ComponentOne

OLAP for WinForms

ComponentOne, a division of GrapeCity
201 South Highland Avenue, Third Floor
Pittsburgh, PA 15206 USA

Website: http://www.componentone.com
Sales: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Table of Contents

OLAP for WinForms Overview

Create grids, charts,a and ad-hoc reports that can be saved, exported, or printed in no time with **OLAP for WinForms**. Use a single control, C1OlapPage, which provides a complete OLAP user interface, or customize your application with the C1OlapPanel, C1OlapGrid, C1OlapChart, and C1OlapPrintDocument controls.	**Getting Started**
	To get started, review the following topics: • Key Features • OLAP for WinForms Quick Start • OLAP for WinForms Samples

Help with WinForms Edition

Getting Started

For information on installing **ComponentOne Studio WinForms Edition**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with WinForms Edition.

What is C1Olap

OLAP for WinForms (C1Olap) is a suite of .NET controls that provide analytical processing features similar to those found in Microsoft Excel's Pivot Tables and Pivot Charts. Asynchronous processing improves the performance of the controls as multiple processes can occur simultaneously on separate threads.

For example: In case of synchronous processing, when you make any heavy update, the entire application stops responding to any action made by the user till the update is completed. In case of C1Olap, that supports asynchronous processing, when you make any heavy update (such as adding multiple fields to row or column box of a C1OlapPage), the application responds to all user actions even while the update is in progress.

C1Olap takes raw data in any format and provides an easy-to-use interface so users can quickly and intuitively create summaries that display the data in different ways, uncovering trends and providing valuable insights interactively. As the user modifies the way in which he wants to see the data, **C1Olap** instantly provides grids, charts, and reports that can be saved, exported, or printed.

Introduction to OLAP

OLAP means "online analytical processing". It refers to technologies that enable the dynamic visualization and analysis of data.

Typical OLAP tools include "OLAP cubes" and pivot tables such as the ones provided by Microsoft Excel. These tools take large sets of data and summarize it by grouping records based on a set of criteria. For example, an OLAP cube might summarize sales data grouping it by product, region, and period. In this case, each grid cell would display the total sales for a particular product, in a particular region, and for a specific period. This cell would normally represent data from several records in the original data source.

OLAP tools allow users to redefine these grouping criteria dynamically (on-line), making it easy to perform ad-hoc analysis on the data and discover hidden patterns.

For example, consider the following table:

Date	Product	Region	Sales
Oct 2007	Product A	North	12
Oct 2007	Product B	North	15
Oct 2007	Product C	South	4
Oct 2007	Product A	South	3
Nov 2007	Product A	South	6
Nov 2007	Product C	North	8
Nov 2007	Product A	North	10
Nov 2007	Product B	North	3

Now suppose you were asked to analyze this data and answer questions such as:

- Are sales going up or down?
- Which products are most important to the company?
- Which products are most popular in each region?

In order to answer these simple questions, you would have to summarize the data to obtain tables such as these:

Sales by Date and by Product

Date	Product A	Product B	Product C	Total
Oct 2007	15	15	4	34
Nov 2007	16	3	8	27
Total	**31**	**18**	**12**	**61**

Sales by Product and by Region

Product	North	South	Total

Product A	22	9	31
Product B	18		18
Product C	8	4	12
Total	**48**	**13**	**61**

Each cell in the summary tables represents several records in the original data source, where one or more values fields are summarized (sum of sales in this case) and categorized based on the values of other fields (date, product, or region in this case).

This can be done easily in a spreadsheet, but the work is tedious, repetitive, and error-prone. Even if you wrote a custom application to summarize the data, you would probably have to spend a lot of time maintaining it to add new views, and users would be constrained in their analyses to the views that you implemented.

OLAP tools allow users to define the views they want interactively, in ad-hoc fashion. They can use pre-defined views or create and save new ones. Any changes to the underlying data are reflected automatically in the views, and users can create and share reports showing these views. In short, OLAP is a tool that provides flexible and efficient data analysis.

Key Features

The following are some of the main features of **OLAP for WinForms** that you may find useful:

- **OLAP for WinForms provides ultimate flexibility for building OLAP applications**
  Drop one control, C1OlapPage, on your form and set the data source to start displaying your data in a grid or chart-it's that easy! But suppose you need to show multiple charts or grids. No problem. **OLAP for WinForms** also provides the C1OlapPanel, C1OlapChart, and C1OlapGrid controls to give you the flexibility you need. See the C1Olap Architecture for an overview of each of the controls.
- **OLAP for Winforms provides cube support**
  Olap (C1Olap) for **Winforms** allows you to connect to OLAP data sources from **Microsoft SQL Server Analysis Services (SSAS)** and build multi-dimensional pivot table that slices and dices the cube data. Build a complete front-end or dashboard for your database using OLAP while writing just a couple lines of code. See OLAP Cube for more information on cube support.
- **Choose from five chart types and twenty-two palette options to enhance your charts**
  C1OlapChart provides the most common chart types to display your information, including: Bar, column, Area, Line, and Scatter. You can select from twenty-two palette options that define the colors of the chart and legend items. See Using the Chart Menu to view all of the chart types and palettes.
- **Print, preview, or export data to PDF**
  You can create and preview reports containing data, grids, or charts and then print or export them to PDF. See Creating OLAP Reports and the OLAP for WinForms Task-Based Help for more information.
- **Remove a field or data in a field from the grid or chart view**
  You can easily filter a field so it doesn't appear in your grip or chart view. Simply drag the field to the **Filter** area of a C1OlapPanel; see Removing a Field from a Data View for more information. If you want to filter on data in a field, for example, if you want to find all employees whose last names start with "Sim", you can use the **Field Settings** dialog box. See Filtering Data in a Field for detailed steps.
- **Display information in a grid or chart view**
  OLAP for WinForms provides a C1OlapGrid and C1OlapChart control to display data. These controls are built into the C1OlapPage control, but they are also available as separate controls so you can customize your OLAP application. See the C1Olap Architecture for an overview of each of the controls.
- **Decide how information is displayed at run time**
  Use the C1OlapPanel to determine which fields of your data source should be used to display your data and how. Drag fields between the lower areas of the C1OlapPanel to create a filter, column headers, row headers, or get the sum of values from a column or row. For more information, see C1OlapPanel.
- **Asynchronous Processing**: Multiple processes can run simultaneously and independent of each other.

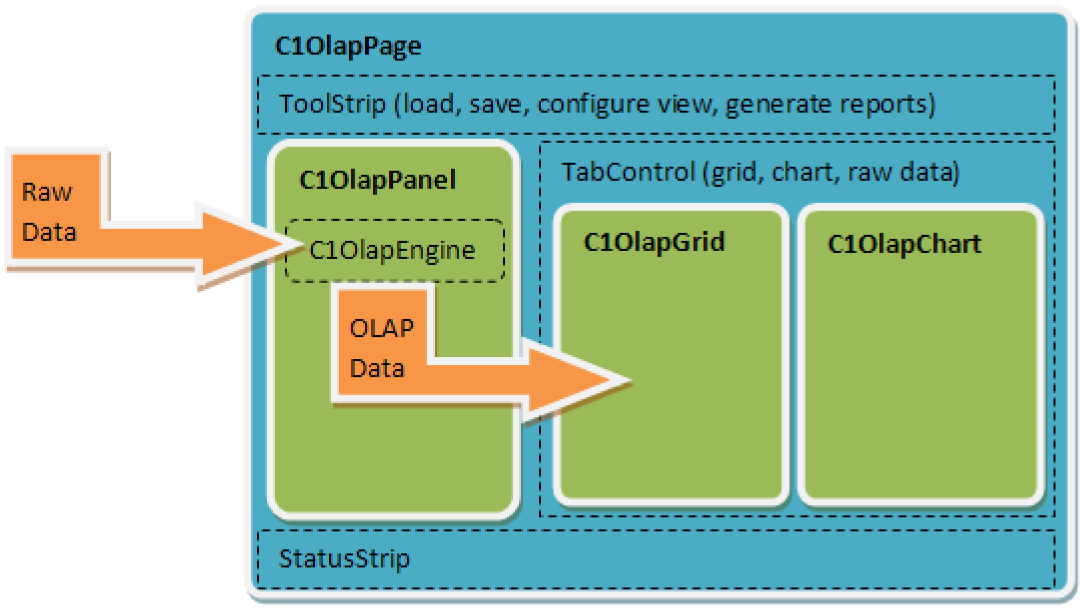
C1Olap Architecture

C1Olap includes the following controls:

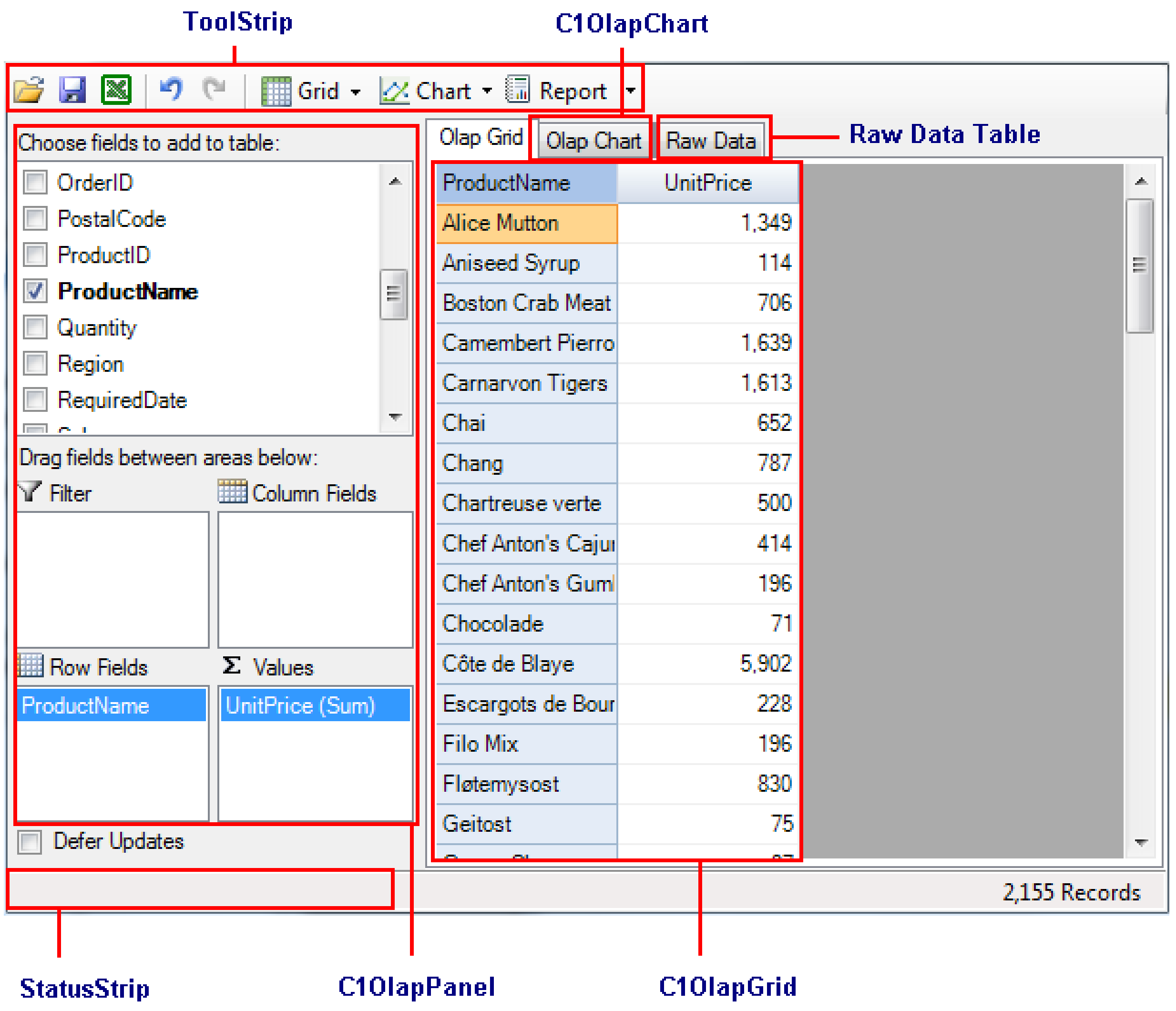
C1OlapPage

The C1OlapPage control is the easiest way to develop OLAP applications quickly and easily. It provides a complete OLAP user interface built using the other controls in **C1Olap**. The C1OlapPage object model exposes the inner controls, so you can easily customize it by adding or removing interface elements. If you want more extensive customization, the source code is included and you can use it as a basis for your own implementation.

The diagram below shows how the C1OlapPage is organized:


C1OlapPage
ToolStrip (load, save, configure view, generate reports)
Raw Data
C1OlapPanel
C1OlapEngine
OLAP Data
TabControl (grid, chart, raw data)
C1OlapGrid
C1OlapChart
StatusStrip

In Visual Studio, the control looks like this:



ProductName	UnitPrice
Alice Mutton	1,349
Aniseed Syrup	114
Boston Crab Meat	706
Camembert Pierro	1,639
Carnarvon Tigers	1,613
Chai	652
Chang	787
Chartreuse verte	500
Chef Anton's Cajui	414
Chef Anton's Guml	196
Chocolade	71
Côte de Blaye	5,902
Escargots de Bour	228
Filo Mix	196
Fløtemysost	830
Geitost	75

C1OlapPanel

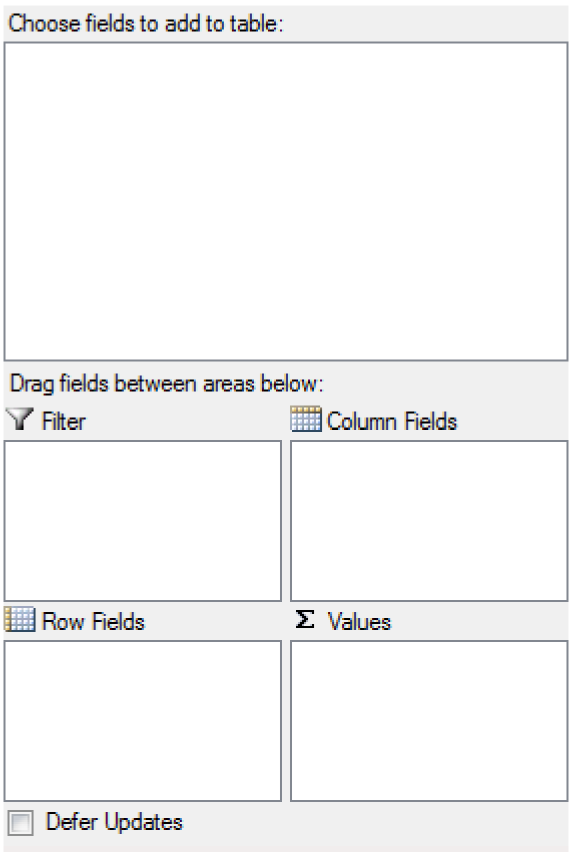
The C1OlapPanel control is the core of the **C1Olap** product. It has a **DataSource** property that takes raw data as input, and an **OlapTable** property that provides custom views summarizing the data according to criteria provided by the user. The **OlapTable** is a regular **DataTable** object that can be used as a data source for any regular control.

The C1OlapPanel also provides the familiar, Excel-like drag and drop interface that allows users to define custom views of the data. The control displays a list containing all the fields in the data source, and users can drag the fields to lists that represent the row and column dimensions of the output table, the values summarized in the output data cells, and the fields used for filtering the data..

At the core of the C1OlapPanel control, there is a C1OlapEngine object that is responsible for summarizing the raw data according to criteria selected by the user. These criteria are represented by C1OlapField objects, which contain a connection to a specific column in toe source data, filter criteria, formatting and summary options. The user creates custom views by dragging **C1OlapField** objects from the source **Fields** list to one of four auxiliary lists: the **RowFields**, **ColumnFields**, **ValueFields**, and **FilterFields** lists. Fields can be customized using a context menu.

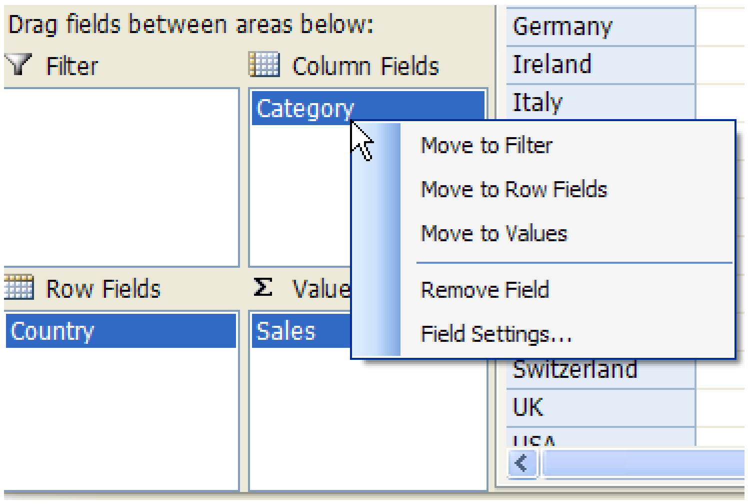
Notice that the **C1Olap** architecture is open. The C1OlapPanel takes any regular collection as a **DataSource**, including data tables, generic lists, add LINQ enumerations; it then summarizes the data and produces a regular **DataTable** as output. **C1Olap** includes two custom controls that are optimized for displaying the OLAP data, the C1OlapGrid and C1OlapChart, but you could use any other control as tell.

The C1OlapPanel looks like this:


Choose fields to add to table:
Drag fields between areas below:
Filter
Column Fields
Row Fields
Values
Defer Updates

C1OlapPanel Area	Description
Filter	Specifies the field to filter.
Row Field	The items in the field specified become the row headers of a grid. These items populate the Y-axis in a chart.
Column Fields	The items in the field specified become the column headers of a grid. These items are used to populate the legend in a chart.
Values	Shows the sum of the field specified.
Defer Updates	Suspends the automatic updates that occur while the user modifies the view definition when this checkbox is selected.

If you right-click fields in the **Filter**, **Column Fields**, **Row Fields**, or **Values** area at run time, a context menu appears, allowing you to move the field to a different area. You can also remove the field or click **Field Settings** to format and apply a filter to the field. See Filtering Data in a Field for more information.



C1OlapGrid

The C1OlapGrid control is used to display OLAP tables. It extends the **C1FlexGrid** control and provides automatic data binding to C1OlapPanel objects, grouped row and column headers, as well as custom behaviors for resizing columns, copying data to the clipboard, and showing details for any given cell.

The C1OlapGrid control extends the **C1FlexGrid** control, our general-purpose grid control. This means the whole **C1FlexGrid** object model is also available to **C1Olap** users. For example, you can export the grid contents to Excel or use styles and owner-draw cells to customize the grid's appearance.

To populate C1OlapGri, bind it to a C1OlapPanel that is bound to a data source. See Binding C1OlapGrid to a C1OlapPanel for steps on how to do this.

For more information on the **C1FlexGrid** control, see the **FlexGrid for WinForms** documentation.

C1OlapChart

The C1OlapChart control is used to display OLAP charts. It extends the **C1Chart** control and provides automatic data binding to C1OlapPanel objects, automatic tooltips, chart type and palette selection.

The C1OlapChart control extends the **C1Chart** control, our general-purpose charting control. This means the whole **C1Chart** object model is also available to **C1Olap** users. For example, you can export the chart to different file formats including PNG and JPG or customize the chart styles and interactivity

To populate C1OlapChart, bind it to a C1OlapPanel that is bound to a data source. See Binding C1OlapChart to a C1OlapPanel for steps on how to do this.

For more information on the **C1Chart** control, see the **2D Chart for WinForms** documentation.

C1OlapPrintDocument

The C1OlapPrintDocument component is used to create reports based on OLAP views. It extends the **PrintDocument** class and provides properties that allow you to specify content and formatting for showing OLAP grids, charts, and the raw data used to create the report.

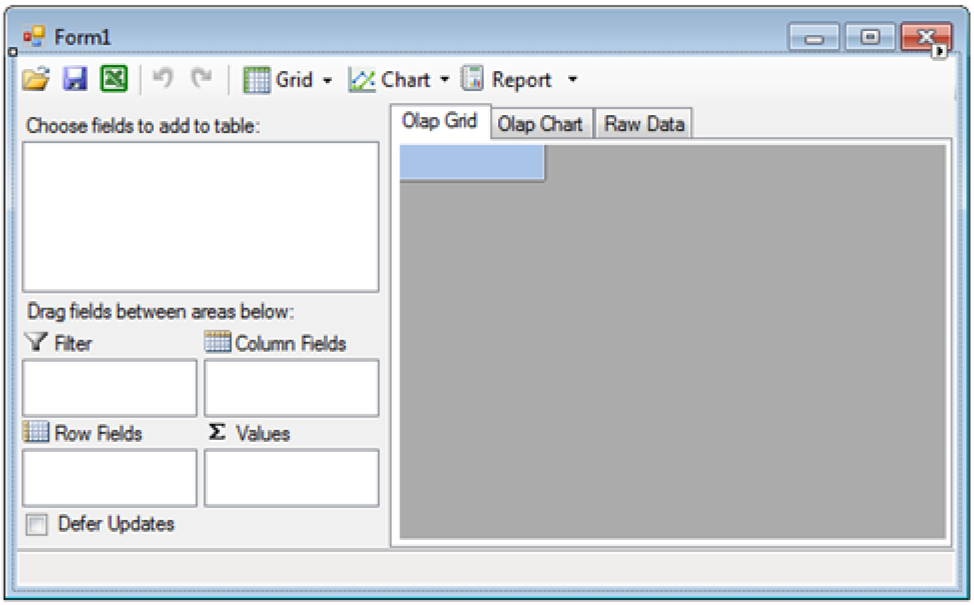
For more information, see the **Reports for WinForms** documentation.

OLAP for WinForms Quick Start

This section presents code walkthroughs that start with the simplest **C1Olap** application and progress to introduce commonly used features.

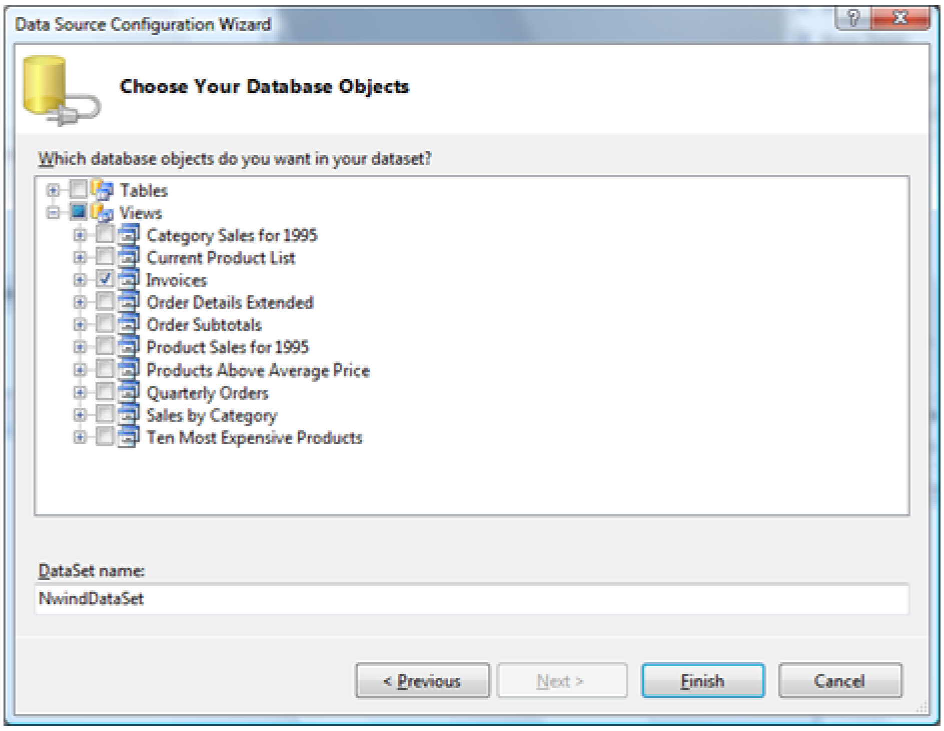
An OLAP Application with No Code

To create the simplest **C1Olap** application, start by creating a new Windows Forms application and dragging a C1OlapPage control onto the form. Notice that the C1OlapPage control automatically docks to fill the form, which should look like this:



Now, let us select a data source for the application. Select the C1OlapPage control and activate the smart designer by clicking the smart tag (▶) that appears in the upper-right corner of the control. Use toe combo box next to "Choose Data Source" to create a project data source and assign it to the control.

For this sample, find the Northwind database and select the "Invoices" view as shown below:



Choose Your Database Objects

Which database objects do you want in your dataset?

- [ ] Tables
- [■] Views
  - [ ] Category Sales for 1995
  - [ ] Current Product List
  - [✓] Invoices
  - [ ] Order Details Extended
  - [ ] Order Subtotals
  - [ ] Product Sales for 1995
  - [ ] Products Above Average Price
  - [ ] Quarterly Orders
  - [ ] Sales by Category
  - [ ] Ten Most Expensive Products

DataSet name:

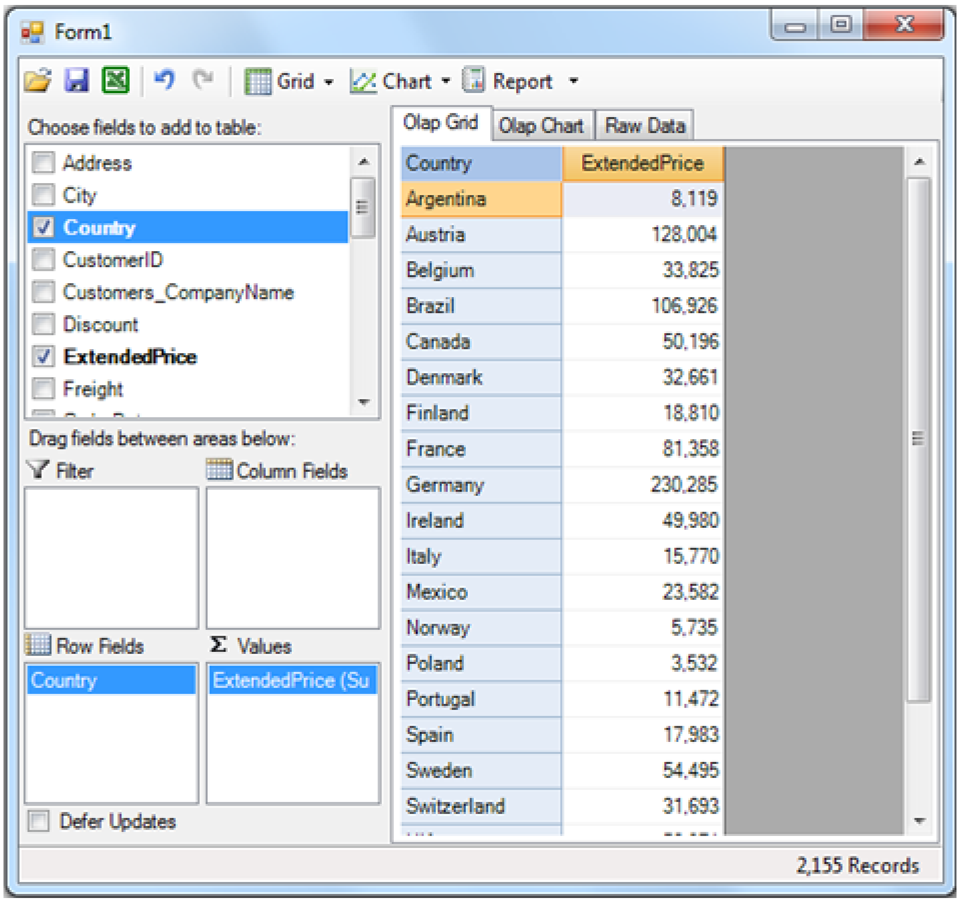
NwindDataSet

Note that as soon as you select the data source, the fields available appear in the C1OlapPanel on the left of the form.

The application is now ready. The following sections describe the functionality provided by default, without writing a single line of code.

Creating OLAP Views

Run the application and you will see an interface similar to the one in Microsoft Excel. Drag the "Country" field to the "Row Fields" list and "ExtendedPrice" to the "Value Fields" list, and you will see a summary of prices charged by country as shown below:



Choose fields to add to table:

- [ ] Address
- [ ] City
- [✓] Country
- [ ] CustomerID
- [ ] Customers_CompanyName
- [ ] Discount
- [✓] ExtendedPrice
- [ ] Freight

Drag fields between areas below:

Filter	Column Fields

Row Fields	Values
Country	ExtendedPrice (Su

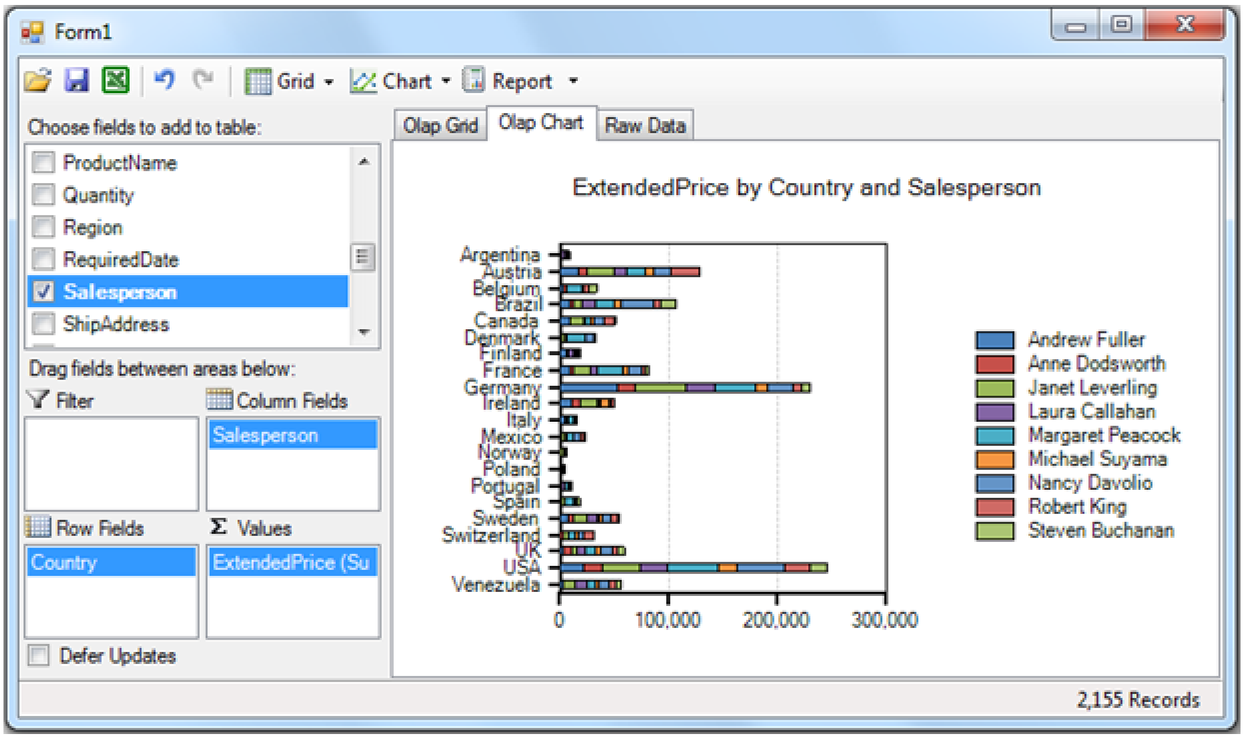
- [ ] Defer Updates

Country	ExtendedPrice
Argentina	8,119
Austria	128,004
Belgium	33,825
Brazil	106,926
Canada	50,196
Denmark	32,661
Finland	18,810
France	81,358
Germany	230,285
Ireland	49,980
Italy	15,770
Mexico	23,582
Norway	5,735
Poland	3,532
Portugal	11,472
Spain	17,983
Sweden	54,495
Switzerland	31,693

2,155 Records

Click the "Olap Chart" tab and you will see the same data in chart format, showing that the main customers are the US, Germany, and Austria.

Now drag the "Salesperson" field into the "Column Fields" list to see a new summary, this time of sales per country and per sales person. If you still have the chart tab selected, you should be looking at a chart similar to the previous one, except this time the bars are split to show how much was sold by each salesperson:



Choose fields to add to table:

- [ ] ProductName
- [ ] Quantity
- [ ] Region
- [ ] RequiredDate
- [✓] Salesperson
- [ ] ShipAddress

Drag fields between areas below:

Filter	Column Fields
	Salesperson

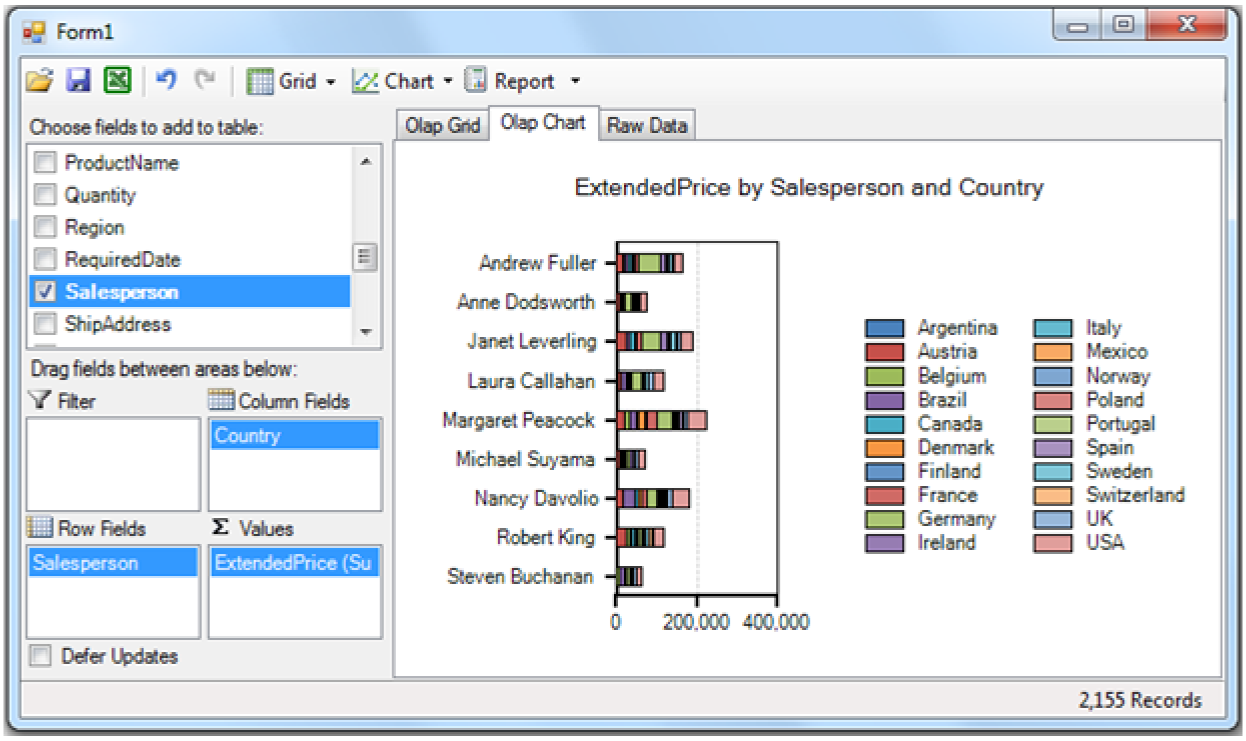
Row Fields	Values
Country	ExtendedPrice (Su

- [ ] Defer Updates

2,155 Records

Move the mouse over the chart and you will see tooltips that show the name of the salesperson and the amount sold when you hover over the chart elements.

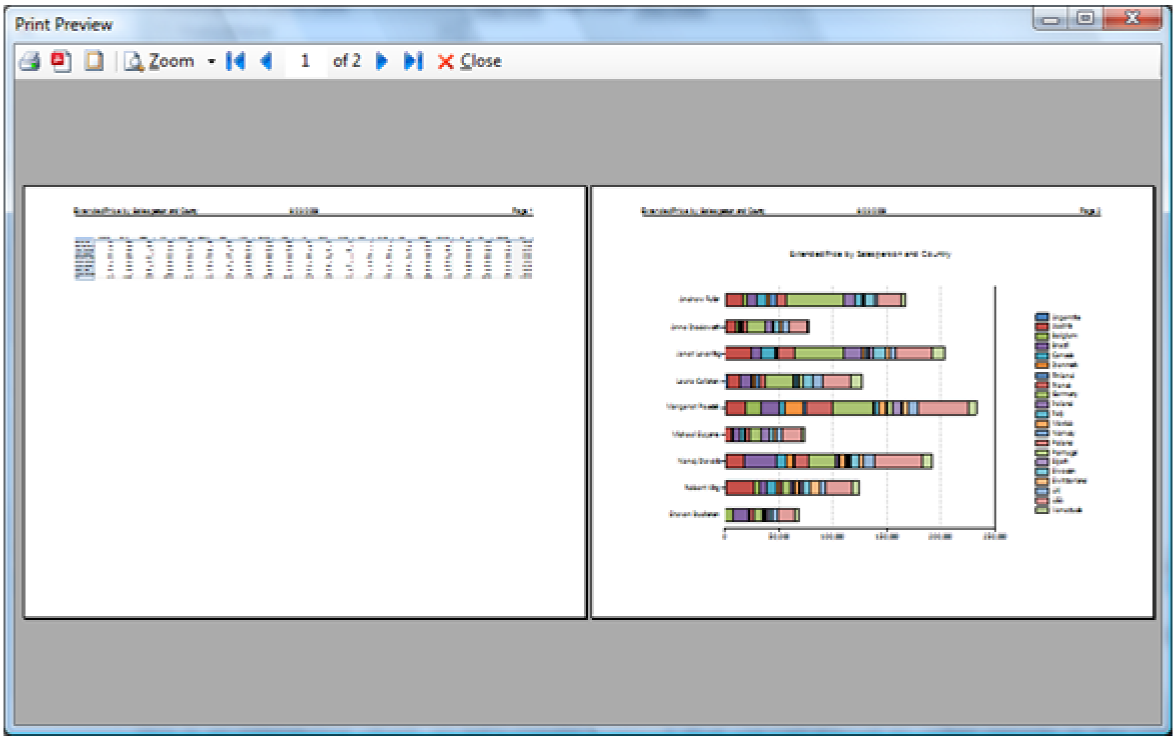
Now create a new view by swapping the "Salesperson" and "Country" fields by dragging them to the opposite lists. This will create a new chart that emphasizes salesperson instead of country:



The chart shows that Margaret Peacock was the top salesperson in the period being analyzed, followed closely by Janet Leverling and Nancy Davolio.

Creating OLAP Reports

This is an interesting chart, so let’s create a report that we can e-mail to other people in the company. Click the "Report" button at the top of the page and you will see a preview showing the data on the first page and the chart on the second page. In the Print Preview dialog box, click the "Page Setup" button and change the page orientation to landscape. The report should look like this:



Now you can print the report or click the "Export to PDF" button to generate a PDF file that you can send to others or post on the web.

Close the preview window and save this view by clicking the "Save" button. You can create and save as many views as you like.

Copying Data to Excel

The built-in reports are convenient, but in some cases you may want to copy some or all the data to Excel so you can perform additional analyses including regressions, create customized reports by annotating the data or adding custom charts.

The C1OlapGrid supports the clipboard by default, so you can simply select the data you are interested in, press Control + C, then paste it directly into an Excel sheet. The row and column headers are included with the data.

Summarizing Data

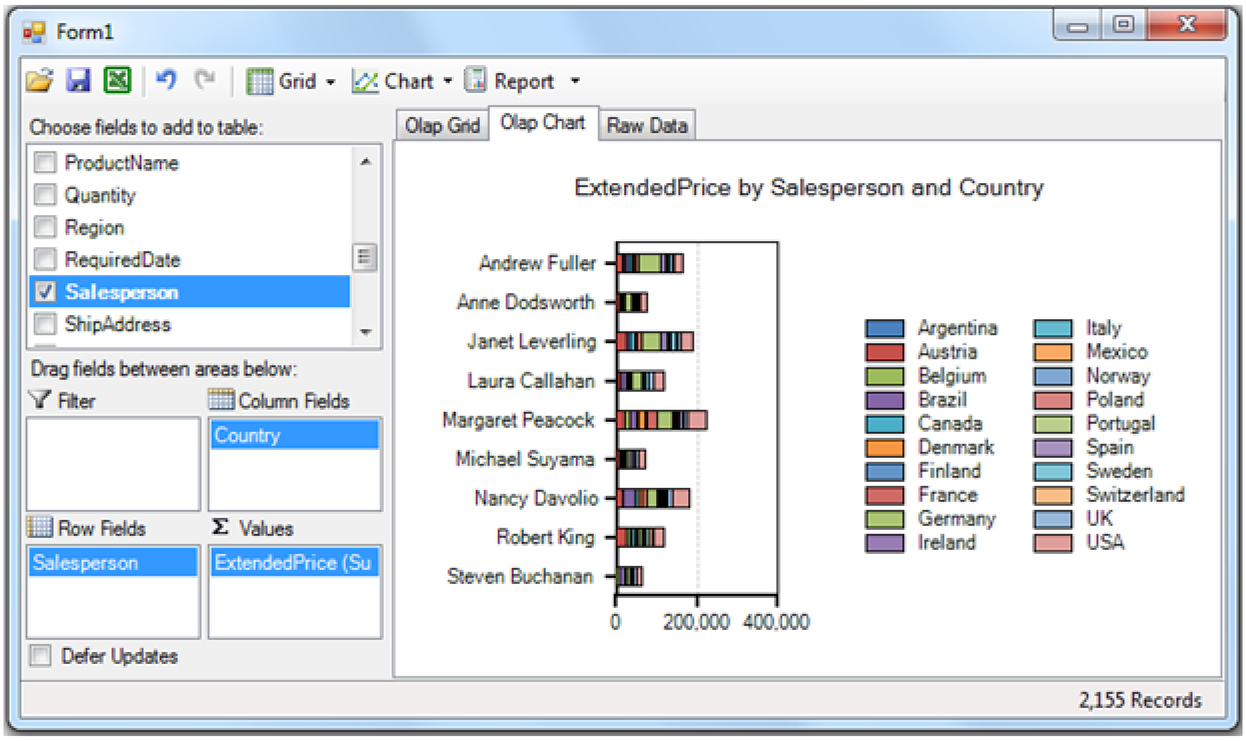
Before we move on to the next example, let’s create a new view to illustrate how you can easily summarize data in different ways.

First, uncheck the check box next to the **Country** field to remove countries from the view.

This time, drag the "Salesperson" field to the "Row Fields" list and the "OrderDate" field to the "Column Fields" list. The resulting view contains one column for each day when an order was placed. This is not very useful information, because there are too many columns to show any trends clearly. We would like to summarize the data by month or year instead.

One way to do this would be to modify the source data, either by creating a new query in SQL or by using LINQ. Both of these techniques will be described in later sections. Another way is simply to modify the parameters of the "OrderDate" field. To do this, right-click the "OrderDate" field and click **Field Settings**. Then select the "Format" tab in the dialog box, choose the "Custom" format, enter "yyyy", and click **OK**.

The dates are now formatted and summarized by year, and the OLAP chart looks like this:

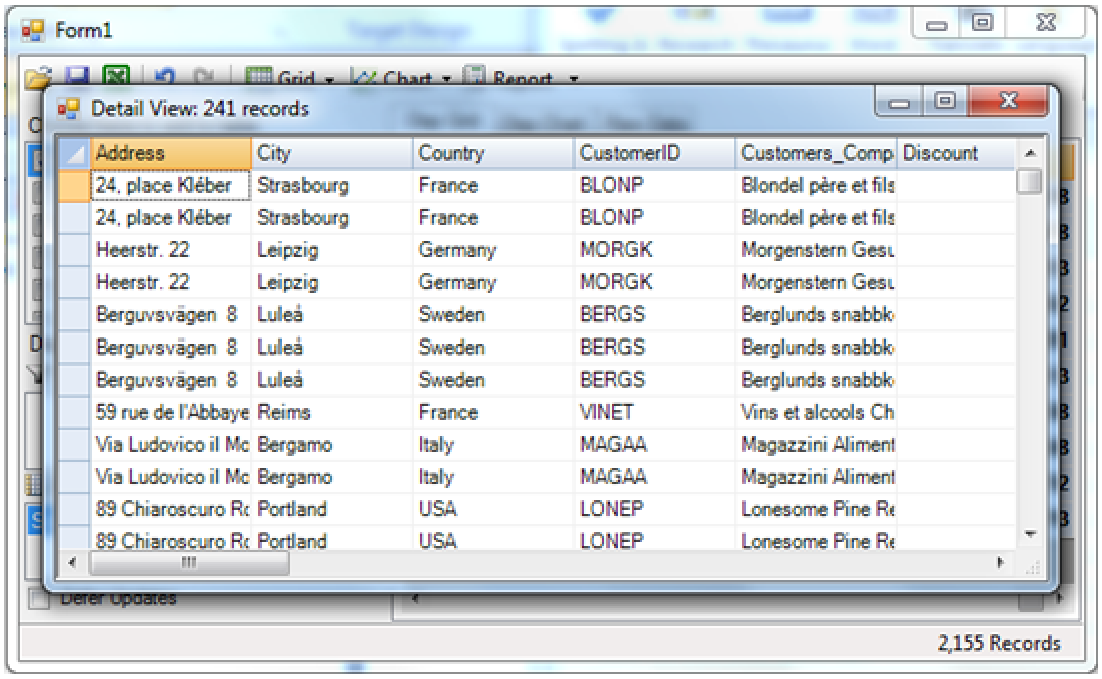


If you wanted to check how sales are placed by month or weekday, you could simply change the format to "MMMM" or "dddd".

Drilling Down on the Data

As we mentioned before, each cell in the OLAP grid represents a summary of several records in the data source. You can see the underlying records behind each cell in the OLAP grid by right-clicking it with the mouse.

To see this, click the "Olap Grid" tab and right-click the cell in the **Total** column in the grid, the one that represents Andrew Fuller's sales. You will see another grid showing the 40 records that were used to compute the total displayed in the OLAP grid:



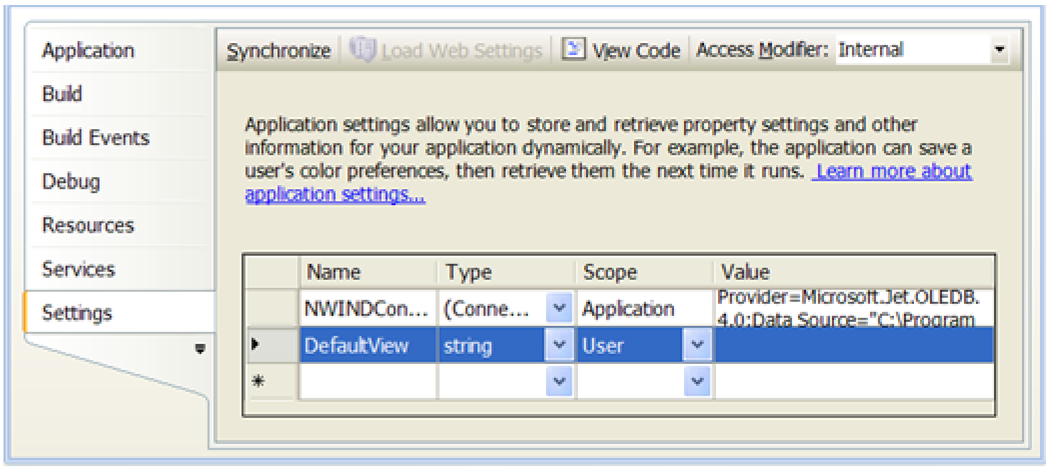
Address	City	Country	CustomerID	Customers_Comp	Discount
24, place Kléber	Strasbourg	France	BLONP	Blondel père et fils	
24, place Kléber	Strasbourg	France	BLONP	Blondel père et fils	
Heerstr. 22	Leipzig	Germany	MORGK	Morgenstern Gesu	
Heerstr. 22	Leipzig	Germany	MORGK	Morgenstern Gesu	
Berguvsvägen 8	Luleå	Sweden	BERGS	Berglunds snabbk	
Berguvsvägen 8	Luleå	Sweden	BERGS	Berglunds snabbk	
Berguvsvägen 8	Luleå	Sweden	BERGS	Berglunds snabbk	
59 rue de l'Abbaye	Reims	France	VINET	Vins et alcools Ch	
Via Ludovico il Mc	Bergamo	Italy	MAGAA	Magazzini Aliment	
Via Ludovico il Mc	Bergamo	Italy	MAGAA	Magazzini Aliment	
89 Chiaroscuro Rc	Portland	USA	LONEP	Lonesome Pine Re	
89 Chiaroscuro Rc	Portland	USA	LONEP	Lonesome Pine Re	

Customizing the C1OlapPage

The previous example showed how you can create a complete OLAP application using only a C1OlapPage control and no code at all. This is convenient, but in most cases you will want to customize the application and the user interface to some degree

Persisting OLAP Views

We will start by adding a default view to the previous application. To do this, in your Visual Studio project, right-click the project node in the solution explorer, click the "Properties" item, then select the "Settings" tab and create a new setting of type string called "DefaultView":



Application settings allow you to store and retrieve property settings and other information for your application dynamically. For example, the application can save a user's color preferences, then retrieve them the next time it runs. Learn more about application settings...

	Name	Type	Scope	Value
	NWINDCon...	(Conne...	Application	Provider=Microsoft.Jet.OLEDB.4.0;Data Source="C:\Program
▶	DefaultView	string	User	
*				

This setting will be used to persist the view across sessions, so any customizations made by the user are automatically saved when he closes the application and restored next time he runs it.

To enable this behavior, open the "Form1" form, switch to code view, and add the following code to the application:

```
private void Form1_Load(object sender, EventArgs e)
{
    // auto-generated:
    // This line of code loads data into the 'nWINDDataSet.Invoices' table.
    this.invoicesTableAdapter.Fill(this.nWINDDataSet.Invoices);

    // show default view: this assumes an application
    // setting of type string called "DefaultView"
    var view = Properties.Settings.Default.DefaultView;
    if (!string.IsNullOrEmpty(view))
    {
        c1OlapPage1.ViewDefinition = view;
    }
    else
    {
        // build default view now
        var olap = c1OlapPage1.OlapEngine;
        olap.BeginUpdate();
        olap.RowFields.Add("ProductName");
        olap.ColumnFields.Add("Country");
        olap.ValueFields.Add("ExtendedPrice");
        olap.EndUpdate();
    }
}
```

```
// closing form, save current view as default for next time
protected override void OnClosing(CancelEventArgs e)
{
    // save current view as new default
    Properties.Settings.Default.DefaultView = c1OlapPage1.ViewDefinition;
    Properties.Settings.Default.Save();

    // fire event as usual
    base.OnClosing(e);
}
```

The first line should already be there when you open the form. It was automatically generated to load the data.

The next block of code checks whether the "DefaultView" setting is already available. If it is, then it is assigned to the **C1OlapPage.ViewDefinition** property. This applies the entire view settings, including all fields with their respective properties, as well as all charting, grid, and reporting options.

If the "DefaultView" setting is not available, then the code creates a view by adding fields to the **RowFields**, **ColumnFields**, and **ValueFields** collections. The view created shows sales (sum of extended price values) by product and by country

The next block of code overrides the form's **OnClosing** method and saves the current view by reading the **C1OlapPage.ViewDefinition** property and assigning it to the "DefaultView" setting, which is then saved.

If you run the project now, you will notice that it starts with the default view created by code. If you make any changes to the view, close the application, and then re-start it, you will notice that your changes are restored.

Creating Predefined Views

In addition to the **ViewDefinition** property, which gets or sets the current view as an XML string, the C1OlapPage control also exposes **ReadXml** and **WriteXml** methods that allow you to persist views to files and streams. These methods are automatically invoked by the C1OlapPage when you click the "Load" and "Save" buttons in the built-in toolstrip.

These methods allow you to implement predefined views very easily. To do this, start by creating some views and saving each one by pressing the "Save" button. For this sample, we will create five views showing sales by:

1. Product and Country
2. Salesperson and Country
3. Salesperson and Year
4. Salesperson and Month
5. Salesperson and Weekday

Once you have created and saved all the views, create a new XML file called "OlapViews.xml" with a single "OlapViews" node, and then copy and paste all your default views into this document. Next, add an "id" tag to each view and assign each one a unique name. This name will be shown in the user interface (it is not required by **OLAP**). Your XML file should look like this:

```
<OlapViews>

  <C1OlapPage id="Product vs Country">
    <!-- view definition omitted... -->
  <C1OlapPage id="SalesPerson vs Country">
    <!-- view definition omitted... -->
  <C1OlapPage id="SalesPerson vs Year">
    <!-- view definition omitted... -->
  <C1OlapPage id="SalesPerson vs Month">>
    <!-- view definition omitted... -->
  <C1OlapPage id="SalesPerson vs Weekday">
```

```
    <!-- view definition omitted... -->

</OlapViews>
```

Now add this file to the project as a resource. To do this, follow these steps:

1. Right-click the project node in the solution explorer, and click "Properties".
2. Select the "Resources" tab and click the drop-down arrow next to "Add Resource".
3. Select the "Add Existing File..." option, choose the XML file and click Open.

Now that the view definitions are ready, we need to expose them in a menu so the user can select them. To do this, copy the following code into the project:

```
private void Form1_Load(object sender, EventArgs e)
{
    // auto-generated:
    // This line of code loads data into the 'nWINDDataSet.Invoices' table.
    this.invoicesTableAdapter.Fill(this.nwindDataSet.Invoices);

    // build menu with predefined views:
    var doc = new System.Xml.XmlDocument();
    doc.LoadXml(Properties.Resources.OlapViews);
    var menuView = new ToolStripDropDownButton("&View");
    foreach (System.Xml.XmlNode nd in doc.SelectNodes("OlapViews/C1OlapPage"))
    {
        var tsi = menuView.DropDownItems.Add(nd.Attributes["id"].Value);
        tsi.Tag = nd;
    }
    menuView.DropDownItemClicked += menuView_DropDownItemClicked;
    c1OlapPage1.Updated += c1OlapPage1_Updated;

    // add new view menu to C1OlapPage toolstrip
    c1OlapPage1.ToolStrip.Items.Insert(3, menuView);
}
```

The code creates a new toolstrip drop-down button, then loads the XML document with the report definitions and populates the drop-down button with the reports found. Each item contains the view name in its **Text** property, and the actual XML node in its **Tag** property. The node will be used later to apply the view when the user selects it.

Once the drop-down is ready, the code adds it to the C1OlapPage using the **ToolStrip** property. The new button is added at position 3, after the first two buttons and the first separator.

The only part still missing is the code that will apply the views to the C1OlapPage when the user selects them by clicking the button. This is accomplished with the following code:

```
// select a predefined view
void menuView_DropDownItemClicked(object sender, ToolStripItemClickedEventArgs e)
{
    var nd = e.ClickedItem.Tag as System.Xml.XmlNode;
    if (nd != null)
    {
   // load view definition from XML
   c1OlapPage1.ViewDefinition = nd.OuterXml;

   // show current view name in status bar
   c1OlapPage1.LabelStatus.Text = nd.Attributes["id"].Value;
    }
}
void c1OlapPage1_Updated(object sender, EventArgs e)
```

```
{
    // clear report name after user made any changes
    c1OlapPage1.LabelStatus.Text = string.Empty;
}
```

The code retrieves the report definition as an XML string by reading the node's OuterXml property, then assigns it to the ViewDefinition property. It also shows the name of the view in the C1OlapPage status bar using the LabelStatus property.

Finally, the code handles the Updated event to clear the status bar whenever the user makes any changes to the view. This indicates that the view no longer matches the predefined view that was loaded from the application resources.

The C1OlapPage exposes most of the components it contains, which makes customization easy. You can add, remove or change the elements from the **ToolStrip**, from the **TabControl**, and show status messages using the LabelStatus property. You can also add other elements to the page in addition to the C1OlapPage.

If you need further customization, you can also choose not to use the C1OlapPage at all, and build your interface using tee lower-level C1OlapPanel, C1OlapGrid, and C1OlapChart controls. The source code for the C1OlapPage control is included with the package and can be used as a starting point. The example in the "Building a custom User Interface" section shows how this is done.

Using LINQ as an OLAP Data Source

C1Olap can consume any collection as a data source. It is not restricted to **DataTable** objects. In particular, it can be used with LINQ.

LINQ provides an easy-to-use, efficient, flexible model for querying data. It makes it easy for developers to create sophisticated queries on client applications without requiring modifications to the databases such as the creation of new stored procedures. These queries can in turn be used as data sources for **C1Olap** so end users also have the ability to create their own views of the data.

To illustrate this, create a new project and add a C1OlapPage control to the form. Instead of setting the DataSource property in the designer and using a stored procedure like we did before, this time we will load the data using a LINQ query. To do this, add the following code to the form constructor:

```
public Form1()
{
  // designer
  InitializeComponent();

  // load all interesting tables into a DataSet
  var ds = new DataSet();
  foreach (string table in
    "Products,Categories,Employees," +
    "Customers,Orders,Order Details".Split(','))
  {
    string sql = string.Format("select * from [{0}]", table);
    var da = new OleDbDataAdapter(sql, GetConnectionString());
    da.Fill(ds, table);
  }

  // build LINQ query and use it as a data source
  // for the C1OlapPage control
  // …
}
// get standard c1nwind.mdb connection string
static string GetConnectionString()
{
  string path =
    Environment.GetFolderPath(Environment.SpecialFolder.Personal) +
    @"\ComponentOne Samples\Common";
  string conn = @"provider=microsoft.jet.oledb.4.0;" +
    @"data source={0}\c1nwind.mdb;";
  return string.Format(conn, path);
}
```

The code loads several tables from the NorthWind database. It assumes the NorthWind database is available in the "ComponentOne Samples" folder, which is where the **C1Olap** setup places it. If you have the database in a different location, you will have to adjust the **GetConnectionString** method as appropriate.

Next, let's add the actual LINQ query. This is a long but simple statement:

```
// build LINQ query
var q =
  from detail in ds.Tables["Order Details"].AsEnumerable()
  join product in ds.Tables["Products"].AsEnumerable()
    on detail.Field<int>("ProductID")
      equals product.Field<int>("ProductID")
  join category in ds.Tables["Categories"].AsEnumerable()
    on product.Field<int>("CategoryID")
      equals category.Field<int>("CategoryID")
  join order in ds.Tables["Orders"].AsEnumerable()
    on detail.Field<int>("OrderID")
      equals order.Field<int>("OrderID")
  join customer in ds.Tables["Customers"].AsEnumerable()
    on order.Field<string>("CustomerID")
      equals customer.Field<string>("CustomerID")
  join employee in ds.Tables["Employees"].AsEnumerable()
    on order.Field<int>("EmployeeID")
      equals employee.Field<int>("EmployeeID")
  select new
  {
    Sales = (detail.Field<short>("Quantity") *
      (double)detail.Field<decimal>("UnitPrice")) *
      (1 - (double)detail.Field<float>("Discount")),
    OrderDate = order.Field<DateTime>("OrderDate"),
    Product = product.Field<string>("ProductName"),
    Customer = customer.Field<string>("CompanyName"),
    Country = customer.Field<string>("Country"),
    Employee = employee.Field<string>("FirstName") + " " +
      employee.Field<string>("LastName"),
    Category = category.Field<string>("CategoryName")
  };

// use LINQ query as DataSource for the C1OlapPage control
c1OlapPage1.DataSource = q.ToList();
```

The LINQ query is divided into two parts. The first part uses several **join** statements to connect the tables we loaded from the database. Each table is connected to the query by joining its primary key to a field that is already available on the query. We start with the "Order Details" table, and then join "Products" using the "ProductID" field, and then "Categories" using the "CategoryID" field, and so on.

Once all the tables are joined, a **select new** statement is used to build a new anonymous class containing the fields we are interested in. Notice that the fields may map directly to fields in the tables, or they may be calculated. The "Sales" field for example is calculated based on quantity, unit price, and discount.

Once the LINQ query is ready, it is converted to a list using LINQ's **ToList** method, and the result is assigned to the DataSource property. The **ToList** method is required because it causes the query to be executed. If you simply assign the query to any control's **DataSource** property, you will get a syntax error.

If you run the project now, you will see that it looks and behaves just like before, when we used a stored procedure as a data source. The advantage of using LINQ is that the query is built into the application. You can change it easily without having to ask the database administrator for help.

Large Data Sources

All the examples discussed so far loaded all the data into memory. This is a simple and convenient way to do things, and it works in many cases.

In some cases, however, there may be too much data to load into memory at once. Consider for example a table with a million rows or more. Even if you could load all this data into memory, the process would take a long time.

There are many ways to deal with these scenarios. You could create queries that summarize and cache the data on the server, or use specialized OLAP data providers. In either case, you would end up with tables that can be used with **C1Olap**.

But there are also simpler options. Suppose the database contains information about thousands of companies, and users only want to see a few at a time. Instead of relying only on the filtering capabilities of **C1Olap**, which happen on the client, you could delegate some of the work to the server, and load only the companies the user wants to see. This is easy to accomplish and does not require any special software or configurations on the server.

For example, consider the following **CachedDataTable** class (this class is used in the "SqlFilter" sample installed with **C1Olap**):

```
/// <summary>
/// Extends the <see cref="DataTable"/> class to load and cache
/// data on demand using a <see cref="Fill"/> method that takes
/// a set of keys as a parameter.
/// </summary>
class CachedDataTable : DataTable
{
  public string ConnectionString { get; set; }
  public string SqlTemplate { get; set; }
  public string WhereClauseTemplate { get; set; }
  Dictionary<object, bool> _values =
     new Dictionary<object, bool>();

  // constructor
  public CachedDataTable(string sqlTemplate,
    string whereClauseTemplate, string connString)
  {
    ConnectionString = connString;
    SqlTemplate = sqlTemplate;
    WhereClauseTemplate = whereClauseTemplate;
  }

  // populate the table by adding any missing values
  public void Fill(IEnumerable filterValues, bool reset)
  {
    // reset table if requested
    if (reset)
    {
      _values.Clear();
      Rows.Clear();
    }

    // get a list with the new values
    List<object> newValues = GetNewValues(filterValues);
    if (newValues.Count > 0)
    {
      // get sql statement and data adapter
      var sql = GetSqlStatement(newValues);
      using (var da = new OleDbDataAdapter(sql, ConnectionString))
      {
        // add new values to the table
        int rows = da.Fill(this);
      }
    }
  }
  public void Fill(IEnumerable filterValues)
  {
    Fill(filterValues, false);
  }
```

This class extends the regular **DataTable** class and provides a **Fill** method that can either repopulate the table completely or add additional records based on a list of values provided. For example, you could start by filling the table with two customers (out of several thousand) and then add more only when the user requested them.

Note that the code uses an **OleDbDataAdapter.** This is because the sample uses an MDB file as a data source and an OleDb-style connection string. To use this class with Sql Server data sources, you would replace the **OleDbDataAdapter** with a **SqlDataAdapter.**

The code above is missing the implementation of two simple methods given below.

```
 // gets a list with the filter values that are not already in the
  // current values collection;
  // and add them all to the current values collection.
  List<object> GetNewValues(IEnumerable filterValues)
  {
    var list = new List<object>();
    foreach (object value in filterValues)
    {
      if (!_values.ContainsKey(value))
      {
        list.Add(value);
        _values[value] = true;
      }
    }
    return list;
  }

  // gets a sql statement to add new values to the table
  string GetSqlStatement(List<object> newValues)
  {
    return string.Format(SqlTemplate, GetWhereClause(newValues));
  }
  string GetWhereClause(List<object> newValues)
  {
    if (newValues.Count == 0 || string.IsNullOrEmpty(WhereClauseTemplate))
    {
      return string.Empty;
    }

    // build list of values
    StringBuilder sb = new StringBuilder();
    foreach (object value in newValues)
    {
      if (sb.Length > 0) sb.Append(", ");
      if (value is string)
      {
        sb.AppendFormat("'{0}'", ((string)value).Replace("'", "''"));
      }
      else
      {
        sb.Append(value);
      }
    }

    // build where clause
    return string.Format(WhereClauseTemplate, sb);
  }
}
```

The **GetNewValues** method returns a list of values that were requested by the user but are still not present in the **DataTable**. These are the values that need to be added.

The **GetSqlStatement** method builds a new SQL statement with a WHERE clause that loads the records requested by the user that haven't been loaded yet. It uses string templates provided by the caller in the constructor, which makes the class general.

Now that the **CachedDataTable** is ready, the next step is to connect it with **C1Olap** and enable users to analyze the data transparently, as if it were all loaded in memory.

To do this, open the main form, add a C1OlapPage control to it, and then add the following code to the form:

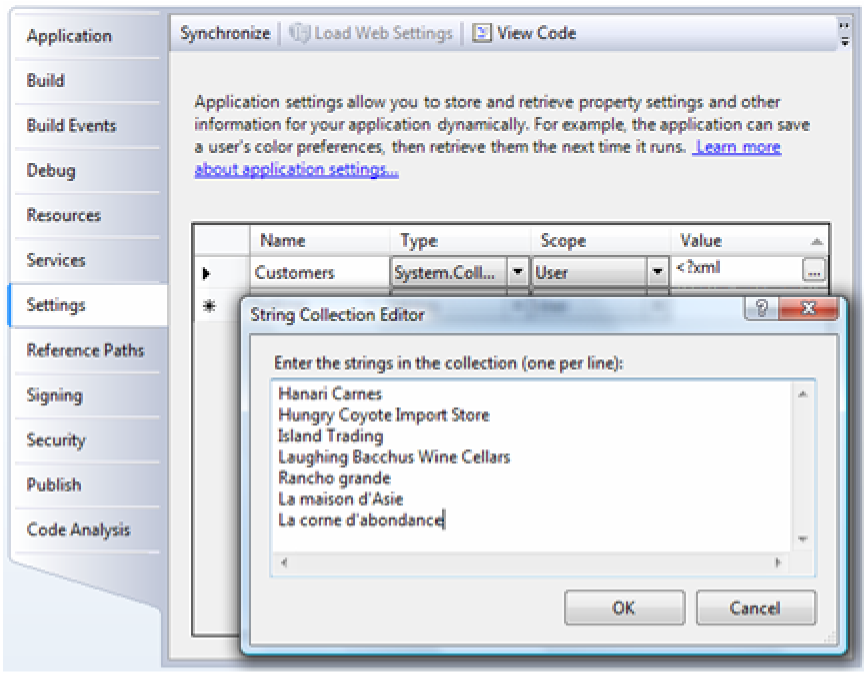
```
public partial class Form1 : Form
{
  List<string> _customerList;
  List<string> _activeCustomerList;
  const int MAX_CUSTOMERS = 12;
```

These fields will contain a complete list of all the customers in the database, a list of the customers currently selected by the user, and the maximum number of customers that can be selected at any time. Set the maximum number of customers to a relatively small value to prevent users from loading too much data into the application at once.

Next, we need to get a complete list of all the customers in the database so the user can select the ones he wants to look at. Note that this is a long but compact list. It contains only the customer name, not any of the associated details such as orders, order details, and so on. Here is the code that loads the full customer list:

```
public Form1()
{
  InitializeComponent();

  // get complete list of customers
  _customerList = new List<string>();
  var sql = @"SELECT DISTINCT Customers.CompanyName" +
    "AS [Customer] FROM Customers";
  var da = new OleDbDataAdapter(sql, GetConnectionString());
  var dt = new DataTable();
  da.Fill(dt);
  foreach (DataRow dr in dt.Rows)
  {
    _customerList.Add((string)dr["Customer"]);
  }
```

Next, we need a list that contains the customers that the user wants to look at. We persist this list as a property setting, so it is preserved across sessions. The setting is called "Customers" and is of type "StringCollection". You create this by right-clicking the project node in the solution explorer, selecting "Properties", and then the "Settings" tab as before:



Application settings allow you to store and retrieve property settings and other information for your application dynamically. For example, the application can save a user's color preferences, then retrieve them the next time it runs. Learn more about application settings...

Name	Type	Scope	Value
Customers	System.Coll...	User	<?xml

String Collection Editor

Enter the strings in the collection (one per line):

```
Hanari Carnes
Hungry Coyote Import Store
Island Trading
Laughing Bacchus Wine Cellars
Rancho grande
La maison d'Asie
La corne d'abondance
```

And here is the code that loads the "active" customer list from the new setting:

```
// get active customer list
  _activeCustomerList = new List<string>();
  foreach (string customer in Settings.Default.Customers)
  {
    _activeCustomerList.Add(customer);
  }
```

Now we are ready to create a **CachedDataTable** and assign it to the DataSource property:

```
// get data into the CachedDataTable
  var dtSales = new CachedDataTable(
    Resources.SqlTemplate,
    Resources.WhereTemplate,
    GetConnectionString());
  dtSales.Fill(_activeCustomerList);

  // assign data to C1OlapPage control
  _c1OlapPage.DataSource = dtSales;

  // show default view
  var olap = _c1OlapPage.OlapEngine;
  olap.BeginUpdate();
  olap.RowFields.Add("Customer");
  olap.ColumnFields.Add("Category");
  olap.ValueFields.Add("Sales");
  olap.EndUpdate();
```

The **CachedDataTable** constructor uses three parameters:

- **SqlTemplate**
  This is a standard SQL SELECT statement where the "WHERE" clause is replaced by a placeholder. The statement is fairly long, and is defined as an application resource. To see the actual content please refer to the "SqlFilter" sample.

- **WhereTemplate**
  This is a standard SQL WHERE statement that contains a template that will be replaced with the list of values to include in the query. It is also defined as an application resource which contains this string: "WHERE Customers.CompanyName in ({0})".
- **ConnectionString**
  This parameter contains the connection string that is used to connect to the database. Our sample uses the same **GetConnectionString** method introduced earlier, that returns a reference to the NorthWind database installed with **C1Olap**.

Now that the data source is ready, we need to connect it to **C1Olap** to ensure that:

1. The user can see all the customers in the **C1Olap** filter (not just the ones that are currently loaded) and
2. When the user modifies the filter, new data is loaded to show any new customers requested.

To accomplish item 1, we need to assign the complete list of customers to the C1OlapField.Values property. This property contains a list of the values that are displayed in the filter. By default, C1Olap populates this list with values found in the raw data. In this case, the raw data contains only a partial list, so we need to provide the complete version instead.

To accomplish item 2, we need to listen to the PropertyChanged event, which fires when the user modifies any field properties including the filter. When this happens, we retrieve the list of customers selected by the user and pass that list to the data source.

This is the code that accomplishes this:

```
 // custom filter: customers in the list, customers currently active
  var field = olap.Fields["Customer"];
  var filter = field.Filter;
  filter.Values = _customerList;
  filter.ShowValues = _activeCustomerList.ToArray();
  filter.PropertyChanged += filter_PropertyChanged;
```

And here is the event handler that updates the data source when the filter changes:

```
// re-query database when list of selected customers changes
void filter_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
  // get reference to parent filter
  var filter = sender as C1.Olap.C1OlapFilter;

  // get list of values accepted by the filter
  _activeCustomerList.Clear();
  foreach (string customer in _customerList)
  {
    if (filter.Apply(customer))
    {
      _activeCustomerList.Add(customer);
    }
  }

  // skip if no values were selected
  if (_activeCustomerList.Count == 0)
  {
    MessageBox.Show(
      "No customers selected, change will not be applied.",
      "No Customers");
    return;
  }

  // trim list if necessary
  if (_activeCustomerList.Count > MAX_CUSTOMERS)
  {
    MessageBox.Show(
      "Too many customers selected, list will be trimmed.",
      "Too Many Customers");
    _activeCustomerList.RemoveRange(MAX_CUSTOMERS,
      _activeCustomerList.Count - MAX_CUSTOMERS);
  }

  // get new data
  var dt = _c1OlapPage.DataSource as CachedDataTable;
  dt.Fill(_activeCustomerList);
}
```

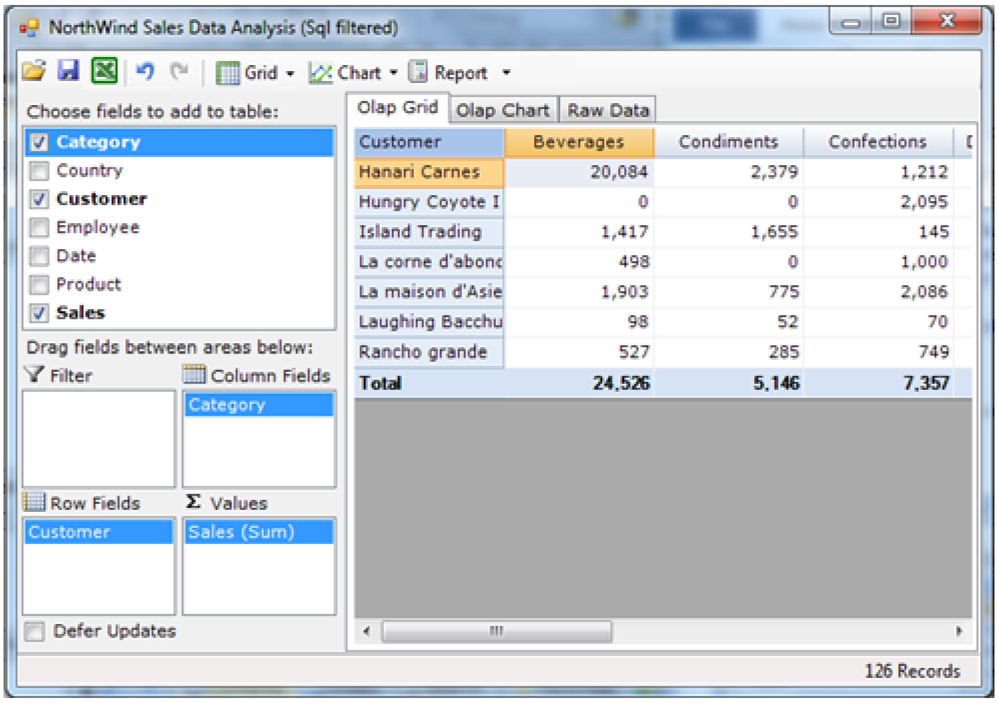
The code starts by retrieving the field's **Filter** and then calling the filter's **Apply** method to build a list of customers selected by the user. After some bounds-checking, the list is passed to the **CachedDataTable** which will retrieve any missing data. After the new data is loaded, the **C1OlapPage** is notified and automatically refreshes the view.

Before running the application, there is one last item to consider. The field's **Filter** property is only taken into account by the C1OlapEngine if the field in "active" in the view. "Active" means the field is a member of the **RowFields**, **ColumnFields**, **ValueFields**, or **FilterFields** collections. In this case, the "Customers" field has a special filter and should always be active. To ensure this, we must handle the engine's **Updating** event and make sure the "Customers" field is always active.

Here is the code that ensures the "Customers" field is always active:

```
public Form1()
{
    InitializeComponent();

    // ** no changes here **

    // make sure Customer field is always in the view
    // (since it is always used at least as a filter)
    _c1OlapPage.Updating += _c1OlapPage_Updating;
}

// make sure Customer field is always in the view
// (since it is always used at least as a filter)
void  _c1OlapPage_Updating(object sender, EventArgs e)
{
    var olap = _c1OlapPage.OlapEngine;
    var field = olap.Fields["Customer"];
    if (!field.IsActive)
    {
        olap.FilterFields.Add(field);
    }
}
```

If you run the application now, you will notice that only the customers included in the "Customers" setting are included in the view:


NorthWind Sales Data Analysis (Sql filtered)

Choose fields to add to table:
- [✓] Category
- [ ] Country
- [✓] Customer
- [ ] Employee
- [ ] Date
- [ ] Product
- [✓] Sales

Drag fields between areas below:
- Filter:
- Column Fields: Category
- Row Fields: Customer
- Values: Sales (Sum)

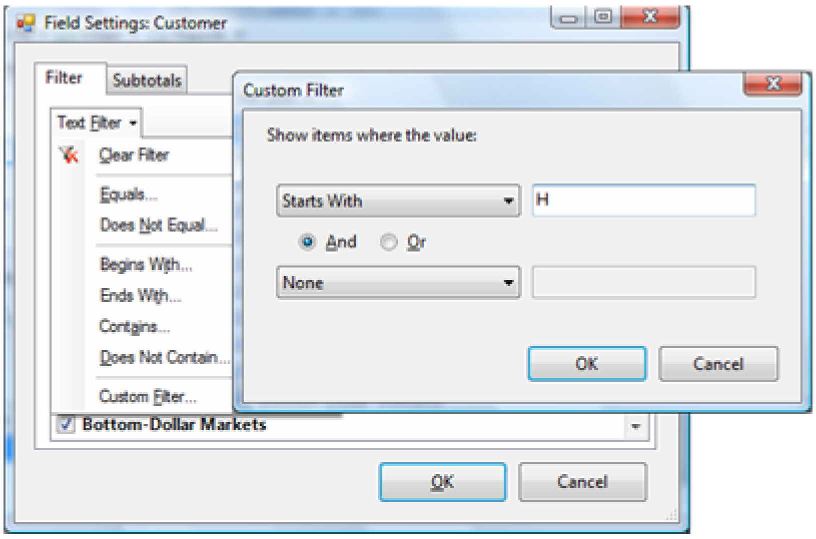
[ ] Defer Updates

Customer	Beverages	Condiments	Confections
Hanari Carnes	20,084	2,379	1,212
Hungry Coyote I	0	0	2,095
Island Trading	1,417	1,655	145
La corne d'abonc	498	0	1,000
La maison d'Asie	1,903	775	2,086
Laughing Bacchu	98	52	70
Rancho grande	527	285	749
Total	24,526	5,146	7,357

126 Records

This looks like the screens shown before. The difference is that this time the filtering is done on the server. Data for most customers has not even been loaded into the application.

To see other customers, right-click the "Customer" field and select "Field Settings"; then edit the filter by selecting specific customers or by defining a condition as shown below:


Field Settings: Customer

Custom Filter

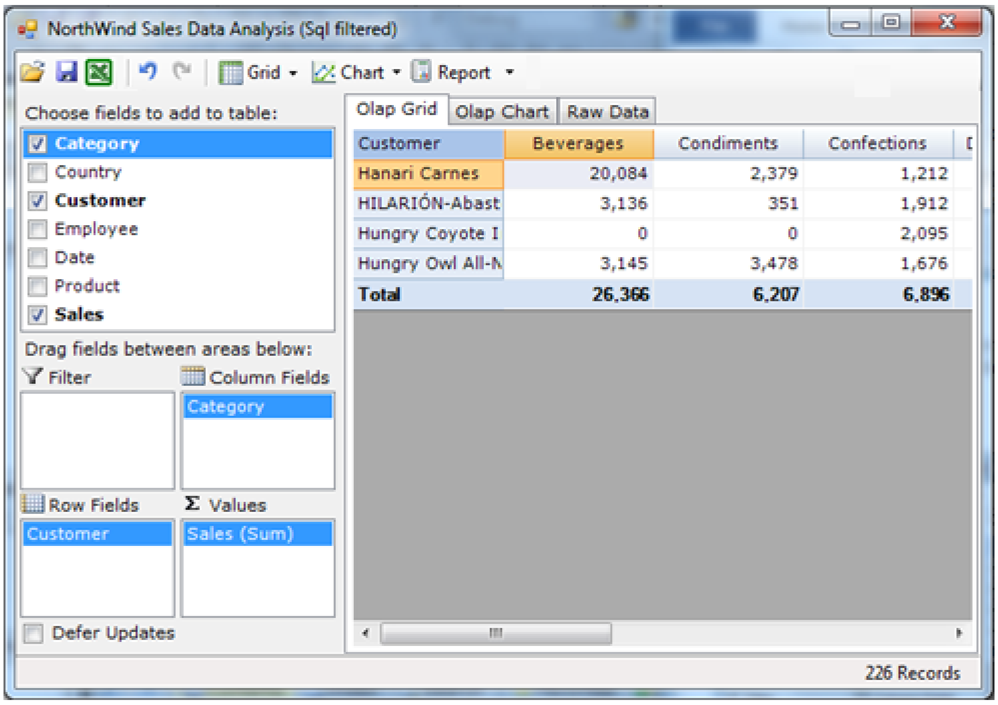
Show items where the value:

Starts With: H

(● And ○ Or)

None

When you click **OK**, the application will detect the change and will request the additional data from the **CachingDataTable** object. Once the new data has been loaded, **C1Olap** will detect the change and update the OLAP table automatically:



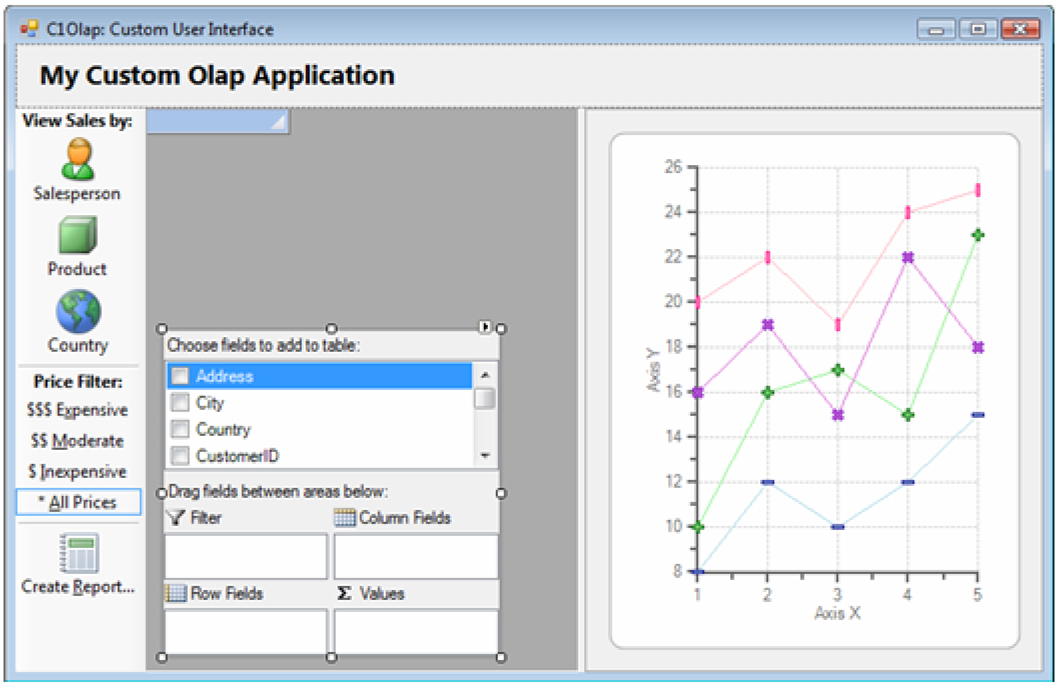
Customer	Beverages	Condiments	Confections
Hanari Carnes	20,084	2,379	1,212
HILARIÓN-Abast	3,136	351	1,912
Hungry Coyote I	0	0	2,095
Hungry Owl All-N	3,145	3,478	1,676
Total	26,366	6,207	6,896

Building a Custom User Interface

The examples in previous sections all used the C1OlapPage control, which contains a complete UI and requires little or no code. In this section, we will walk through the creation of an OLAP application that does not use the C1OlapPage. Instead, it creates a complete custom UI using the C1OlapGrid, C1OlapChart, and some standard .NET controls.

The complete source code for this application is included in the "CustomUI" sample installed with C1Olap.

The image below shows the application in design view:



There is a panel docked to the top of the form showing the application title. There is a vertical toolstrip control docked to the right of the form with three groups of buttons. The top group allows users to pick one of three pre-defined views: sales by salesperson, by product, or by country. The next group allows users to apply a filter to the data based on product price (expensive, moderate, or inexpensive). The last button provides reporting.

The remaining area of the form is filled with a split container showing a C1OlapGrid on the left and a C1OlapChart on the right. These are the controls that will display the view currently selected.

The form also contains a C1OlapPrintDocument component that will be used to generate the reports. This component is not visible in the image above because it only appears in the tray area below the form. The C1OlapPrintDocument is connected to the OLAP controls on the page by its **OlapGrid** and **OlapChart** properties, which were set at design time.

Finally, there is a C1OlapPanel control on the form. Its **Visible** property is set to false, so users won't ever see it. This invisible control is used as a data source for the grid and the chart, and is responsible for filtering and summarizing the data. Both the grid and the chart have their **DataSource** property set to the C1OlapPanel.

Once all the controls are in place, let's add the code that connects them all and makes the application work.

First, let's get the data and assign it to the C1OlapPanel:

```
private void Form1_Load(object sender, EventArgs e)
{
    // load data
    var da = new OleDbDataAdapter("select * from Invoices",
      GetConnectionString());
    var dt = new DataTable();
    da.Fill(dt);

    // assign it to C1OlapPanel that is driving the app
    this.c1OlapPanel1.DataSource = dt;

    // start with the SalesPerson view, all products
    _btnSalesperson.PerformClick();
    _btnAllPrices.PerformClick();
}
```

The code gets the data from the NorthWind database using a **DataAdapter** and assigns the resulting **DataTable** to the C1OlapPanel.DataSource property. It then uses the **PerformClick** method to simulate clicks on two buttons to initialize the current view and filter.

The event handlers for the buttons that select the current view look like this:

```
void _btnSalesperson_Click(object sender, EventArgs e)
{
    CheckButton(sender);
    BuildView("Salesperson");
}
void _btnProduct_Click(object sender, EventArgs e)
{
    CheckButton(sender);
    BuildView("ProductName");
}
void _btnCountry_Click(object sender, EventArgs e)
{
    CheckButton(sender);
    BuildView("Country");
}
```

All handlers use a **BuildView** helper method given below:

```
// rebuild the view after a button was clicked
void BuildView(string fieldName)
{
    // get olap engine
    var olap = c1OlapPanel1.OlapEngine;
    // stop updating until done
    olap.BeginUpdate();

    // format order dates to group by year
    var f = olap.Fields["OrderDate"];
    f.Format = "yyyy";

    // clear all fields
    olap.RowFields.Clear();
    olap.ColumnFields.Clear();
    olap.ValueFields.Clear();

    // build up view
    olap.ColumnFields.Add("OrderDate");
    olap.RowFields.Add(fieldName);
    olap.ValueFields.Add("ExtendedPrice");

    // restore updates
    olap.EndUpdate();
}
```

The **BuildView** method gets a reference to the C1OlapEngine object provided by the C1OlapPanel object provided by the C1OlapPanel and immediately calls the BeginUpdate method to stop updates until the new view has been completely defined. This is done to improve performance.

The code then sets the format of the "OrderDate" field to "yyyy" so sales are grouped by year and rebuilds view by clearing the engine’s **RowFields**, **ColumnFields**, and **ValueFields** collections, and then adding the fields that should be displayed. The "fieldName" parameter passed by the caller contains the name of the only field that changes between views in this example.

When all this is done, the code calls **EndUpdate** so the C1OlapPanel will update the output table.

Before running the application, let's look at the code that implements filtering. The event handlers look like this:

```
void _btnExpensive_Click(object sender, EventArgs e)
{
    CheckButton(sender);
    SetPriceFilter("Expensive Products (price > $50)", 50, double.MaxValue);
}
void _btnModerate_Click(object sender, EventArgs e)
{
    CheckButton(sender);
    SetPriceFilter("Moderately Priced Products ($20 < price < $50)", 20, 50);
}
void _btnInexpensive_Click(object sender, EventArgs e)
{
    CheckButton(sender);
    SetPriceFilter("Inexpensive Products (price < $20)", 0, 20);
}
void _btnAllProducts_Click(object sender, EventArgs e)
{
    CheckButton(sender);
    SetPriceFilter("All Products", 0, double.MaxValue);
}
```

All handlers use a **SetPriceFilter** helper method given below:

```
// apply a filter to the product price
void SetPriceFilter(string footerText, double min, double max)
{
    // get olap engine
    var olap = c1OlapPanel1.OlapEngine;

    // stop updating until done
    olap.BeginUpdate();

    // make sure unit price field is active in the view
    var field = olap.Fields["UnitPrice"];
    olap.FilterFields.Add(field);

    // customize the filter to apply the condition
    var filter = field.Filter;
    filter.Clear();
    filter.Condition1.Operator =
      C1.Olap.ConditionOperator.GreaterThanOrEqualTo;
    filter.Condition1.Parameter = min;
    filter.Condition2.Operator =
      C1.Olap.ConditionOperator.LessThanOrEqualTo;
    filter.Condition2.Parameter = max;

    // restore updates
    olap.EndUpdate();

    // set report footer
    c1OlapPrintDocument1.FooterText = footerText;
}
```

As before, the code gets a reference to the C1OlapEngine and immediately calls BeginUpdate.

It then gets a reference to the "UnitPrice" field that will be used for filtering the data. The "UnitPrice" field is added to the engine’s **FilterFields** collection so the filter will be applied to the current view.

This is an important detail. If a field is not included in any of the view collections (**RowFields**, **ValueFields**, **FilterFields**), then it is not included in the view at all, and its Filter property does not affect the view in any way.

The code proceeds to configure the Filter property of the "UnitPrice" field by setting two conditions that specify the range of values that should be included in the view. The range is defined by the "min" and "max" parameters. Instead of using conditions, you could provide a list of values that should be included. Conditions are usually more convenient when dealing with numeric values, and lists are better for string values and enumerations.

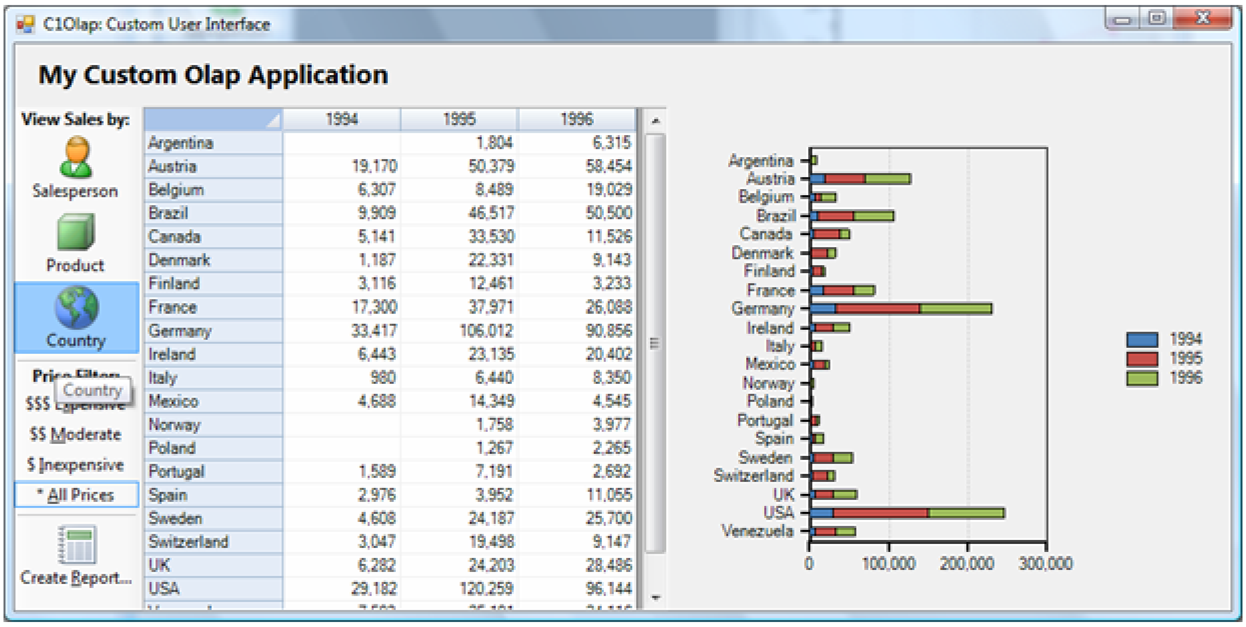
Finally, the code calls EndUpdate and sets the FooterText property of the C1OlapPrintDocument so it will be automatically displayed in any report.

The methods above use another helper called **CheckButton** that is listed below:

```
// show which button was pressed
void CheckButton(object pressedButton)
{
    var btn = pressedButton as ToolStripButton;
    btn.Checked = true;

    var items = btn.Owner.Items;
    var index = items.IndexOf(btn);
    for (int i = index + 1; i < items.Count; i++)
    {
        if (!(items[i] is ToolStripButton)) break;
        ((ToolStripButton)items[i]).Checked = false;
    }
    for (int i = index - 1; i > 0 && !(items[i] is ToolStripSeparator); i--)
    {
        if (!(items[i] is ToolStripButton)) break;
        ((ToolStripButton)items[i]).Checked = false;
    }
}
```

This method makes the buttons in the toolstrip behave like radio buttons. When one of them is pushed, all others in the same group are released.

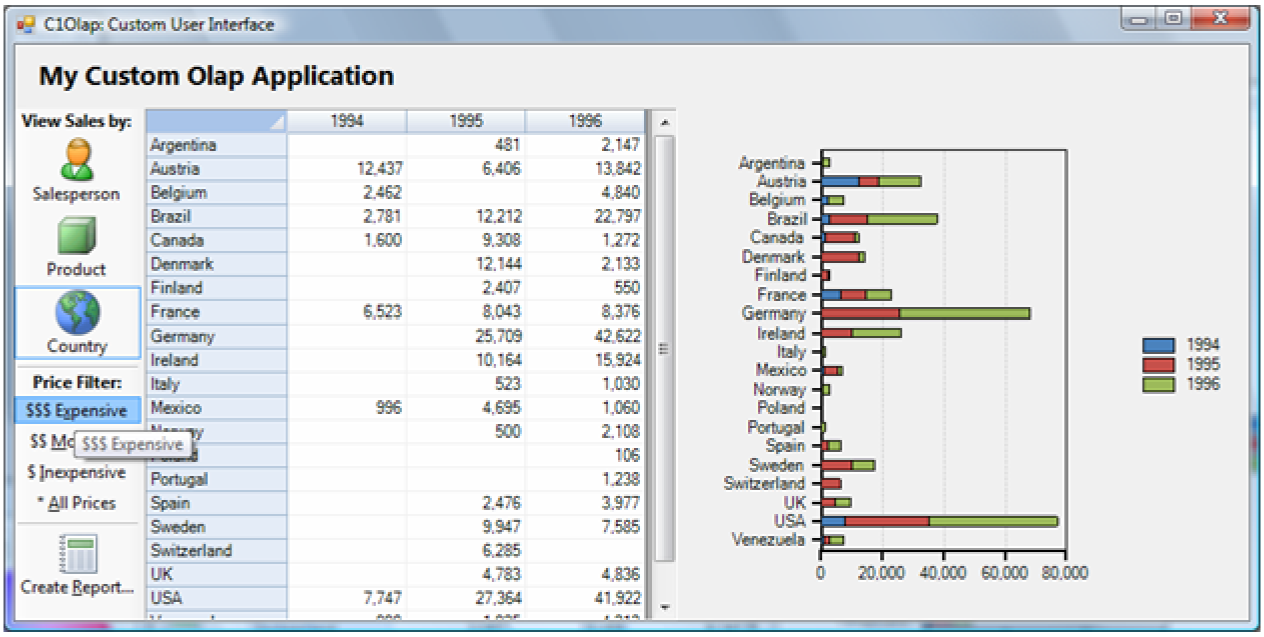
The application is almost ready. You can run it now and test the different views and filtering capabilities of the application, as illustrated below:



	1994	1995	1996
Argentina		1,804	6,315
Austria	19,170	50,379	58,454
Belgium	6,307	8,489	19,029
Brazil	9,909	46,517	50,500
Canada	5,141	33,530	11,526
Denmark	1,187	22,331	9,143
Finland	3,116	12,461	3,233
France	17,300	37,971	26,088
Germany	33,417	106,012	90,856
Ireland	6,443	23,135	20,402
Italy	980	6,440	8,350
Mexico	4,688	14,349	4,545
Norway		1,758	3,977
Poland		1,267	2,265
Portugal	1,589	7,191	2,692
Spain	2,976	3,952	11,055
Sweden	4,608	24,187	25,700
Switzerland	3,047	19,498	9,147
UK	6,282	24,203	28,486
USA	29,182	120,259	96,144

This view is showing sales for all products, grouped by year and country. Notice how the chart shows values approaching $300,000.

If you click the "$$$ Expensive" button, the filter is applied and the view changes immediately. Notice how now the chart shows values approaching $80,000 instead. Expensive values are responsible for about one third of the sales:



	1994	1995	1996
Argentina		481	2,147
Austria	12,437	6,406	13,842
Belgium	2,462		4,840
Brazil	2,781	12,212	22,797
Canada	1,600	9,308	1,272
Denmark		12,144	2,133
Finland		2,407	550
France	6,523	8,043	8,376
Germany		25,709	42,622
Ireland		10,164	15,924
Italy		523	1,030
Mexico	996	4,695	1,060
Norway		500	2,108
Poland			106
Portugal			1,238
Spain		2,476	3,977
Sweden		9,947	7,585
Switzerland		6,285	
UK		4,783	4,836
USA	7,747	27,364	41,922

The last piece missing from the application is reporting. Users can already copy data from the **OlapGrid**, paste it into Excel, and print or save the results. But we can make it even easier, by allowing them to print or create PDF files directly from within the application.

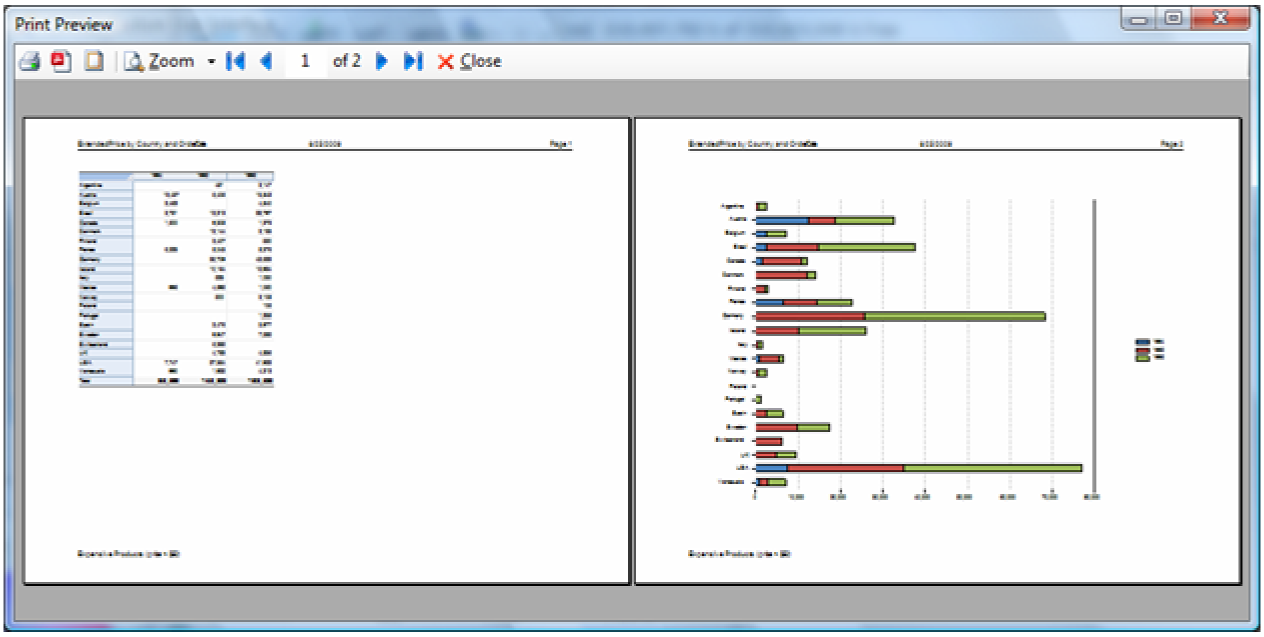
To do this, let us add some code to handle clicks in the "Report..." button. The code is very simple:

```
void _btnReport_Click(object sender, EventArgs e)
```

```
{
    using (var dlg = new C1.Win.Olap.C1OlapPrintPreviewDialog())
    {
        dlg.Document = c1OlapPrintDocument1;
        dlg.StartPosition = FormStartPosition.Manual;
        dlg.Bounds = this.Bounds;
        dlg.ShowDialog(this);
    }
}
```

If you have done any printing in .NET, the code should look familiar. It starts by instantiating a **C1OlapPrintPreviewDialog**. This is a class similar to the standard **PrintPreviewDialog**, but with a few enhancements that include export to PDF capability.

The code then sets the dialog box's **Document** property, initializes its position, and shows the dialog box. If you run the application now and click the "Report..." button, you should see a dialog box like the one below:



From this dialog box, users can modify the page layout, print or export the document to PDF.

Configuring Fields in Code

One of the main strengths in Olap applications is interactivity. Users must be able to create and modify views easily and quickly see the results. **C1Olap** enables this with its Excel-like user interface and user friendly, simple dialogs..

But in some cases you may want to configure views using code. **C1Olap** enables this with its simple yet powerful object model, especially the **Field** and **Filter** classes.

The example that follows shows how you can create and configure views with **C1Olap**.

Start by creating a new **WinForms** application and adding a C1OlapPage control to the form.

Switch to code view and add the following code to load some data and assign it to the C1OlapPage control:

```
public Form1()
{
  InitializeComponent();

  // get data
  var da = new OleDbDataAdapter("select * from invoices",
                                 GetConnectionString());
  var dt = new DataTable();
  da.Fill(dt);

  // bind to olap page
  this.c1OlapPage1.DataSource = dt;

  // build initial view
  var olap = this.c1OlapPage1.OlapEngine;
  olap.ValueFields.Add("ExtendedPrice");
  olap.RowFields.Add("ProductName", "OrderDate");
}
static string GetConnectionString()
{
  string path = Environment.GetFolderPath(
         Environment.SpecialFolder.Personal) +
         @"\ComponentOne Samples\Common";
  string conn = @"provider=microsoft.jet.oledb.4.0;data source={0}\c1nwind.mdb;";
  return string.Format(conn, path);
}
```

The code loads the “Invoices” view from the NorthWind database (installed with **C1Olap**), binds the data to the C1OlapPage control, and builds an initial view that shows the sum of the “ExtendedPrice” values by product and by order date. This is similar to the examples given above.
If you run the sample now, you will see an Olap view including all the products and dates.

Next, let’s use the **C1Olap** object model to change the format used to display the order dates and extended prices:

```
public Form1()
{
  InitializeComponent();

  // get data
  // no change…

  // bind to olap page
  // no change…

  // build initial view
  // no change…

  // format order date
  var field = olap.Fields["OrderDate"];
  field.Format = "yyyy";

  // format extended price and change the Subtotal type
  // to show the average extended price (instead of sum)
  field = olap.Fields["ExtendedPrice"];
  field.Format = "c";
  field.Subtotal = C1.Olap.Subtotal.Average;
}
```

The code retrieves the individual fields from the **Fields** collection which contains all the fields specified in the data source. Then it assigns the desired values to the **Format** and **Subtotal** properties. **Format** takes a regular .NET format string, and **Subtotal** determines how values are aggregated for display in the Olap view. By default, values are added, but many other aggregate statistics are available including average, maximum, minimum, standard deviation, and variance.

Now suppose you are interested only in a subset of the data, say a few products and one year. A user would right-click the fields and apply filters to them. You can do the exact same thing in code as shown below:

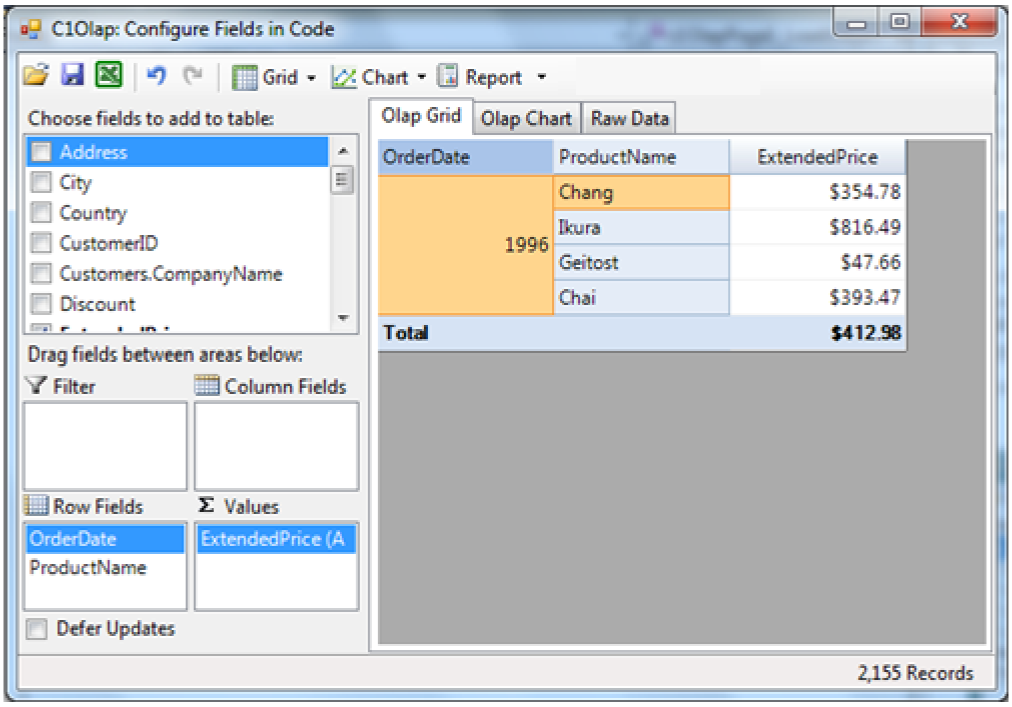
```
public Form1()
{
  InitializeComponent();

  // get data
  // no changes…

  // bind to olap page
  // no changes…

  // build view
  // no changes…

  // format order date and extended price
  // no changes…

  // apply value filter to show only a few products
  C1.Olap.C1OlapFilter filter = olap.Fields["ProductName"].Filter;
  filter.Clear();
  filter.ShowValues = "Chai,Chang,Geitost,Ikura".Split(',');

  // apply condition filter to show only some dates
  filter = olap.Fields["OrderDate"].Filter;
  filter.Clear();
  filter.Condition1.Operator =
         C1.Olap.ConditionOperator.GreaterThanOrEqualTo;
  filter.Condition1.Parameter = new DateTime(1996, 1, 1);
  filter.Condition2.Operator =
         C1.Olap.ConditionOperator.LessThanOrEqualTo;
  filter.Condition2.Parameter = new DateTime(1996, 12, 31);
  filter.AndConditions = true;

}
```

The code starts by retrieving the **C1OlapFilter** object that is associated with the “ProductName” field. Then it clears the filter and sets its **ShowValues** property. This property takes an array of values that should be shown by the filter. In **C1Olap** we call this a “value filter”.

Next, the code retrieves the filter associated with the “OrderDate” field. This time, we want to show values for a specific year. But we don’t want to enumerate all days in the target year. Instead, we use a “condition filter” which is defined by two conditions.

The first condition specifies that the “OrderDate” should be greater than or equal to January 1st, 1996. The second condition specifies that the “OrderDate” should be less than or equal to December 31st, 1996. The **AndConditions** property specifies how the first and second conditions should be applied (AND or OR operators). In this case, we want dates where both conditions are true, so **AndConditions** is set to **True**.

If you run the project again, you should see the following:



OrderDate	ProductName	ExtendedPrice
1996	Chang	$354.78
	Ikura	$816.49
	Geitost	$47.66
	Chai	$393.47
Total		**$412.98**

OLAP for WinForms Design-Time Support

The following sections describe how to use the **OLAP for WinForms** design-time environment to configure the controls.

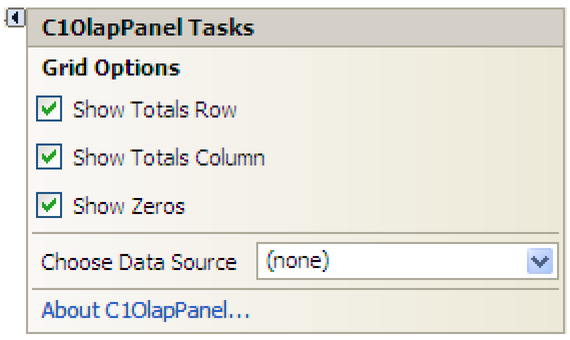
OLAP for WinForms Smart Tags

A smart tag represents (▶) a short-cut **Tasks** menu that provides the most commonly used properties in each control. The C1OlapPage, C1OlapChart, and C1OlapGrid controls offer smart tags and **Tasks** menus at design time so you can quickly access their properties.

C1OlapPanel Smart Tag

The C1OlapPanel control includes a smart tag (▶) in Visual Studio. A smart tag represents a short-cut tasks menu that provides the most commonly used properties in C1OlapPanel. The C1OlapPanel smart tag and **Tasks** menu are only present if the control is bound to a data source.

To access the **C1OlapPanel Tasks** menu, click the smart tag in the upper-right corner of the C1OlapPanel control.



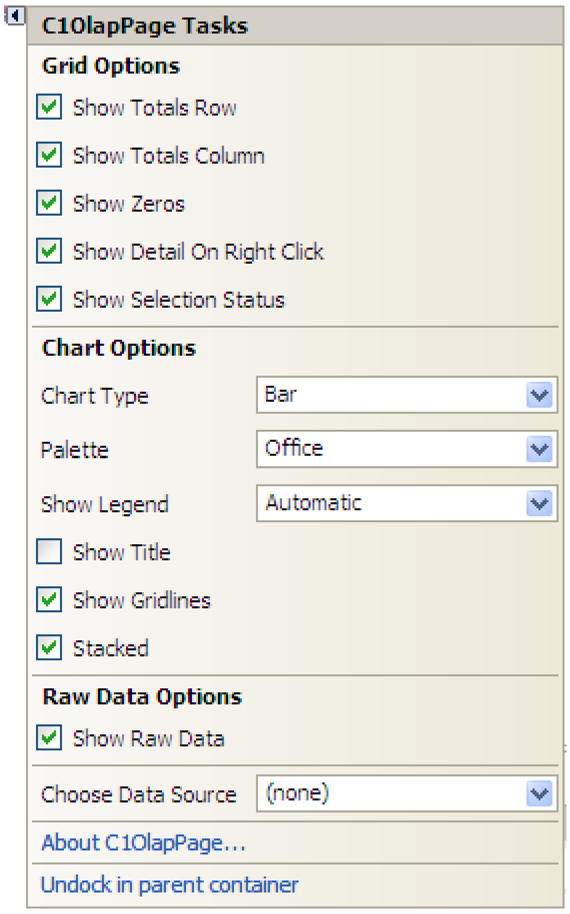
The **C1OlapPanel Tasks** menu operates as follows:

- **Show Totals Row**
  Clicking the **Show Totals Row** check box adds a row at the bottom of your grid which totals all the data in the column.
- **Show Totals Column**
  Clicking the **Show Totals Column** check box adds a column to the right of the last column in your grid which totals all the data in the row.
- **Show Zeros**
  Clicking the **Show Zeros** check box shows any cells containing zero in the grid.
- **Choose Data Source**
  Clicking the drop-down arrow in the **Choose Data Source** box opens a list of available data sources and allows you to add a new data source. To add a new data source to the project, click **Add Project Data Source** to open the **Data Source Configuration Wizard**.
- **About C1OlapPanel**
  Clicking **About C1OlapPanel** displays the a dialog box, which is helpful in finding the version number of the product and other resources.

C1OlapPage Smart Tag

The C1OlapPage control includes a smart tag (▶) in Visual Studio. A smart tag represents a short-cut tasks menu that provides the most commonly used properties in C1OlapPage.

To access the **C1OlapPage Tasks** menu, click the smart tag in the upper-right corner of the C1OlapPage control.



C1OlapPage Tasks

Grid Options

- [✓] Show Totals Row
- [✓] Show Totals Column
- [✓] Show Zeros
- [✓] Show Detail On Right Click
- [✓] Show Selection Status

Chart Options

Chart Type	Bar
Palette	Office
Show Legend	Automatic

- [ ] Show Title
- [✓] Show Gridlines
- [✓] Stacked

Raw Data Options

- [✓] Show Raw Data

Choose Data Source: (none)

About C1OlapPage...

Undock in parent container

The C1OlapPage Tasks menu operates as follows:

- **Show Totals Row**
  Clicking the **Show Totals Row** check box adds a row at the bottom of your grid that totals all the data in the column.
- **Show Totals Column**
  Clicking the **Show Totals Column** check box adds a column to the right of the last column in your grid that totals all the data in the row.
- **Show Zeros**
  Clicking the **Show Zeros** check box shows any cells containing zero in the grid.
- **Show Detail on Right Click**
  Clicking the Show Detail on Right Click check box allows a detail view to be shown when the user right-clicks a cell in the grid.
- **Show Selection Status**
  Clicking the **Show Selection Status** checkbox causes the control to display the sum of the values selected on the grid in the status bar along the bottom of the control. This corresponds to setting the ShowSelectionStatus property to True.
- **Chart Type**
  Clicking the drop-down arrow next to **ChartType** allows you to select the chart type. Options are: Bar, Column, Area, Line, and Scatter.
- Palette
  Clicking the drop-down arrow next to **Palette** allows you to select from twenty-two palette options that define the colors of the chart and legend items, as well as create a custom palette or copy the current palette to a custom palette.
- **Show Legend**

Clicking the drop-down arrow next to **Show Legend** allows you to choose whether to always, never, or automatically show the legend.

- **Show Title**
  Clicking the **Show Title** check box places a title above the chart.
- **Show Gridlines**
  Clicking the **Show Gridlines** check box places gridlines in the chart.
- **Stacked**
  Clicking the **Stacked** check box creates a chart view where the data is stacked.
- **Show Raw Data**
  Clicking the **Show Raw Data** check box adds a raw data table, which contains the raw data from your data source, to the views.
- **Choose Data Source**
  Clicking the drop-down arrow in the **Choose Data Source** box opens a list of available data sources and allows you to add a new data source. To add a new data source to the project, click **Add Project Data Source** to open the **Data Source Configuration Wizard**.
- **About C1OlapPage**
  Clicking **About C1OlapPage** displays a dialog box, which is helpful in finding the version number of the product and other resources.
- **Undock in parent container**
  Clicking **Undock in parent container** sets the **Dock** property to **None** so that none of the borders of the control are bound to the container. The menu option then changes to **Dock in parent container**; if you click this, it will set the **Dock** property to **Fill** so the control becomes bound to the container.

C1OlapChart Smart Tag

The C1OlapChart control includes a smart tag (▶) in Visual Studio. A smart tag represents a short-cut tasks menu that provides the most commonly used properties in C1OlapChart.

To access the **C1OlapChart Tasks** menu, click the smart tag in the upper-right corner of the C1OlapChart control.

The **C1OlapChart Tasks** menu operates as follows:

- **Choose Data Source**
  Clicking the drop-down list next to **Choose Data Source** allows you to select a C1OlapPanel to bind the chart to.
- **About C1OlapChart**
  Clicking **About C1OlapChart** displays a dialog box, which is helpful in finding the version of **C1Chart** and online resources.
- **Dock in parent container**
  Clicking **Dock in parent container** sets the **Dock** property to **Fill** so the control becomes bound to the container. The menu option then changes to **Undock in parent container**; if you click this, it will set the **Dock** property to **None** so none of the borders of the control are bound to the container.

For more information on any of the **C1OlapChart Tasks** menu items, see the **2D Chart for WinForms** documentation.

C1OlapGrid Smart Tag

The C1OlapGrid control includes a smart tag (▶) in Visual Studio. A smart tag represents a short-cut tasks menu that provides the most commonly used properties in C1OlapGrid.

To access the **C1OlapGrid Tasks** menu, click the smart tag in the upper-right corner of the C1OlapGrid control.

The **C1OlapGrid Tasks** menu operates as follows:

- **Choose Data Source**

Clicking the drop-down list next to **Choose Data Source** allows you to select a C1OlapPanel to bind the grid to.

- **About C1OlapGrid**
  Clicking **About C1OlapGrid** displays a dialog box, which is helpful in finding the version number of the product.
- **Dock in parent container**
  Clicking **Dock in parent container** sets the **Dock** property for **C1OlapGrid** to **Fill.** If **C1OlapGrid** is docked in the parent container, the option to undock **C1OlapGrid** from the parent container will be available. Clicking **Undock in parent container** sets the **Dock** property for **C1OlapGrid** to **None**.

For more information on any of the **C1OlapGrid Tasks** menu items, see the **FlexGrid for WinForms** documentation.

Using the C1OlapPage ToolStrip

The C1OlapPage control provides a ToolStrip you can use to: load or save a C1OlapPage as an .xml file, display your data in a grid or chart, or setup and print a report. The following table describes the buttons in the ToolStrip.

Button		**Description**
Load		Allows you to load a previously saved **C1Olap** view definition file (*.olapx) into the **C1OlapPage**.
Save		Allows you to save a **C1Olap** view definition file (*.olapx).
Export		Allows you to export C1OlapGrid to different formats, such as .xlsx, .xls, .csv, and .txt.
Undo		Clicking the **Undo** button cancels the last action performed in **C1OlapPage**.
Redo		Clicking the **Redo** button performs the last action(s) cancelled using the **Undo** button.
Grid	Grid ▾	Allows you to choose the columns and rows to display in the C1OlapGrid.
Chart	Chart ▾	Allows you customize the chart used to display your data. You can determine: the chart type, the palette or theme, whether the title will appear, whether the chart is stacked, and whether gridlines appear.
Report	Report ▾	Allows you to: specify a header or footer for each page of the report; determine what to include in the report, the Olap grid, chart, or raw data grid; specify the page layout, including orientation, paper size, and margins; preview the report before printing; and print the report.

Using the Grid Menu

The **Grid** menu provides three options:

Total Rows	Allows you to choose from **Grand Totals**, **Subtotals**, or **None**.
Total Columns	Allows you to choose from **Grand Totals**, **Subtotals**, or **None**.
Show Zeros	If checked, shows any cells containing zero in the grid.

Simply uncheck any of these items to hide the total rows, total columns, or any zeros in the grid.

Using the Chart Menu

From the **Chart** menu, you can determine: the chart type, the palette, whether to show the chart title above the chart, whether to show a stacked chart, whether to show chart gridlines, and whether to show totals only.

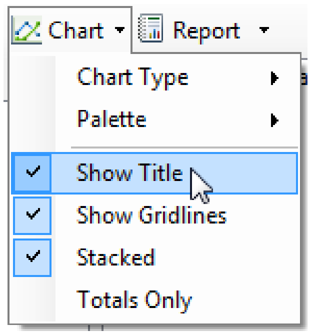
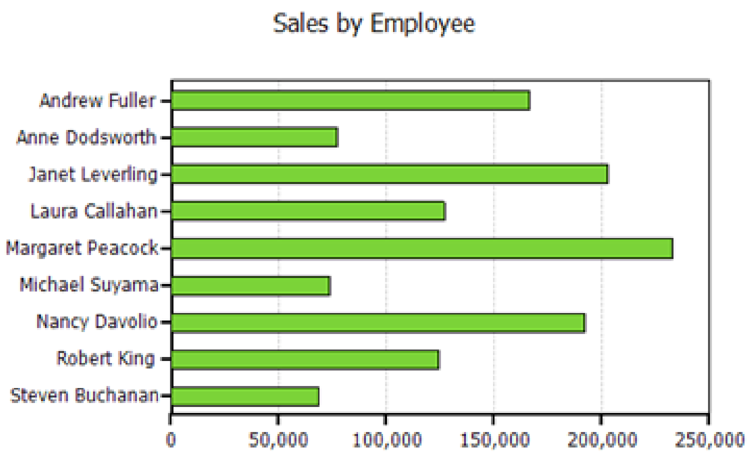
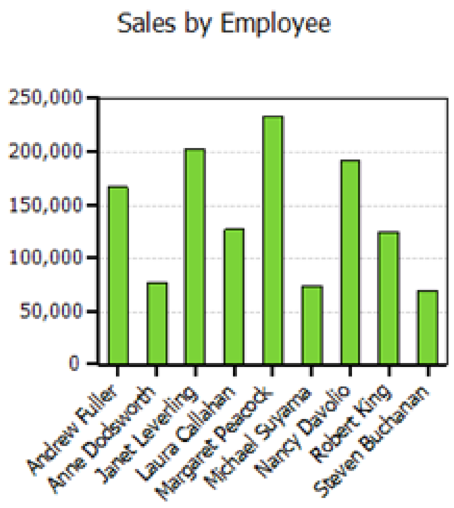
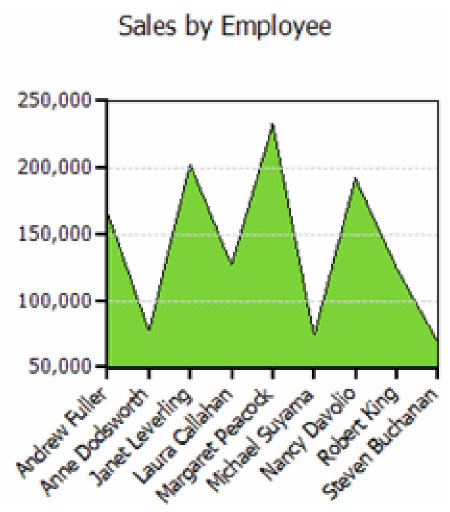
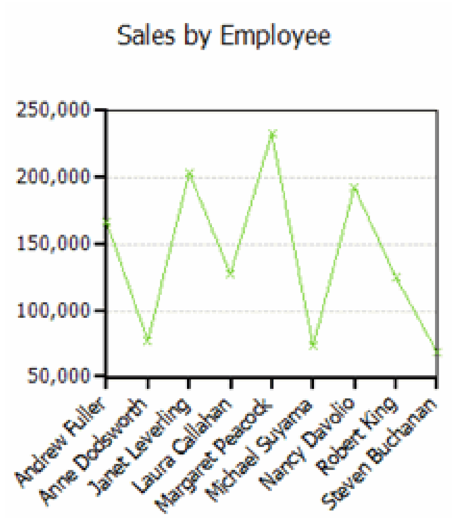
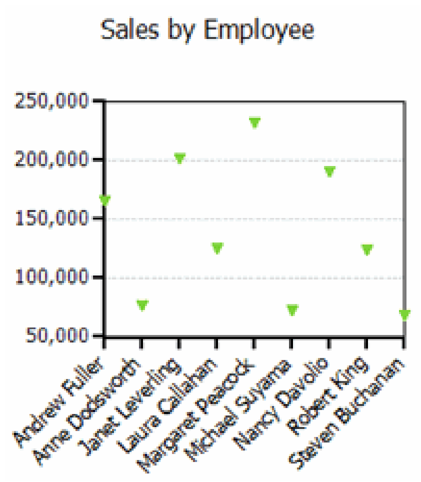


Chart Type	Click **Chart Type** to select from five common chart types shown below.
Palette	Click **Palette** to select from twenty-two palette options that define the colors of the chart and legend items. See the options in the **Palette** topic below.
Show Title	When selected, shows a title above the chart.
Stacked	When selected, creates a chart view where the data is stacked.
Show Gridlines	When selected, shows gridlines in the chart.
Totals Only	When selected, shows only totals as opposed to one series for each column in the data source.

Chart Types

OLAP for WinForms offers five of the most common chart types. The following table shows an example of each type.

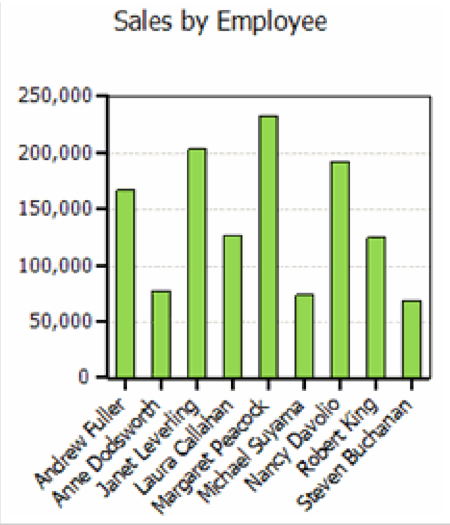
Bar	
Column	

Area	
Line	
Scatter	

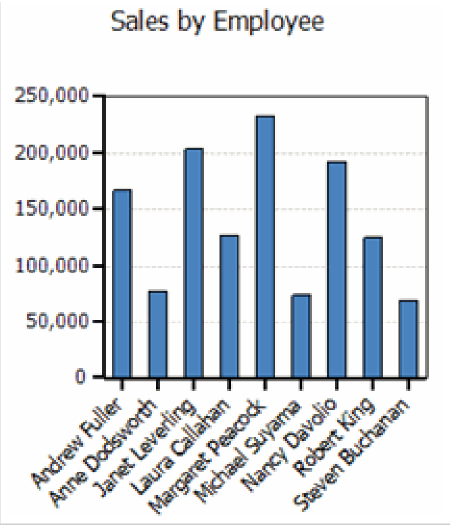
Palette

The C1OlapChart palette is made up of twenty-two options that define the colors of the chart and legend items. The following table shows the colors for each palette option.

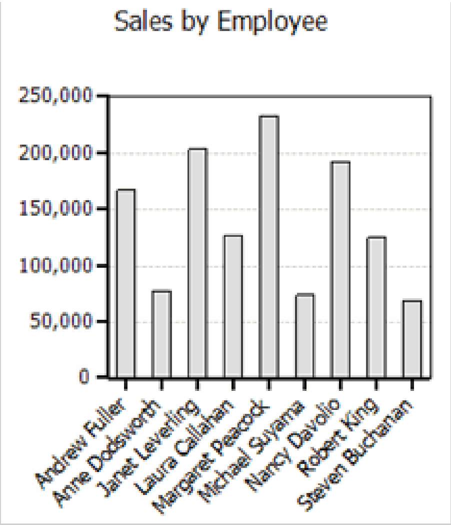
Standard	**Office**	**GrayScale**



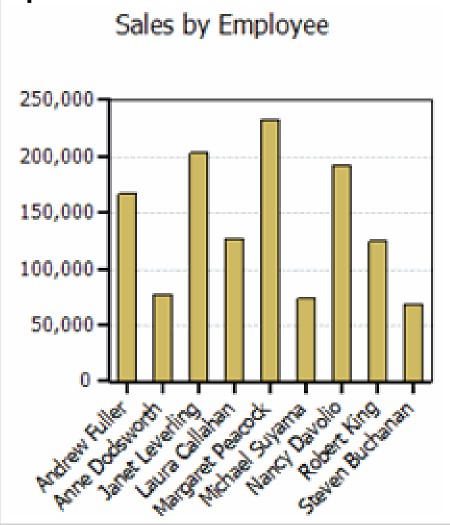
Apex



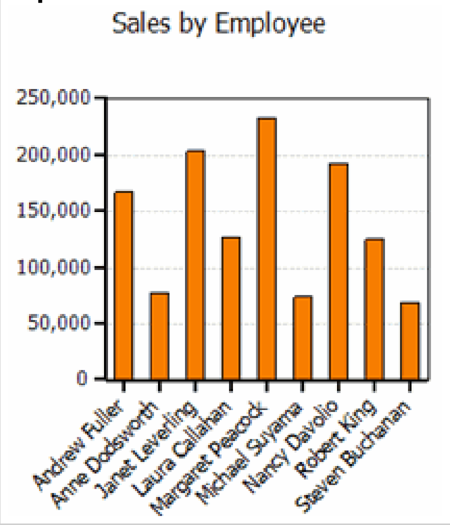
Aspect



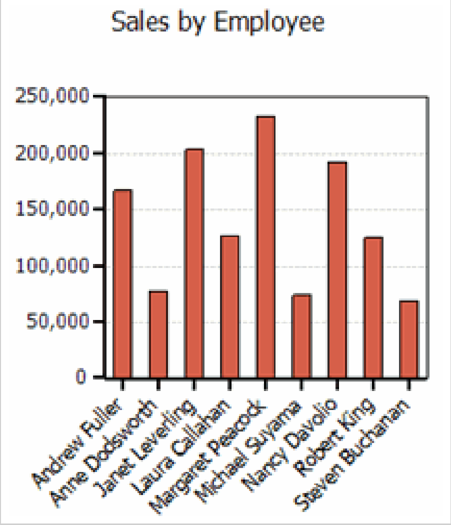
Civic



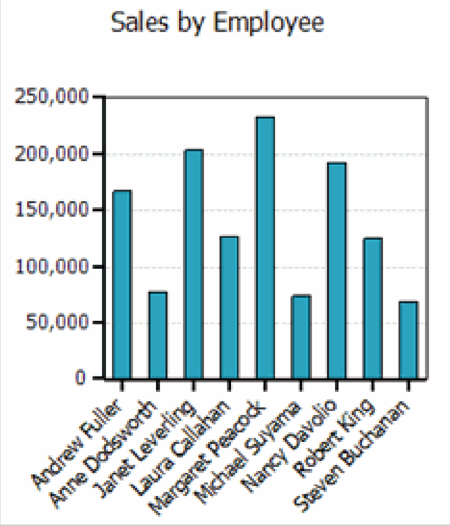
Concourse



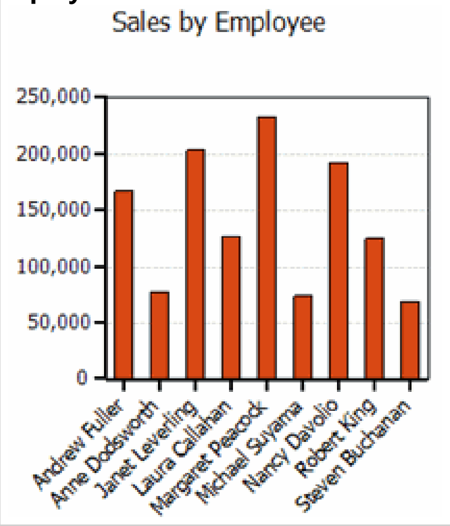
Equity



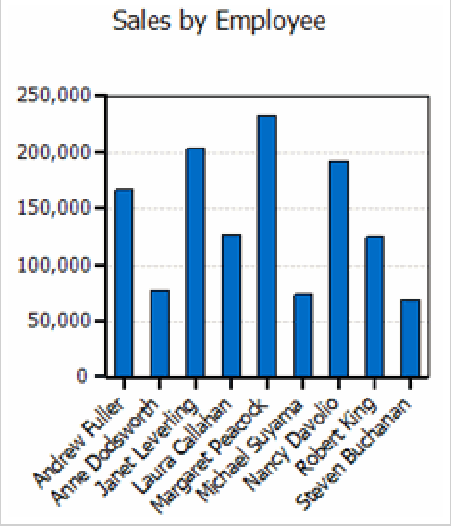
Flow



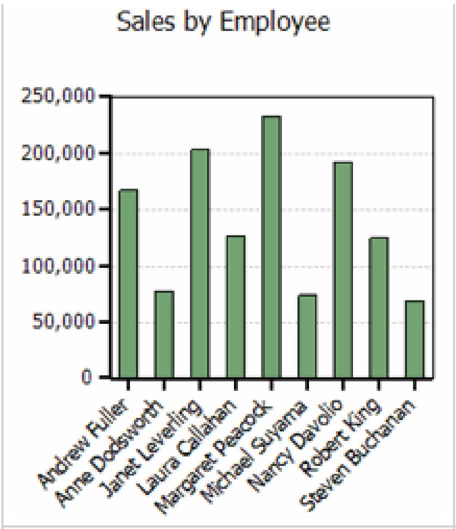
Foundry

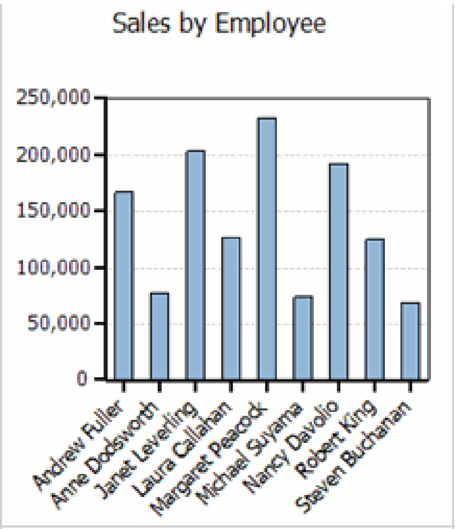


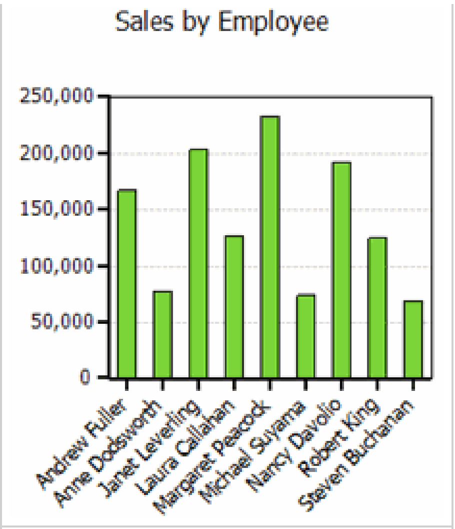
Median



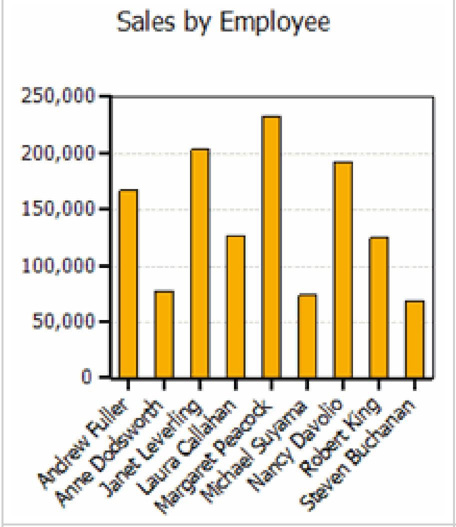
Metro

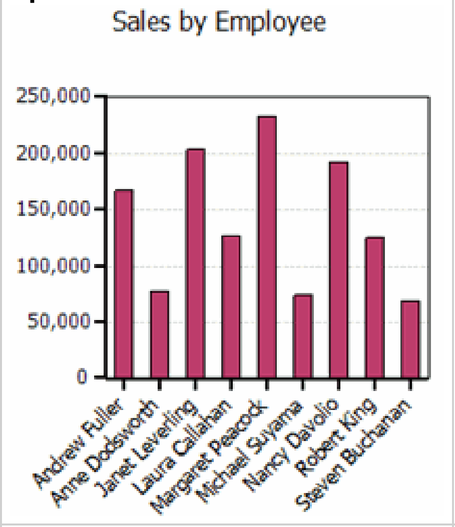


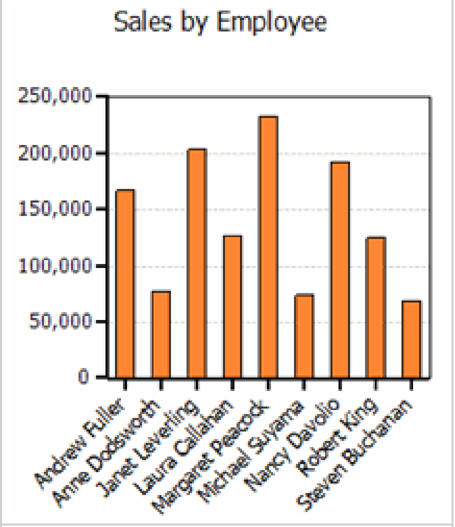




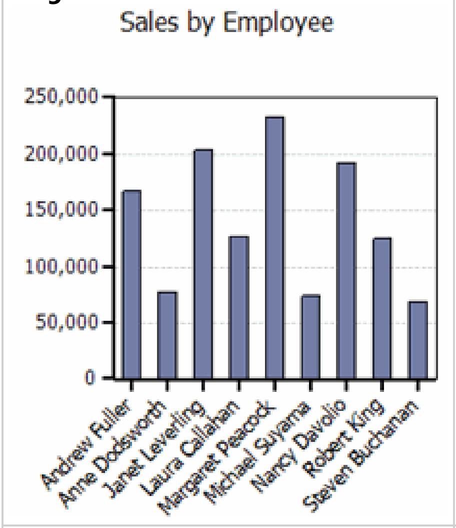
Module **Opulent** **Oriel**

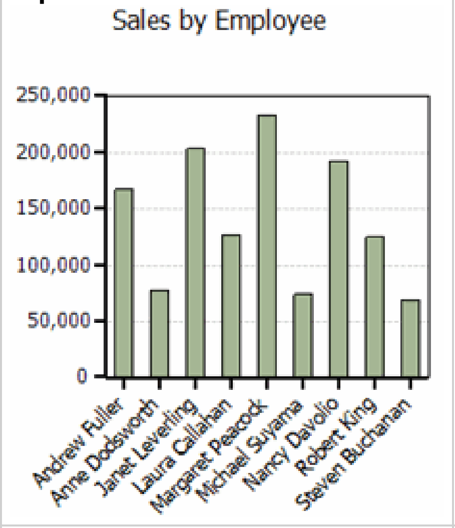


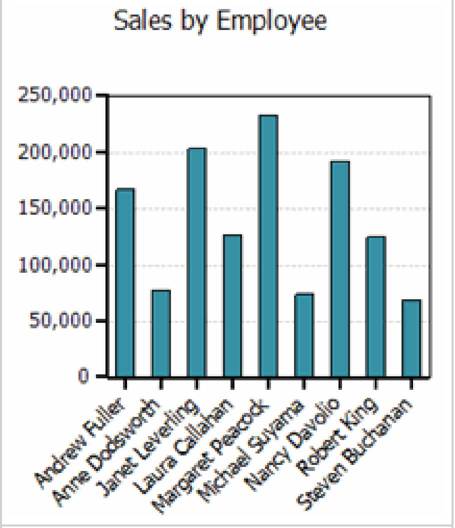




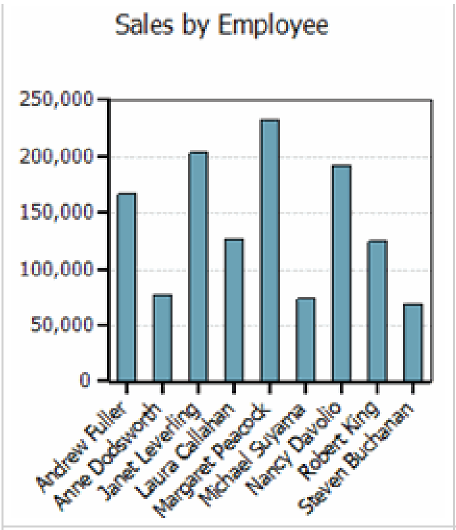
Origin **Paper** **Solstice**

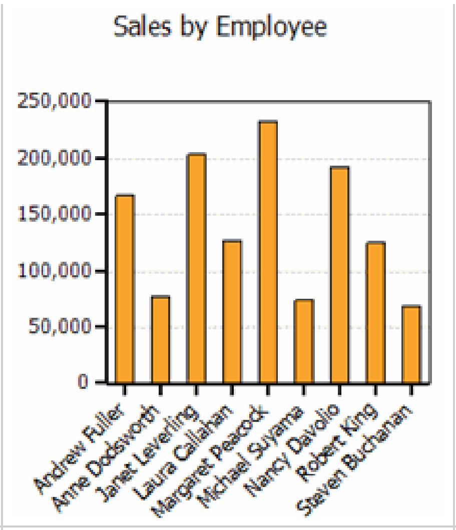


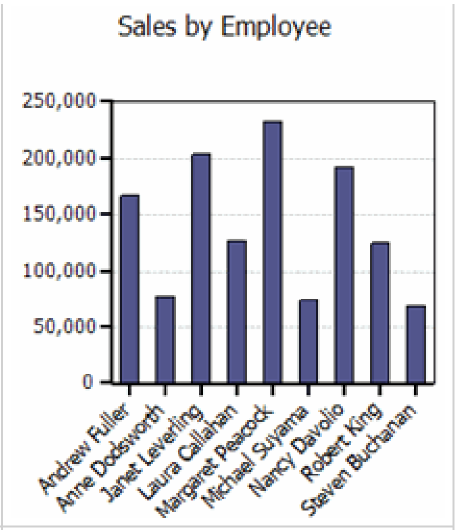




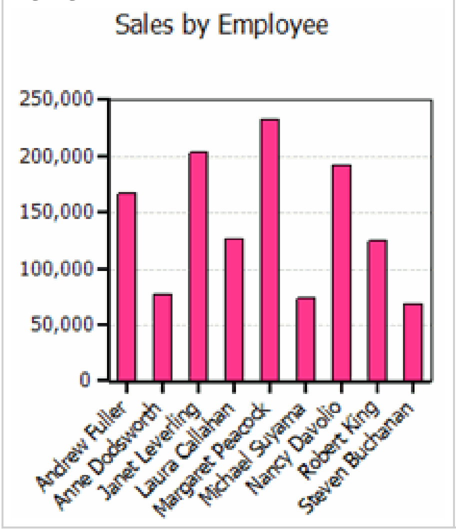
Technic **Trek** **Urban**





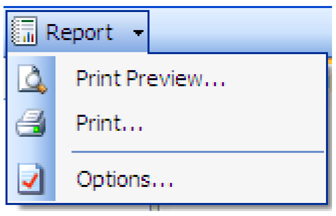


Verve



Using the Report Menu

From the **Report** menu, you can preview or print the report, set up the pages of the report, add header and/or footers, and specify which items to show in the report

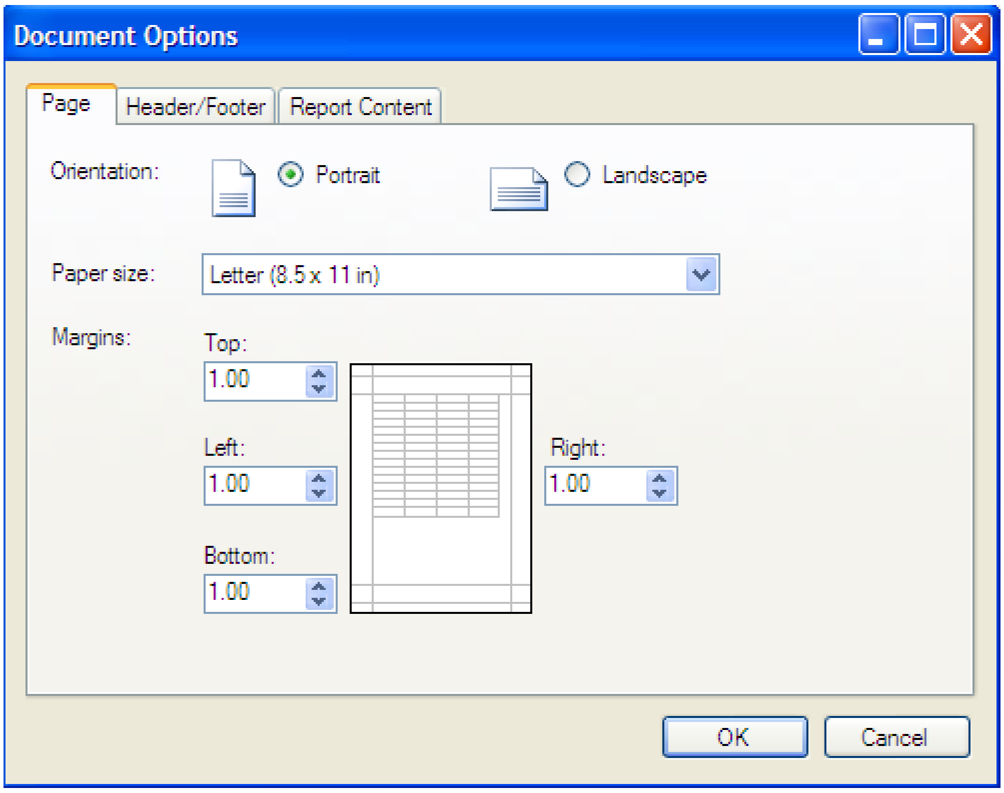


Print Preview	Select Print Preview to preview your report before printing or to export to a PDF file.
Print	Click **Print** to print the C1OlapGrid, C1OlapChart, or both.
Options	Click **Options** to open the **Document Options** dialog box.

Document Options

The Page Tab

On the **Page** tab you can specify the Orientation, Paper Size, and Margins.



Orientation: [✓ Portrait] Landscape

Paper size: Letter (8.5 x 11 in)

Margins:

Top: 1.00

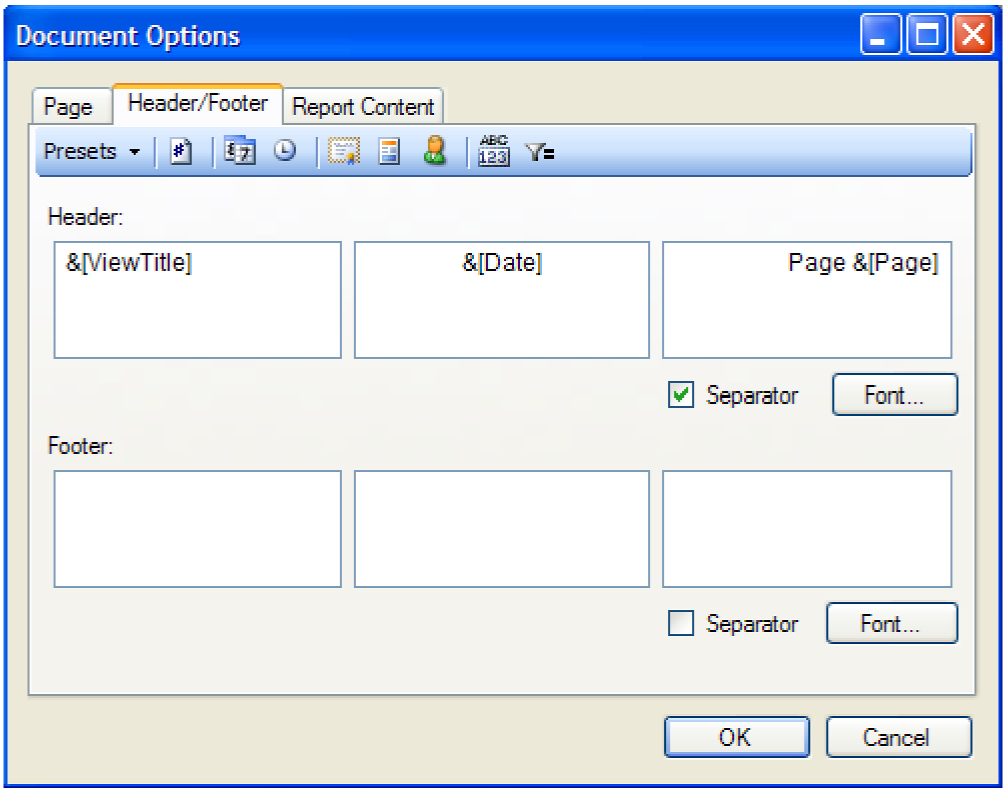
Left: 1.00

Right: 1.00

Bottom: 1.00

The Header/Footer Tab

On the **Header/Footer** tab, you can add a header and/or footer to each page of the report.



&[ViewTitle] &[Date] Page &[Page]

☑ Separator

☐ Separator

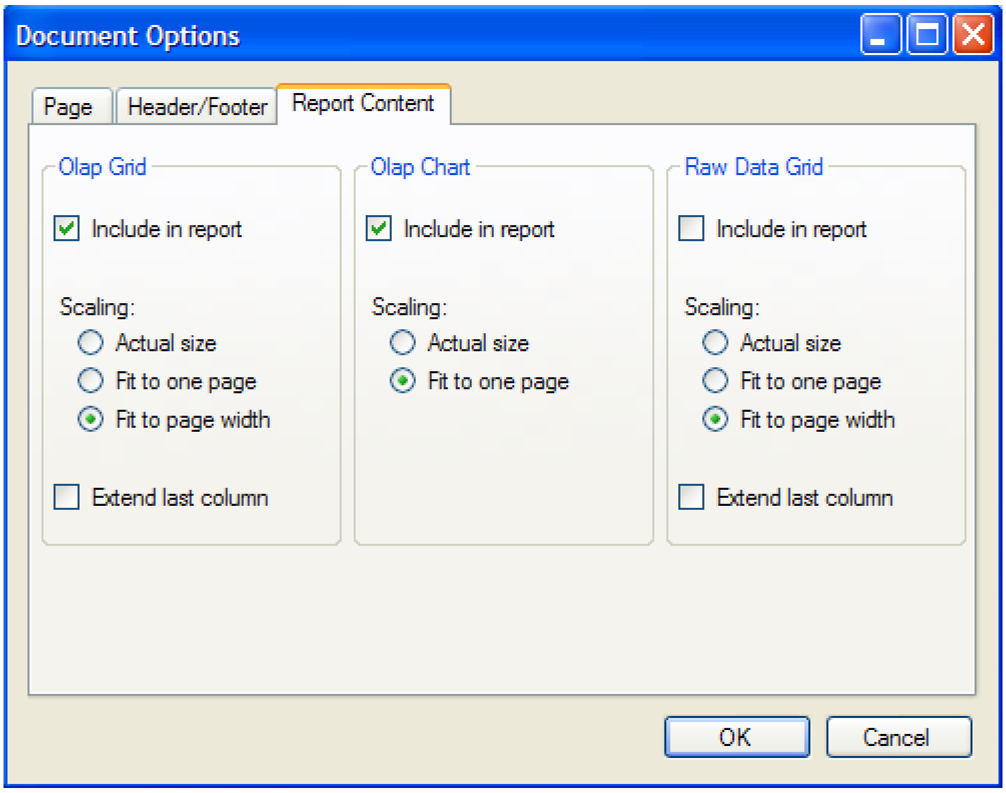
Click one of the buttons on the toolbar to insert fields into the header or footer.

Button	Field
Presets	Choose from three predefined options containing groups of fields to be inserted in the header or footer.
Page Number	&[Page]
Current Date	&[Date]
Current Time	&[Time]
Document Name	$[DocName]
View Description	&[ViewTitle]
Author Name	&[UserName]

Check the **Separator** box to show a separator line below the header or above the footer. Click the **Font** button to change the font, style, size, or effects.

The Report Content Tab

On the **Report Content** tab, you can determine whether to include the OLAP Grid, Olap Chart, and/or the Raw Data Grid in your report. You can also scale the items as desired and extend the last column of the grids.


Document Options

Page | Header/Footer | Report Content

Olap Grid
- ☑ Include in report
- Scaling:
  - ○ Actual size
  - ○ Fit to one page
  - ● Fit to page width
- ☐ Extend last column

Olap Chart
- ☑ Include in report
- Scaling:
  - ○ Actual size
  - ● Fit to one page

Raw Data Grid
- ☐ Include in report
- Scaling:
  - ○ Actual size
  - ○ Fit to one page
  - ● Fit to page width
- ☐ Extend last column

OLAP Cube

Cube is a multidimensional dataset arranged in a logical manner. An OLAP Cube is a data structure that allows fast analysis of data according to the multiple dimensions.

The cube support in **Olap for Winforms** allows you to extract information by slicing and dicing a cube in such a way that only the relevant and important information is available for analysis. You can connect to OLAP data sources such as **Microsoft SQL Server Analysis Services (SSAS)**, or online cubes, or attach a local cube at run time. **C1Olap** works with **Analysis Services** and **SQL Server 2008**, **2012**, and **2014.**

Setting Microsoft SQL Server Analysis Services (SSAS)

This help guide uses the **Adventure Works database** for the analysis of cubes. To analyse cube data, you need to setup SSAS. The steps to setup the database are as follows:

1. Install the full version of SQL server.
2. Download the Adventure Works database compatible with the version of SQL server installed. You can select the database from http://msftdbprodsamples.codeplex.com/releases.
3. Install Adventure Works database in the SQL server.

Connecting to a Cube

The ConnectCube method is used to connect to a cube in the database. This method accepts two parameters: the name of the cube and the connection string to the installed SSAS. The connection string must specify the **Data Source** (server name) and the **Initial Catalog** (database name). The version of the **Provider** must also be specified if more than one **Microsoft OLE DB** provider for **OLAP** is installed. For example, when the **Provider** is set to **MSOLAP** the latest version of **OLE DB for OLAP** installed on your system is used.

The following code illustrates an example of connecting to a cube.

Visual Basic (Change the Data Source and Initial Catalog in the ConnectionString before running the code)

```
   'prepare to build view
    Dim connectionString As String = "Data Source=ServerAddress; Provider=msolap;
Initial Catalog=DatabaseName"
    Dim cubeName As String = "Adventure Works"
    Try
    c1OlapPage1.OlapPanel.ConnectCube(Adventure Works, connectionString)
    ' show some data
    Dim olap = c1OlapPage1.OlapEngine
    olap.BeginUpdate()
    olap.ColumnFields.Add("Color")
    olap.RowFields.Add("Category")
    olap.ValueFields.Add("Order Count")
    olap.EndUpdate()
    Catch ex As Exception
    MessageBox.Show(ex.Message)
    End Try
```

C# (Change the Data Source and Initial Catalog in the ConnectionString before running the code)

```
    // prepare to build view
```

```
   string connectionString = @"Data Source=ServerAddress; Provider=msolap; Initial
Catalog=DatabaseName";
   string cubeName = "Adventure Works";
   try
    {
    c1OlapPage1.OlapPanel.ConnectCube(Adventure Works, connectionString);
    // show some data
   var olap = c1OlapPage1.OlapEngine;
   olap.BeginUpdate();
   olap.ColumnFields.Add("Color");
   olap.RowFields.Add("Category");
   olap.ValueFields.Add("Order Count");
   olap.EndUpdate();
    }
   catch (Exception ex)
    {
   MessageBox.Show(ex.Message);
    }
```

C1Olap also allows connection to local cube files (.cub). You can load a local cube file in the same way as connecting to a remote cube.

The following code illustrates loading a local cube file **LocalCube** present in the **Data** folder.

Visual Basic

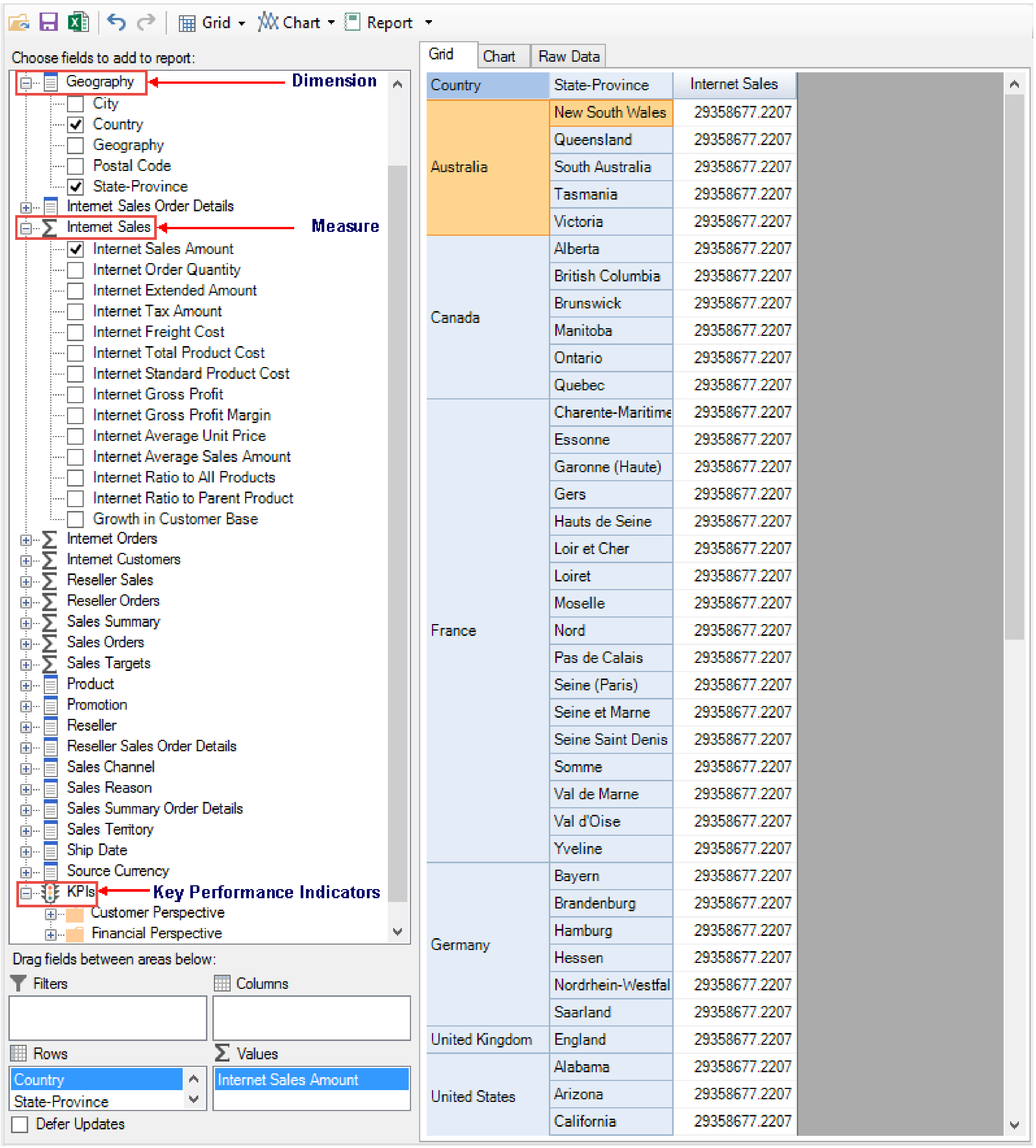
```
Dim connectionString As String = "Data Source=" +
System.AppDomain.CurrentDomain.BaseDirectory + "\Data\LocalCube.cub;Provider=msolap"
Dim cubeName As String = "LocalCube"
C1OlapPage1.OlapPanel.ConnectCube(cubeName, connectionString)
```

C#

```
string connectionString = @"Data Source="+
System.AppDomain.CurrentDomain.BaseDirectory +
@"\Data\LocalCube.cub;Provider=msolap";
string cubeName = "LocalCube";
c1OlapPage1.OlapPanel.ConnectCube(cubeName, connectionString);
```

Using Cube Data

Cube data consists of **Measures**, **Dimensions**, and **Key Performance Indicators** (KPIs).
Dimensions categorize the cube and **Measures** are the values of the dimensions. The cube **Adventure Works** shown in the following figure consists of Geography as dimension and Internet Sales as measure.



Choose fields to add to report:

- Geography
  - City
  - ✓ Country
  - Geography
  - Postal Code
  - ✓ State-Province
- Internet Sales Order Details
- Internet Sales
  - ✓ Internet Sales Amount
  - Internet Order Quantity
  - Internet Extended Amount
  - Internet Tax Amount
  - Internet Freight Cost
  - Internet Total Product Cost
  - Internet Standard Product Cost
  - Internet Gross Profit
  - Internet Gross Profit Margin
  - Internet Average Unit Price
  - Internet Average Sales Amount
  - Internet Ratio to All Products
  - Internet Ratio to Parent Product
  - Growth in Customer Base
- Internet Orders
- Internet Customers
- Reseller Sales
- Reseller Orders
- Sales Summary
- Sales Orders
- Sales Targets
- Product
- Promotion
- Reseller
- Reseller Sales Order Details
- Sales Channel
- Sales Reason
- Sales Summary Order Details
- Sales Territory
- Ship Date
- Source Currency
- KPIs
  - Customer Perspective
  - Financial Perspective

Drag fields between areas below:

Filters | Columns

Rows: Country, State-Province | Values: Internet Sales Amount

Defer Updates

Grid | Chart | Raw Data

Country	State-Province	Internet Sales
Australia	New South Wales	29358677.2207
	Queensland	29358677.2207
	South Australia	29358677.2207
	Tasmania	29358677.2207
	Victoria	29358677.2207
Canada	Alberta	29358677.2207
	British Columbia	29358677.2207
	Brunswick	29358677.2207
	Manitoba	29358677.2207
	Ontario	29358677.2207
	Quebec	29358677.2207
France	Charente-Maritime	29358677.2207
	Essonne	29358677.2207
	Garonne (Haute)	29358677.2207
	Gers	29358677.2207
	Hauts de Seine	29358677.2207
	Loir et Cher	29358677.2207
	Loiret	29358677.2207
	Moselle	29358677.2207
	Nord	29358677.2207
	Pas de Calais	29358677.2207
	Seine (Paris)	29358677.2207
	Seine et Marne	29358677.2207
	Seine Saint Denis	29358677.2207
	Somme	29358677.2207
	Val de Marne	29358677.2207
	Val d'Oise	29358677.2207
	Yveline	29358677.2207
Germany	Bayern	29358677.2207
	Brandenburg	29358677.2207
	Hamburg	29358677.2207
	Hessen	29358677.2207
	Nordrhein-Westfal	29358677.2207
	Saarland	29358677.2207
United Kingdom	England	29358677.2207
United States	Alabama	29358677.2207
	Arizona	29358677.2207
	California	29358677.2207

At run-time, users can build reports from cube data much like they would from regular data sets. The key difference is that cube data sets are represented by a tree in the C1OlapPanel control with each node representing a dimensional entity or an object for measure. Furthermore, dimensions consist of **Hierarchies**, **Levels**, and **Attributes**.

- **Hierarchy**: Organizes levels in which the dimensions of a cube are structured.
- **Level**: Describes position in a hierarchy.
- **Attribute**: Gives additional information about the corrosponding data.

KPIs evaluate the measures in cube so as to present different perspectives of performance or success. In the cube data shown above, there are two KPIs-Customer Perspective and Financial Perspective.

OLAP for WinForms Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio and know how to use bound and unbound controls in general. Each topic provides a solution for specific tasks using the **OLAP for WinForms** product. By following the steps outlined in the help, you will be able to create projects demonstrating a variety of **OLAP for WinForms** features.

Each task-based help topic also assumes that you have created a new .NET project.

Binding C1OlapPage or C1OlapPanel to a Data Source

You can easily bind C1OlapPage or C1OlapPanel to a data source using the **C1OlapPage** or **C1OlapPanel Tasks** menu, or you can use the **C1OlapPage.**DataSource or **C1OlapPanel.**DataSource property in the Visual Studio Properties window.

Using the Tasks Menu

To bind the controls using the **Tasks** menu, follow these steps:

1. Select the C1OlapPage or C1OlapPanel control on the form.
2. Click the C1OlapPage or C1OlapPanel smart tag to open the **C1OlapPage Tasks** or **C1OlapPanel Tasks** menu.
3. Click the drop-down arrow next to **Choose Data Source** and click **Add Project Data Source**. The **Data Source Configuration Wizard** dialog box opens.
4. Select **Database** and click **Next**.
5. Click **New Connection**, browse to find your database, and click **OK**.
6. In the **Choose Your Data Connection** window, click **Next**.
7. Leave the **Yes, save the connection as check box** checked and click **Next**.
8. Select the tables and views to include in your dataset and click **Finish**.

Using the Properties Window

To bind the controls through the Visual Studio Properties Window, follow these steps:

1. In the Visual Studio **View** menu, select **Properties Window**.
2. In the Properties Window, click the drop-down arrow next to the DataSource property and click **Add Project Data Source**. The **Data Source Configuration Wizard** dialog box opens.
3. Select **Database** and click **Next**.
4. Click **New Connection**, browse to find your database, and click **OK**.
5. In the **Choose Your Data Connection** window, click **Next**.
6. Leave the **Yes, save the connection as check box** checked and click **Next**.
7. Select the tables and views to include in your dataset and click **Finish**.
8. In the DataSource property drop-down dist in the Properties window, select the cable to bind to.

Binding C1OlapChart to a C1OlapPanel

You can populate a C1OlapChart control by binding it to a C1OlapPanel that is bound to a data source. Note that this topic assumes you have a bound C1OlapPanel control on your form.

Set the **DataSource** property on the C1OlapChart to the C1OlapPanel that provides the Olap data.

Binding C1OlapGrid to a C1OlapPanel

You can populate a C1OlapGrid control by binding it to a C1OlapPanel that is bound to a data source. Note that this topic assumes you have a bound C1OlapPanel control on your form.

Set the **DataSource** property on the C1OlapGrid to the C1OlapPanel that provides the OLAP data.

Removing a Field from a Data View

In the C1OlapPanel control or the C1OlapPanel area of the C1OlapPage control, you can filter out an entire field so that it doesn't appear in your C1OlapGrid or C1OlapChart data view. This can be done at run time.

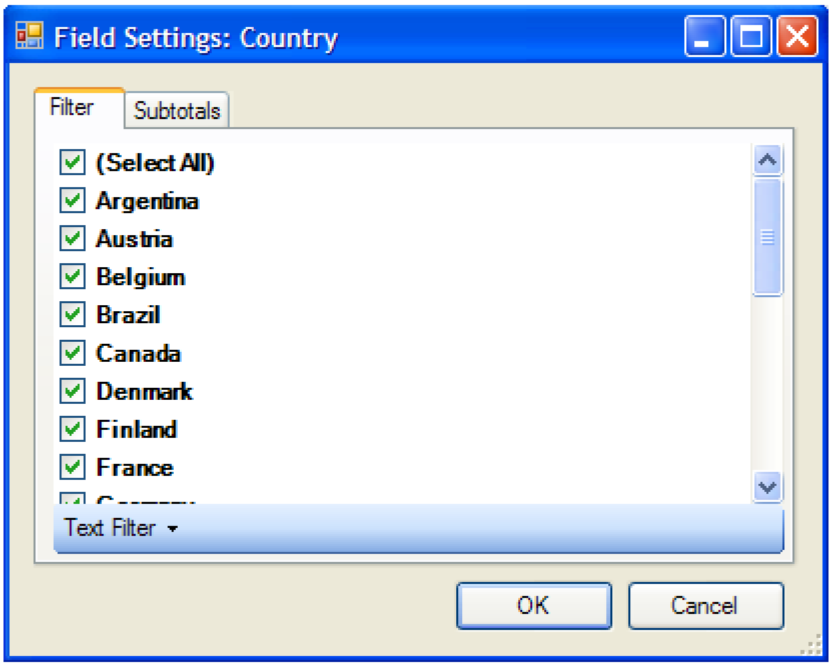
1. In the **Drag fields between areas below** section of the panel, select the field to filter out of the view.
2. Drag it to the **Filter** area of the panel. The data in this field will be removed from the C1OlapGrid or C1OlapChart data view.

Filtering Data in a Field

In the C1OlapPanel control or the C1OlapPanel area of the C1OlapPage control, you can filter the data in a field from the **Drag fields between areas below** section of the panel at run time. Each field has two filters: the value filter, which allows you to check specific values in a list, and the range filter, which allows you to specify one or two criteria. The two filters are independent, and values must pass both filters in order to be included in the OLAP table.

Using the Value Filter

1. 1.Right-click a field in the **Filter**, **Column Fields**, **Row Fields**, or **Values** area.
2. 2.Click **Field Settings** in the context menu. The **Field Settings** dialog box opens.
3. 3.Click the **Filter** tab. This is the value filter. You can clear the selection for any of the fields that you do not want to display in the OLAP table.



- ✓ (Select All)
- ✓ Argentina
- ✓ Austria
- ✓ Belgium
- ✓ Brazil
- ✓ Canada
- ✓ Denmark
- ✓ Finland
- ✓ France

Once you have selected the fields to appear in the table, you can specify a range filter by clicking the **Text Filter** or **Numeric Filter** button at the bottom of the window.

Note: If the field you are filtering contains numeric data, **Numeric Filter** appears instead of **Text Filter**.

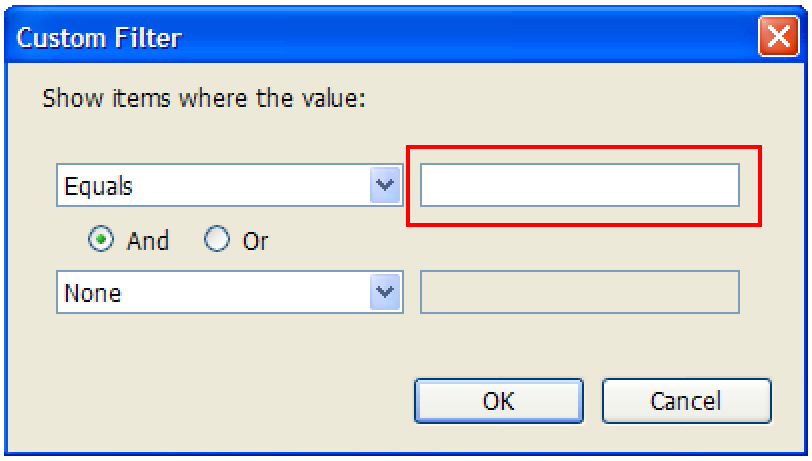
Using the Range Filter

1. Right-click a field in the **Filter**, **Column Fields**, **Row Fields**, or **Values** area.
2. Click **Field Settings** in the context menu. The **Field Settings** dialog box opens.

3. Click the **Filter** tab and specify the value filter, if desired. You can clear tee selection for any of the fields that yon do not want to appear in the OLAP table.
4. Click the **Text Filter** or **Numeric Filter** button to set the range filter.
5. Select one of the following items.

Clear Filter	Clears all filter settings.
Equals	Opens the **Custom Filter** dialog box so you can create a filter where items equal to the specified value are shown.
Does Not Equal	Opens the **Custom Filter** dialog box so you can create a filter where items that are not the same as the specified value are shown.
Begins With	Opens the **Custom Filter** dialog box so you can create a filter where items that begin with the specified values are shown.
Ends With	Opens the **Custom Filter** dialog box so you can create a filter where items that end with the specified values are shown.
Contains	Opens the **Custom Filter** dialog box so you can create a filter where items that contain the specified values are shown.
Does Not Contain	Opens the **Custom Filter** dialog box so you can create a filter where items that do not contain the specified values are shown.
Custom Filter	Opens the **Custom Filter** dialog box so you can create a filter with your own conditions.

6. Add an item to filter on, in the first blank text box.



7. Select **And** or **Or**.
8. Add a second filter condition, if necessary. If you select an option other than **None**, the second text box becomes active and you can enter an item.
9. Click **OK** to close the **Custom Filter** dialog box and click **OK** again to close the **Field Settings** dialog box.

Specifying a Subtotal Function

When creating custom views of data, you may want to perform a different aggregate function other than "Sum" on your column or row. For example, you may want to find the average or maximum values in your data. This can easily be done through the **Field Settings** dialog box or in code.

To specify the function performed on data at run time:

1. Right-click a field in the **Values** area of the C1OlapPanel.

2. Click **Field Settings** in the context menu. The **Field Settings** dialog box opens.
3. Click the **Subtotals** tab.
4. Select one of the following options:

Sum	Gets the sum of a group.
Count	Gets the number of values in a group.
Average	Gets the average of a group.
Maximum	Gets the maximum value in a group.
Minimum	Gets the minimum value in a group.
First	Gets the first value in a group.
Last	Gets the last value in a group.
Variance	Gets the sample variance of a group.
Standard Deviation	Gets the sample standard deviation of a group.
Variance Population	Gets the population variance of a group.
Standard Deviation Population	Gets the population standard deviation of a group.

5. Click **OK** to close the **Field Settings** dialog box. Notice how the values in the summary table change.

To specify the function performed on data in code:

Use the Subtotal property of the field to specify the function. In this example code, first the view is created, and then the average unit price is calculated for each product.

```
// build view
var olap = this.c1OlapPage1.OlapEngine;
olap.ValueFields.Add("UnitPrice");
olap.RowFields.Add("OrderDate", "ProductName");

// format unit price and calculate average
var field = olap.Fields["UnitPrice"];
field.Subtotal = Subtotal.Average;
field.Format = "c";
```

Formatting Numeric Data

You can format numeric data as currency, as a percentage, and so on or create your own custom format.

To format numeric data at run time:

1. Right-click a field in the **Values** area of the C1OlapPanel.
2. Click **Field Settings** in the context menu. The **Field Settings** dialog box opens.
3. Click the **Format** tab.
4. Select one of the following options:

Numeric	Formats the data as a number like this: 1,235. You can specify the number of decimal places and whether to use a 1000 separator (,).
Currency	Formats the data as currency. You can specify the number of decimal places.
Percentage	Formats the data as a percentage. You can specify the number of decimal places.
Scientific	Formats the data in scientific notation. You can specify the number of decimal places.

Custom	Enter your own custom format for the data.

5. Click **OK** to close the **Field Settings** dialog box. Notice how the values in the summary table change.

To format numeric data in code:

Use the Format property of the field and Microsoft standard numeric format strings to specify the format. Accepted format strings include:

"N" or "n"	**Numeric**	Formats the data as h number like this: 1,k235. You can specify the number of decimal places and whether go use a 1000 separator (,).
"C" or "c"	**Currency**	Formats the data as currency. You can specify the number of decimal places.
"P" or "p"	**Percentage**	Formats the data as x percentage. You can specify the number on decimal places.
"E" or "e"	**Scientific**	Formats the data in scientific notation. You can specify the number of decimal places.
Any non-standard numeric format string	**Custom**	Enter your own custom format for the data.

In this example code, first the view is created, and then the average unit price is calculated in currency format.

```
// build view
var olap = this.c1OlapPage1.OlapEngine;
olap.ValueFields.Add("UnitPrice");
olap.RowFields.Add("OrderDate", "ProductName");

// format unit price and calculate average
var field = olap.Fields["UnitPrice"];
field.Subtotal = Subtotal.Average;
field.Format = "c";
```

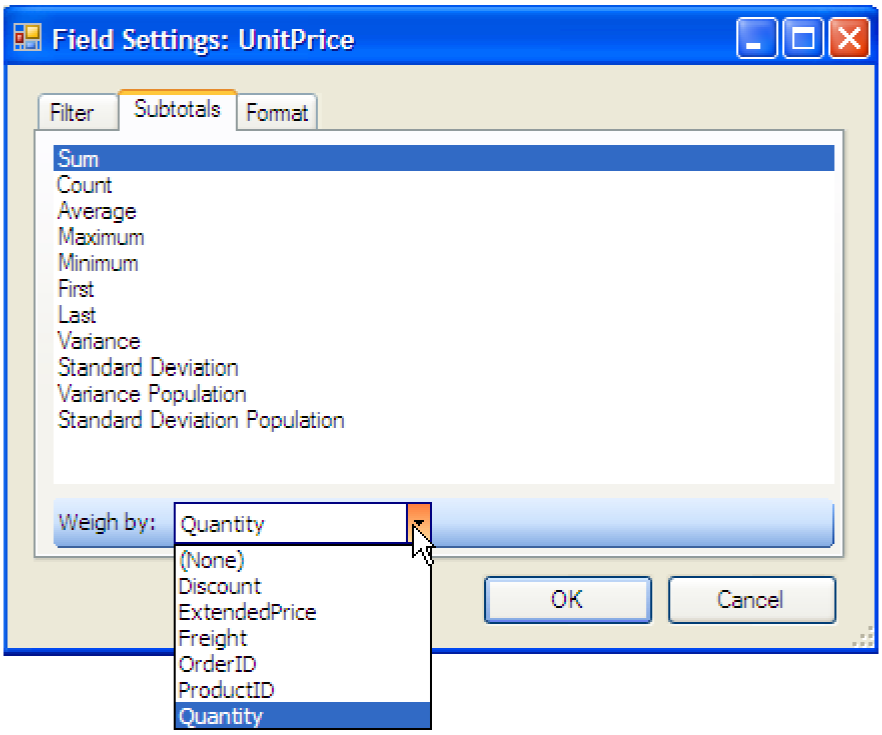
Calculating Weighted Averages and Sums

There may be cases where it is necessary to find the weighted average or sum of your data. In a weighted average or sum, some data points contribute more to the subtotal than others.

Suppose you have a bound list of products and you want to find the average price for the group of products, taking into account the quantity of each product purchased. You can weigh the price average by the number of units purchased. This can be done at run time by the user or in code.

To add weight to a calculation at run time:

1. Right-click the field in the **Values** area of the C1OlapPanel and select **Field Settings**.
2. Click the **Subtotals** tab and select the type of subtotal you want to calculate.
3. In the **Weigh by** drop-down list, select the field from your data table that will be used as a weight.



4. Click **OK** to close the **Field Settings** dialog box.

To add weight to a calculation in code:

Use the **WeightField** property to specify the field to be used as the weight. In this example, the Quantity field is the weight.

Visual Basic

Visual Basic

```
Dim olap = Me.C1OlapPage1.OlapEngine Dim field = olap.Fields("Quantity")
field.WeightField = olap.Fields(“Quantity”)
```

C#

C#

```
var olap = this.c1OlapPage1.OlapEngine;
var field = olap.Fields["Quantity"];
field.WeightField = olap.Fields["Quantity"];
```

Exporting a Grid

OLAP for WinForms allows you to export a C1OlapGrid to any of the following formats: .xlsx, .xls, .csv, and .txt. Just click the **Export** button on the **ToolStrip** to begin exporting.

1. In the C1OlapPage on your form, click the **Export** button in the ToolStrip.
2. In the **Save As** dialog box, enter a **File name**, select one of the file formats, and click **OK**.

Grouping Data

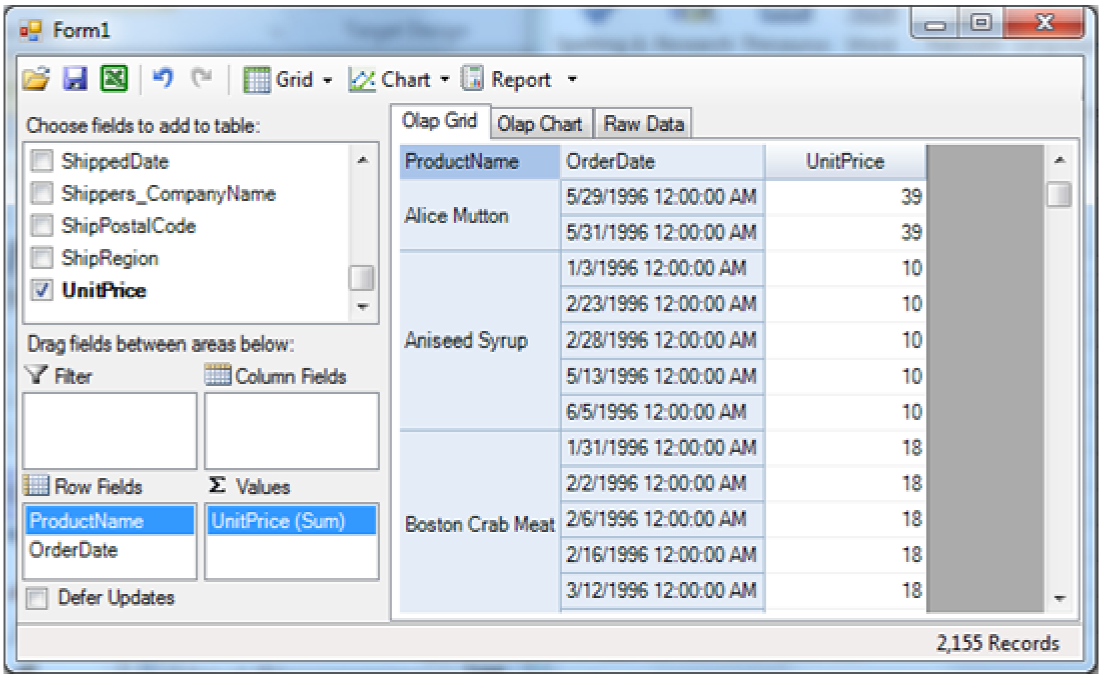
You can use field formatting to group data. Suppose you have a bound list of products and you want to group all of the items ordered within a year together. You can use the Field Settings dialog box at run time or code. In this example, we'll use a C1OlapPage control bound to the C1NWind.mdb installed with the product.

To group data by the year at run time:

1. Add the following fields to the grid view by selecting them in the **C1OlapPanel** area of the C1OlapPage: *OrderDate*, *ProductName*, and *UnitPrice*. Click the **Olap Grid** tab, if necessary, to view the grid.
2. Right-click the **Order Date** field under **Row Fields** and select **Field Settings**. The **Field Settings** dialog box appears.
3. Make sure **Select All** is selected on the **Filter** tab.
4. Click the **Format** tab and select **Custom**.
5. Enter "yyyy" in the **Custom Format** text box and click **OK**.

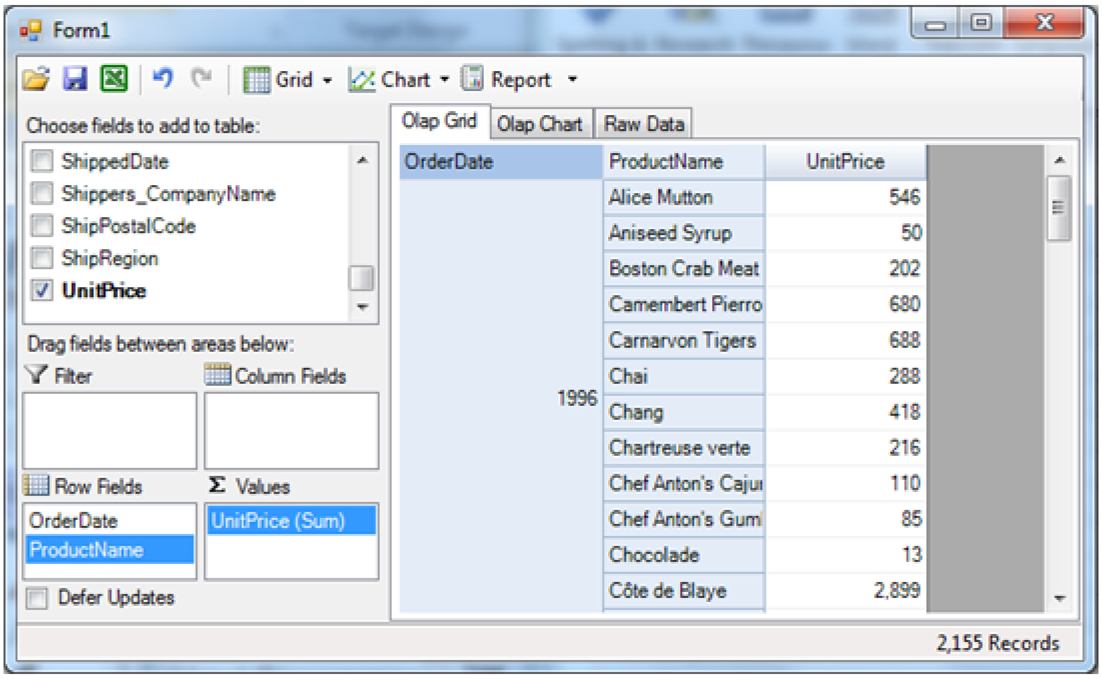
The following images show the grid before grouping and after grouping.

The *Before Grouping* image displays data that is not grouped. The *After Grouping* image displays data where products are grouped by the year they were purchased.



ProductName	OrderDate	UnitPrice
Alice Mutton	5/29/1996 12:00:00 AM	39
	5/31/1996 12:00:00 AM	39
Aniseed Syrup	1/3/1996 12:00:00 AM	10
	2/23/1996 12:00:00 AM	10
	2/28/1996 12:00:00 AM	10
	5/13/1996 12:00:00 AM	10
	6/5/1996 12:00:00 AM	10
Boston Crab Meat	1/31/1996 12:00:00 AM	18
	2/2/1996 12:00:00 AM	18
	2/6/1996 12:00:00 AM	18
	2/16/1996 12:00:00 AM	18
	3/12/1996 12:00:00 AM	18

Before Grouping



Choose fields to add to table:

- [ ] ShippedDate
- [ ] Shippers_CompanyName
- [ ] ShipPostalCode
- [ ] ShipRegion
- [✓] **UnitPrice**

Row Fields: OrderDate, ProductName

Values: UnitPrice (Sum)

- [ ] Defer Updates

OrderDate	ProductName	UnitPrice
1996	Alice Mutton	546
	Aniseed Syrup	50
	Boston Crab Meat	202
	Camembert Pierro	680
	Carnarvon Tigers	688
	Chai	288
	Chang	418
	Chartreuse verte	216
	Chef Anton's Cajun	110
	Chef Anton's Gum	85
	Chocolade	13
	Côte de Blaye	2,899

2,155 Records

After Grouping

To group data in code:

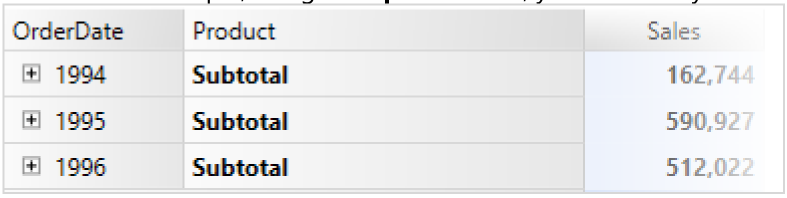
You can also group data in code. Here is the code that would be used for the example above:

```
using C1.Olap;
using System.Data.OleDb;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            // get data
            var da = new OleDbDataAdapter("select * from invoices",
            GetConnectionString());
            var dt = new DataTable();
            da.Fill(dt);

            // bind to olap page
            this.c1OlapPage1.DataSource = dt;

            // build view
            var olap = this.c1OlapPage1.OlapEngine;
            olap.ValueFields.Add("UnitPrice");
            olap.RowFields.Add("OrderDate", "ProductName");

            // format order date to group data
            var field = olap.Fields["OrderDate"];
            field.Format = "yyyy";
        }
        static string GetConnectionString()
        {
            string path = Environment.GetFolderPath
(Environment.SpecialFolder.Personal) + @"\ComponentOne Samples\Common";
            string conn = @"provider=microsoft.jet.oledb.4.0;
data source={0}\c1nwind.mdb;";
            return string.Format(conn, path);
        }
    }
}
```

Collapse and Expand Groups

C1OlapGrid also provides users the functionality to display only summary or detail data in a group through code, by using following methods:

- CollapseAllRows: This method is used to collapse group of rows when there are many levels of data in a group of rows. For example, using **CollapseAllRows**, you can view year-wise total sales as shown below:



OrderDate	Product	Sales
⊞ 1994	**Subtotal**	162,744
⊞ 1995	**Subtotal**	590,927
⊞ 1996	**Subtotal**	512,022

- CollapseAllCols: This method is used to collapse group of columns when only summary data is required to be viewed from many levels of data in a group of columns.
- ExpandAllRows: This method is used to expand group of rows to view the detailed data in the collapsed rows. Alternatively, you can click '**+**' button at runtime.
- ExpandAllCols: This method is used to expand group of columns to view the detailed data in the collapsed columns. Alternatively, you can click '**+**' button at runtime.

The following codes illustrates how to set these properties:

- To collapse group of rows

VB
```
c1OlapPage1.OlapGrid.CollapseAllRows()
```

C#
```
c1OlapPage1.OlapGrid.CollapseAllRows();
```

- To expand group of rows

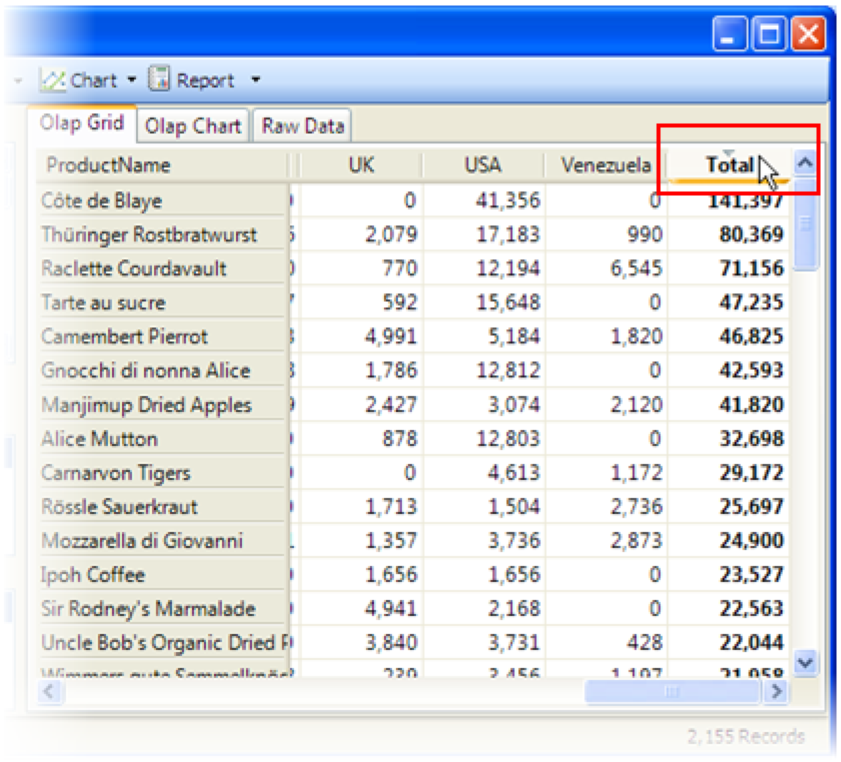
VB
```
c1OlapPage1.OlapGrid.ExpandAllRows()
```

C#
```
c1OlapPage1.OlapGrid.ExpandAllRows();
```

Similarly, properties for collapsing and expanding of group of columns can be set.

Sorting Olap Data

By default, results in the Olap output table are sorted by key, for example, "Argentina", "Brazil", and so on. This is not always the most useful way to show the data. Users may prefer to see the results sorted by sales value for example. To allow this, set the AllowSorting property on the C1OlapGrid to **True** (default). This will allow users to sort the data by clicking on the column headers, just like a regular grid. Clicking the header repeatedly changes the sort orders from ascending to descending to unsorted.



ProductName	UK	USA	Venezuela	Total
Côte de Blaye	0	41,356	0	141,397
Thüringer Rostbratwurst	2,079	17,183	990	80,369
Raclette Courdavault	770	12,194	6,545	71,156
Tarte au sucre	592	15,648	0	47,235
Camembert Pierrot	4,991	5,184	1,820	46,825
Gnocchi di nonna Alice	1,786	12,812	0	42,593
Manjimup Dried Apples	2,427	3,074	2,120	41,820
Alice Mutton	878	12,803	0	32,698
Carnarvon Tigers	0	4,613	1,172	29,172
Rössle Sauerkraut	1,713	1,504	2,736	25,697
Mozzarella di Giovanni	1,357	3,736	2,873	24,900
Ipoh Coffee	1,656	1,656	0	23,527
Sir Rodney's Marmalade	4,941	2,168	0	22,563
Uncle Bob's Organic Dried F	3,840	3,731	428	22,044
Wimmers gute Semmelknöd	220	2,456	1,107	21,958

Creating a Report

In the C1OlapPage control, you can set up and print a report using the **Report** menu at run time.

To create the report, follow these steps:

1. Click the drop-down arrow next to **Report** on the C1OlapPage ToolStrip.
2. Select **Options**. The **Document Options** dialog box appears.
3. On the **Page** tab, select a page **Orientation**, **Paper size**, and set the **Margins** as desired.
4. Click the **Header/Footer** tab.
5. Place the cursor in the header or footer text box where you want to add text or a predefined header/footer item.
6. Click one of the buttons on the toolbar to insert the desired field.
7. Click the **Report Content** tab.
8. Check the check box next to the items you want included in the report. You can also select a radio button to change the scaling of the grid or chart.
9. Click **OK** to close the **Document Options** dialog box.

Printing a Report

To print the report using the C1OlapPage control at run time, follow these steps:

1. Click the drop-down arrow next to **Report** on the C1OlapPage ToolStrip.

2. Select **Print**. The **Print** dialog box appears.
3. Choose a printer from the **Name** drop-down list and click **OK**.