ComponentOne OLAP

What is C1Olap

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.

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.

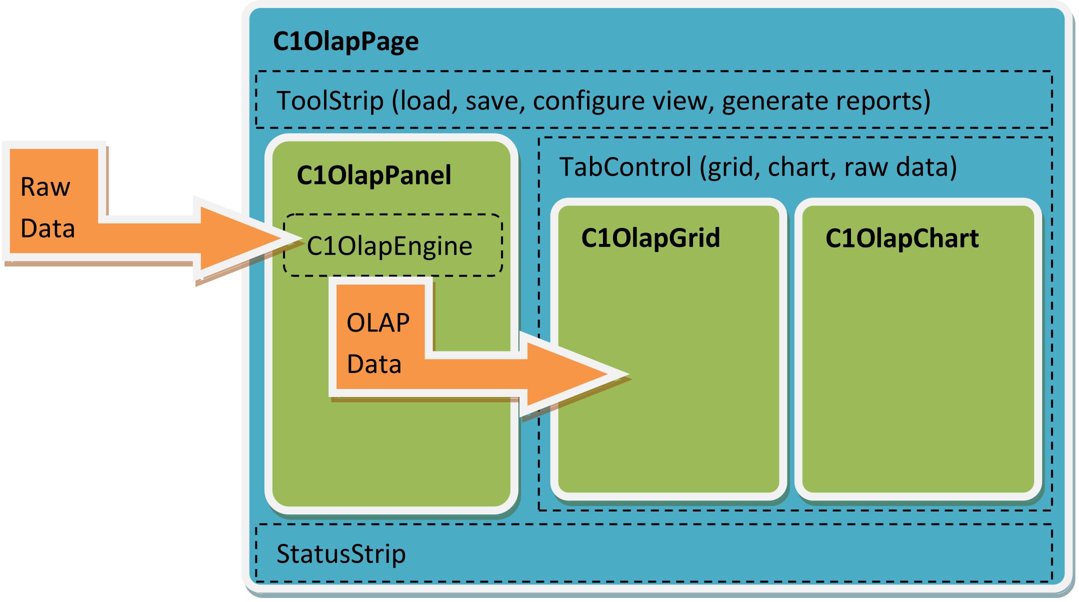
C1Olap Architecture

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 remove 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

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 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 a specific column in the 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, and 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 well.

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.

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.

C1OlapPrintDocument

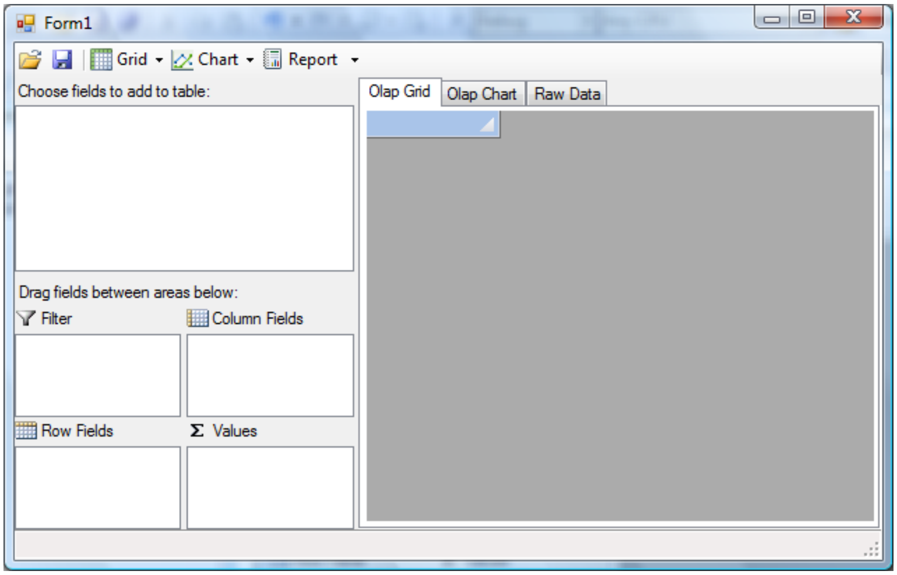
The **C1OlapPrintDocument** component is used to create reports based on OLAP views. It extends the **PrintDocument** class and provides properties that allow you to specify content and formatting for showing OLAP grids, charts, and the raw data used to create the report.

Quickstart

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

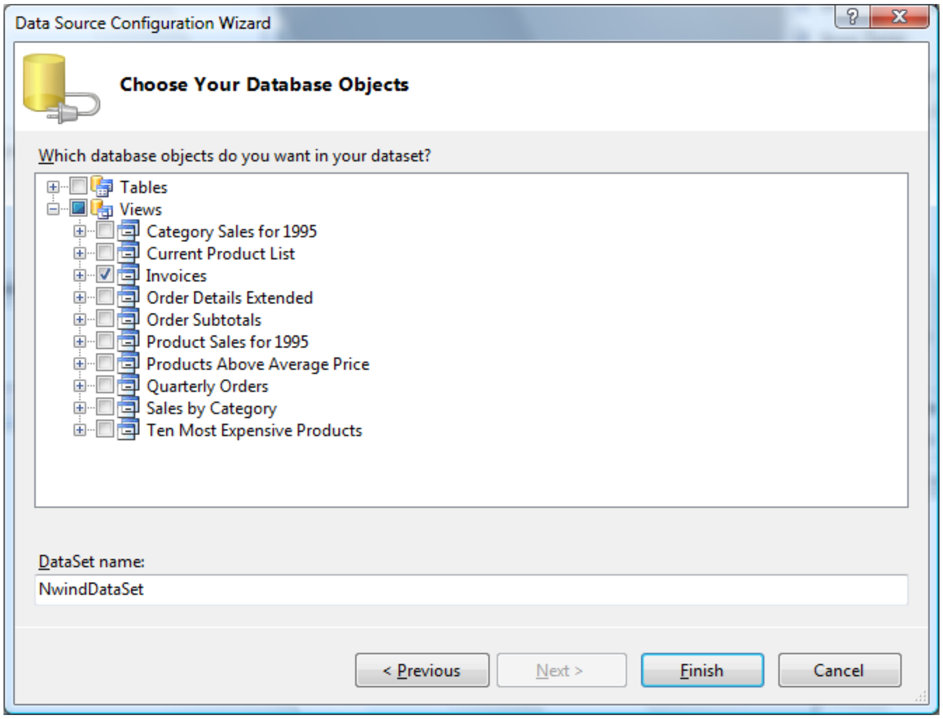
An Olap application with no code

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 small triangle that appears at the top right corner of the control. Use the 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:


Data Source Configuration Wizard
Choose Your Database Objects
< Previous
Next >
Finish
Cancel

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

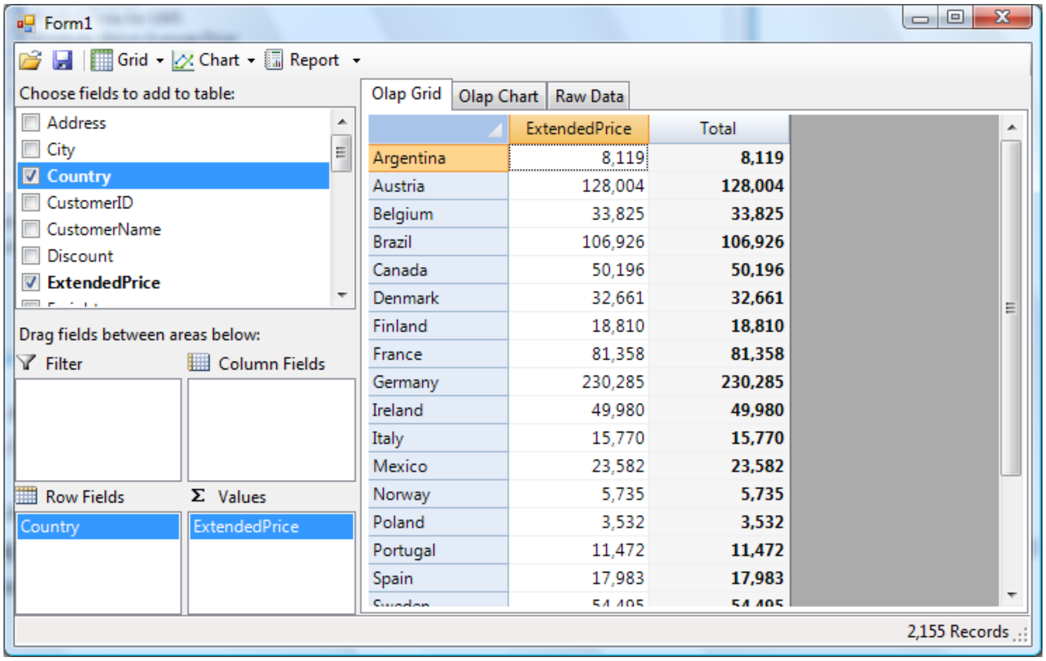
DataSet name: NwindDataSet

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
☐ CustomerName
☐ Discount
☑ ExtendedPrice

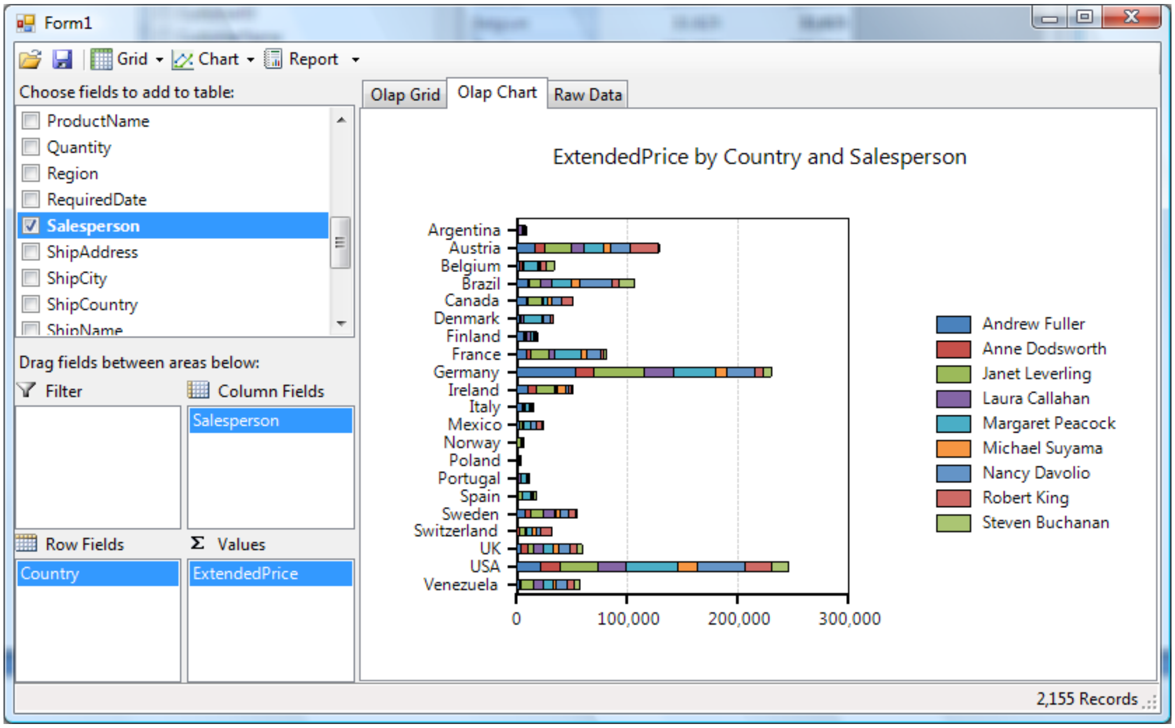
Row Fields: Country

Values: ExtendedPrice

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

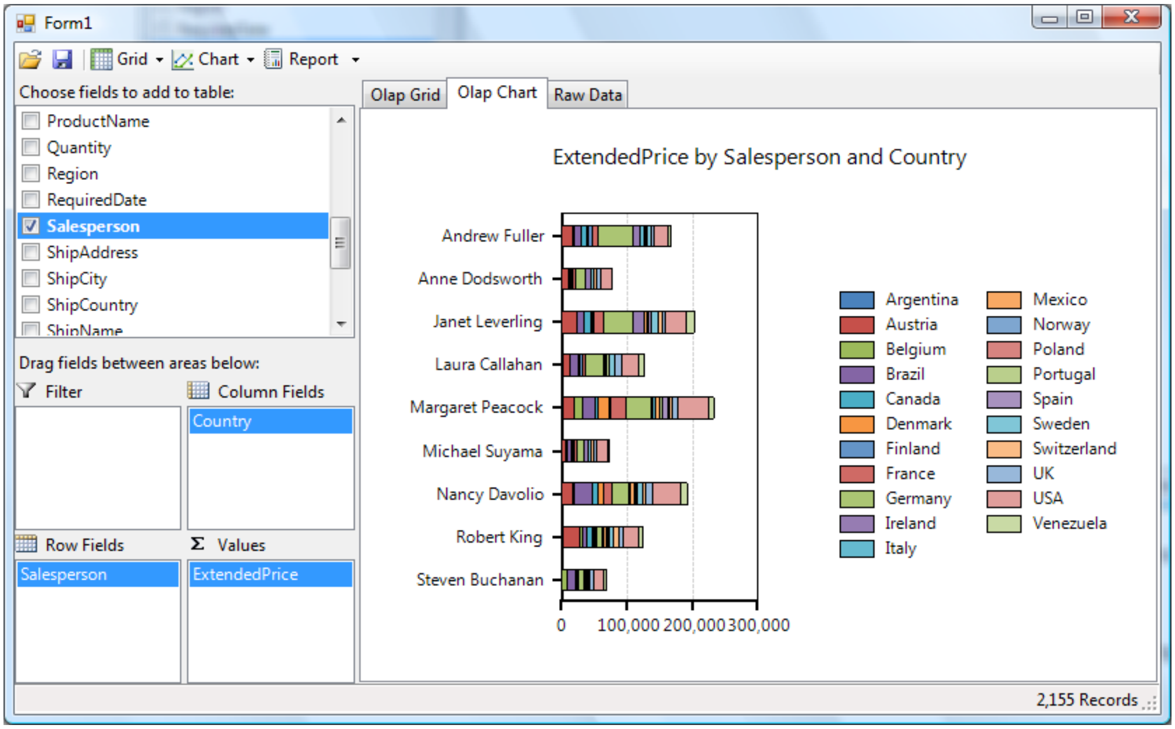
Click the "Olap Chart" tab and you will see the same data in chart format, showing that the main customers are the US, Germany, and Austria.

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:



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.

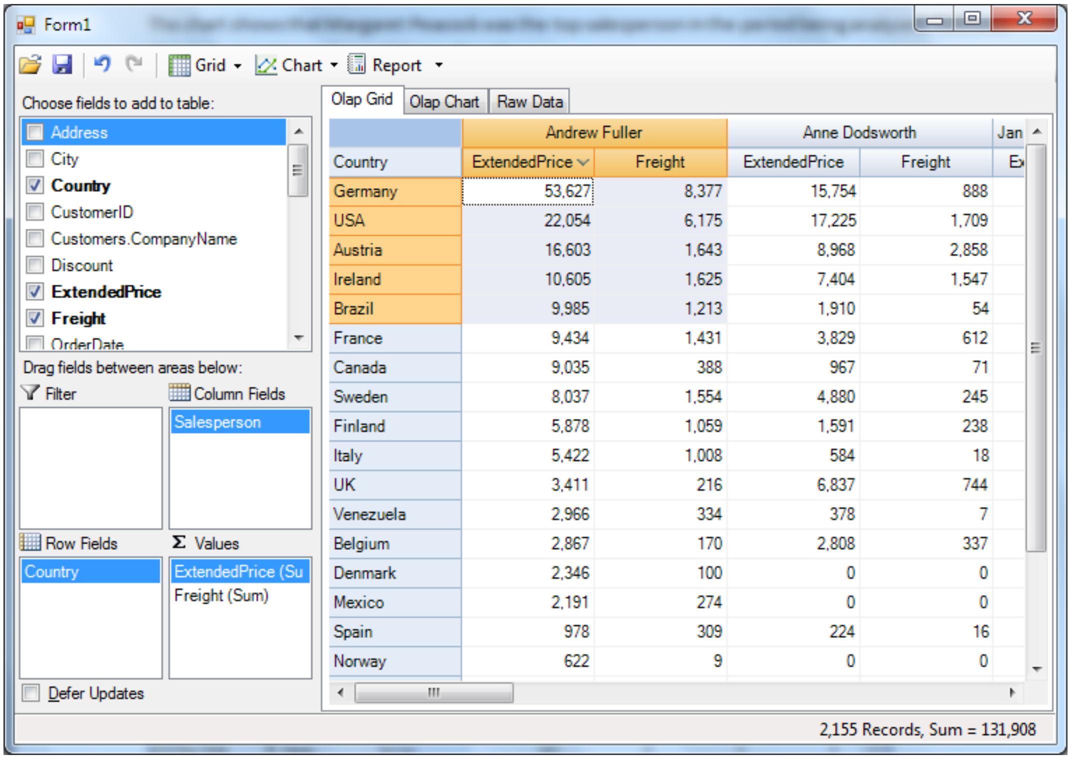
Multiple Value Fields

By default, **C1Olap** is configured to allow only one value field per view. When the user adds a value field to the **Values** list in the **C1OlapPanel**, the new field replaces any pre-existing ones.

In some cases, you may want to allow users to summarize multiple value fields at once. To enable this, you have to modify the value of the **MaxItems** property on the **C1OlapEngine.ValueFields** collection. For example, the code below initializes a **C1OlapPage** to display a summary of sales price and freight by country and by salesperson:

```
// get a reference to the olap engine
var olap = this.c1OlapPage1.OlapEngine;

// allow up to two value fields
olap.ValueFields.MaxItems = 2;

// summarize ExtendedPrice and Freight
olap.ValueFields.Add("ExtendedPrice", "Freight");

// by Country and by SalesPerson
olap.RowFields.Add("Country");
olap.ColumnFields.Add("SalesPerson");
```

The result is shown below:



Choose fields to add to table:

☐ Address
☐ City
☑ **Country**
☐ CustomerID
☐ Customers.CompanyName
☐ Discount
☑ **ExtendedPrice**
☑ **Freight**
☐ OrderDate

Column Fields: Salesperson

Row Fields: Country

Values: ExtendedPrice (Su
Freight (Sum)

☐ Defer Updates

	Andrew Fuller		Anne Dodsworth		Jan
Country	ExtendedPrice	Freight	ExtendedPrice	Freight	Ex
Germany	53,627	8,377	15,754	888	
USA	22,054	6,175	17,225	1,709	
Austria	16,603	1,643	8,968	2,858	
Ireland	10,605	1,625	7,404	1,547	
Brazil	9,985	1,213	1,910	54	
France	9,434	1,431	3,829	612	
Canada	9,035	388	967	71	
Sweden	8,037	1,554	4,880	245	
Finland	5,878	1,059	1,591	238	
Italy	5,422	1,008	584	18	
UK	3,411	216	6,837	744	
Venezuela	2,966	334	378	7	
Belgium	2,867	170	2,808	337	
Denmark	2,346	100	0	0	
Mexico	2,191	274	0	0	
Spain	978	309	224	16	
Norway	622	9	0	0	

2,155 Records, Sum = 131,908

Note that the **MaxItems** property can also be used to limit the maximum number of fields that the user can add to the **RowFields**, **ColumnFields**, and **FilterFields** collections. By default, **MaxItems** is set to 1 for the **ValueFields** collection and to -1 on all others (which means allow any number of fields).

Conditional Field Formatting

In many applications it may be useful to highlight cells on the grid according to the values they contain. For example, you may want to show values above a specific threshold with a green background or using a bold font.

C1Olap supports conditional formatting with three style properties available on the **C1OlapField** class:

C1OlapField.Style: This property contains a style that is applied by default to cells that display the field values. It is mostly useful in views that contain multiple value fields, so users can easily see which columns belong to each value field.

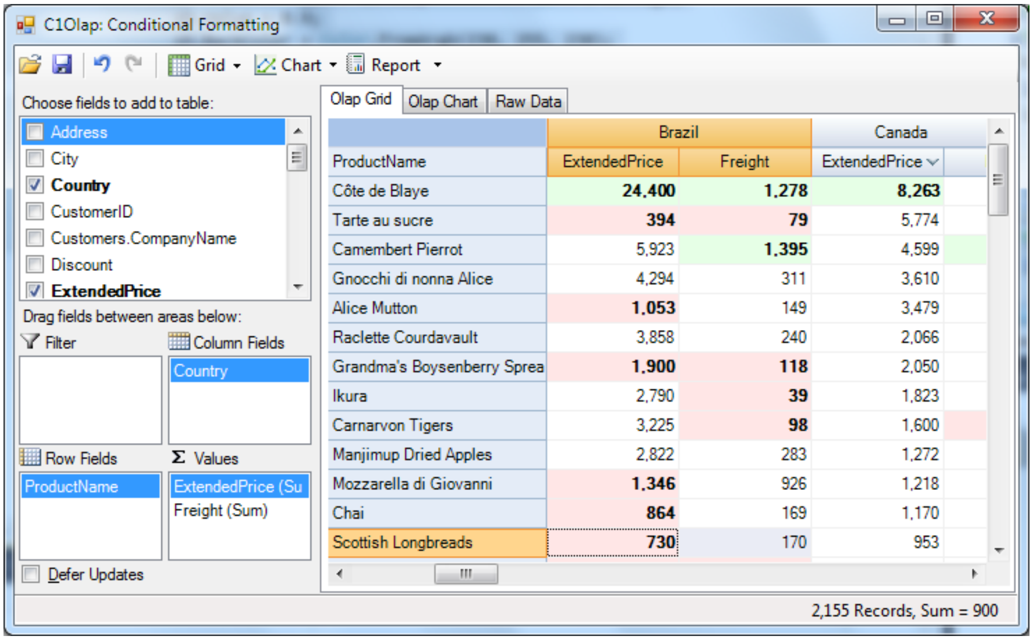
C1OlapField.StyleHigh: This property contains a style that is applied to cells that contain values above a specified threshold. The threshold may be a specific value (e.g. 1,000) or it may be specified as a percentage (e.g. 90%). It is often useful to specify thresholds as percentages because they automatically adapt to the range of values being displayed. For example, a high threshold specified as 90% will apply the style to values between 90 and 100 if the field contains values between 0 and 100, and will apply the style to values between 900 and 1,000 if the field contains values between 0 and 1,000.

C1OlapField.StyleLow: This property contains a style that is applied to cells that contain values below a specified threshold. Again, the threshold may be a specific value (e.g. 10) or it may be specified as a percentage (e.g. 10%).

The code below creates conditional styles for all value fields in the view and causes cells in the bottom 10% of the value range to be displayed in red, and cells in the top 10% of the value range to be displayed in green:

```
// set up conditional formats in code
foreach (var f in olap.ValueFields)
{
    // show bottom 10% values in bold with a red background
    var sl = f.StyleLow;
    sl.ConditionType = C1.Olap.ConditionType.Percentage;
    sl.Value = 0.1;
    sl.BackColor = Color.FromArgb(255, 230, 230);
    sl.FontBold = true;

    // show top 10% values in bold with a green background
    var sh = f.StyleHigh;
    sh.ConditionType = C1.Olap.ConditionType.Percentage;
    sh.Value = 0.9;
    sh.BackColor = Color.FromArgb(230, 255, 230);
    sh.FontBold = true;
}
```

The result is shown below:



Choose fields to add to table:

☐ Address
☐ City
☑ **Country**
☐ CustomerID
☐ Customers.CompanyName
☐ Discount
☑ **ExtendedPrice**

Drag fields between areas below:

Column Fields: Country

Row Fields: ProductName

Values: ExtendedPrice (Su
Freight (Sum)

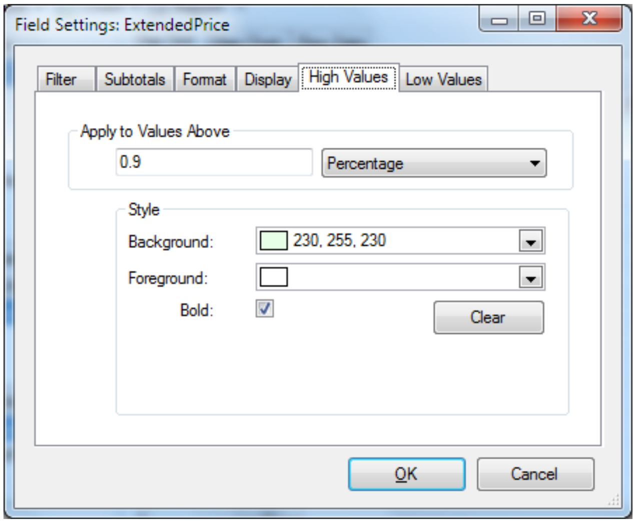
☐ Defer Updates

	Brazil		Canada
ProductName	ExtendedPrice	Freight	ExtendedPrice
Côte de Blaye	**24,400**	**1,278**	**8,263**
Tarte au sucre	**394**	**79**	5,774
Camembert Pierrot	5,923	**1,395**	4,599
Gnocchi di nonna Alice	4,294	311	3,610
Alice Mutton	**1,053**	149	3,479
Raclette Courdavault	3,858	240	2,066
Grandma's Boysenberry Sprea	**1,900**	**118**	2,050
Ikura	2,790	**39**	1,823
Carnarvon Tigers	3,225	**98**	1,600
Manjimup Dried Apples	2,822	283	1,272
Mozzarella di Giovanni	**1,346**	926	1,218
Chai	**864**	169	1,170
Scottish Longbreads	**730**	170	953

2,155 Records, Sum = 900

Conditional formats may also be created and edited at run-time by users. Right-clicking a field in the **C1OlapPanel** and selecting the "Field Settings…" option from the context menu brings up the field properties dialog, which contains tabs for the **Style**, **StyleHigh**, and **StyleLow** properties of the field.

This is what the field editor looks like:



Apply to Values Above: 0.9 Percentage

Style

Background: 230, 255, 230

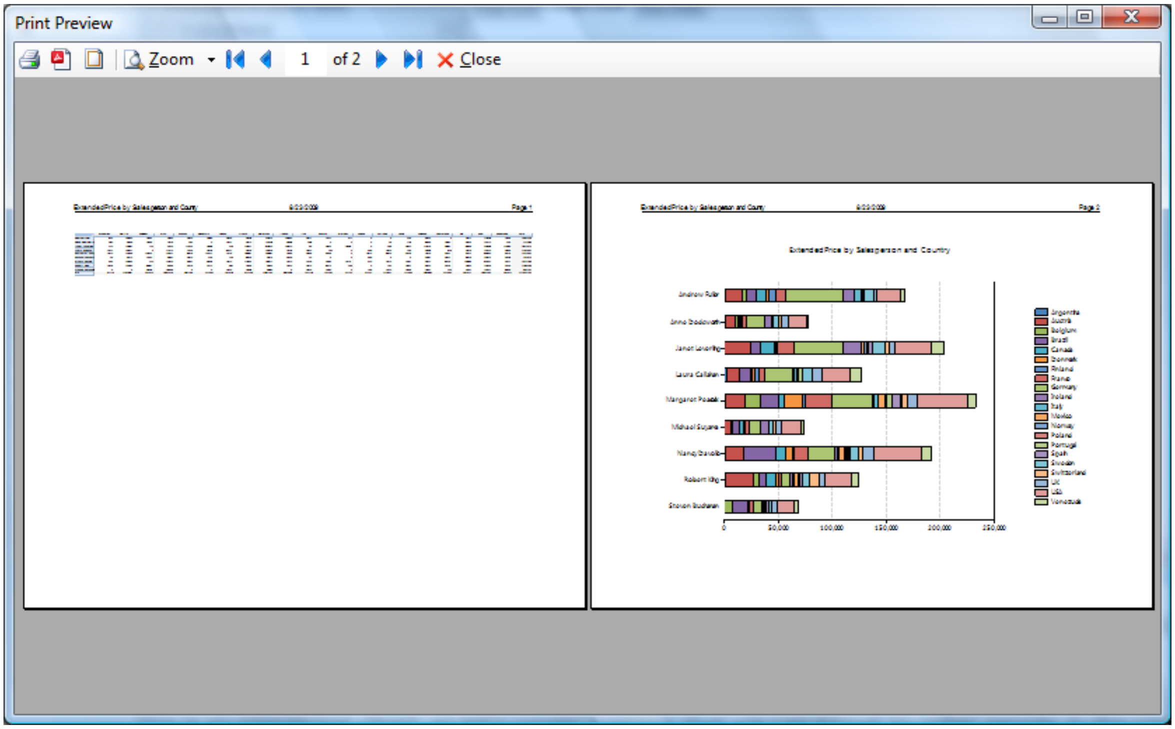
Foreground:

Bold: ☑

In addition to **C1Olap's** built-in conditional formatting capabilities, remember that the **C1OlapGrid** control derives from the **C1FlexGrid** grid, which means you can use all the **C1FlexGrid** features in your **C1Olap** applications. These features include owner-draw cells which allow for complete customization over how cells are displayed to the user. For example, you could easily customize the grid to shows icons for high and low values.

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 preview dialog, 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.

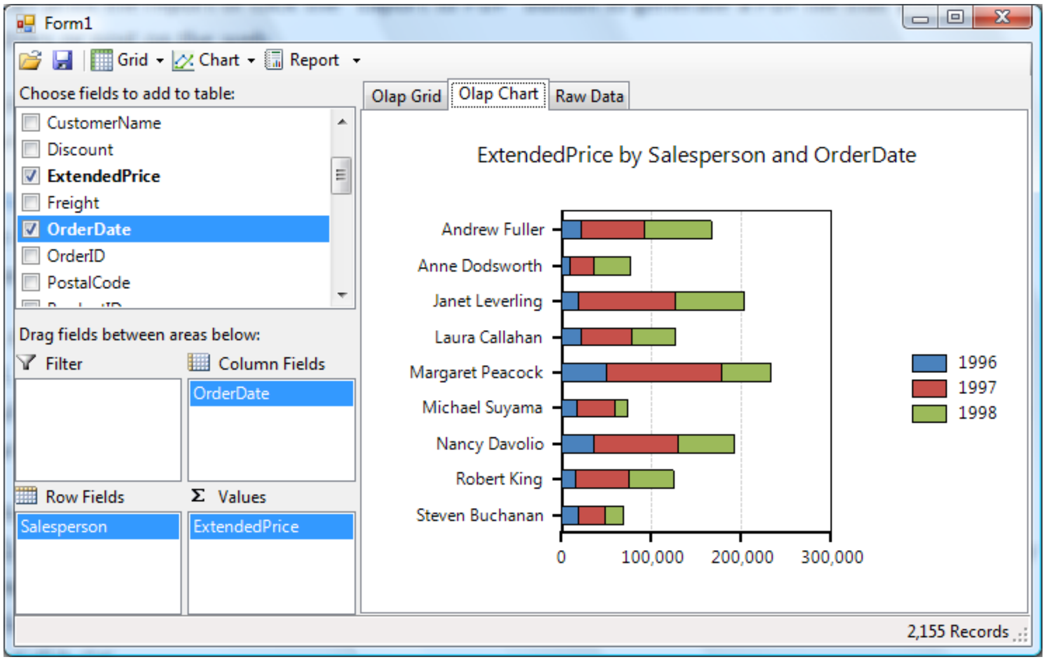
Summarizing Data

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.

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 select the “Field Settings” menu. Then select the “Format” tab in the dialog, 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:



Choose fields to add to table:

☐ CustomerName
☐ Discount
☑ ExtendedPrice
☐ Freight
☑ OrderDate
☐ OrderID
☐ PostalCode

Drag fields between areas below:

Filter:

Column Fields: OrderDate

Row Fields: Salesperson

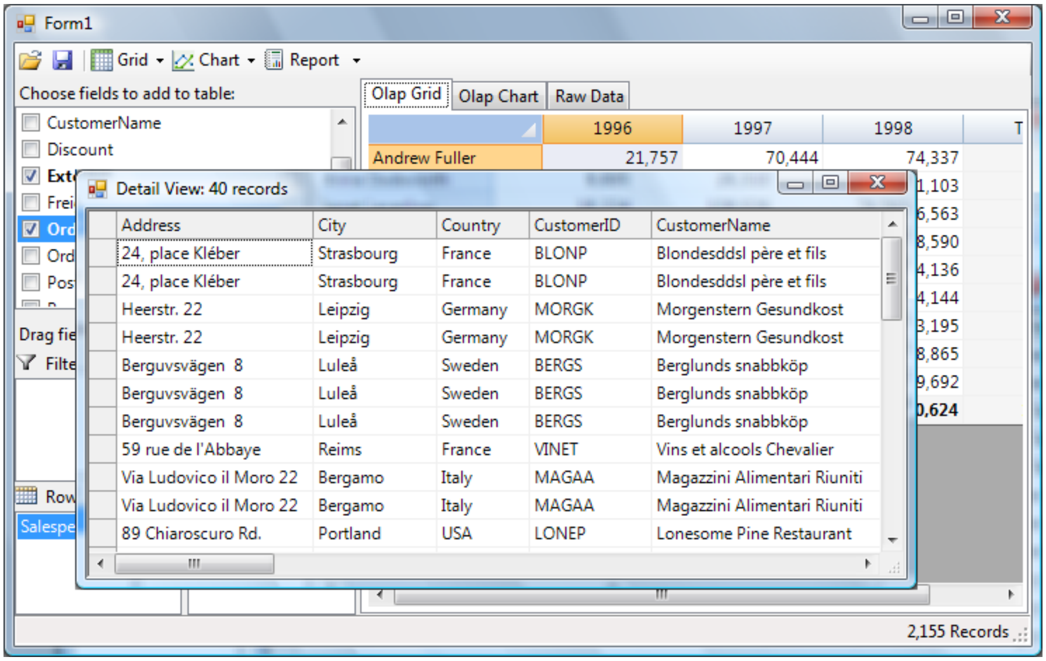
Values: ExtendedPrice

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 first cell on the grid, the one that represents Andrew Fullers’s sales in 1996. You will see another grid showing the 40 records that were used to compute the total displayed in the Olap grid:



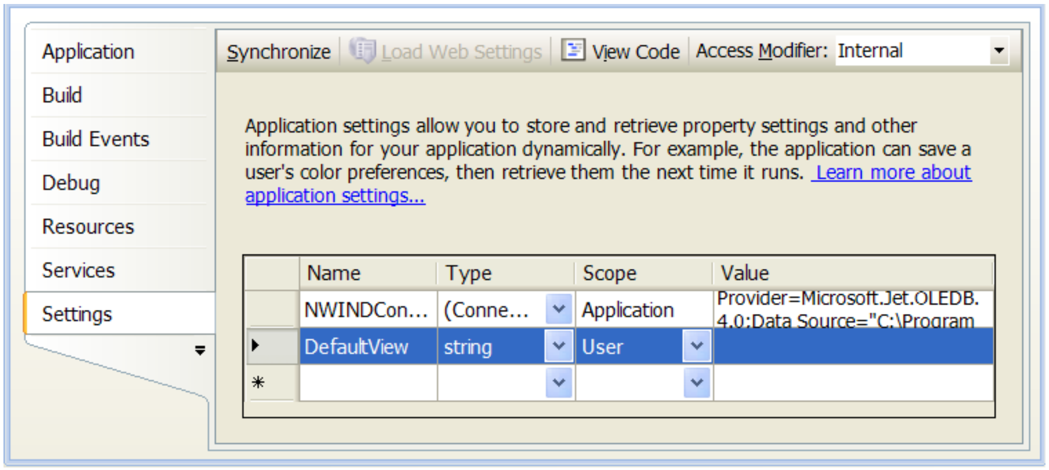
Address	City	Country	CustomerID	CustomerName
24, place Kléber	Strasbourg	France	BLONP	Blondesddsl père et fils
24, place Kléber	Strasbourg	France	BLONP	Blondesddsl père et fils
Heerstr. 22	Leipzig	Germany	MORGK	Morgenstern Gesundkost
Heerstr. 22	Leipzig	Germany	MORGK	Morgenstern Gesundkost
Berguvsvägen 8	Luleå	Sweden	BERGS	Berglunds snabbköp
Berguvsvägen 8	Luleå	Sweden	BERGS	Berglunds snabbköp
Berguvsvägen 8	Luleå	Sweden	BERGS	Berglunds snabbköp
59 rue de l'Abbaye	Reims	France	VINET	Vins et alcools Chevalier
Via Ludovico il Moro 22	Bergamo	Italy	MAGAA	Magazzini Alimentari Riuniti
Via Ludovico il Moro 22	Bergamo	Italy	MAGAA	Magazzini Alimentari Riuniti
89 Chiaroscuro Rd.	Portland	USA	LONEP	Lonesome Pine Restaurant

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, 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);
}
asdasdasdas
```

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 crated and saved all the views, create a new XML file called "DefaultViews.xml" with a single "OlapViews" node, 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 **C1Olap**). Your XML file should look like this:

```xml
<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, right-click the project node in the solution explorer, select the "Properties" item, then the "Resources" tab, the "Add Resource", then "Add Existing File…" option. Select the XML file and click OK.

Now that the view definitions are ready, we need to exposed 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)
{
    // no changes here
    // …

    // 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 **C1OlapPage.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 the 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 nwind 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. 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, then join “Products” using the “ProductID” field, 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 **C1OlapPage.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 **GetSqkStatement** 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, then add the following code to the form:

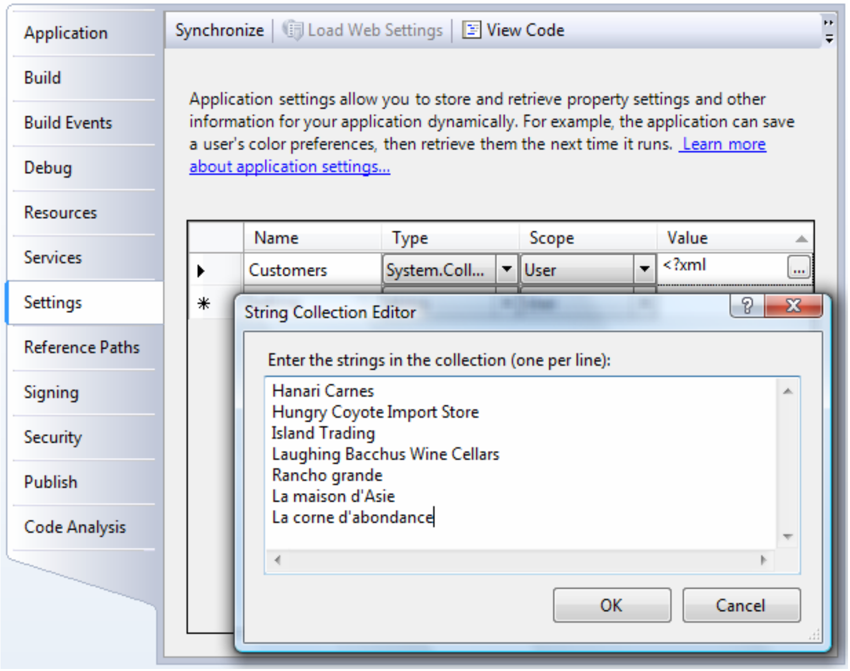
```
public partial class Form1 : Form
{
  List<string> _customerList;
  List<string> _activeCustomerList;
  const int MAX_CUSTOMERS = 12;
```

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 list but compact list. It contains only the customer name, not any of the associated details such as orders, order details, etc. 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 customer 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”, 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

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 **C1OlapPage.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 **C1OlapField.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);
}
```

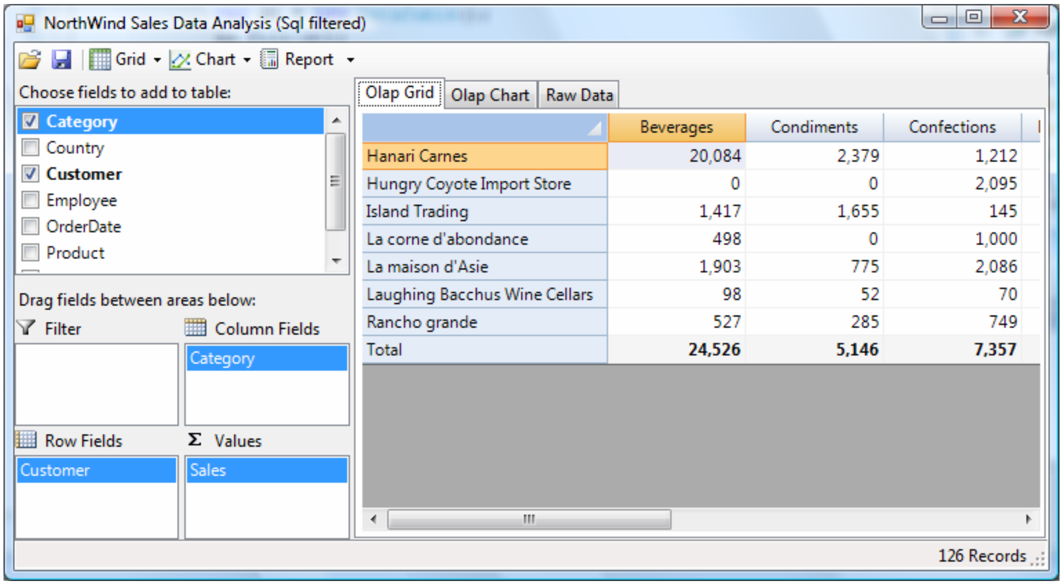
The code starts by retrieving the field’s **Filter** and then calling the filter’s **Apply** method to build a list of customers selected by the user. After some bounds-checking, the list is passed to the **CachedDataTable** which will retrieve any missing data. After the new data is loaded, the **C1OlapPage** 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)
Grid
Chart
Report
Choose fields to add to table:
Drag fields between areas below:
Filter
Column Fields
Row Fields
Values
Olap Grid
Olap Chart
Raw Data
126 Records

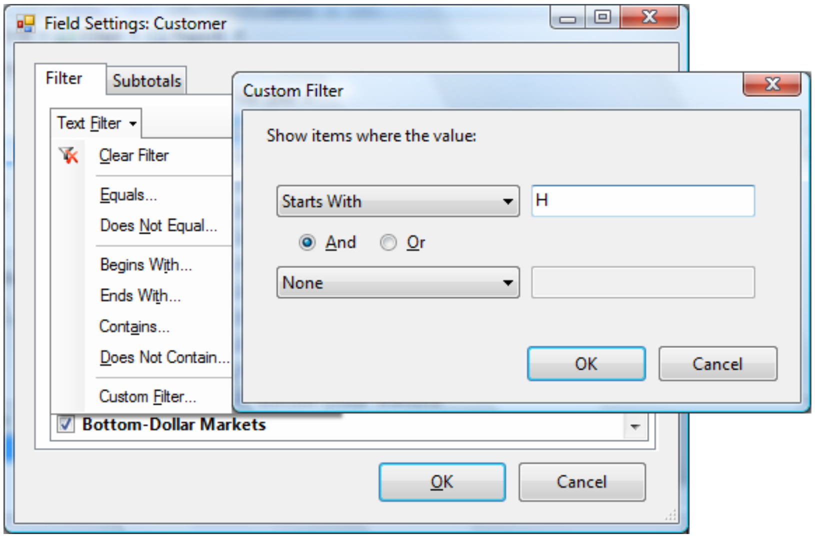
☑ Category
☐ Country
☑ Customer
☐ Employee
☐ OrderDate
☐ Product

Column Fields: Category
Row Fields: Customer
Values: Sales

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

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:



Show items where the value:

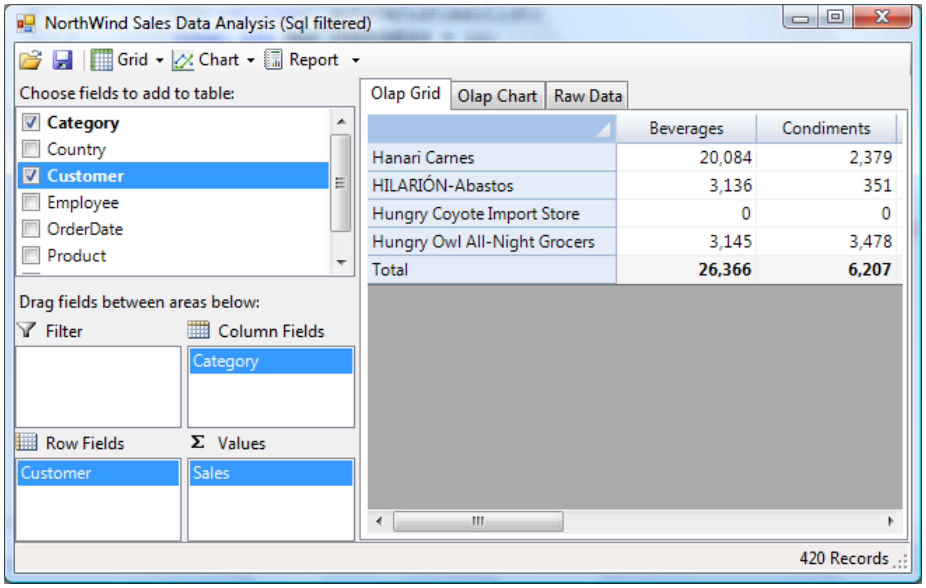
Starts With: H

● And ○ Or

None

☑ Bottom-Dollar Markets

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:



Choose fields to add to table:

☑ Category
☐ Country
☑ Customer
☐ Employee
☐ OrderDate
☐ Product

Column Fields: Category

Row Fields: Customer

Values: Sales

	Beverages	Condiments
Hanari Carnes	20,084	2,379
HILARIÓN-Abastos	3,136	351
Hungry Coyote Import Store	0	0
Hungry Owl All-Night Grocers	3,145	3,478
Total	**26,366**	**6,207**

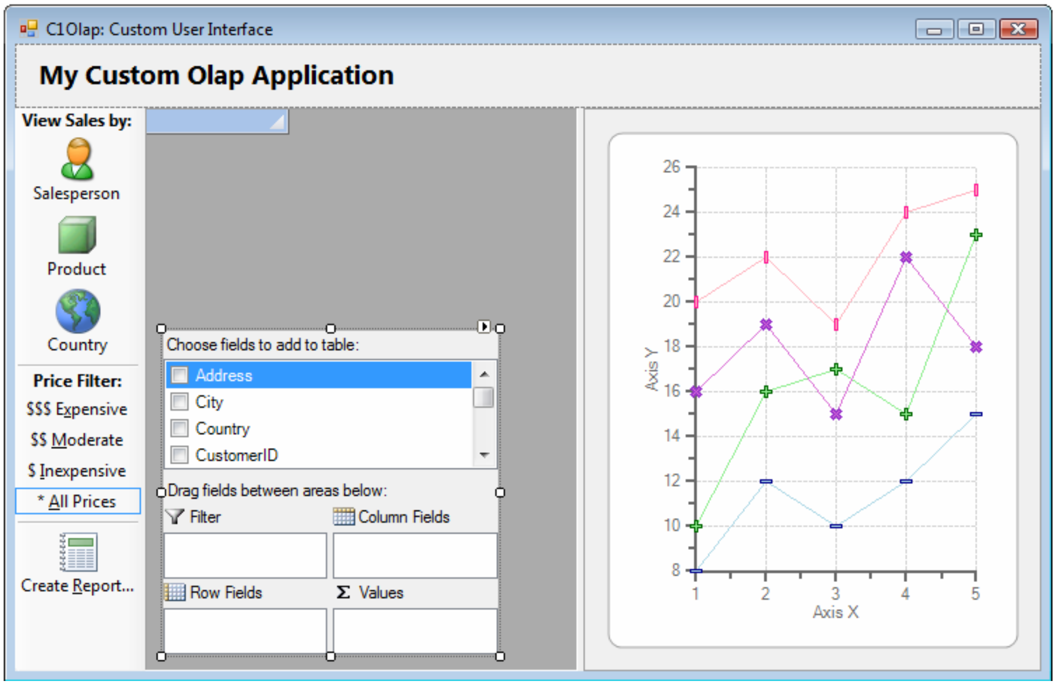
420 Records

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 if 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** 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, 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 updadtes
    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**, **ColumnFields**, **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.

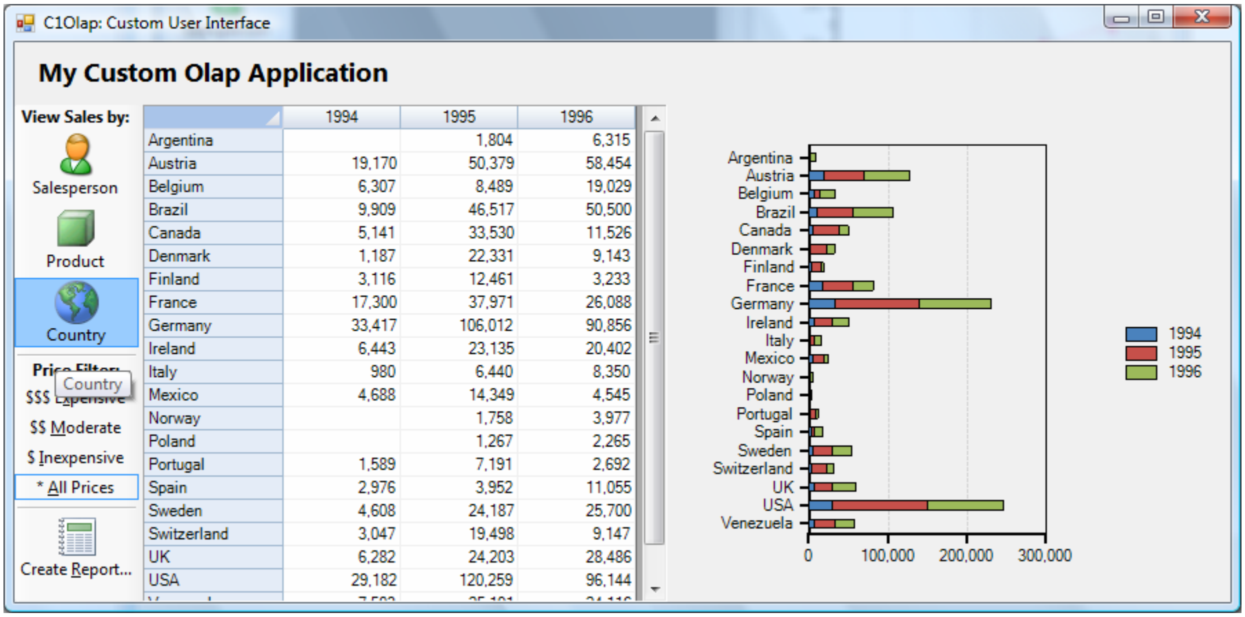
Finally, the code calls **EndUpdate** and sets the **FooterText** property of the **C1OlapPrintDocument** so it will be automatically displayed in any reports.

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.

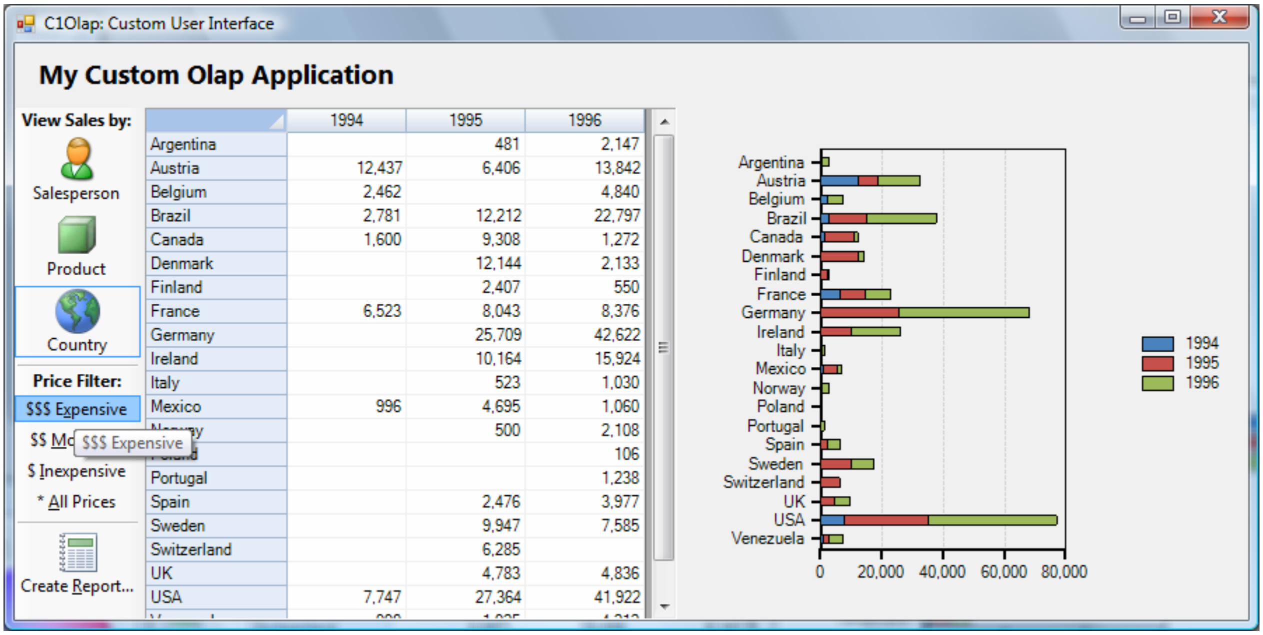
The application is almost ready. You can run it now and test the different views and filtering capabilities of the application, as illustrated below:



	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

View 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
[illegible]		500	2,108
[illegible]			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

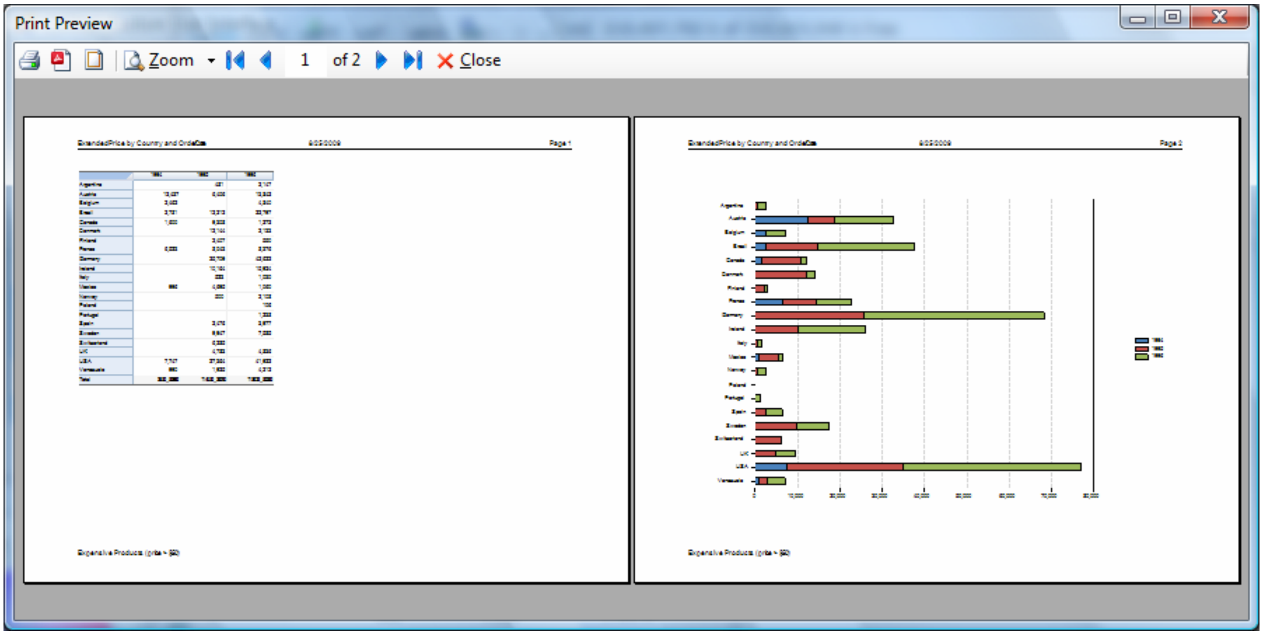
The last piece missing from the application is reporting. Users can already copy data from the **OlapGrid,** paste it into Excel, and print or save the results. But we can make it even easier, by allowing them to print or create PDF files directly from within the application.

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’s **Document** property, initializes its position, and shows the dialog. If you run the application now and click the “Report…” button, you should see a dialog like the one below:



From this dialog, 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 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:

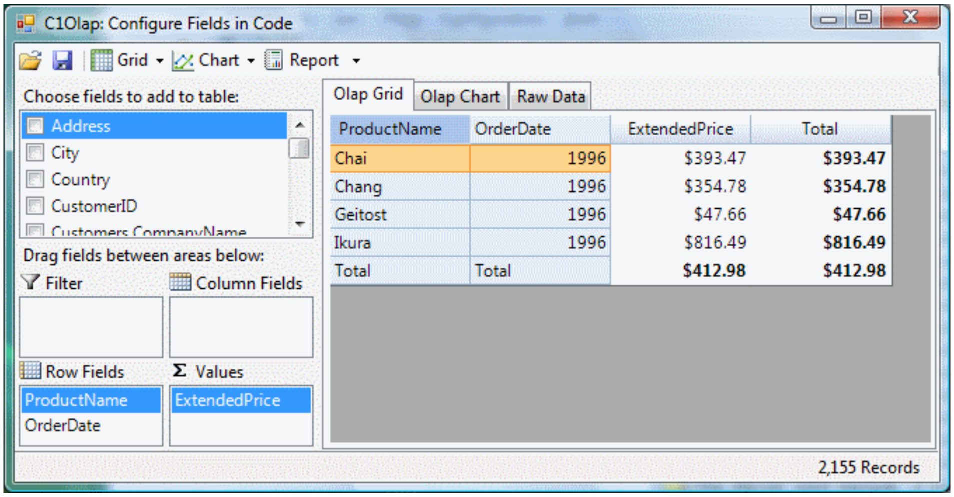
```
public Form1()
{
  InitializeComponent();

  // 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:


C1Olap: Configure Fields in Code
Grid
Chart
Report
Olap Grid
Olap Chart
Raw Data
Drag fields between areas below:
2,155 Records

Choose fields to add to table:

- ☐ Address
- ☐ City
- ☐ Country
- ☐ CustomerID
- ☐ Customers CompanyName

Filter

Column Fields

Row Fields
- ProductName
- OrderDate

Values
- ExtendedPrice

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