Crystal Xcelsius

Best Practices and Workflows for Building Enterprise Solutions

Overview

This document is a collection of best practices and workflows for building Xcelsius enterprise solutions. This document presumes a good understanding of both Xcelsius modeling and universe design.

Contents

Best practices

Spreadsheet design

- Plan ahead. Think about the data and components you want to use to visualize this data.
- Name the tabs something other than Sheet1, Sheet2, etc. Names make logic spread across multiple tabs easier to locate.
- Place directly related content on the same tab. Place less related content on different tabs.
- Place your initial content ~25 rows from the top and ~5 columns to the right of the A column. This leaves room for control logic you will probably add later.
- Logical content should flow in a ***rightward*** direction.
- Query result content should flow in a ***downward*** direction because it allows you to use the row attribute in the Query as a Web Service query definition. This functionality enables you to configure the output of a multi-column query without having to configure each column individually.
- Do not data enter any content into the spreadsheet that could otherwise be fetched from the database using a query. The spreadsheet should only contain visual indicators of
    - query and insert cell ranges (where the data goes, using color or numbers).
    - display cell ranges (what data is displayed by the component).

  There are exceptions to this rule (but not many). Column headers, in mono-lingual solutions, are one of the exceptions. In general, the more data contained within the spreadsheet, the poorer the performance of the model (assuming good database performance).

Xcelsius model design

- Avoid using a font size smaller than 12 point. If the required content exceeds the available canvas space, use dynamic visibility to not display the components at the same time.
- Use a small number of different components to preserve UI consistency.
- Use a consistent font and color scheme.
- Use a 1 second fade-in entry effect on all components for better visual effect.
- Include a toggle and dynamic visibility for Help text, displaying instructions on how to use the solution.

- Use a radio button or label-based selector when the number of values to select from is small. These two components require only a single click to make a selection. Most of the other components require two or more clicks to make a selection.
- For list of values (LOV) and fact data ranges that have the potential to grow over time, consider adding 10% more cell rows to the end of both the query target range as well as to the selector or display component range. To hide the fact that these extra cells exist within the component ranges (such that the blank cells don't display), select the **Ignore Blank Rows** option.
- In the Object Browser, name each component, referencing its purpose.

Query design

These best practices generally apply to either Query as a Web Service or Live Office. The techniques for building Xcelsius solutions differ *quite dramatically* from the best practices used in reporting. With Business Objects reporting products, you may build queries that fetch hundreds or thousands of rows and do one or more of the following with those result sets:

- Aggregate within the report, using report variables. This approach is common with Crystal Reports.
- Aggregate within the report, using the microcube as the data source. This approach is common with Web Intelligence documents.
- Drill down from summary content to detail content, navigating the hierarchy within the microcube, never accessing the database a second time.
- Create multi-page or multi-tab reports, fetching content for every single tab and page, but not actually view every page or tab in that report
- Schedule the reports to run in advance of their actual usage.
- Burst and distribute personalized versions of the same report to thousands of users.

These are all possible because Business Objects reporting products not only display data, but they can also store and process data.

However, Xcelsius cannot process data in the same way as a reporting product nor can it store very much data. It's purely a visualization tool that requires nearly all data it displays to be delivered to it ***presentation-ready*** by the queries it runs. The following best practices take these issues into consideration:

- Fetch only the data needed for the display component, ***at that specific point in the users viewing sequence***, and no more. Do not 'overfetch', in anticipation of what a user ***might*** view. In each query, fetch from the database only what the user ***will*** view, at that point in time.

- Highly parameterize and/or highly aggregate every query in the model to reduce result set size.
- Fetch only the data needed at the level of aggregation at which it is need it at. If a lower level of detail is later required, fetch it using a subsequent query.
- Drill down to additional detail by passing the parent selection into a child query that uses the selection as input.
- Instead of building fewer queries containing more rows and columns (in typical Web Intelligence fashion), build more queries containing fewer rows and columns. Make every user click a query, if necessary.
- Do nearly all calculation logic in the database (during the ETL process) or universe (with database functions), not the Excel layer. Models will perform better as a result.
- Optimize the underlying database for pure query speed, modeling the database specifically to the dashboard solution it will support. If the database is too slow, build another, better and faster one (or use a cube).
- Never retrieve more than ~500 rows in a single query. Flash cannot handle result sets much larger than this. Find a way to parameterize, aggregate, or subdivide the query to reduce the result set size.
- If requirements call for more than 500 rows, form an OpenDoc URL and use a URL link component to pass the request to a prompted Web Intelligence or Crystal Report.

Basic workflows

Described below are a series of workflows that can be used to optimize model ease of use and appearance. These workflows are often a combination of universe, query, spreadsheet, and Xcelsius techniques. The workflows are described in increasing order of complexity.

NOTE	These workflows presume the data is in some form of a star, snowflake or merged fact and dimension table schema. All of the SQL examples use Microsoft SQL Server syntax and will work on other databases by altering the functions used. For any of the workflows requiring advanced universe modification, OLAP sources are not supported. In many cases however, the same result can be achieved within the query panel.

Standard workflow

Here is the generic workflow that is possible on nearly any available data:

1. Build a universe as you would normally for reporting:
   i. Join facts to dimensions.
   ii. Create dimension, measure and filter objects.
2. Identify dimensions that could be used with Xcelsius selector components.
   - Select one or more dimensions with fewer than ~50 unique values.
   - Dimensions like product and geography work well.
3. Identify dimensions to be used in Xcelsius display component(s):
   - Time dimension for time series analysis.
   - Low cardinality dimensions for ranked display.

NOTE	A display dimension is optional for score carding and other single value display components like gauges, trend indicators, and indicator lights.

4. Identify the measure(s) to be used within the display components.
5. Create the spreadsheet that controls the interactivity using the best practices previously described:
   i. Create username and password cells and hard code the BusinessObjects Enterprise values initially.
   ii. Create target ranges for each list of values.
   iii. Create insert cells for each of the selectors.
   iv. Create target ranges for the fact query results.
   v. Create a concatenated key combining all the selected dimension values. This key will be used to trigger the fact table query.
6. Create one query per dimension value to be used as LOV for the display selectors:
   i. Make sure LOV queries produce distinct values and not any repeating values.
   ii. Again, make sure the LOV is of a reasonable size: fewer than 50 values. Longer LOV are too cumbersome to scroll.
7. Create a query for the facts, to be used to display measure values
   - Displaying as result objects, the display dimension and measures.

- Prompting for each of the dimension values used in the selectors.

8. Design the Xcelsius model:
    i. Load the spreadsheet.
    ii. Add and configure the Web Services Connector components (WSC):
        a. Use one WSC per query.
        b. Use the WSDL URLs generated by Query as a Web Service.
        c. For the LOV queries, set the output values to the LOV target ranges.
        d. For the fact query, set the output value to fact table target range.
        e. For the fact table query, set the input values to each selection target.
        f. For LOV queries, set to Refresh On Load.
        g. For the fact query, set to Refresh on Insert of the concatenated trigger key.
        h. Hide the WSC components, using dynamic visibility = a blank cell.
    iii. Add and configure the selector components.
        a. Configure each to display the LOV query target range.
        b. Configure each to insert a label into the selection target cell.
    iv. Add and configure the display components. Configure each to display the fact query output target range

This workflow can be enhanced in a variety of ways:

1. Add more fact queries and display components on the same screen. The selectors can be reused, passing their selections to these new fact queries, even if those fact queries use a completely different data source. Relational and OLAP content can easily be integrated into the same model.

2. Add more queries and components, conditionally displayed, using dynamic visibility.

3. Link to a prompted report. Using the selections, an OpenDoc URL can be formed. The OpenDoc URL can be called using a URL link component to enable context sensitive drilling to detail in a Crystal or Web Intelligence report.

Unique lists of values for selectors

Unique LOV are required when using many of the selector components in Xcelsius. LOV are commonly used with combo boxes, radio buttons, and label menus, for example. To make the LOV dynamic (and/or secure), queries to display the values are typically created.

If when creating a query, the results produce no duplicate values (because the source has no duplicates for that dimension value), no query modification is required. This is often the case with snowflake schemas, where each dimension value resides in its own table. If however the query for a dimension values produces duplicate values (for example, country is repeated because the query uses the Customer table as its source), a SELECT DISTINCT is required to produce a unique list of values.

Currently, the Query as a Web Service query panel is not able to add the DISTINCT condition to queries it creates. To work around this issue, force a GROUP BY onto the query. Simply add any measure object to the query producing the LOV. The grouping will make the list of values unique. Because the measure object used to force the grouping does not actually need to be used in the Xcelsius model, do not configure a target cell for the measure in the worksheet.

An alternative approach

If no measure objects exist or the query performance to aggregate the measure value is too slow because the summarization source is large, an alternate approach is as follows:

1. In the universe, create a measure object containing this SQL statement:

   ```
   COUNT(*)
   ```

2. Click the **Tables** button and choose the same table that the column used to source the dimension value is in.
3. Use this measure, along with the dimension value in the LOV query.

The result is this query:

```
Select dimension_value, count(*) from dimension_table group
by dimension_value
```

This approach is useful in star schemas where dimension tables contain repeating values for certain dimensions.

An ‘all values’ selection

In some situations, a user wants to ignore a particular selector. For example, the user might want ‘All Products’ or ‘All Employees’ as opposed to selecting a specific product or employee. Since the query engine does not support the notion of optional prompts, there are two options for solving this problem:

- Create one query for each dimensional selector combination. For a three-dimensional selection, the query prompt combinations are as follows:
    - Product, Customer, Time
    - Product, Customer
    - Product, Time
    - Customer, Time
    - Product
    - Customer
    - Time
    - No prompts

  Creating eight different queries to achieve this result is possible but not optimal because of the amount of query setup. Additional spreadsheet logic would also be required to isolate which of the eight queries to run based on the selections a user has made.

- Create intelligent filters that make an 'All' selection possible. This can be accomplished with a single query and multiple universe filters. Complete these steps:
    1. Create one universe filter per dimension.
    2. Within the filter, use SQL like this:
       ```
       Product = @Variable('Product') or
       @Variable('Product') = 'All Products'
       ```
    3. In the spreadsheet, hardcode the 'All Products' value and ensure that it is matched to the 'All' value in the universe filter.
    4. Place the values for that same dimension (either using a query or by hard coding) directly below the 'All Products' cell.
    5. Point the display component (that is, a combo box) to the entire range of values, inclusive of the All value as well as the actual values.
    6. Set the component default value to the **All** choice.

  The result is that when the All selection is made, the resulting query predicate contains `'All Products' = 'All Products'`. Because this is the logical equivalent of 'get everything', all values for that dimension are included in the query. The prompt is not actually ignored; it just has no constraining ability. When a non-All selection is made, the query works in the usual way.

The 'some but not one or all' selection challenge

Currently, while it is possible to create queries containing the IN operator, the Web Service component in Xcelsius does not support multiple cell values for a single query input parameter. For example, there is currently no way to let a user select multiple values from a list (for example, 'I want Red and Green products but not Blue products') and submit that request to a Web Services query with an IN list operator. Only the first value in the range will be recognized. Below are a few workarounds to this issue.

The list builder workaround

One workaround to this challenge is to use a list builder component in conjunction with an associated prompted Web Intelligence, Crystal, or Desktop Intelligence report. By using a URL link component with OpenDoc syntax, the multiple values generated by the list builder can be passed to the report. While the results are not displayed with Xcelsius components, the selection is made with Xcelsius.

One limitation to this approach is that the OpenDoc URL cannot exceed 1024 characters in length. This is an Excel string length limitation. This limitation can be managed by using keys instead of long descriptions in the report prompts and query string.

The checkbox workaround

An Xcelsius-only workaround is to use a series of checkbox components, each representing a single dimension value. For example, using three checkboxes to represent a country, all, one, or two countries could be selected. These selections could then be used in conjunction with a universe SQL filter, such as the following:

```
dbo.store.store_country in (@Variable('USA'),
@Variable('Canada'),@Variable('Mexico'))
```

Each checkbox is configured to return either its dimension value when checked or '0' (or other non-dimension value) when unchecked to a single cell. For example, when the USA checkbox is selected, "USA" is inserted into the target cell). The result is that zero, one, two, or three checkboxes can be checked and the proper rows will be returned by the query in all cases. For example, when the USA and Mexico checkboxes are checked, the SQL becomes the following:

```
dbo.store.store_country in ('USA',’0’,'Mexico')
```

The limitations with this workaround are as follows:

- Individual checkboxes for each dimension values must be configured. This is labor intensive where there are many dimension values.
- When new dimension values appear in the data or are removed, checkboxes must be added or removed from the model.
- Changes in the checkboxes require changing the universe Filter as well.

As a result, this approach should be used on dimensions with a limited number of relatively static dimension values (for example, 20 or fewer).

Content display control

Xcelsius components display the data returned by a query until that data is replaced by the data from a subsequent query. This product functionality causes certain issues. For example:

- Initial query returns sales for Red products (because Red is the initial value selected) into a chart component.
- User changes the selection from Red to Blue products.
- Until the query completes (a few seconds or more), the chart component displays the sales for Red products yet the selection indicates Blue products. This creates uncertainty and confusion: “Is the query done?” “I’m I looking at Red Sales or Blue Sales?”

In parameterized report queries, returning as a results object the same object that was parameterized is not very common. For example, if the report prompted the user for a country, it would be unnecessary to also return the country object as part of the result set because each row in that result column would be the same.

Returning the value prompted for serves one purpose. The presence of the value indicates that the query has completed as requested. Query completion can then be used to control whether other related content is displayed. Below is sample workflow to illustrate how this accomplished:

1. Create a query that prompts for country.
2. In that same query, also return the country object.
3. Target the country return value to a single cell because it always will return a single value.
4. Create a display control cell, referencing the cell a selection is inserted into (the selected_cell), as well as the cell containing the value returned from the query:

   ```
   =if(Country_Selected = Country_Returned,1,0)
   ```
5. For the cells used by the display component (a chart, for example), use logic like this in every cell in the display range:

   ```
   =if(Country_Display = 1,data_cell,””)
   ```

For display ranges impacted by multiple changing dimension values, use the **AND** function:

```
=if(AND(Country_Display = 1, Product_Display = 1
Month_Display = 1),data_cell,””)
```

The result of this additional logical control is that the moment a selection change is made by the user, the displayed content is hidden (though it is still resident in the model), eliminating the possibility of end user confusion. When the query actually completes, the selector matches the selection returned and the data displays as expected.

An alternate approach is to hide the data content based on the query load status of the queries providing data to the component. The effect is similar to the above approach, using this syntax:

```
=if(Query Load Status = ‘idle’,data_cell,””)
```

Delayed query triggering

Interactive analysis often requires that queries be triggered by the selection of a value delivered by a previous query. For example, initial query, set to refresh on load, returns customer and sales. The subsequent query prompts for customer, returns products and sales for the selected customer, and is triggered on a change to the selected customer.

In most cases, this workflow does not need any special treatment. In some circumstances however (that is, when the model first opens), the second query will trigger ***before*** there is an actual value to fill the prompt. In those cases, the second query will run but will return no rows because no match for a null value was found.

One workaround is to introduce an intermediate query, between the two queries. The purpose of this query is to introduce a short delay in the triggering sequence to enable the input value for the second query to be available before the query is triggered. This intermediate query can be as simple as a returning a timestamp from the database, and can be done by creating a query with an object containing this SQL. For example::

```
Max(getdate())
```

The revised query triggering sequence then becomes as follows:

1. Initial query is set to refresh on load.
2. Default value displayed in a component is inserted into target cell.
3. The timestamp query is configured to trigger on a change to the target cell.
4. Timestamp value is inserted into a cell.
5. Final query is configured to trigger on a change to the timestamp cell, using the target cell as input.

This intermediate query introduces sufficient latency to insure that the values necessary to run the second query are consistently available in the input cells.

Dynamically cascading hierarchical lists of values

It can be difficult to find a single value in a long list. Use hierarchical relationships within the data to make the selection process easier:

1. Identify any hierarchical relationships in the data and determine the top to bottom sequence. An example is Region > Country > City.
2. Create one query per dimensional level, returning the LOV for that level, as follows:
   i. Level 1 query: no parameters and refresh on open.
   ii. Level 2 query: prompt for Level 1 value and refresh on Insert of Level 1 selection.
   iii. level 3 query: prompt for leve1 and level 2 values, create a concatenated key combining level 1 and level 2 selections, and refresh on insert of the level 1/level 2 key value.
   iv. Level 4 query: prompt for level 1, 2, and 3 values, create a concatenated key combining level 1, 2, and level 3 selections, and refresh on insert of the level 1, 2, or 3 key value.
   v. Continue until the necessary levels of the hierarchy are represented by queries and components.

The results of this are a series of shorter, in-context LOV, dynamically changing based on each selection a user makes above the bottom level. For example, this workflow is possible:

1. "Europe" is chosen from the geo combo box. The country combo box then displays
   - Germany
   - United Kingdom
   - France
2. "Germany" is chosen. The city combo box then displays
   - Berlin
   - Frankfurt
   - Munich

This workflow is easier than finding a single city in a long list of values. Additionally, because each LOV is short, the model performs better than if a single LOV is used.

Constrained dimension values

Not every dimension selection combination retrieves rows from the fact table because sometimes not every dimensional combination exists. To reduce the attempts to find fact data when you're unfamiliar with best combinations to use, consider constraining each dimension value to only those values residing in the fact table. This can be done in two different ways:

- Force the join for the dimension object to the fact table. This technique applies this join to all queries, including report queries that use this dimension object.
- Create a filter object containing the join between the fact and dimension table and use this filter object. This technique applies this join only to the queries that use this filter.

This workflow reduces the chance of ***no-data-found*** but it does not eliminate it. This is especially the case with large number of dimensions. In these cases, use this workflow:

1. Create a COUNT(*) measure object in the universe, pointing to the fact table.
2. In a report, create a query that shows every dimension object used in your Xcelsius fact table query, along with the COUNT(*) measure.
3. Sort in descending order on the Count measure object in the report. At the top of the block are the most common dimensional combinations.

Dynamic dimension objects

The interactivity of a solution can often be enhanced by enabling the user to easily change the dimensions (not just dimension values) appearing in a component. For example:

- Trend line chart with a radio button that toggles between quarters, months, and weeks on the x-axis of the chart.
- Horizontal bar chart showing sales by product or geography. A radio button is used to toggle between the two dimensions.
- List view component displaying multiple columns where each dimension can be changed by the user using a selector component. For example, the column 1 display options might be gender, age, and income while the column 2 options might be job title, marital status, and address.

Using dynamic dimension objects, these types of solutions can be created with a single query. The dimension object SQL would look something like this:

```
Case when @Variable(‘Time Dimension’) = ‘Quarter’ then
sales_qtr when @Variable(‘Time Dimension’) = ‘Month’ then
sales_month else sales_week end
```

By using a fixed range of values (for example, quarter, month, or week) that can be entered into the worksheet, the proper parameters can be passed using selectors into the query to return the proper dimension value. The query is triggered on an insert into this target cell.

Make sure to select the **Ignore Blank Rows** option on the display component because the dimension lists will nearly always have differing numbers of unique values.

Dynamic measure objects

One common scenario involves the comparison of two different measures, often with respect to a dimension (for example, the time dimension). While it is possible to retrieve multiple measures in a single query, retrieving a large number of columns and rows simultaneously may degrade model performance. Dynamic measure objects reduce the number of columns returned to the model, without reducing flexibility and interactivity.

Below is a simple example. Using two dynamic measure objects, the user has access to six different measure columns, yet views only two at a time in the display component.

Measure 1

```
sum(dbo.Current_Facts.store_@Prompt('Measure1','N',{'sales'
,'cost','margin'},mono,free))
```

Measure 2

```
Sum(Case when @Variable(‘Measure2’) = ‘Quota’ then
Sales_Quota when @Variable(‘Measure2’) = ‘Plan’ then
Sales_Plan else Sales_ Opportunity end)
```

As with dynamic dimension objects, the measure name literals (for example, sales, cost, and margin) are data entered into worksheet cells and used with selector components to fill the input values required by the dynamic measure objects. Queries are triggered on the concatenation of the two selections.

In the above examples, two different techniques were used. In Measure 1, the variable column name part (sales, cost, and margin) is concatenated with the fixed column name part name (store_) to complete the full column name. In Measure 2, CASE logic is used. The CASE approach is best when the columns to be selected are not within the same table.

NOTE	Each measure column within a dynamic measure object should have the same display type (integer, decimal, percentage, currency, etc). This is because the measure display type within a chart component is fixed per chart component. For example, if the chart is configured to display currency, it cannot dynamically change to a percentage display. By creating measure objects with the same display type, this issue is avoided.

The alerts challenge

You may require that your dashboards display Alert messages when there are important changes to business conditions. Xcelsius has the following alerting capabilities:

- Display ticker-style key information: “USA Sales are up 25%”.

- Display messages based on the comparison of actual versus goal or previous period data.
- Hide messages using rule-based messages.
- Personalize messages for thousands of users.
- Drill down on the message for further detail about the alert.

This type of alerting solution is possible using this workflow:

1. Create a query that returns a dimension and two measures. For example, geographic or product dimension and today's sales and yesterday's sales measures. The measure values could even come from two different sources, using two queries instead of one.
2. In Excel (or universe), compare today's sales to yesterday's sales:

   ```
   =if(Today's Sales>Yesterday's Sales,'Up',Down')
   ```
3. Iin Excel, concatenate the dimension (for example, USA), the measure name (for example, sales), the trend (for example, up or down) and the numeric difference (for example, 25%) into a single text string.
4. Use a ticker component to display the range of messages, one row per dimension value.
5. Configure the ticker to insert 'USA' into an OpenDoc URL when clicked.
6. Create a prompted report to provide the additional drilldown detail.
7. Add an URL Link Component to run the OpenDoc URL when the user clicks it.

The large scale personalization is accomplished by adding a USER_NAME = @Variable('BOUSER') condition to the query.

Sorted list of values

When a list of dimension values is returned by a query, a specific sort order is often required. Currently, the Query as a Web Service query panel does not have the ability to include an ORDER BY within the SQL. The Web Intelligence query panel also does not have this functionality.

To work around this limitation, to embed the ORDER BY clause into a derived table and then use the objects from this table in the query. The workflow to do this is as follows:

1. Create the universe as you would normally, creating joins, measures, and dimensions.
2. Using the Query Panel within Designer, choose
   i. The dimension upon which you want to sort, as the first object in the Results panel.

   ii. Measures, if any.

   iii. Filters, if any.

3. Copy the SQL from this query and paste it into a derived table.
4. Edit the query as follows:
   i. Add “**`Top 100`**” between SELECT and the name of the dimension column.
   ii. Add Alias names to each column. For example: **`AS Sales_Amount`**.
   iii. Add “**`ORDER BY Dimension_Value ASC`** (or **`DESC`**)” to the end of the query.
5. Expose the dimension and measure objects from the derived table, but do not re-aggregate the measures (that is, put aggregation around the measure objects).
6. Create the query using these new objects but do not further filter the query. Below is query before the ORDER BY is added and embedded into the derived table

```
SELECT
  cast(datepart(yyyy,dbo.Current_Facts.sales_date) as
char(4)),
  sum(dbo.Current_Facts.store_sales)
FROM
  dbo.Current_Facts,
  dbo.store
WHERE
  ( dbo.Current_Facts.store_id=dbo.store.store_id  )
  AND  (
  dbo.store.store_country  =
@Prompt('Country','A',,mono,free)
  )
GROUP BY
  cast(datepart(yyyy,dbo.Current_Facts.sales_date) as
char(4))
```

Below is the query **after** the ORDER BY is embedded into the Derived Table:

```
SELECT
  Country_Sales.Year_Qtr,
  Country_Sales.Sales
FROM
  ( SELECT Top 100
  cast(datepart(yyyy,dbo.Current_Facts.sales_date) as
char(4)) +'-Q'+
cast(datepart(q,dbo.Current_Facts.sales_date) as
char(1)) as Year_Qtr,

  sum(dbo.Current_Facts.store_sales) as Sales

FROM

  dbo.Current_Facts,

  dbo.store

WHERE

 dbo.Current_Facts.store_id=dbo.store.store_id

  AND  dbo.store.store_country  =
@Prompt('Country','A',,mono,free)

GROUP BY

  cast(datepart(yyyy,dbo.Current_Facts.sales_date) as
char(4)) +'-Q'+
cast(datepart(q,dbo.Current_Facts.sales_date) as
char(1))


ORDER BY

cast(datepart(yyyy,dbo.Current_Facts.sales_date) as
char(4)) +'-Q'+
cast(datepart(q,dbo.Current_Facts.sales_date) as
char(1)) ASC) Quarterly_Sales
```

Ranked selection

Using the proper query, combined with a slider or any other integer selection component makes Top *N* models easy to create. The following technique combines ranked selection performed on a derived table together with a sort performed on the same derived table. Below is a sample query:

```
SELECT

  TopN_Sorted.Customer_Name,

  TopN_Sorted.Sales_Amt

FROM

  ( SELECT top @Prompt('How Many
Customers?','N',,mono,free)

customer.lname as Customer_Name,

sum(Current_Facts.store_sales) as Sales_Amt

FROM

  dbo.Current_Facts, customer

WHERE customer.customer_id=Current_Facts.customer_id

GROUP BY

customer.lname
ORDER BY
sum(Current_Facts.store_sales)  DESC) TopN_Sorted
```

The result is the exact number of customers requested, sorted in descending order.

Parameterized operators

Queries can be made more interactive by replacing hard coded SQL operators with parameterized operators. An example of a hard coded operator is

```
…WHERE customer.sales >= 100…
```

A query containing this condition always returns rows with sales greater than or equal to 100 but what if a dashboard requires also finding rows where sales were less than 100? In this case, a parameterized operator is needed to enable the end user to make their operator choice, within the confines of a single query.

A parameterized operator, created within a universe filter, is as follows:

```
…WHERE customer.sales
@Prompt('Operator','N',{'<=','>='},mono,free) 100…
```

By using an Xcelsius selector to pass the '<=' or '>=' values into the query at runtime, increased interactivity is possible.

NOTE	The values enclosed within curly brackets do not represent all the total possible operators that could be used in the query. The @Prompt logic simply needs one or more values in order to parse properly. The Xcelsius selector could pass ***any*** operator (<>, =, for example) to the query, even if it is not enclosed within curly brackets. A data type of 'N' within the @Prompt is required. By using 'N', no quotes are required around the operator value.

Parameterized sort order

Giving the user control over the sort order of a list of dimension values is often desired. While the list view component gives the user the ability to sort dimension values, no other components gives end users this sorting capability.

Controlling the sort order is simply a matter of replacing the ASC or DESC parameter at the end of the query with a prompt and using a selector component to give sort control to the user. Below is an example:

```
SELECT Top 100
  dbo.product_class.product_department as
Product_Department,
  sum(dbo.Current_Facts.store_sales) as Sales
FROM
  dbo.Current_Facts,
  dbo.product,
  dbo.product_class
WHERE

dbo.product.product_class_id=dbo.product_class.product_clas
s_id
  AND dbo.Current_Facts.product_id=dbo.product.product_id
GROUP BY
  dbo.product_class.product_department
order by sum(dbo.Current_Facts.store_sales)
@Prompt('Sort','N',{'ASC','DESC'},mono,free)
```

NOTE	This query must be placed in a derived table to enable parameterized sorting .

Query pivoting for multi-dimensional display

Most Xcelsius components can easily display a resultset containing **a** single dimension and multiple measures (for example, sales, margin and cost grouped by month).

To display a single measure across multiple dimensions requires additional universe and query preparation. For example, what if the requirement is to display sales by product and time? To create the proper query result set for Xcelsius, the measure must be 'dimensionalized'. To do this, one of the dimensions must be selected for use in a measure object. The choice of which dimension to use largely depends on two issues:

- The number of unique values that the dimension has.
- How frequently new dimension values are added or removed.

If the dimension has a small number of unique values (that is, fewer than 20) and new values are rarely if ever added, this dimension is a good candidate for becoming a measure. The measure object's SQL statements for three products are as follows:

```
Sum(CASE WHEN Product = 'Red' then Sales Else 0 end)
Sum(CASE WHEN Product = 'Green' then Sales Else 0 end)
Sum(CASE WHEN Product = 'Blue' then Sales Else 0 end)
```

These three measures could then be used with the time dimension in the same query to represent two different dimensions: product and time. The result is a crosstab (time down the side, products across the top, sales in the body of the crosstab) that can be used by Xcelsius.

This approach weakens as the number of dimension values increases because additional measure objects must be created. It also suffers when the frequency of LOV changes is high because of effort to maintain the universe, query, and model. A lower maintenance approach is to use the time dimension within the measure objects. There are two ways to do this:

Absolute time period measures

Here, each measure represents a specific period of time (January, February, Q1, Q3, for example) and generally occupies a specific position within the worksheet. Here is an example of the required SQL statements:

```
SELECT
  dbo.product_class.product_family,
  sum(case when datepart(q,dbo.Current_Facts.sales_date) =
1 then dbo.Current_Facts.store_sales else 0 end) as
Q1_Sales,
  sum(case when datepart(q,dbo.Current_Facts.sales_date) =
2 then dbo.Current_Facts.store_sales else 0 end) as
Q2_Sales,
  sum(case when datepart(q,dbo.Current_Facts.sales_date) =
3 then dbo.Current_Facts.store_sales else 0 end) as
Q3_Sales,
  sum(case when datepart(q,dbo.Current_Facts.sales_date) =
4 then dbo.Current_Facts.store_sales else 0 end) as
Q4_Sales
FROM
  dbo.Current_Facts,
  dbo.product,
  dbo.product_class
WHERE
  (
dbo.product.product_class_id=dbo.product_class.product_clas
s_id  )
  AND  (
dbo.Current_Facts.product_id=dbo.product.product_id  )
GROUP BY
  dbo.product_class.product_family
```

In this example, each measure column is sales for a specific quarter while the dimension value is product. With this approach, no object, query, or model maintenance is required when products are added. One disadvantage is that early in a calendar period (that is, Year), not all the measures will return data.

NOTE	Note that this example shows temporal database functions being used. A custom calendar table could be used in place of these functions.

Relative time period measures

Here, each measure represents a relative period of time (for example, this month's sales or last month's sales) and the values will shift to the left as time progresses (the right-most measure is current month, 2nd from the right is Previous Month). The SQL statements might look like the following:

```
SELECT

  dbo.product_class.product_family,

  sum(case when dbo.Current_Facts.sales_date between
dateadd(mm,-3,DATEADD(mm,
DATEDIFF(mm,0,dbo.Current_Facts.sales_date), 0)) and
dateadd(dd, - 1, dateadd(m, 1, DATEADD(mm, DATEDIFF(mm, 0,
dateadd(mm,-3,DATEADD(mm,
DATEDIFF(mm,0,dbo.Current_Facts.sales_date), 0))), 0)))
then dbo.Current_Facts.store_sales else 0 end) as
3_Months_Ago_Sales,


  sum(case when dbo.Current_Facts.sales_date between
dateadd(mm,-2,DATEADD(mm,
DATEDIFF(mm,0,dbo.Current_Facts.sales_date), 0)) and
dateadd(dd, - 1, dateadd(m, 1, DATEADD(mm, DATEDIFF(mm, 0,
dateadd(mm,-2,DATEADD(mm,
DATEDIFF(mm,0,dbo.Current_Facts.sales_date), 0))), 0)))
then dbo.Current_Facts.store_sales else 0 end) as
2_Months_Ago_Sales,


  sum(case when dbo.Current_Facts.sales_date between
dateadd(mm,-1,DATEADD(mm, DATEDIFF(mm,0,getdate()), 0)) and
dateadd(dd,-1,DATEADD(mm, DATEDIFF(mm, 0,getdate()), 0))
then dbo.Current_Facts.store_sales else 0 end) as
Last_Month_Sales,


  sum(case when dbo.Current_Facts.sales_date>= DATEADD(mm,
DATEDIFF(mm, 0, getdate()), 0) then
dbo.Current_Facts.store_sales else 0 end) as
Current_Month_Sales


FROM

  dbo.Current_Facts,

  dbo.product,

  dbo.product_class


WHERE


dbo.product.product_class_id=dbo.product_class.product_clas
s_id

  AND  dbo.Current_Facts.product_id=dbo.product.product_id
```

```
GROUP BY
  dbo.product_class.product_family
```

In this example, each measure column is Sales for a relative month while the dimension value is product. This is accomplished by isolating within each measure only the activity between the first and last dates of periods in the past (months, in this example). With this approach, no object, query, or model maintenance is required when products are added. Unlike the absolute period approach, every column can always return data regardless of the time of the year.

NOTE	Since each measure represents a measure for a relative period of time, proper labeling with an absolute date label is important. Without labeling, it is unclear which quarter, month, and day are being displayed.

To provide the proper period labeling, two approaches are available:

- With an Excel formula in a series of cells (subtracting periods from TODAY).
- With a query to produce a series of date labels (using a calendar table).

Drilldown to detail in reports

Many enterprise solutions require the ability to drill to the detail behind the summary information. While Xcelsius is well-suited for displaying small amounts of information, reports are better suited for the subsequent drilling workflow and the associated volume of data. Below is the workflow for creating context sensitive drilldown:

1. Select an existing or create a Web Intelligence, Desktop Intelligence, or Crystal report that contains prompts for character strings, date values, or integers.

2. Enter data into the spreadsheet such as report name(s), prompt name(s), and report types (if using two or more different reports of different types to drill to).

3. Concatenate the following attributes using the ampersand (&) symbol:
   - “http://”
   - System Name literal
   - Folder structure literal containing openDoc.jsp
   - DocName literal
   - Doc Type literal
   - Report name (either hard coded or referencing a report selection)
   - Report type (either hard coded or referencing a cell value)
   - Prompt names (matching those created in the report)
   - Cells whose values contain the prompt names

The resulting OpenDoc URL resemble the following:

```
http://cdixir2:8080/businessobjects/enterprise115/desktopla
unch/opendoc/openDocument.jsp?sDocName=Product_Detail&sType
=wid&lsSCountry=USA&lsSProduct=Food&lsSEnding
Date=12/31/2004&lsSMeasure 1=Margin&lsSMeasure 1
Rollup=Sum&lsSSelect a Product Level=Product Name&lsSMonths
Ago=12
```

In the URL example above, the Web Intelligence report (named Product_Detail, type wid) has seven prompts.

NOTE	Note that the maximum length of the URL string supported by Excel is 1024 characters. To test if the cell is exceeding this limit, use the LEN function (ie. =LEN(URL cell)) in another cell.

Within the Xcelsius model, use a URL link component to access the OpenDoc URL cell in the step above. Name the button “Run Report” or use the selections and concatenation to make the button name dynamic.

6. Export the SWF file to InfoView from Crystal Xcelsius by going to **File > Export > Business Objects Enterprise**.
7. Test the result from in InfoView:
   i. Open the SWF file exported in the previous step
   ii. Make any selection
   iii. Click the Run report button
   iv. View the detail displayed within the report

Finding more information

Business Objects Diamond Technical Community

http://diamond.businessobjects.com

Business Objects Technical Support

http://technicalsupport.businessobjects.com

For more information and resources, refer to the product documentation and visit the support area of the web site at

http://www.businessobjects.com/

www.businessobjects.com