the information in `Declare-Connection`.<br>● `schema`—Name of the cube.<br>● `getdata`—Name of the table in the cube.<br>You must include `Connection`, `Schema`, and `GetData` in all cubes. |
| `Begin-select loops=5000`<br>`Time &time (+2,1)`<br>`Product &prod (0,15)` | Begins a SELECT paragraph.<br>● `loops=5000`—(Optional) Specifies the number of rows to retrieve. After the specified number is processed, the `SELECT` loop exits.<br>● `&time,&prod`—The dimensions in the cube. |
| `Measures.Profit &prof (0,45) edit 999999.99`<br>`"Measures.Store Cost" &cog (0,70) edit 999999.99`<br>`"Measures.Store Sales" &sale (0,60) edit 999999.99` | Required to retrieve the numeric data.<br>Measures are hierarchical in design; each dot defines another level or member.<br>The measures in this example are `Profit`, `Store Cost`, and `Store Sales`. |
| From MSOLAP | `From`—Required for Production Reporting.<br>`MSOLAP`—(Optional) Documents the data source of a select statement. |

| Code | Explanation |
| --- | --- |
| `End-Select` | Completes `Begin-Select`. |
| `End-Execute` | Completes `Begin-Execute`. |
| `End-Procedure` | Completes `Begin-Procedure`. |

Part III

# Appendices

In Appendices:


- Sample Resource Files
- Using the HTTP-enabled XML DDO Driver

# A Sample Resource Files

**In This Appendix**

SAP DataSource Property Description ..... 185
SAP DataSource Messages ..... 190

## SAP DataSource Property Description

```
# Title:        Production Reporting Objects -- SAP/R3 Access
# Version:
# Copyright:    Copyright (c) 2005
# Company:      Oracle
# Description:  SAP/R3 Access Data Source Property Descriptions
#

logon.Name=Logon
logon.Description=Logon is required to establish a connection with an SAP R3data source.
\
The SAP R3data source takes a number of parameters as part of the connection string. \
Included among these are the operator (or user) identification and the operator (or
user) \
password. The user identification and password are included as logon information. The
remaining \
connection attributes are part of the connection string provided with the data source \
description (see the SAP R3 DataSource interface documentation for a complete
description). \
To summarize, the SAP R3 connection string may have a form similar to: \
SAPR3:JNI:ASHOST=hs0311 SYSNR=53. \
Where the ASHOST would be replaced by the data source host identifier; the SYSNR would
be replaced by \
the system number for the host. \
The client (CLIENT=), user identification (USER=), password (PASSWD=) and language
(LANG=) \
are appended to the connection string. These are usually retained as properties. A user
\
interface will normally prompt for these values.
logon.Indices=logon.client \
user \
password \
logon.language \
logon.gateway \
logon.noloadbalancing \
logon.loadbalancing \
logon.advanced
logon.Required=true
```

```
logon.client.Name=Client
logon.client.Description=SAP logon client identifier.
logon.client.ClassName=java.lang.String
logon.client.Required=true

logon.language.Name=Language
logon.language.Description=SAP logon language (1-byte SAP language or 2-byte ISO
language).
logon.language.ClassName=java.lang.String

logon.gateway.Name=Gateway
logon.gateway.Description=The attributes of this node provide gateway specifications.
logon.gateway.Indices=logon.dest \
logon.gwhost \
logon.gwserv

logon.loadbalancing.Name=Load Balancing
logon.loadbalancing.Description=The attributes of this node provide load balancing
specifications.
logon.loadbalancing.Indices=logon.mshost \
logon.r3name \
logon.group

logon.noloadbalancing.Name=No Load Balancing
logon.noloadbalancing.Description=The attributes of this node provide additional no load
balancing specifications.
logon.noloadbalancing.Indices=logon.sysnr \
logon.ashost

logon.advanced.Name=Advanced
logon.advanced.Description=The attributes of this node provide advanced and diagnostic
specifications.
logon.advanced.Indices=logon.type \
logon.check \
logon.trace

logon.type.Name=RFC Server Type
logon.type.Description=The RFC server type. This parameter should be left empty or set
to 3, the default.
logon.type.ClassName=java.lang.String
logon.type.ValidationType=2
logon.type.Validator=com.sqribe.comutil.StringValidator
logon.type.ValidationValues=3 E 2

logon.check.Name=SAP Logon Check
logon.check.Description=SAP logon check during Open. The default is 1. The default
should always be used.
logon.check.ClassName=java.lang.String
logon.check.ValidationType=2
logon.check.Validator=com.sqribe.comutil.StringValidator
logon.check.ValidationValues=1 0

logon.trace.Name=RFC Trace
logon.trace.Description=Establish an RFC trace log. The default is 0, do not establish a
trace log.
logon.trace.ClassName=java.lang.String
```

```
logon.trace.ValidationType=2
logon.trace.Validator=com.sqribe.comutil.StringValidator
logon.trace.ValidationValues=1 0

logon.dest.Name=SAP Logical Destination
logon.dest.Description=Destination in saprfc.ini, when using saprfc.ini. If the RFC
server is an R/2 system \
this destination must also be defined in the 'sideinfo' for the SAP gateway. This
parameter value should \
only be specified when an saprfc.ini file is being referenced; otherwise, it should be
left empty.
logon.dest.ClassName=java.lang.String

logon.gwhost.Name=SAP Gateway Host
logon.gwhost.Description=The host name of the SAP gateway, when the server is R/2 or
External. \
This parameter value should only be specified when the server is R/2 or External;
otherwise, it should be left empty.
logon.gwhost.ClassName=java.lang.String

logon.gwserv.Name=SAP Gateway Service
logon.gwserv.Description=The service of the SAP gateway, when the server is R/2 or
External. \
This parameter value should only be specified when the server is R/2 or External;
otherwise, it should be left empty.
logon.gwserv.ClassName=java.lang.String

logon.mshost.Name=SAP Message Server Host
logon.mshost.Description=The host name of the SAP message server, when using load
balancing. \
This parameter value should only be specified when the server is R/3 and load balancing
is being used; \
otherwise, it should be left empty.
logon.mshost.ClassName=java.lang.String

logon.r3name.Name=R/3 Load Balancing Server Name
logon.r3name.Description=The SAProuter string for the R/3 system, when using load
balancing. \
This parameter value should only be specified when the server is R/3 and load balancing
is being used; \
otherwise, it should be left empty.
logon.r3name.ClassName=java.lang.String

logon.group.Name=Application Server Group Name
logon.group.Description=The name of the group of application servers, when using load
balancing. \
This parameter value should only be specified when the server is R/3 and load balancing
is being used; \
otherwise, it should be left empty.
logon.group.ClassName=java.lang.String

logon.ashost.Name=External Server Name
logon.ashost.Description=The host name of a specific application server, when using R/3
without load balancing. \
This parameter value should only be specified when the server is R/3, the server is an
external application \
server, and load balancing is not being used; \
```

```
otherwise, it should be left empty.
logon.ashost.ClassName=java.lang.String

logon.sysnr.Name=External Server System Number
logon.sysnr.Description=The SAP system number of a specific application server, when
using R/3 without load balancing. \
This parameter value should only be specified when the server is R/3, the server is an
external application \
server, and load balancing is not being used; \
otherwise, it should be left empty.
logon.sysnr.ClassName=java.lang.String

filters.Name=SAP Hierarchy Filters
filters.Description=SAP hierarchy filters provide a mechanism for controlling the amount
\
of information retrieved as part of the hierarchy. For example, the hierarchy may
include \
all business objects or only business objects with BAPIs. In another case, the hierarchy
\
may only include business objects with BAPIs that have a status of "released."
filters.Indices=filters.BAPI \
filters.Objects \
filters.Interfaces \
filters.Organization \
filters.Implemented \
filters.Released \
filters.Modelled \
filters.Obsolete \
filters.Delegated

filters.BAPI.Name=Object Type Filter
filters.BAPI.Description=When activated, this filter will restrict the hierarchy to
business \
objects to those containing BAPIs. When deactivated, the hierarchy, potentially, will
contain \
all business objects.
filters.BAPI.ClassName=java.lang.Boolean
filters.BAPI.ValidationType=2
filters.BAPI.Validator=com.sqribe.comutil.BooleanValidator
filters.BAPI.ValidationValues=true false

filters.Objects.Name=Object Filter
filters.Objects.Description=If this filter and the Interfaces filter are deactivated,
i.e., \
set to false, all SAP Business Object types and interface types are returned. When this
\
filter is activated, then only Business Object types are returned. \
When not specified, this filter defaults to "false".
filters.Objects.ClassName=java.lang.Boolean
filters.Objects.ValidationType=2
filters.Objects.Validator=com.sqribe.comutil.BooleanValidator
filters.Objects.ValidationValues=true false

filters.Interfaces.Name=Object Filter
filters.Interfaces.Description=If this filter and the Objects filter are deactivated,
i.e., \
```

```
set to false, all SAP Business Object types and interface types are returned. When this
\
filter is activated, then only Interface type identifiers are returned. \
When not specified, this filter defaults to "false".
filters.Interfaces.ClassName=java.lang.Boolean
filters.Interfaces.ValidationType=2
filters.Interfaces.Validator=com.sqribe.comutil.BooleanValidator
filters.Interfaces.ValidationValues=true false

filters.Organization.Name=Organizational Unit Filter
filters.Organization.Description=Filter of organizational unit object types. \
A value of "true" indicates that all organizational unit object types are returned. \
A value of "false" indicates that no organizational unit object types are returned. \
When not specified, this filter defaults to "false".
filters.Organization.ClassName=java.lang.Boolean
filters.Organization.ValidationType=2
filters.Organization.Validator=com.sqribe.comutil.BooleanValidator
filters.Organization.ValidationValues=true false

filters.Implemented.Name=With Implemented Filter
filters.Implemented.Description=Filter for implemented SAP Business Object types.
Implmented \
object types have not been officially released but can be used in a runtime environment.
Use \
this filter when the "Released" filter is set to "Released". The value, "true", means
that object \
types with status, "implemented", are also returned. The value, "false", means that
objects \
types with status, "implemented", are not returned. \
When not specified, the default value is "true".
filters.Implemented.ClassName=java.lang.Boolean
filters.Implemented.ValidationType=2
filters.Implemented.Validator=com.sqribe.comutil.BooleanValidator
filters.Implemented.ValidationValues=true false

filters.Released.Name=Released Object Type Filter
filters.Released.Description=Filter for released SAP Business Object types. \
The value, "true", means that "released" object types are returned. \
The value, "false", means that "released" object types are not returned. \
When not specified, the default value is "false".
filters.Released.ClassName=java.lang.Boolean
filters.Released.ValidationType=2
filters.Released.Validator=com.sqribe.comutil.BooleanValidator
filters.Released.ValidationValues=true false

filters.Modelled.Name=Modelled Object Type Filter
filters.Modelled.Description=Filter for modelled SAP Business Object types. \
The value, "true", means that all modelled object types are returned. \
The value, "false", means that no modelled object types are returned. \
When not specified, the default value is "false".
filters.Modelled.ClassName=java.lang.Boolean
filters.Modelled.ValidationType=2
filters.Modelled.Validator=com.sqribe.comutil.BooleanValidator
filters.Modelled.ValidationValues=true false

filters.Obsolete.Name=With Obsolete Filter
filters.Obsolete.Description=Filter for obsolete SAP Business Object types. Use \
```

```
this filter when the "Released" filter is set to "Released". The value, "true", means
that object \
types with status, "obsolete", are also returned. The value, "false", means that objects
\
types with status, "obsolete", are not returned. \
When not specified, the default value is "false".
filters.Obsolete.ClassName=java.lang.Boolean
filters.Obsolete.ValidationType=2
filters.Obsolete.Validator=com.sqribe.comutil.BooleanValidator
filters.Obsolete.ValidationValues=true false

filters.Delegated.Name=With Delegated Object Type Filter
filters.Delegated.Description=Filter for delegated SAP Business Object types. \
The value, "true", means that all object types, including delegated object types, are
returned. \
The value, "false", means that no delegated object types are returned. \
When not specified, the default value is "true".
filters.Delegated.ClassName=java.lang.Boolean
filters.Delegated.ValidationType=2
filters.Delegated.Validator=com.sqribe.comutil.BooleanValidator
filters.Delegated.ValidationValues=true false
```

# SAP DataSource Messages

```
# Title:        Production Reporting Objects -- SAPR3acc
# Version:
# Copyright:    Copyright (c) 1999
# Company:      SQRIBE Technologies
# Description:  SAP R3 Access Common Messages
#

# The connection string protocol value is not recognized by the SAP R3 DAO driver
# 0    class name
# 1 method name
# 2 specified protocol name
# 3 expected protocol name
# 4 the connection string
InvalidProtocolName.text=The specified connection protocol, "{2}", is not supported by \
this driver. This driver supports the "{3}" protocol. The connection string is: {4}.
InvalidProtocolName.logtext={0}.{1}(): The specified connection protocol, "{2}", is not
supported by \
this driver. This driver supports the "{3}" protocol. The connection string is: {4}.

# The connection string subprotocol value is not recognized by the SAP R3 DAO driver
# 0    class name
# 1 method name
# 2 specified subprotocol name
# 3 expected subprotocol name
# 4 the connection string
InvalidSubProtocolName.text=The specified connection subprotocol, "{2}", is not
supported by \
this driver. This driver supports the "{3}" subprotocol. The connection string is: {4}.
InvalidSubProtocolName.logtext={0}.{1}(): The specified connection subprotocol, "{2}",
is not supported by \
this driver. This driver supports the "{3}" subprotocol. The connection string is: {4}.
```

```
# A parameter in the connection string was not recognized
# 0    class name
# 1 method name
# 2 specified parameter key
# 3 specified parameter value
# 4 the connection string
UnknownLogonParameter.text=The specified parameter, "{2}={3}", is not recognized by \
this driver. The connection string is: {4}. The parameter will be passed to the data
source.
UnknownLogonParameter.logtext={0}.{1}(): The specified parameter, "{2}={3}", is not
recognized by \
this driver. The connection string is: {4}. The parameter will be passed to the data
source.

# Unexpected SAP R3 RFC error occurred. It has been converted to an SAP R3 exception.
# 0    class catching the exception
# 1 method catching the exception
# 2 error key
# 3 error group
# 4 error message
SAPR3Exception.text=SAP R3 message: key={2}, group={3}, and msg={4}.
SAPR3Exception.logtext={0}.{1}(): SAP R3 exception caught: key={2}, group={3}, and msg=
{4}.

# SAPR3Field internal error set field with improper field type
# 0    class catching the exception
# 1 method catching the exception
# 2 the string dump of the underlying field
SAPR3FieldInternalError.text=Unexpected field instance of {2}.
SAPR3FieldInternalError.logtext={0}.{1}(): Unexpected field instance of {2}.

# BOR Tree retrieval error message
# 0    class name
# 1 method name
# 2 the SAP R/3 message text
BorTreeMsg.text={2}.
BorTreeMsg.logtext={0}.{1}(): {2}.

# ParameterException container message, indicating errors occurring during container
marshalling
# 0 class name
# 1 method name
# 2 messages
ParameterException.text={2}
ParameterException.logtext={0}.{1}(): {2}

# There are too many parameters in the parameter list
# 0 class name
# 1 method name
# 2 input parameter count from metadata
# 3 parameter list count
TooManyParameters.text=Too many parameters. There are {2} parameters in the metadata \
but there are {3} parameters in the parameter list.
TooManyParameters.logtext={0}.{1}: Too many parameters. There are {2} parameters in the
metadata \
but there are {3} parameters in the parameter list.
```

```
# There are too few parameters in the parameter list
# 0 class name
# 1 method name
# 2 input parameter count from metadata
# 3 parameter list count
TooFewParameters.text=Too few parameters. There are {2} parameters in the metadata \
but there are {3} parameters in the parameter list.
TooFewParameters.logtext={0}.{1}: Too few parameters. There are {2} parameters in the
metadata \
but there are {3} parameters in the parameter list.

# Conflicting field types for the expected and actual parameters
# 0 class name
# 1 method name
# 2 input parameter field type
# 3 parameter list field type
ConflictingParameterFieldTypes.text=Conflicting parameter field types. The metatdata \
parameter field type is {2}. The input parameter field type is {3}.
ConflictingParameterFieldTypes.logtext={0}.{1}: Conflicting parameter field types. The
metatdata \
parameter field type is {2}. The input parameter field type is {3}.

# Parameter format error
# 0 class name
# 1 method name
# 2 expected parameter name
# 3 expected parameter field type
# 4 received parameter name
# 5 received parameter field type
# 6 exception
ParameterFormatError.text=Expected parameter {2} has a field type of {3}. The parameter
list \
parameter {4} has a field type of {5}. The exception was {6}.
ParameterFormatError.logtext={0}.{1}: Expected parameter {2} has a field type of {3}.
The parameter list \
parameter {4} has a field type of {5}. The exception was {6}.

# Required parameter is null
# 0 class name
# 1 method name
# 2 parameter list index
RequiredParameterEmpty.text=Parameter {2} is required parameter, but is empty.
RequiredParameterEmpty.logtext={0}.{1}: Parameter {2} is required parameter, but is
empty.

# Invocation of object returned an error message
# 0 class name
# 1 method name
# 2 object type
# 3 error code
# 4 error type
# 5 workarea
# 6 message
# 7 text
InvokeErrorMessage.text=Object, {2}, invocation returned error message: \
Code: {3}, Type: {4}, Workarea: {5}, Message: {6}, Text: {7}.
```

```
InvokeErrorMessage.logtext={0}.{1}: Object, {2}, invocation returned error message: \
Code: {3}, Type: {4}, Workarea: {5}, Message: {6}, Text: {7}.

# Handle is not invalid or null. This is mostly likely an internal processing error
# 0 class name
# 1 method name
# 2 table object state
InvalidHandle.text=The RFC table handle is not valid or is null. \
The handle either has not been created or was previously destroyed. \
Table object state: {2}.
InvalidHandle.logtext={0}.{1}: The RFC table handle is not valid or is null. \
The handle either has not been created or was previously destroyed. \
Table object state: {2}.

# Rowset is null
# 0 class name
# 1 method name
# 2 parameter name
RowsetIsNull.text=Expected rowset for parameter, {2}, is null.
RowsetIsNull.logtext={0}.{1}: Expected rowset for parameter, {2}, is null.

# Fetch of structure failed
# 0 class name
# 1 method name
# 2 parent object name (e.g., the schema procedure name)
# 3 structure name
# 4 exception message
StructureFetchFailed.text=An exception was caught while retrieving the \
description of the structure, {3}, for object, {2}. The exception message is: {4}.
StructureFetchFailed.logtext={0}.{1}: An exception was caught while retrieving the \
description of the structure, {3}, for object, {2}. The exception message is: {4}.

# Field not found. This should not occur. It means that the metadata returned
# from SAP is not valid (or the permissions of the user are really messed up).
# 0 class name
# 1 method name
# 2 column name
# 3 field name
FieldNotFound.text=The field, {3}, for column, {2}, was not found.
FieldNotFound.logtext={0}.{1}(): The field, {3}, for column, {2}, was not found.

# CallResults failed for a HelpValues retrieval. This should not occur.
# It means that the metadata returned from SAP is not valid (or the permissions
# of the user are really messed up).
# 0 class name
# 1 method name
# 2 object name
# 3 procedure name
# 4 parameter name
# 5 field name
# 6 exception message
HelpCallResultsFailed.text=The BAPI call to obtain help description \
information for object type, {2}, procedure, {3}, parameter, {4}, and field, {5}, \
failed. Exception: {6}.
HelpCallResultsFailed.logtext={0}.{1}(): The BAPI call to obtain help description \
information for object type, {2}, procedure, {3}, parameter, {4}, and field, {5}, \
failed. Exception: {6}.
```

```
# The help description metadata is invalid.
# 0 class name
# 1 method name
# 2 column name
# 3 exception message
HelpMetaDataInvalid.text=The help description metadata is invalid for \
column, {2}. Exception: {3}.
HelpMetaDataInvalid.logtext={0}.{1}(): The help description metadata is invalid for \
column, {2}. Exception: {3}.

# The help description values are invalid.
# 0 class name
# 1 method name
# 2 column name
# 3 exception message
HelpValuesInvalid.text=The help description values are invalid for \
column, {2}. Exception: {3}.
HelpValuesInvalid.logtext={0}.{1}(): The help description values are invalid for \
column, {2}. Exception: {3}.

# The Help description return value
# 0 class name
# 1 method name
# 2 object name
# 3 procedure name
# 4 parameter name
# 5 field name
# 6 message text (dump of return value row)
HelpReturnValue.text=The request for help description for \
object={2}, procedure={3}, parameter={4}, field={5}, produced message: {6}.
HelpReturnValue.logtext={0}.{1}(): The request for help description for \
object={2}, procedure={3}, parameter={4}, field={5}, produced message: {6}.

# Expected parameters based upon the metadata information
# 0 class name
# 1 method name
# 2 parameter metadata dump
ExpectedParameterMetadata.logtext={0}.{1}() The expected parameter metadata is: {2}.

# Actual parameter list
# 0 class name
# 1 method name
# 2 parameter list dump
ActualParameterList.logtext={0}.{1}() The actual parameter list is: {2}.
```

# B Using the HTTP-enabled XML DDO Driver

**In This Appendix**


Usage ..... 195
Accessing XML Files via HTTP Using the Production Reporting DDO Query Editor ..... 197
Limitation ..... 197


**Note:**

See "XML Support in Production Reporting" in Volume 1 of the *Hyperion SQR Production Reporting Developer's Guide* for a description of XML support in Production Reporting.

## Usage

The XML DDO driver checks for the presence of a property URL, and reads data from that URL if specified. If the URL property is not present, the XML DDO driver reads data from the XML files in the directory specified in the connection string. You do not need a special DDO registry entry to access URLs. Instead, you can make a DDO registry entry that points to a local directory, and a Production Reporting program can use that entry to access XML files via URLs.

➤ To use the XML DDO driver:

1. **Define a registry entry**
2. **Declare a connection**
3. **Use Getdata= in the begin-execute section**

Review the following sections for specific information on each of the above steps, as well as how to use alter-connection and specify URLs at runtime.

### Define a Registry Entry

Define a DDO Registry entry which uses the XML DDO driver. Use the DDO Registry Editor to create one, or edit the *properties\Registry.properties* file manually. It should contain an entry such as:

```
SampleXML.desc=Sample XML files
SampleXML.class=com.sqribe.xmlacc.XMLDataSource
SampleXML.lib=
```

```
SampleXML.load=
SampleXML.conn=D:\\XML_Data\\SampleXML
```

## Declare a Connection

Declare a connection to your data source in the BEGIN-SETUP section of your Production Reporting program. For example:

```
begin-setup
declare-connection xml
   DSN=SampleXML
end-declare
end-setup
```

This allows access to the files in the directory specified by the connection string in the registry. (D:\XML_Data\SampleXML in the example above.)

To use the new HTTP features of the driver, include PARAMETERS= URL=<your_url> in the connection declaration:

```
begin-setup
declare-connection xml
   DSN=SampleXML
   PARAMETERS= URL=http://server/path/filename.xml;
end-declare
end-setup
```

## Use Getdata= in the Begin-Execute Section

In the BEGIN-EXECUTE portion of your Production Reporting DDO program, specify the connection. Make sure to use the file name (without the extension) as the schema name parameter to GetData:

```
begin-execute
   connection=xml
   GetData='filename'
```

## An Alternate Method

Instead of specifying the URL in a DECLARE-CONNECTION block, you can specify it in ALTER-CONNECTION. For example:

```
alter-connection
   NAME=xml
   PARAMETERS= URL=http://server/path/filename.xml;
```

The ALTER-CONNECTION method is particularly useful if you specify the connection on the Production Reporting DDO command line. In this case, you can leave out the DECLARE-CONNECTION statements, and simply alter the default connection:

```
alter-connection
```

```
NAME=default
PARAMETERS= URL=http://server/path/filename.xml;
```

### Specifying URLs at Runtime

To use a URL from a variable, build a parameter string which begins 'URL=' and ends ';', and use it in an alter-connection statement. For example:

```
let $params = 'URL=' || $url || ';'
alter-connection
   NAME=xml
   PARAMETERS=$params
```

## Accessing XML Files via HTTP Using the Production Reporting DDO Query Editor

Editing the property files slightly makes it easier to use the Production Reporting DDO Query Editor to test this driver. Edit the following file and uncomment the lines defining the logon property:

`properties\com_sqribe_xmlacc_XMLDataSource_PropertyDescriptions.properties`

Defining the logon property this way, including setting `logon.Required=true`, forces the Production Reporting DDO Query Editor to display a dialog allowing entry of the URL value at runtime.

If you also edit the following file and uncomment the `URL=line`, the logon dialog box displays a default value that you can update.

`properties\com_sqribe_xmlacc_XMLDataSource_Properties.properties`

This saves you typing while you test using the Production Reporting DDO Query Editor; however, it also prevents use of the XML driver to access the directory of files specified in the connection string. This is because the URL property is always defined for *all* XML data sources, and thus the driver always uses the URL provided.

**Note:**

Do not define a logon property and specify a default URL on a production installation, because it may affect existing Production Reporting DDO programs which access XML files on the file system. These changes are most useful in a development environment, when you use the Production Reporting DDO Query Editor to view the structure, schema names, and content of XML files obtained through HTTP.

## Limitation

When using Oracle's Hyperion® SQR® Production Reporting DDO with the XML DDO driver, **the XML DDO driver uses a validating DOM parser.** This involves significant memory

usage, and means that large XML files (approaching a three megabytes in size) may trigger out-of-memory errors in the JVM.

# Index


## A

access to messages, 75
adding
  a specification in the registry, 88
  connection specification information, 74
allocateFields() method, 71
application security, 12
architecture for Production Reporting DDO drivers, 16

## B

base classes, 11
BEGIN-SELECT, 135
bi-directional gateway, 73
bwacc driver, 132

## C

call() method, 44
calling a procedure to obtain data, 15
calling procedures, 44
CallResults interface, 15
capabilities, 28
capabilities files, 56
capability, code example, 29
class hierarchy, used by message facility, 57
close() method, 61
code example
  access to messages, CSV driver, 76
  call () method, 44
  CSV driver, 55
  registry entry, 59
  retrieving data and processing the data, 19
  set uer and password, 25
  to check for the procedure's return value, 36
  using a selector to pick specific dimensions, 50
column metadata
  providing, 62
  retrieving, 34
columns, 37
common connection implementation method, 73
connection
  establishing, 25
  interface, 13, 25
connection interface, implementing, 60
createField() method, 71
createSchemas() method, 61
creating a new data source, edit a registry file, 22
CSV
  driver, 55
  driver template, 75
  interface, 58
CSVFileclass methods, 62

## D

data access strategies, 17
data objects
  definition of, 14
  obtaining information about, 33
data source
  capabilities, 28
  implementation mehods, 73
  name, 87
  specification, 87
database cursors, 44
DataSource interface, 59
DataSourceManager, 20
DataSourceManagerAdmin, 22
DDO
  access from other programming languages, 12
  API, 19
  defined, 12
  SDK, 12
defining a new data source, 22
dimension properties, accessing, 135

driver manager, 17

## E

error notification, 77
establishing a connection, 25
executing a command to obtain data, 15
executing commands, 42

## G

getData() method, 66
getDBTypeName() method, 65
getField() method, 64, 65, 70
getFieldCount() method, 70
getRow() method, 70
getSchemaObjectColumns() method, 62

## H

hierarchical and multidimensional data, 47

## I

IO access to the file system, 59

## L

listDir()method, 61
listing procedure parameters, 35
log messages, 76

## M

managing multiple registries, 21
message conventions, 76
message file
  creating, 57
  demo_csv_CSV.properties, 57
metadata
  access, 12
  function, 14
  obtaining, 30
multidimensional data, 47
multiple calls, 20
multiple registries, 22

## N

names
  of the capabilities files, 57
  of the properties files, 56
next() method, 70

## O

obtaining data, 34
  getData () method, 34
  three methods, 15
obtaining metadata, 30
obtaining metadata about procedures, 35
open system, 12
open() call, 25
open()method, 59

## P

platforms, for SAP BW, 131
procedure, 14
procedureresult sets, 36
processing
  logon properties, 25
  results using the CallResults interface, 15
  results using the rowset interface, 40
properties files, 56
property descriptions file, 56
property sheet, 13

## R

registering a new data source, 13
registry, 21
registry.properties file, adding a data source to, 132
remote data access, 12
requesting, 38
rethrow convenience methods, 76
retrieving column metadata, 34
retrieving multidimensional data
  using a regular selector, 50
  using MDSelector, 51
return value, 36
Rowset interface, 16

## S

sample code, loading the registry, 21
SAP BW
  adding a data source, 132
  copying files to the /lib directory, 132
  supported platforms, 131

schema, 30
  listing data objects in a schema, 32
  traversing schemas, 31
schemas, 14
SDK, 12
searching the Registry for a data source, 20
selecting and filtering, 42
selector, to pick the desired columns, 42
setSchemas() method, 61
static rethrow() method, 61

## T

transaction interface, 14
transactions, 46

## U

user messages, 76
using getData to obtain data, 15

## W

writing a DDO driver, 55