Multipass SQL

A Step By Step Guide for BOE XIr2

Version 1.0
April 15, 2008

Background

What is it?

Multi-pass SQL refers to the creation of multiple queries against a database to accomplish complex comparisons to answer sophisticated questions. These inquiries may seem difficult or surprisingly easy. Often business questions are easy to ask, but very difficult to answer without thinking in terms of multiple passes against the database. The logical approach to answering such questions is to break it down into its components and then pool the answers to each component together for the final response. Multi-pass SQL appears in many situations. These include:

1. Sharing dimensions across multiple fact tables.
2. Mixing the grains of measurement in the same query such as year-to-date and month-to-date.
3. Defined calculations that require an end result as one of its factors.
4. The need of semi-additive measures such as account balances and ending inventory.
5. Analysis on a subset of data and possibly comparing it to the whole or another subset.

How is it done?

The concept is that to achieve a specific analysis a series of queries is executed against a database. The results of each query are then combined in such a way to ensure that the desired analysis is attained. The traditional approach is to issue multiple queries against the database while storing the intermediate results in temporary tables. Once all the queries are completed the temporary tables are queried for the final result. However, Ralph Kimball has defined multi-pass capability as the ability of "the query tool (to) break the report down into a number of simple queries that are processed separately by the DBMS" and "then automatically (combining) the results of the separate queries in an intelligent way". [1]

Why is it important?

"SQL does not have the ability to perform multidimensional calculations in single statements, and complex multi-pass SQL is necessary to achieve more than the most trivial multidimensional functionality."[2] The main objective is to provide the end user the answer to their questions. If multi-pass SQL can be achieved in way that also simplifies the query experience for the end user, this all the better. So obviously the preferred method is that end user can obtain the correct results by building one query instead of several.

Does Business Objects support multi-pass?

Yes. According to Kimball, “Tools like Cognos, Business Objects, and Microstrategy are quite capable of handling multi-pass SQL.” [3]

What about our competition?

Microstrategy has stressed the importance of multi-pass SQL for years. The competitive tour section of their website states that Business Objects has “limited support for advanced analysis”. This is due to its “single-pass SQL engine which does not fully leverage database processing features” resulting in “limited support for advanced analysis”. This statement is made for all their competitive vendors that access relational data.

It is best to assume that all BI vendors can achieve some level of multi-pass SQL. The difference is in which situations multi-pass can be invoked, simplicity of work flow, and the methodology used to achieve multi-pass. In addition to these concerns one must also be aware of on which groups of users is placed the burden of the multi-pass SQL solution.

The groups can roughly be separated into report consumers, report creators, and semantic layer developers. In a pure ad hoc scenario the report consumers and creators are the same. Some solutions may require the report consumer to respond to many prompts opening the door to incorrect answers or inconsistency between users. Other solutions may place the burden on the report creators in the form of multiple queries. Similar reports may require the same queries to be created over and over again. Ideally the solution is implemented at the semantic layer as the solution would be created once in a central location ensuring consistency across reports and queries. Usually solutions at the semantic layer also ensure the most simplicity when a report is executed.

How does Business Object’s handle multi-pass?

Business Objects creates multi-pass SQL in a variety of ways depending upon the situation requiring it. Let’s take the previously mentioned multi-pass situations one by one.

Sharing dimensions across multiple fact tables

A Universe parameter exists that forces multiple SQL statements to be generated when measures are retrieved from different tables. Even though the parameter is labeled as “Multiple SQL statements for each measure” it only applies when the measures being retrieved involve different tables. It is enabled (checked) by default.

Mixing the grains of measurement in the same query

In cases when a measure needs to according to different time spans such as *current month* and *last month* a combination of alias tables and contexts will produce multiple SQL statements to be produced. The fact table is aliased once for each time span with separate measure objects created from each alias. The measure objects can then be combined in a single query along with the required dimensions. The existence of the contexts will cause multiple SQL statements to be created.

Defined calculations that require an end result as one of its factors

Ratios are a good example of such a calculation. A ratio of each category's sales to total sales can be accomplished by using a derived table to calculate the total sales amount. The total for all sales must be obtained first. Then the total sales amount is then retrieved for each category and divided by the overall total previously retrieved.

Need of semi-additive measures

Semi-additive measures include ending inventory and account balances. Ending inventory can be summed across product or location but not time. In these situations a derived table is built for each required timeframe. Aggregate awareness is then used to navigate between the derived tables. The use of the "query drill" report property also allows one drill from year end balance to quarter end balance to month end balance and so on.

Analyzing a subset of data

Often situations exist that require the analysis of a subset of data. The analysis of that subset may then be compared to the whole set of data or to another subset. Such situations may require that the most recent status update or transaction record be retrieved for each account. A derived table can be used to filter out unwanted data, leaving only the most current status or transaction record. Using such a filter within the query panel allows analysis to proceed using the full range of Business Objects' capabilities.

Each one of these scenarios will be discussed in further detail.

What is a derived table?

"A derived table is a SQL construct consisting of a SELECT statement embedded in the FROM clause of another select statement. Derived table support is required for full ANSI-92 SQL conformance. A variety of names are used to refer to derived tables including: table subqueries, nested queries, and table value constructors (the formal ANSI-92 SQL name). Use of derived tables reduces the total number of SQL passes required to answer a complex query. Multi-pass queries that are multi-dimensional, contain metric qualification, split facts, or outer joins can be constructed with derived tables."[4]

Derived tables are not the same as volatile tables. A volatile table is a form of temporary table which is created in memory and whose life extends only for the current session. As volatile tables do not access the database system catalogues, they are not logged and can not be recovered. After its creation, a volatile table is treated just as any another table in a select statement.

In the multi-pass scenario, derived tables can be used to join selection/calculation created in the FROM clause which is then joined to the result set of the main query. As such, calculations can be moved from the report level to the Universe. For example, instead of requiring a ratio to be created as a report variable it can be a Universe object which can be selected within the query panel. Reducing the need for report variables simplifies the report creation process.

Scenarios

Example Data

The initial test is a very simple. The database contains one dimension table (MP_T1_Cat) and two fact tables, MP_T1_Sales and MP_T1_Calls. Figures 1 through 4 display the raw data, row by row. Figures 5 and 6 show the summary totals by category id.

<u>Detail</u>

MP_T1_Cat

Cat_Id	Category
1	Electronics
2	Food
3	Gifts
4	Health & Beauty
5	Household
6	Kid's Korner
7	Travel

Figure 1: Category Data

MP_T1_Sales

Cat_Id	Month_Id	Sales
1	90	12684
1	90	10310
1	89	9047
1	89	7874
2	90	10938
3	90	22376
3	90	8024
3	89	5962
4	89	11707
5	90	22190
5	89	32280
5	90	34304
6	89	2762
6	90	2740
7	90	9289

Figure 2: Sales Data

MP_T1_Calls

Cat_id	Month_Id	Nbr_Calls
1	90	10
1	89	5
1	90	15
2	89	4
2	90	6
3	89	4
3	90	8
3	89	12
3	90	16
3	89	20
4	90	3
4	89	6
4	90	9
4	89	12
5	90	1
5	89	2
5	90	3
5	89	4
5	90	5
5	89	6
5	90	7
5	89	8
5	90	9
5	89	10
6	90	10
6	89	20
6	90	30
7	89	7
7	90	14

Figure 3: Call Data

<u>Summary</u>

Cat_Id	Category
1	Electronics
2	Food
3	Gifts
4	Health & Beauty
5	Household
6	Kid's Korner
7	Travel

Figure 4: Categories

Cat_Id	Sales
1	39915
2	10938
3	36362
4	11707
5	88774
6	5502
7	9289

Figure 5: Sales Summary

Cat_id	Nbr_Calls
1	30
2	10
3	60
4	30
5	55
6	60
7	21

Figure 6: Call Summary

Scenario 1: Multiple Fact Tables

<u>Universe</u>

The initial Universe is shown Figure 7.

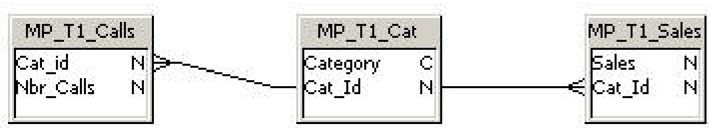


Figure 7: Initial Table Joins for Universe

A few objects are created for the Universe as shown in Figure 8.

Figure 8: Initial Universe Objects

The *Nbr Calls* object is identified as *sum(MP_T1_Calls.Nbr_Calls)* and the *Sales* object is defined as *sum(MP_T1_Sales.Sales)*.

Suppose one needs to report on the number of sales calls made by product category along with the dollar sales by category. Initially one may be tempted to try a simple SQL query as shown below.

```
SELECT
MP_T1_CAT.Category, MP_T1_CAT.Cat_Id,
sum(MP_T1_Calls.Nbr_Calls),
sum(MP_T1_Sales.Sales)
FROM
MP_T1_Cat, MP_T1_Calls, MP_T1_Sales
WHERE
MP_T1_Cat.Cat_Id = MP_T1_Calls.Cat_id
AND MP_T1_Cat.Cat_Id = MP_T1_Sales.Cat_Id
GROUP BY MP_T1_CAT.Category,
MP_T1_CAT.Cat_Id
```

Figure 9: Erroneous attempt to return aggregated columns from two tables in a single SELECT

However such a query will not return correct results. Since the two fact tables are sharing the dimension table the query will retrieve rows from each fact table multiple times. If the dimension value occurs 3 times in the second fact table then that dimensional value in the first fact table will be overstated by a fact of 3. This is discussed in more detail later (Figure 19). This is a limitation of SQL, not of Business Objects or any other BI tool, and is the same across all relational databases.

Using the traditional approach of multi-pass SQL, five (5) steps are required. The first step is to create a table to house temporary results.

```
CREATE TABLE Sales_and_Calls
      (Cat_Id integer, Nbr_Calls integer, Sales decimal(10,2))
```

Next, the number of calls by category is inserted into the work table.

```
INSERT INTO Sales_and_Calls (Cat_Id, Nbr_Calls, Sales)
      SELECT MP_T1_Calls.Cat_ID,
      sum(MP_T1_Calls.Nbr_Calls), 0
      FROM MP_T1_Calls
      GROUP BY  MP_T1_Calls.Cat_Id
```

The third phase is to load the sales by category into the table.

```
INSERT INTO Sales_and_Calls (Cat_Id, Nbr_Calls, Sales)
      SELECT MP_T1_Sales.Cat_ID, 0,
      sum(MP_T1_Sales.Sales)
      FROM MP_T1_Sales
      GROUP BY MP_T1_Sales.Cat_Id
```

At this point the values within the temporary table are shown below in Figure 10.

	Cat_Id	Nbr_Calls	Sales
1	1	30	0.00
2	1	0	39915.00
3	2	10	0.00
4	2	0	10938.00
5	3	60	0.00
6	3	0	36362.00
7	4	0	11707.00
8	4	30	0.00
9	5	0	88774.00
10	5	55	0.00
11	6	0	5502.00
12	6	60	0.00
13	7	0	9289.00
14	7	21	0.00

Figure 10: Values within the temporary table after two passes

At this time all the required results exist in the temporary table. The next stage is to retrieve the results for the report by joining the temporary table to the category descriptions.

```
SELECT MP_T1_Cat.Cat_Id, Category, sum(Nbr_Calls), sum(Sales)
FROM Sales_and_Calls JOIN MP_T1_Cat
ON MP_T1_Cat.Cat_Id = Sales_and_Calls.Cat_Id
GROUP BY MP_T1_Cat.Cat_Id, Category
ORDER BY MP_T1_Cat.Cat_Id
```

The results that are finally obtained are correct.

	CAT_ID	CATEGORY	Sum(Nbr_Calls)	Sum(Sales)
1	1	Electronics	30	39915.00
2	2	Food	10	10938.00
3	3	Gifts	60	36362.00
4	4	Health & Beauty	30	11707.00
5	5	Household	55	88774.00
6	6	Kid's Korner	60	5502.00
7	7	Travel	21	9289.00

Figure 11: Final results using a traditional multi-pass approach

And last but not least, one must always cleanup after themselves. The final step is to delete the temporary work table.

```
DROP TABLE Sales_and_Calls
```

In this simple example, two passes of the database schema were required in addition to pulling the final results for reporting. Along with this activity is the creation and deletion of the temporary table. This "traditionalist" approach is very database centric and assumes very limited capability on the reporting/analysis tier. Not being bound by such limitations Business Objects approaches the solution from a different angle.

An initial query is built with the Web Intelligence query panel (Figure 12).

Figure 12: Web Intelligence Query Panel result objects

The query does return the correct results (Figure 13) when compared to Figures 5 and 6.

Category Id	Category	Nbr Calls	Sales
1	Electronics	30	39,915
2	Food	10	10,938
3	Gifts	60	36,362
4	Health & Beauty	30	11,707
5	Household	55	88,774
6	Kid's Korner	60	5,502
7	Travel	21	9,289

Figure 13: Web Intelligence query results

So how was the SQL generated to obtain the correct results? Two query statements were generated (Figure 14 and Figure 15) with the result sets combined by the Business Objects server. These two SELECT statements bear a strong resemblance to those used to insert data into the temporary table when using the traditional approach.

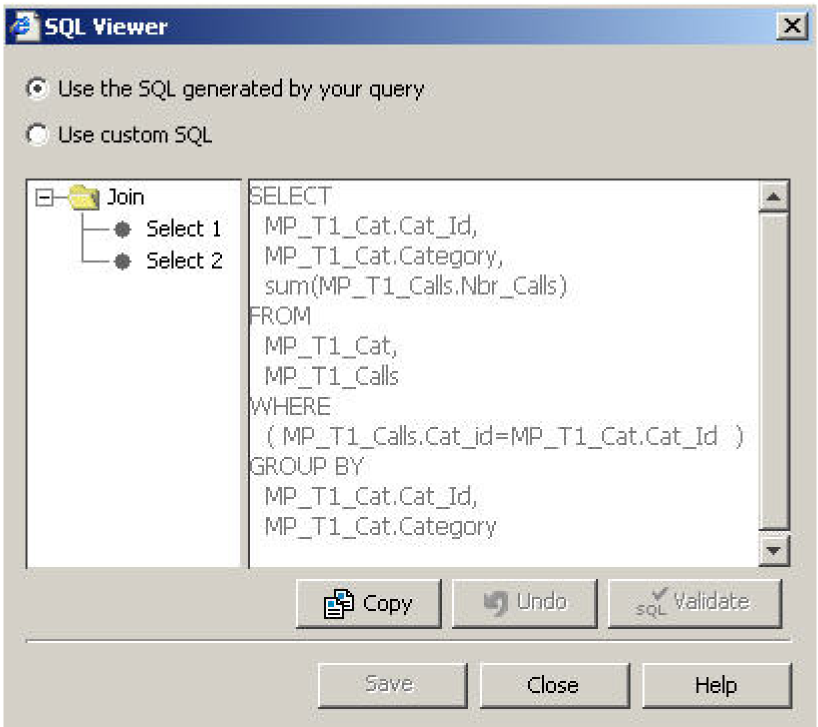


Figure 14: First select returns the number of calls by category

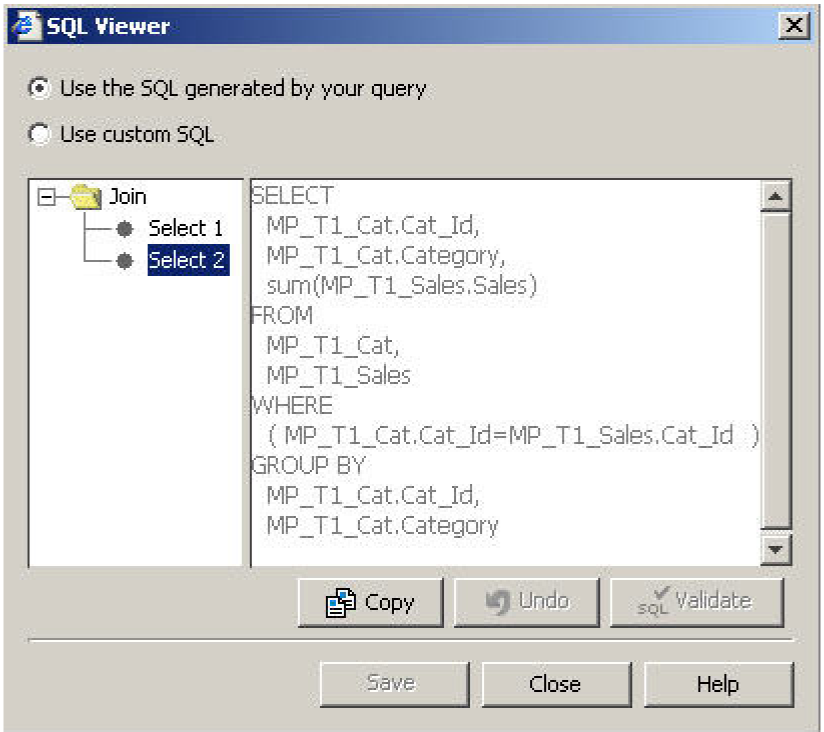


Figure 15: Second select returns the total sales by category

Why did this occur? The defaulting setting for a Universe is to have *Multiple SQL statements for each measure* enabled (Figure 16). As mentioned before a more accurate definition of this parameter is to have multiple SQL statements for each table from which measures are defined.

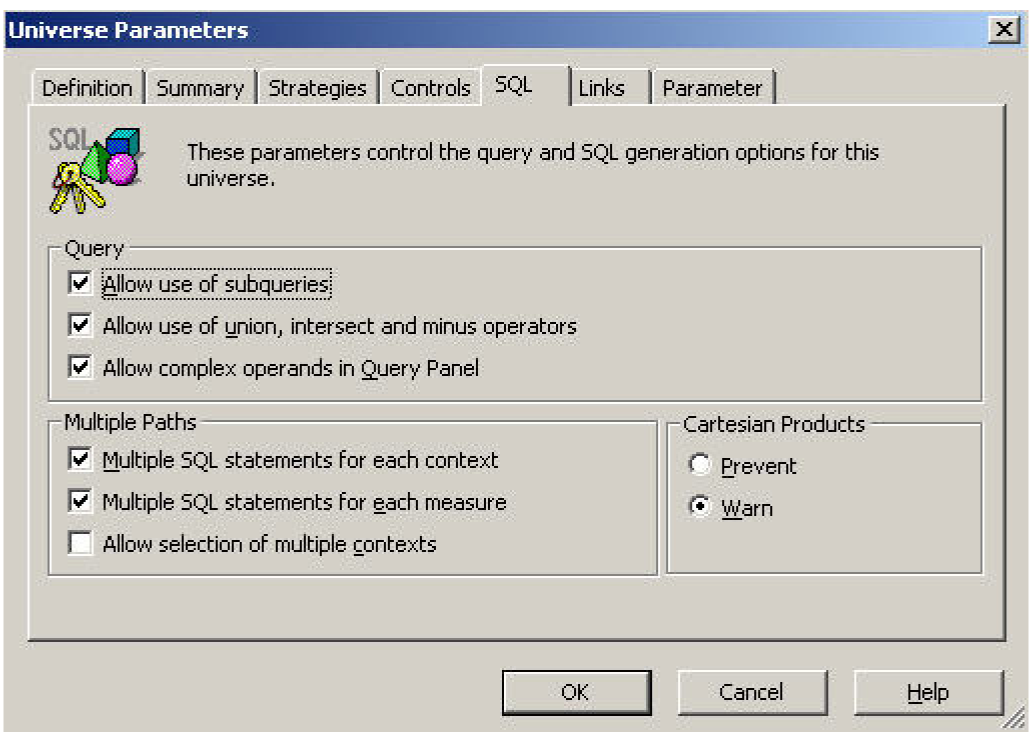


Figure 16: Universe Parameters, SQL tab

What happens is if this option is disabled as done in Figure 17?

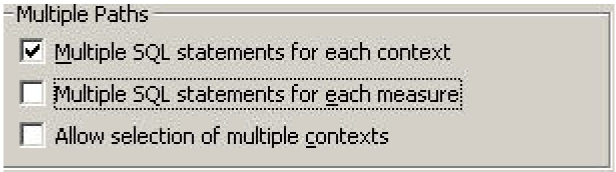


Figure 17: "Multiple SQL Statements for each measure" option disabled

With the exact same objects selected the query engine generates a single statement (Figure 18). This is very similar to the SQL shown in Figure 9.

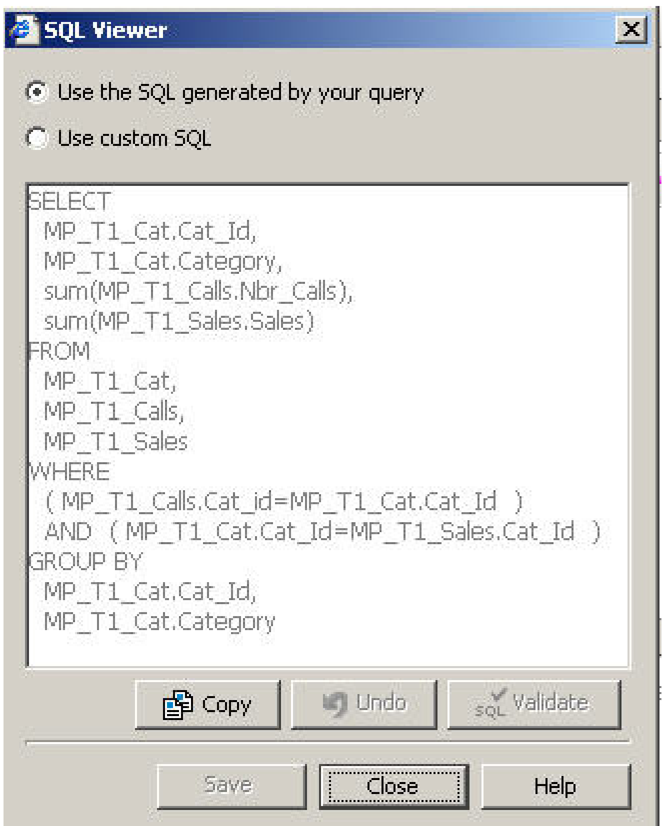


Figure 18: Generated SQL with "Multiple SQL Statements for each measure" disabled

Will the results still be correct? The answer is *No* as proven by the results shown in Figure 19.

Category Id	Category	Nbr Calls	Sales
1	Electronics	120	119,745
2	Food	10	21,876
3	Gifts	180	181,810
4	Health & Beauty	30	46,828
5	Household	165	887,740
6	Kid's Korner	120	16,506
7	Travel	21	18,578

Figure 19: Incorrect results due to "Multiple SQL Statements for each measure" disabled

What happened? This is an example of a *Chasm Trap* so often mentioned as one of the issues that be overcome by proper Universe design. Let's take a closer look at the totals for *Electronics*. As seen in Figure 13, the correct number of calls is 30 and the correct amount of sales is 39, 915. This means that calls have been overstated by a factor of 4 (120/40) and sales have been overstated by a factor of 3 (119,745/39,915). Looking at the Electronics category, according to Figure 2, there are 4 entries in the sales fact table while Figure 3 shows there are 3 entries in the calls fact table. So the number of calls has been replicated 4 times, one time for each entry in the sales fact table. The total sales have been replicated 3 times, one time for each entry in the calls fact table. A chasm trap is a limitation of SQL, not due to anything within Business Objects.

What if there is a need to disable the *Multiple SQL statements for each measure* option? Is there a way to still obtain correct results? The answer is **Yes**; Contexts can be used to also deliver correct results. Contexts define multiple paths between tables. If cardinality has been properly identified they can be automatically generated (Figure 20).

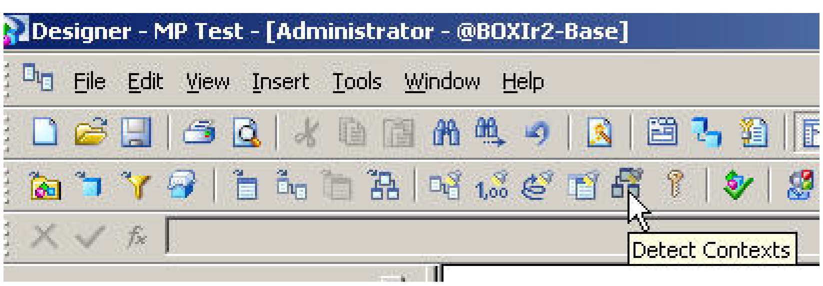


Figure 20: Automatic detection of candidate contexts

Choosing a candidate context will highlight all joins and tables (Figure 21) that will be members of that context if created.

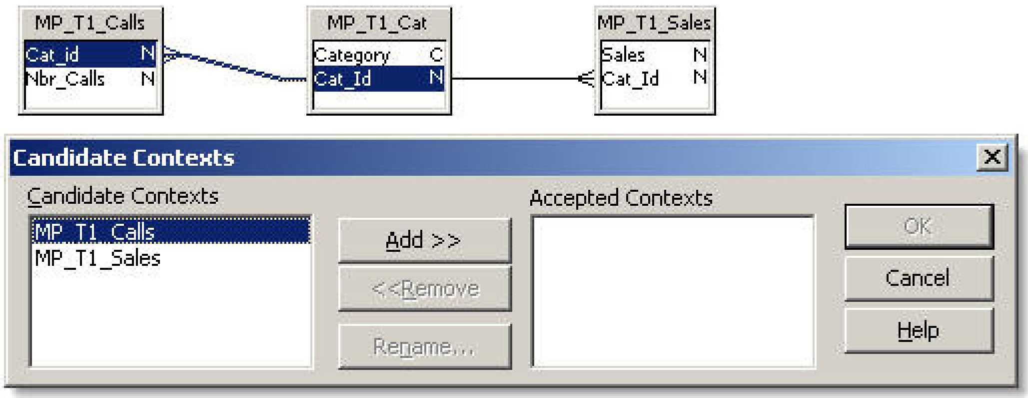


Figure 21: Tables and joins to be included in a context

Using the control and shift keys multiple candidate contexts can be added at one time (Figure 22). Highlight the desired contexts and select **Add >>**. Once all desired contexts appear under **Accepted Contexts** then click **OK**.

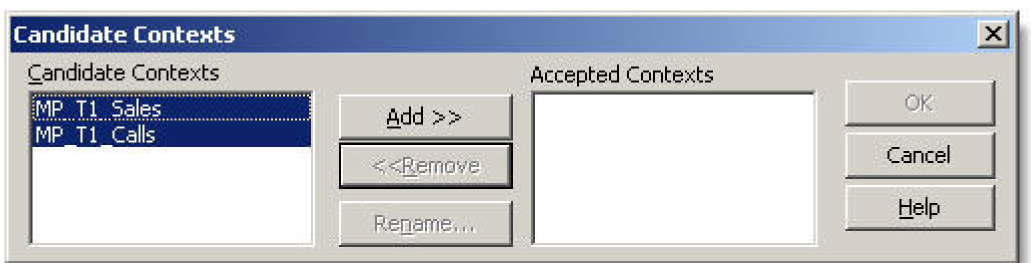


Figure 22: Adding multiple contexts

To view existing contexts, select **List Mode** from the **View** menu option (Figure 23).

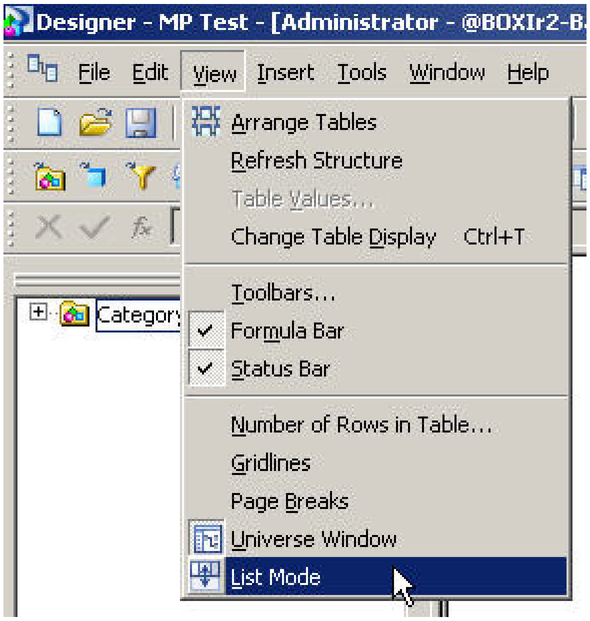


Figure 23: Activating List Mode

Viewing existing contexts is important as their number increases (Figure 24).

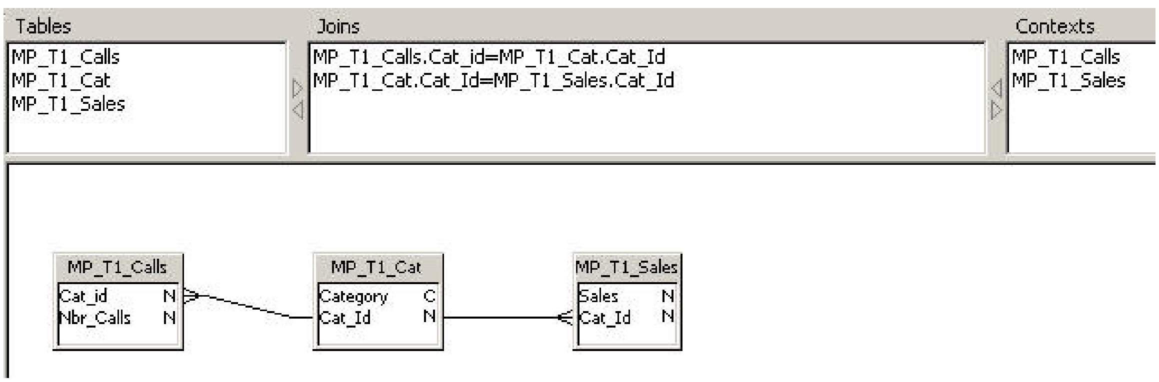


Figure 24: Universe design panel with List Mode activated

Selecting a context from this window will highlight the joins and tables that are members of the selected context. The joins that are members of a context can be modified by altering the **Context Properties**. In situations involving many contexts, it is highly recommended to not manually add or remove joins to a context. If new tables are added to the Universe, deleting all existing contexts and then detecting all required contexts is the recommended approach. Occasionally the joins that make up a context may need to be manually adjusted after detection. Any manual adjustments to a context are lost when that context is deleted. If it is necessary to manual adjust joins within a context be sure to document these joins. Shortcut joins are a good example of when manual adjustments may be necessary. Shortcut joins are not made part of a context when the automatic detection is used. So if a shortcut join needs to be part of a context it must be manually added to the context.

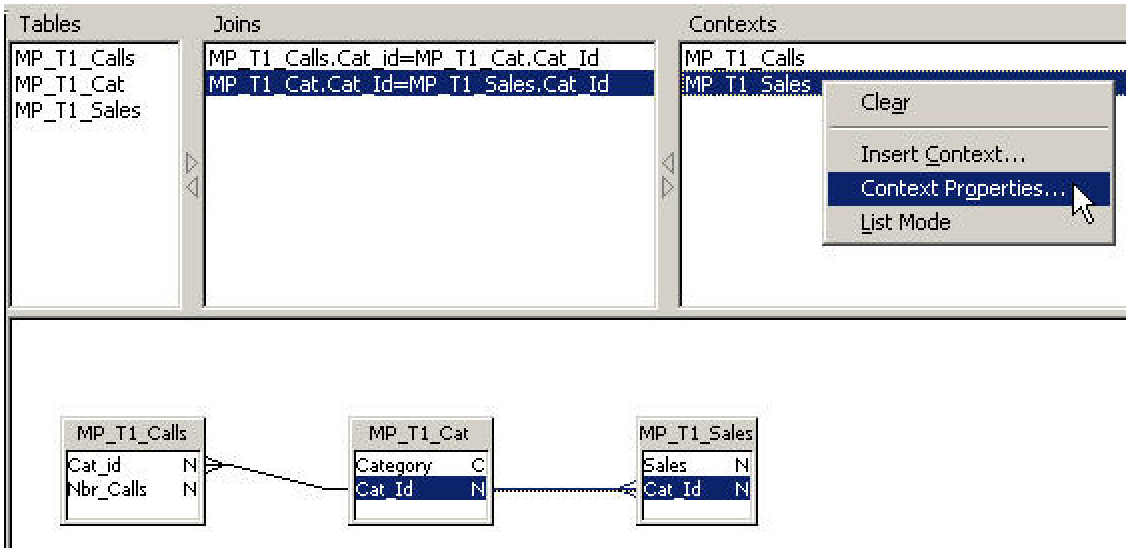


Figure 25: Accessing Context Properties

With the two contexts now defined and the *Multiple SQL statements for each measure* option disabled the correct SQL (Figure 14 and Figure 15) is generated leading to the same correct results seen in Figure 13.

So we have now seen an example of to resolve the first multi-pass scenario, sharing dimensions across multiple fact tables. In fact we have seen two methods. The suggested approach is to leave the *Multiple SQL statements for each measure* option enabled unless there is specific need to do otherwise. Plus use the ability to automatically detect and create contexts within the Universe. This two pronged approach is recommended. But how is this multi-pass SQL? We have seen that in these circumstances, multiple SQL statements are generated. The returned result sets are then combined by the Business Objects server. If we go back to Kimball's definition of multi-pass, the query has definitely been broken down into simple queries, and executed separately by the RDBMS. The result sets have then been automatically combined to deliver the correct results.

As a side note, there is now a Universe parameter, JOIN_BY_SQL, that when set to **Yes** will combine multiple SQL statements into one. Each of the previously separate queries becomes a derived table in the FROM clause with the COALESCE function used to combine each of the SQL statements along the common dimensions. In this scenario only one result set will be returned to Business Objects. So when using this option the RDBMS executes the multiple queries and combines the result set. In general better overall performance will be seen using this option set to **Yes**.

Scenario 2: Varying Grains of Measurement

The next multi-pass scenario is the mixing of grains of measurement in the same query. Often reports will need to compare the number of units sold this month with the number of units sold last month. To work through this scenario we will introduce a processing calendar into the database schema. For this example the data is depicted in Figure 26.

Month_ID	Year_Month	Month_Rpt_Desc	Year_Month_Desc	Year_Nbr	Month_Nbr	Qtr_Nbr	Qtr_Rpt_Desc	Year_Rpt_Desc
90	200406	Current Month	2004/06	2004	6	2	Current Qtr	Current Year
89	200405	Current Month - 1	2004/05	2004	5	2	Current Qtr	Current Year
88	200404	Current Month - 2	2004/04	2004	4	2	Current Qtr	Current Year
87	200403	Current Month - 3	2004/03	2004	3	1	Current Qtr - 1	Current Year
86	200402	Current Month - 4	2004/02	2004	2	1	Current Qtr - 1	Current Year
85	200401	Current Month - 5	2004/01	2004	1	1	Current Qtr - 1	Current Year
48	200312	Current Month - 6	2003/12	2003	12	4	Current Qtr - 2	Current Year - 1
47	200311	Current Month - 7	2003/11	2003	11	4	Current Qtr - 2	Current Year - 1
46	200310	Current Month - 8	2003/10	2003	10	4	Current Qtr - 2	Current Year - 1
45	200309	Current Month - 9	2003/09	2003	9	3	Current Qtr - 3	Current Year - 1
44	200308	Current Month - 10	2003/08	2003	8	3	Current Qtr - 3	Current Year - 1
43	200307	Current Month - 11	2003/07	2003	7	3	Current Qtr - 3	Current Year - 1
42	200306	Current Month - 12	2003/06	2003	6	2	Current Qtr - 4	Current Year - 1
41	200305	Current Month - 13	2003/05	2003	5	2	Current Qtr - 4	Current Year - 1
40	200304	Current Month - 14	2003/04	2003	4	2	Current Qtr - 4	Current Year - 1
39	200303	Current Month - 15	2003/03	2003	3	1	Current Qtr - 5	Current Year - 1
38	200302	Current Month - 16	2003/02	2003	2	1	Current Qtr - 5	Current Year - 1
123	200301	Current Month - 17	2003/01	2003	1	1	Current Qtr - 5	Current Year - 1
36	200212	Current Month - 18	2002/12	2002	12	4	Current Qtr - 6	Current Year - 2
35	200211	Current Month - 19	2002/11	2002	11	4	Current Qtr - 6	Current Year - 2
34	200210	Current Month - 20	2002/10	2002	10	4	Current Qtr - 6	Current Year - 2
33	200209	Current Month - 21	2002/09	2002	9	3	Current Qtr - 7	Current Year - 2
32	200208	Current Month - 22	2002/08	2002	8	3	Current Qtr - 7	Current Year - 2
31	200207	Current Month - 23	2002/07	2002	7	3	Current Qtr - 7	Current Year - 2

Figure 26: Processing Calendar Data

Using the traditional solution is very similar to the previous multi-pass scenario and also requires five (5) steps are. The first step is to create a table to house temporary results.

```
CREATE TABLE This_and_Last
   (Cat_Id integer, This_Month_Calls integer, Last_Month_Calls integer)
```

Next, the number of calls by category for the current month is inserted into the work table.

```
INSERT INTO This_and_Last (Cat_Id, This_Month_Calls, Last_Month_Calls)
      SELECT MP_T1_Calls.Cat_ID, sum(MP_T1_Calls.Nbr_Calls), 0
      FROM MP_T1_Calls, MP_T1_Calendar
      WHERE (MP_T1_Calendar.Month_ID = MPP_T1_Calls.Month_Id
      AND   (MP_T1_Calendar.Month_Rpt_Desc ='Current Month')
      GROUP BY MP_T1_Calls.Cat_Id
```

The third phase is to load the number of calls for the previous month by category into the table.

```
INSERT INTO This_and_Last (Cat_Id, This_Month_Calls, Last_Month_Calls)
      SELECT MP_T1_Calls.Cat_ID, 0, sum(MP_T1_Calls.Nbr_Calls)
```

```
      FROM MP_T1_Calls, MP_T1_Calendar
      WHERE (MP_T1_Calendar.Month_ID = MP_T1_Calls.Month_Id)
      AND   (MP_T1_Calendar.Month_Rpt_Desc = 'Current Month - 1')
      GROUP BY MP_T1_Calls.Cat_Id
```

The values currently within the temporary table are shown in Figure 27.

	Cat_Id	This_Month_Calls	Last_Month_Calls
1	1	25	0
2	1	0	5
3	2	6	0
4	2	0	4
5	3	24	0
6	3	0	36
7	4	0	18
8	4	12	0
9	5	0	30
10	5	25	0
11	6	0	20
12	6	40	0
13	7	0	7
14	7	14	0

Figure 27: Values within the temporary table after two passes

At this point the required results are in the temporary table. The next stage is to retrieve the results for the report by joining the temporary table to the category descriptions.

```
SELECT MP_T1_Cat.Cat_Id, Category, sum(This_Month_Calls),
      sum(Last_Month_Calls)
FROM  This_and_Last JOIN MP_T1_Cat
      ON MP_T1_Cat.Cat_Id = This_and_Last.Cat_Id
GROUP BY MP_T1_Cat.Cat_Id, Category
ORDER BY MP_T1_Cat.Cat_Id
```

The results that are finally obtained are correct.

	CAT_ID	CATEGORY	Sum(This_Month_Calls)	Sum(Last_Month_Calls)
1	1	Electronics	25	5
2	2	Food	6	4
3	3	Gifts	24	36
4	4	Health & Beauty	12	18
5	5	Household	25	30
6	6	Kid's Korner	40	20
7	7	Travel	14	7

Figure 28: Final results using a traditional multi-pass approach

And last but not least, one must always cleanup after themselves. The final step is to delete the temporary work table.

```
DROP TABLE This_and_Last
```

In this example, two passes of the database schema were required in addition to pulling the final results for reporting. Along with this activity is the creation and deletion of the temporary table. This “traditionalist” approach is very database centric and assumes very limited capability on the reporting/analysis tier. Not being bound by such limitations Business Objects offers a few possible solutions.

One way to solve this situation is to create multiple queries, one for this month and another for last month. As Web Intelligence now allows multiple queries this is now a legitimate solution. The first step is to pull the calendar into the Universe (Figure 29).

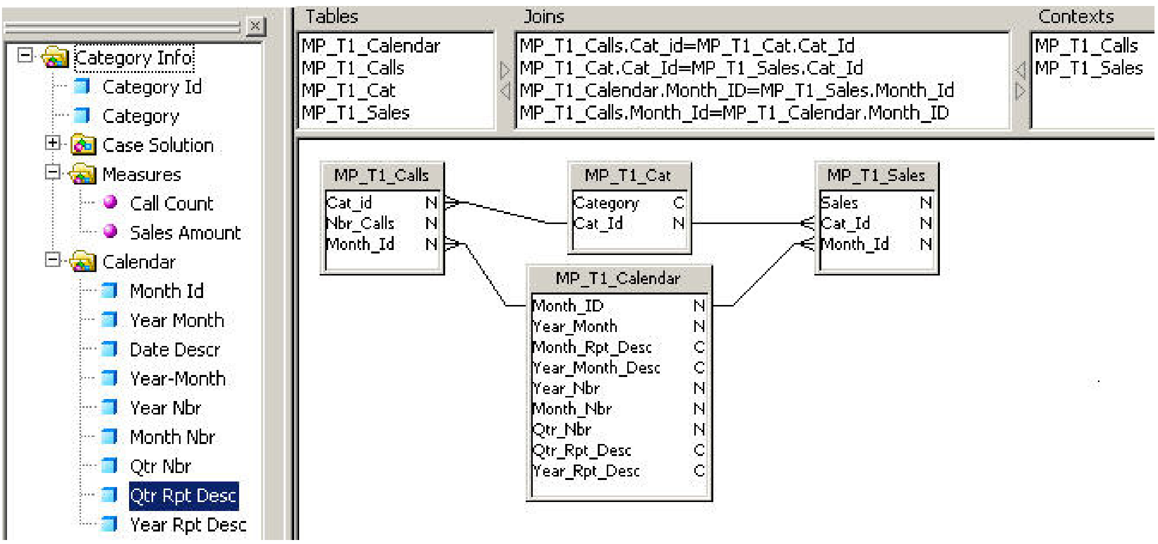


Figure 29: Calendar added to universe

The queries would differ only within *Query Filters* section of the query panel; the *Result Objects* in most cases are identical (Figure 30 and Figure 31).

Figure 30: Current Month query

Figure 31: Previous Month query

There area few reasons why this is not the optimal solution. The first is that the burden is placed on the report developer. Multiple queries have to be created; one query for each desired timeframe is required. If this is a common request then this quickly becomes a tiresome process. Secondly, this solution will only work with Web Intelligence and Desktop Intelligence; It will not work with Crystal Reports or Query as a Web Service. In addition dimensions are automatically merged but measures are not (Figure 32). In Figure 32 the *Sales Amount* initially displayed is from the previous month query. The *Sales Amount* for the current month is not shown at all.

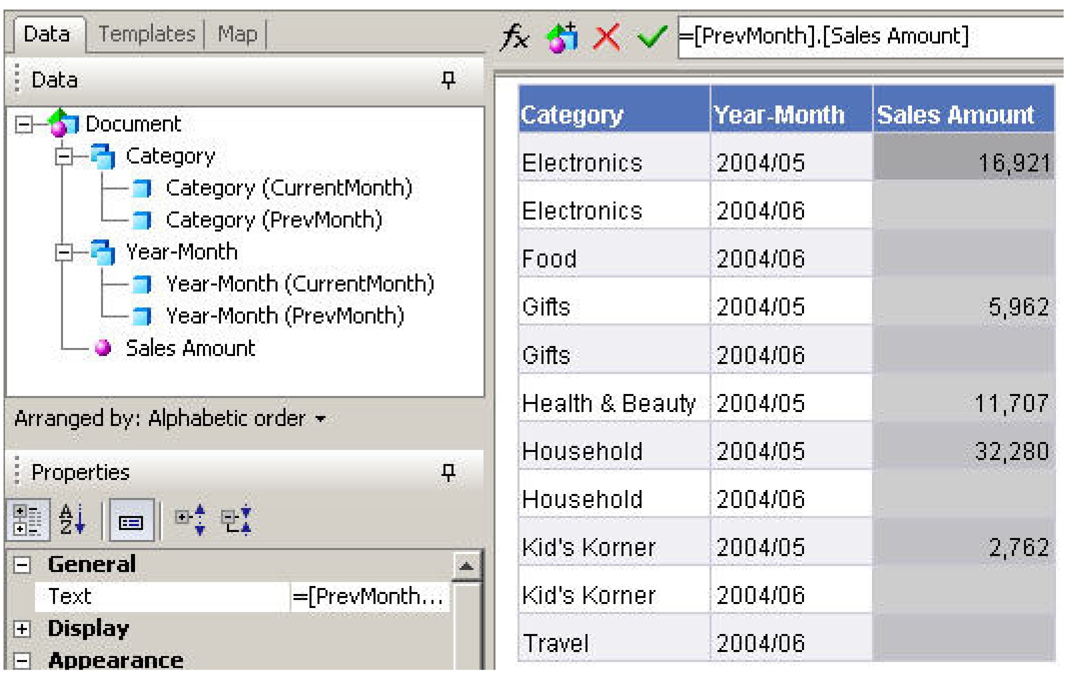


Figure 32: Initial query results using multiple queries

To correct this and also provide meaningful column headers two report variables, one for current month (Figure 33) and another for previous month (Figure 34), are created.

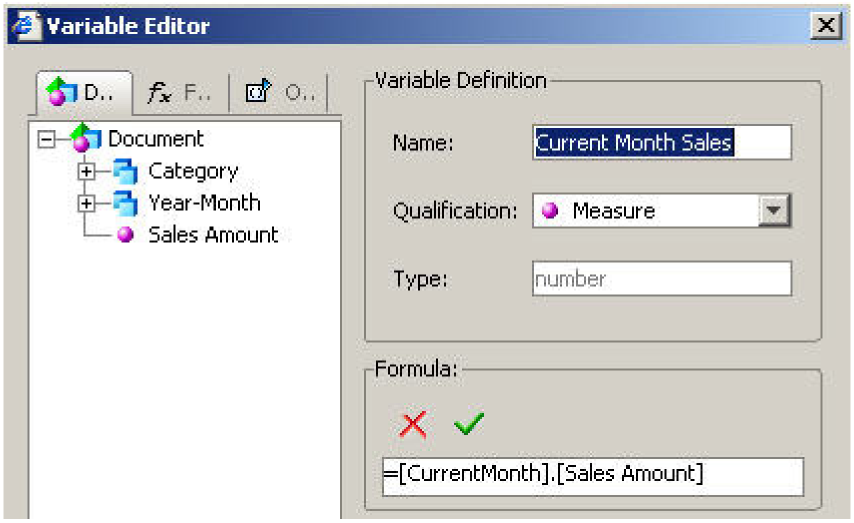


Figure 33: Report variable for current month sales

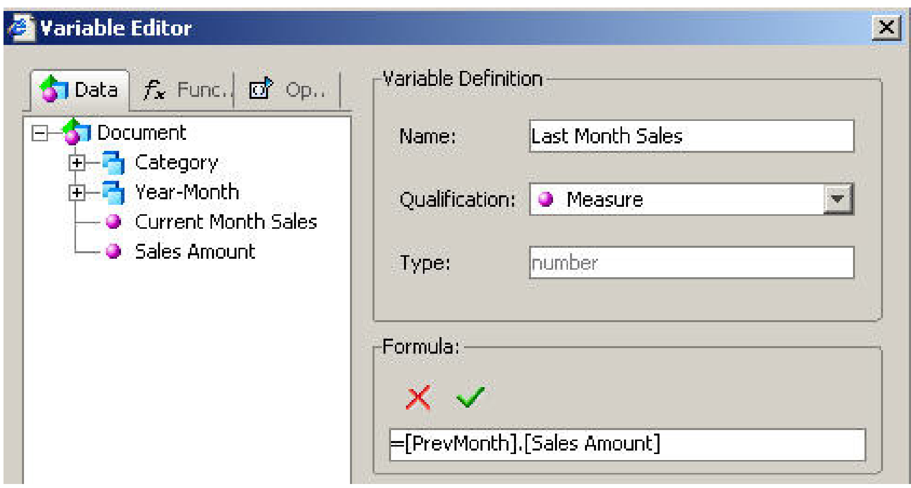


Figure 34: Report variable for previous month sales

Once the variables are created they can be used to create the correct and expected results (Figure 35).

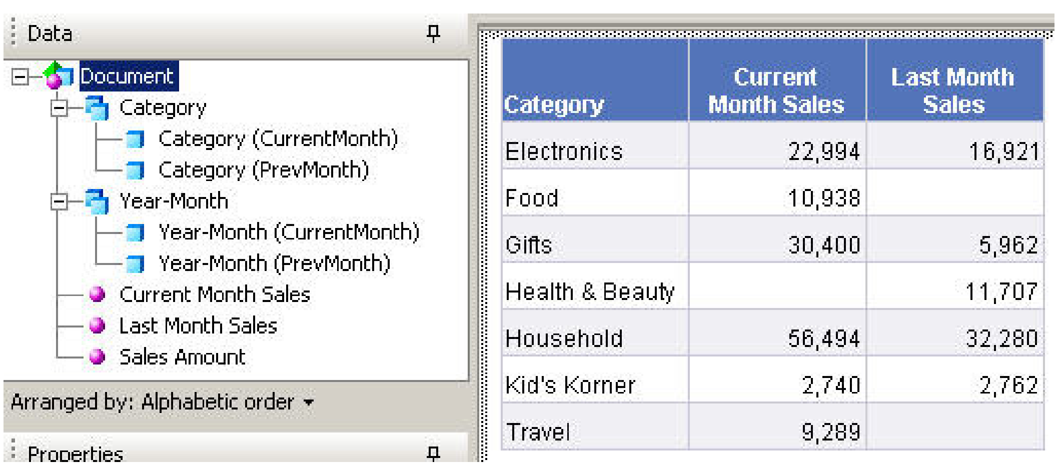


Figure 35: Final results using multiple queries

To satisfy Kimball's definition of multi-pass one could also argue that this solution fails the combining of the result sets in an "intelligent way" test. But if separate Universe objects can be created that would then force multiple SQL statements to be generated then Kimball's definition would be met.

One common solution that does this is to embed the desired time comparison within the Universe object by using a case statement. As seen in Figure 35 a comparison is made against the date table for the previous month indicator. If this is the case then the sales amount is included in the sum otherwise a zero is substituted.

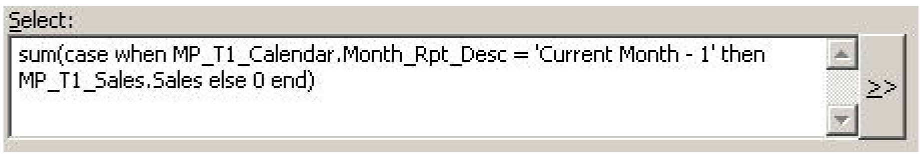


Figure 36: Definition of Last Month Sales using a CASE statement

Figure 37 shows the object definition for the sales amount for the current month. When contrasted to the definition for the previous month calls the difference lies in the timeframe comparison.

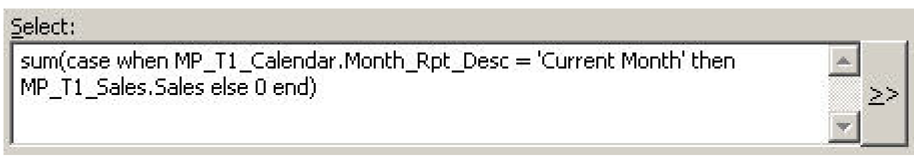


Figure 37: Definition of This Month Sales using a CASE statement

When the objects are used in a query panel (Figure 38) only one SQL statement is generated (Figure 39).

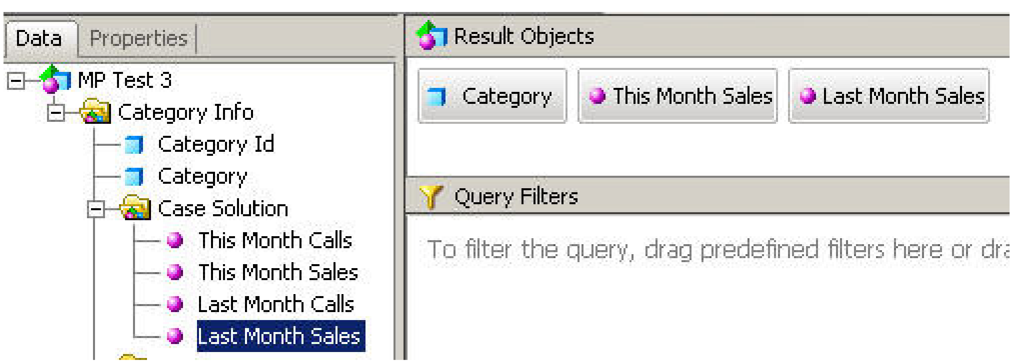


Figure 38: Query using CASE based measures

The one generated SQL statement (Figure 39) retrieves the sales for this and last month.

```
SELECT
  MP_T1_Cat.Category,
  sum(case when MP_T1_Calendar.Month_Rpt_Desc = 'Current Month' then MP_T1_Sales.Sales else 0 end),
  sum(case when MP_T1_Calendar.Month_Rpt_Desc = 'Current Month - 1' then MP_T1_Sales.Sales else 0 end)
FROM
  MP_T1_Cat,
  MP_T1_Sales,
  MP_T1_Calendar
WHERE
  ( MP_T1_Cat.Cat_Id=MP_T1_Sales.Cat_Id  )
  AND  ( MP_T1_Calendar.Month_ID=MP_T1_Sales.Month_Id  )
GROUP BY
  MP_T1_Cat.Category
```

Figure 39: Generated SQL for number of sales amount for current and previous month

The returned results (Figure 40) are correct.

Category	This Month Sales	Last Month Sales
Electronics	22,994	16,921
Food	10,938	0
Gifts	30,400	5,962
Health & Beauty	0	11,707
Household	56,494	32,280
Kid's Korner	2,740	2,762
Travel	9,289	0

Figure 40: Results using CASE based measures

The results were correct, only one query had to be built, no report filters as required, and no report variables were needed. So why not use this method?

The issue with this solution has nothing to do with accuracy of results but with performance. Every row of the fact table has to be retrieved in order to make the date comparison. So in essence a CASE based measure is forcing a full table scan on the fact table. When fact tables can routinely contains millions of rows this is not an ideal resolution. To eliminate the table scan on the fact table the rows of the fact table meeting the time criteria should be determined by query filter based on a join to the calendar table. In most situations an index will exist on the column used in the query filter further increasing query performance. As the generated SQL shows (Figure 39), all the measures from each fact table are included in one SELECT statement regardless of their denoted timeframe. Ideally if separate Universe objects could be created that would then force a separate SELECT statement to be generated for each timeframe then not only would Kimball's definition would be met but be met in an optimal performing way.

So now there are two criteria that have been identified: separate Universe objects for each timeframe and identification of fact table rows by a filter on the join to the calendar table. To accomplish this, the first step is to bring in the tables for the Universe (Figure 41).

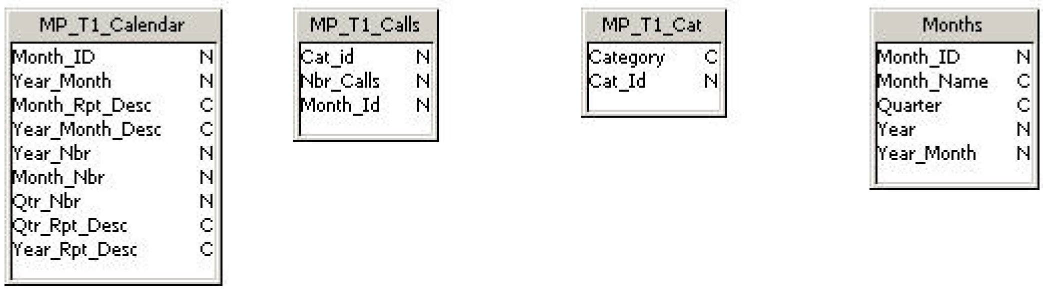


Figure 41: Tables required for Universe

An alias for each grain of measurement is created of the fact table (Figure 42).

Figure 42: Table alias creation

One alias is created to represent this month calls (Figure 43). A similar alias is created to represent last month calls.

Figure 43: Create alias for this month

Once the alias tables have been created (Figure 44), the original fact table is no longer part of the Universe,

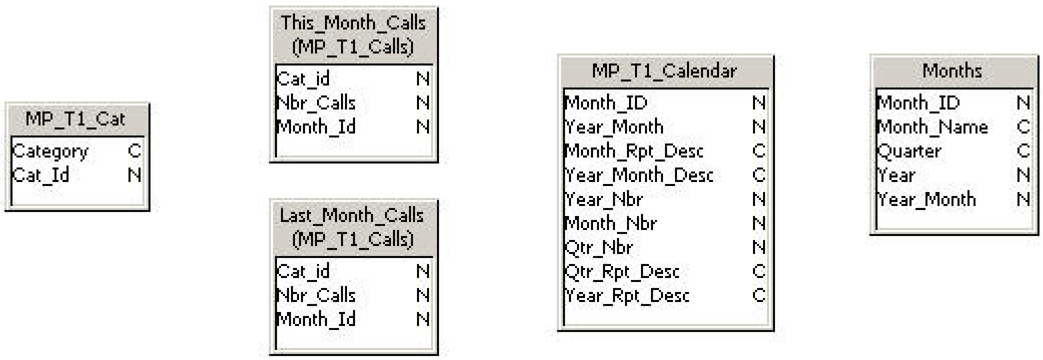


Figure 44: Alias tables present

The next step is to create the table joins and automatically detect the contexts. The result of these operations is shown in Figure 45.

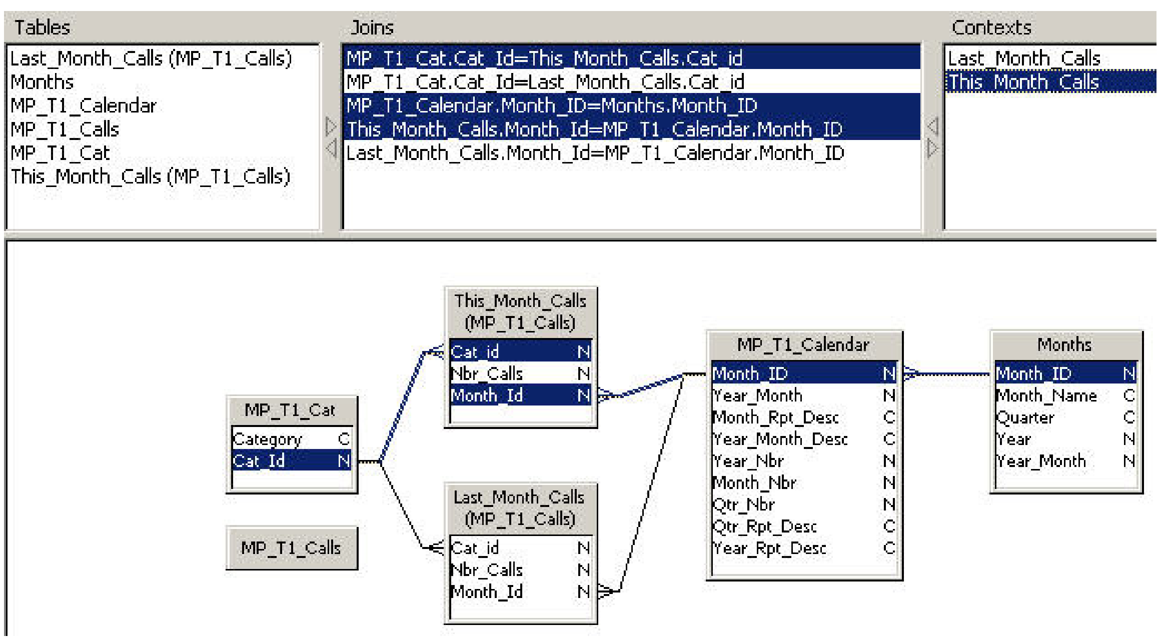


Figure 45: Alias tables, table joins, and detected contexts

The join (Figure 46) to *This_Month_Calls* references only the *Month_Id* column calendar column. The same is true for *Last_Month_*Calls (Figure 47). At this time there is not any difference in the way the aliases have been used in the Universe.

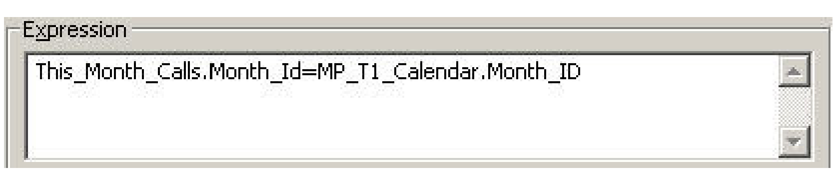


Figure 46: Join between This_Month_Calls and MP_T1_Calendar

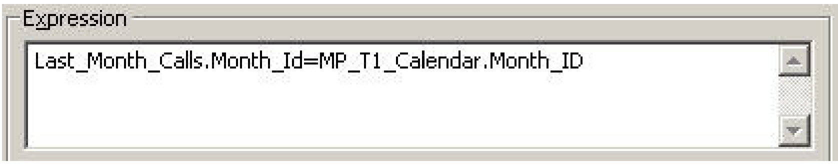


Figure 47: Join between Last_Month_Calls and MP_T1_Calendar

So why create the aliases? The difference will be seen when objects are created for query panel use (Figure 48).

Figure 48: Objects for testing different grains of measurement

The object *This Month Calls* (Figure 49) references one alias while *Last Month Calls* (Figure 50) references the other alias. In addition a *Where* clause has been specified for each object. The *Where* clause enforces the relative time period contained in the calendar table. When used in conjunction with the join specified between the alias and calendar table only the *Month_Id*s for that relative period are returned from the alias table. So for *This_Month_Calls* only the rows having a *Month_Id* that matches those in the calendar table specified as *Current Month* are returned.

Figure 49: Definition of the object: This Month Calls

Figure 50: Definition of the object: Last Month Calls

So why not include the object's *Where* clause as part of the join to the calendar table? That would be easier than having to specify a *Where* clause on individual objects It indeed would be easier but then that join would only included in the generated SQL if objects from both the calendar and the alias tables were included as part of the query.

Building the query to return both the number of calls for this month and last month is now an easier process. A single query without any prompts or query filters can produce the necessary SQL to return the proper results (Figure 51).

Figure 51: Query panel result objects for this and last month calls

The contexts detected upon use of the alias tables result in the creation of multiple SQL statements (Figure 52 and Figure 53).

Figure 52: Generated SQL for This Month Calls

Figure 53: Generated SQL for Last Month Calls

After the SELECT statements are executed, the two result sets are combined by the Business Objects server for use in a report (Figure 54). Keep in mind that a JOIN_BY_SQL parameter setting of **Yes** can push more of the processing to the database.

Category	This Month Calls	Last Month Calls
Electronics	25	5
Food	6	4
Gifts	24	36
Health & Beauty	12	18
Household	25	30
Kid's Korner	40	20
Travel	14	7

Figure 54: Resulting report for the number of this and last month calls

By also having the calendar table (MP_T1_Calendar) as part of the contexts, the calendar objects can also be used in the query. If the *Year Month* object can also used in the query (Figure 55), the number of calls by month allow for a quick verification of the results.

Figure 55: The Year Month object added to the query

The new results when formatted as a cross tab (Figure 56) show that all the calls for last month occurred in 2004/05 while all the call for this month were in 2004/06. With the sample data provided this is what was expected.

Year Month	2004/05		2004/06	
Category	This Month Calls	Last Month Calls	This Month Calls	Last Month Calls
Electronics		5	25	
Food		4	6	
Gifts		36	24	
Health & Beauty		18	12	
Household		30	25	
Kid's Korner		20	40	
Travel		7	14	

Figure 56: This and last month calls by month

What if the sales for this month and last months are also needed? The same process is repeated. The sales table is added to the Universe from which two aliases are created, one for this month and another for last month. The new aliases are then joined to the dimension tables after which the detect contexts operation is executed again. Once the contexts are added, measure objects are then added to represent this month sales and last month sales. As the number of facts grow and the number of the various time dimensions increase, the measure objects should be arranged in separate folders to allow easier navigation. All of this can be viewed in Figure 57.

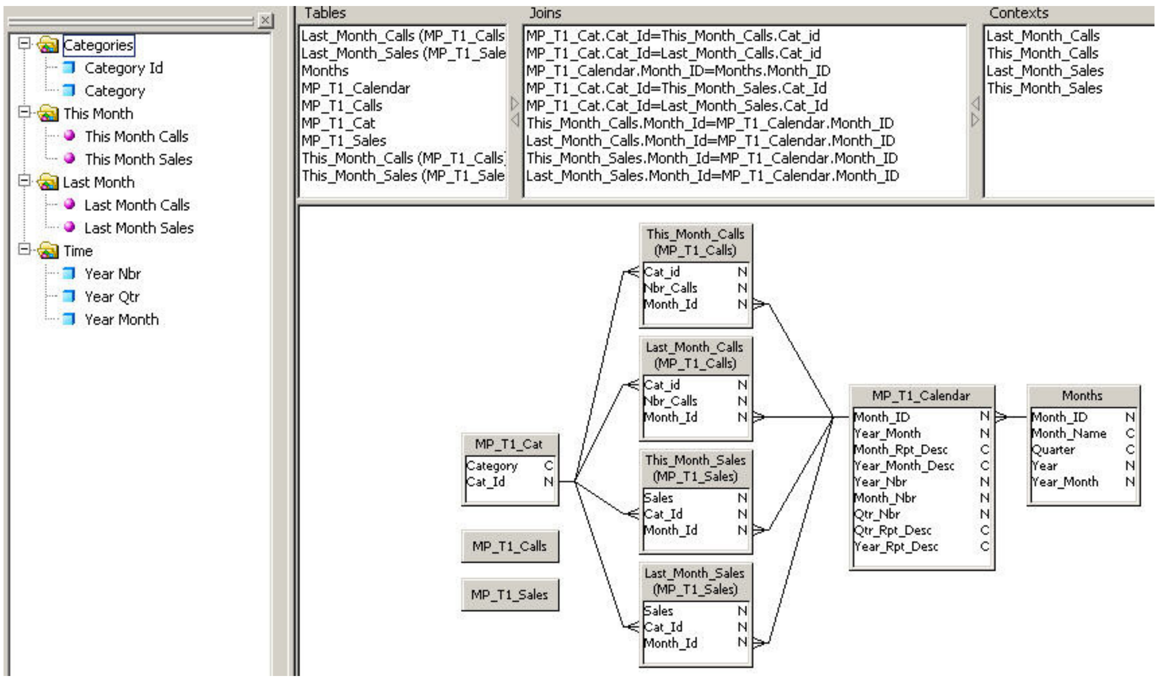


Figure 57: Universe with additional fact table aliases, contexts, and measure objects.

Now to produce a report to show both the sales amounts and the number of calls for this month and last month, only one query still needed as depicted in Figure 58.

Figure 58: Query panel for this and last month sales and calls

Without the additional Universe design, it is very possible that four queries would have to be built manually. Instead the alias tables and context create the four queries for us (Figure 59).

Figure 59: Four queries generated

The result sets are again combined by the Business Objects server for presentation in a report (Figure 60).

Category	This Month Calls	This Month Sales	Last Month Calls	Last Month Sales
Electronics	25	22,994	5	16,921
Food	6	10,938	4	
Gifts	24	30,400	36	5,962
Health & Beauty	12		18	11,707
Household	25	56,494	30	32,280
Kid's Korner	40	2,740	20	2,762
Travel	14	9,289	7	

Figure 60*: Report showing this and last figures for sales and calls.*

Many other objects could be created to represent many other time timeframes by creating objects from various alias tables and adjusting their *Where* clause.

Object	**Alias**	**Where Clause**
Current Qtr Sales	This_Qtr_Sales	MP_T1_Calendar.Qtr_Rpt_Desc = 'Current Qtr'
Same Qtr Last Year Sales	Same_Qtr_LY_Sales	MP_T1_Calendar.Qtr_Rpt_Desc = 'Current Qtr - 3'
Current Year Sales	This_Yr_Sales	MP_T1_Calendar.Year_Rpt_Desc = 'Current Year'
Last Year Sales	Last_Yr_Sales	MP_T1_Calendar.Year_Rpt_Desc = 'Current Year - 1'
Rolling 3 Month Sales	Roll3_Month_Sales	MP_T1_Calendar.Month_Rpt_Desc = 'Current Month' OR MP_T1_Calendar.Month_Rpt_Desc = 'Current Month - 1' OR MP_T1_Calendar.Month_Rpt_Desc = 'Current Month - 2'

Why bother with alias tables? Why not just create the objects with their unique *Where* clauses from the original table? The alias tables are created to require the need for contexts. The presence of context forces the query engine to generate separate SQL statements for each time grain. Without the separate SQL statements the *Where* clauses are combined with the *AND* operator. With the alias tables separate SQL statements are created for each time grain separating the *Where* clauses. Without the context, the query from Figure 64 to return the Category, This Month Calls, and Last Month Calls would not return any data. To prove this let's create a simple test Universe without any alias tables (Figure 61).

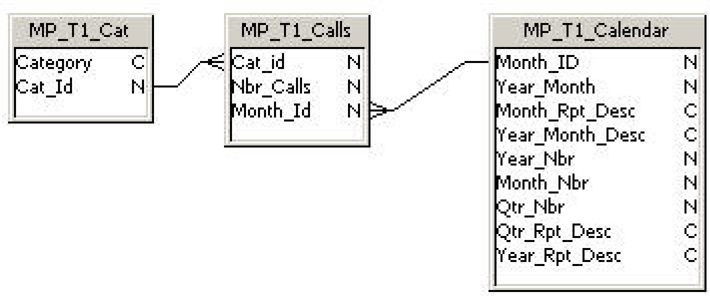


Figure 61: Simple test Universe without any alias tables or contexts.

The objects for *This Month Calls* (Figure 62) and *Last Month Calls* (Figure 63) are now created from the MP_T1_Calls table instead of an alias.

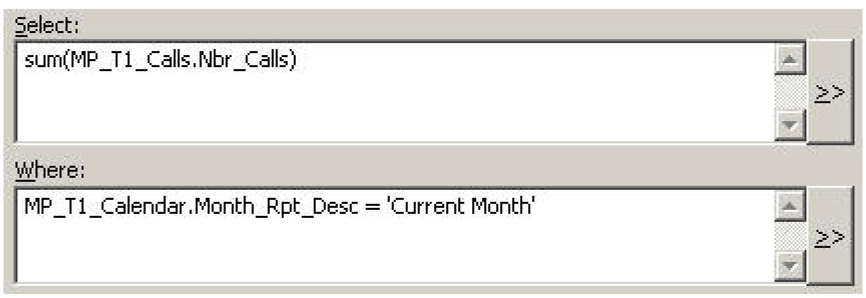


Figure 62: This Month Calls without an alias table

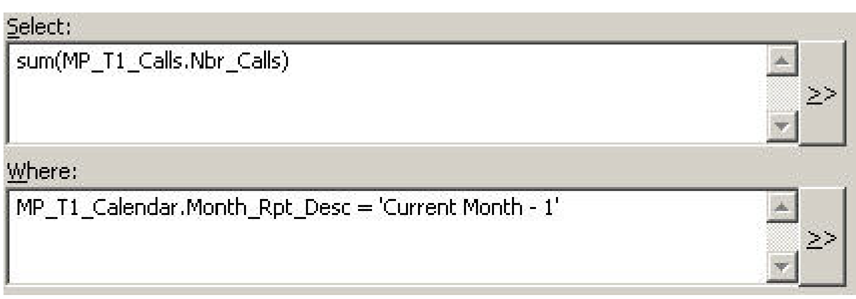
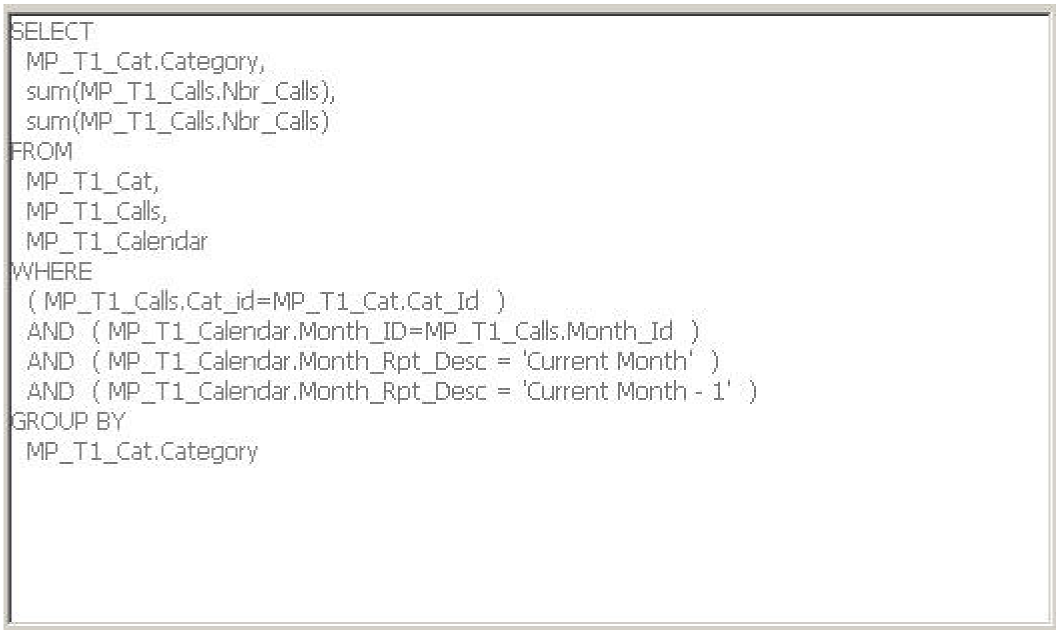


Figure 63: Last Month Calls without an alias table

Now a query is completed to return the *Category*, *This Month Calls*, and *Last Month Calls* (Figure 64). Viewing the SQL one can see that the *Where* clauses have been *AND*ed together.



```
SELECT
  MP_T1_Cat.Category,
  sum(MP_T1_Calls.Nbr_Calls),
  sum(MP_T1_Calls.Nbr_Calls)
FROM
  MP_T1_Cat,
  MP_T1_Calls,
  MP_T1_Calendar
WHERE
  ( MP_T1_Calls.Cat_id=MP_T1_Cat.Cat_Id  )
  AND  ( MP_T1_Calendar.Month_ID=MP_T1_Calls.Month_Id  )
  AND  ( MP_T1_Calendar.Month_Rpt_Desc = 'Current Month'  )
  AND  ( MP_T1_Calendar.Month_Rpt_Desc = 'Current Month - 1'  )
GROUP BY
  MP_T1_Cat.Category
```

Figure 64: SQL with combined WHERE clauses

As there is not any *Month_Id* with a *Month_Rpt_Desc* that is both ‘Current Month’ and ‘Current Month – 1’ no data should be returned. Upon execution that is the message returned (Figure 65).

Figure 65: No data available

To accomplish the second multi-pass scenario of mixing the grains of measurement in the same query liberal use of alias tables and contexts are often the answer. These techniques will force the creation of multiple SQL statements whose result sets are combined together. In summary, the steps are:

1. Create an alias of the fact table for each required measurement grain
2. Join the alias tables to all appropriate dimension tables
3. Detect and create the contexts
4. Create the Universe measure objects from the aliased fact tables
5. Place the required *Where* clauses on the measure objects that indicate the appropriate measurement grain

Scenario 3: Advanced Calculations

Two of the five multi-pass scenarios have now been covered. Next up on the list is the ability to define calculations that require an end result as one of its factors. Often these are calculations that one would normally accomplish at the report level. With recent additions to the Web Intelligence variable editor, it is close to being on par with Desktop Intelligence. So why would one want to deviate from this strategy? There are several reasons that come quickly to mind.

1. The first reason is forcing the user to duplicate the same variable calculation from report to another. The possibly for a simple type is ever present. If the formula were to change many reports have to be touched in order for all reports to have the same results.
2. The next reason is that it gives the impression of being hard to use, especially during a demonstration. After visiting the query panel to build the query and watching the results be displayed there is another step, a visit to the variable editor. It presents a better solution to be able to pull out calculations from the Universe while in the query panel.
3. The “One version of the truth” mantra falls apart when a prospect or end user has to repeatedly create the same variable from one report to another. Not having the ability to share report variables across different reports conflicts with one of our selling points. If these calculations can be available in the Universe then “One version of the truth” is once again valid.
4. Some competitors have the ability to create these types of variables within their semantic layer. Not showing such capability puts us at a disadvantage.

So are report variables worthless? The best way to position the variable editor may be is that it gives the end user the ability to create variables that are not yet present in the semantic layer. It ties in nicely with the “Not being held hostage by IT” message. If the owners of the semantic layer have not yet created a variable that is needed for reporting, the report creation is not held up waiting on a new version of the semantic layer. The report can still be created and published while the request for a new semantic layer variable works its way through the system. Some of our competitors do not have a calculation engine within their tool set, depending on the database for all calculations. If the database can not produce the calculation, then there is not another option. This also applies in situations when it may not be possible to create the calculations within the database. In addition report variables work extremely well with reports with the drilling option activated. Report variables can be created to maintain their context while objects are added to or removed from the report. The next release of Business Objects, XI 3.0, will offer some additional capabilities in this regard but currently report variables are the best option.

In this example the ration of a category's sales to total sales is needed for a report. The initial query contains only two objects (Figure 66).

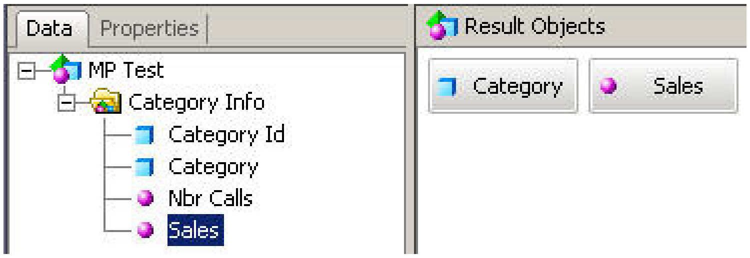


Figure 66: Result objects for ratio example

The results are returned as two columns (Figure 67). A third column showing the ratio to total sales is needed.

Category	Sales
Electronics	39,915
Food	10,938
Gifts	36,362
Health & Beauty	11,707
Household	88,774
Kid's Korner	5,502
Travel	9,289

Figure 67: Initial results

To add the ratio for each category's sales to the overall total sales is much easier than other possible calculations as it is one of the predefined calculations. Simply highlight the *Sales* column, click on the *Percentage* option (Figure 68).

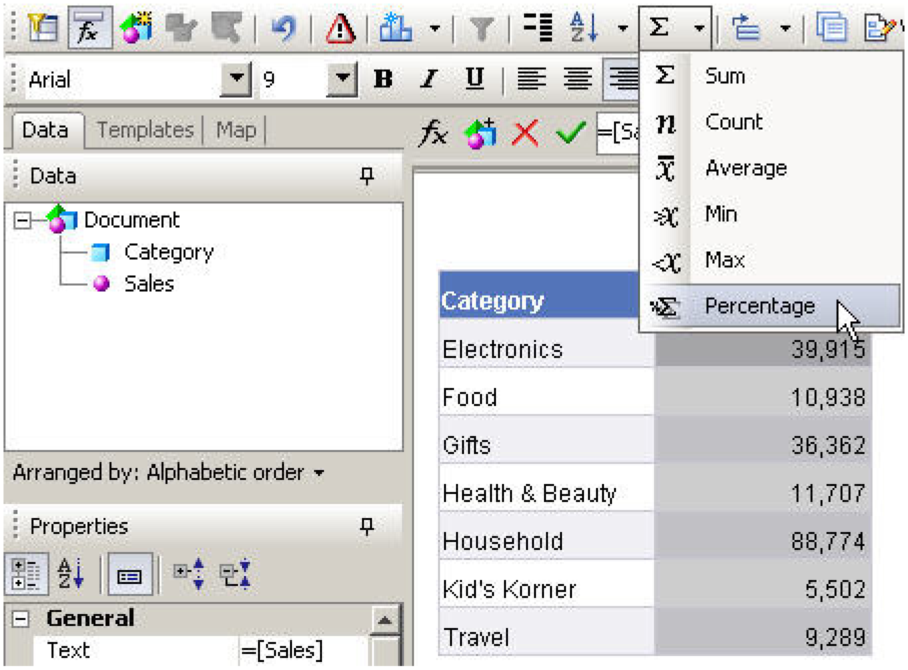


Figure 68: Add the percentage to total sales calculation

Now each category's contribution to total sales as a percentage is displayed on the report (Figure 69). In this case the variable editor did not even have to accessed, allowing a simple workflow to be presented.

Category	Sales	Percentage
Electronics	39,915	19.71%
Food	10,938	5.40%
Gifts	36,362	17.96%
Health & Beauty	11,707	5.78%
Household	88,774	43.84%
Kid's Korner	5,502	2.72%
Travel	9,289	4.59%
	Percentage:	100.00%

Figure 69*: Report with each category's contribution to total sales*

How would such a calculation be achieved using traditional multi-pass SQL? At a minimum four (4) steps are required.

1. Create a temporary table containing a single column for the grand total of sales.

```
CREATE TABLE Total_Sales
      (All_Sales decimal(10,2))
```

2. Insert into this temporary table the grand total of all sales.

```
INSERT INTO Total_Sales (All_Sales)
      SELECT sum(MP_T1_Sales.Sales)
      FROM MP_T1_Sales
```

3. Select the total sales, the category dimension table for the category name, and also the ratio which is sum by category divided by the sum from the temporary table.

```
SELECT Category, sum(MP_T1_Sales.Sales),
      (cast(sum(MP_T1_Sales.Sales) as
      decimal(10,4))/(All_Sales) ) * 100 as Pct_of_Sales
      FROM MP_T1_Sales, Total_Sales, MP_T1_Cat
      WHERE MP_T1_Cat.Cat_Id = MP_T1_Sales.Cat_Id
      GROUP BY Category, All_Sales
      ORDER BY Category
```

4. Drop the temporary table.

```
      DROP TABLE Total_Sales
```

A Cartesian product will occur in step 3 when the result from the temporary table is used as no join exists between the temporary and the sales fact table. The report results are still correct since there is only one row within the temporary table.

A derived table is required to deliver the sales ratio as a Universe object. The derived table is created in the Universe. The SQL is shown in Figure 70. In this case the category id, sum of sales for the category id, total sales, and the ratio of the category sales to the total sales. The derived within the Universe contains a derived table itself. The `SELECT sum(MP_T1_Sales.Sales) as All_Sales FROM MP_T1_Sales)as Company` portion of the derived table, contained in the primary FROM clause, corresponds to step 2 of the multi-pass scenario. The `(sum(MP_T1_Sales.Sales) / Company.All_Sales) as Sales_Ratio` of the first SELECT clause corresponds to step 3 of the multi-pass scenario.

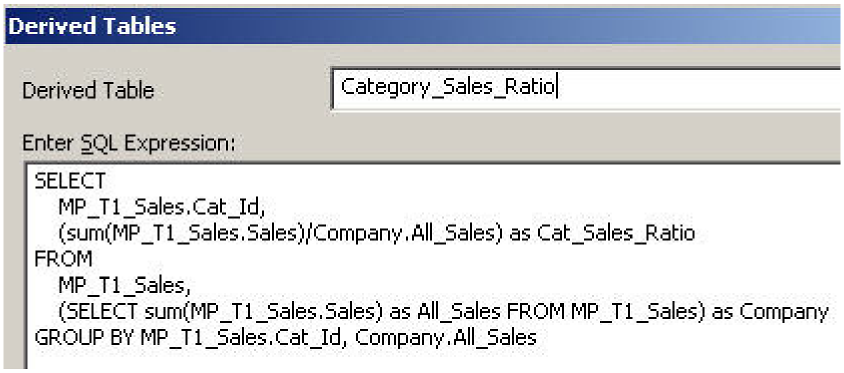


Figure 70: Derived table syntax for sales ratio

The next step is to join the derived table to the dimension table. Once this is done be sure that it is also included in a context. Both of these can be seen in Figure 71.

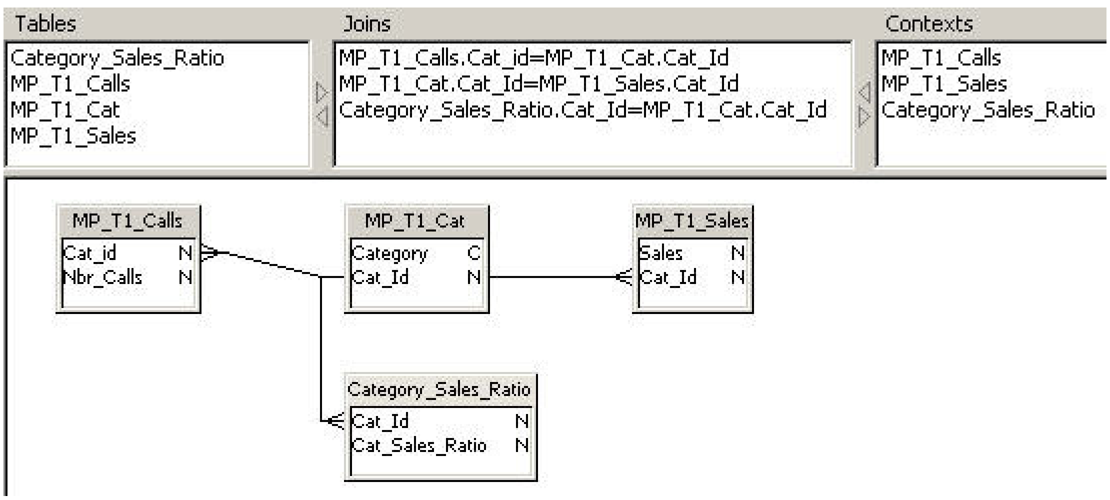


Figure 71: Derived table, Category_Sales_Ratio, with context and join identified

Objects can be created from the derived table. In this case the *Category Sales Ratio* object (Figure 72) is created so that the ratio can be used in a query panel.

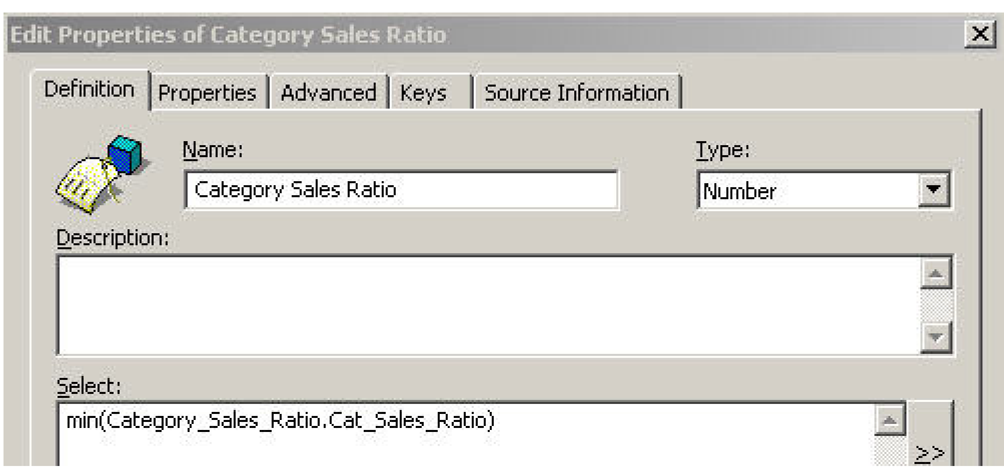


Figure 72: Definition of the Category Sales Ratio object

An aggregate function, in this example the *min* function, is added so that the object will not be added to the GROUP BY clause. The syntax of a SELECT statement requires that all objects that are not part of an aggregate function must be part of the GROUP BY clause. If the aggregate function is omitted, *Category Sales Ratio* will need to be part of the GROUP BY clause since no aggregate function is present. When Business Objects combines SQL statements, if the GROUP BY clauses of the SQL statements are identical then a *Join* operation is performed to combine the result sets. If the GROUP BY clauses of the SQL statements differ then a *Synchronization* operation is performed to combine the result sets. Only if a *Join* operation is possible will the JOIN_BY_SQL parameter of **Yes** be respected. If the result set combination action is *Synchronization* the JOIN_BY_SQL parameter is ignored regardless of its setting. When this occurs the generated SQL can not be used by Crystal Reports nor Query as a Web Service. The *min* function used in this example has no bearing on the results returned by a query using this object as there is only one value per category (Figure 73). As only one row is returned per category the following aggregate functions could be used: avg, max, min, and sum.

Cat_Id	Cat_Sales_Ratio
1	.197123
2	.054018
3	.179576
4	.057816
5	.438418
6	.027172
7	.045874

Figure 73: Values returned by the derived table, Category_Sales_Ratio

Within the query panel, the object is used as any other (Figure 74).

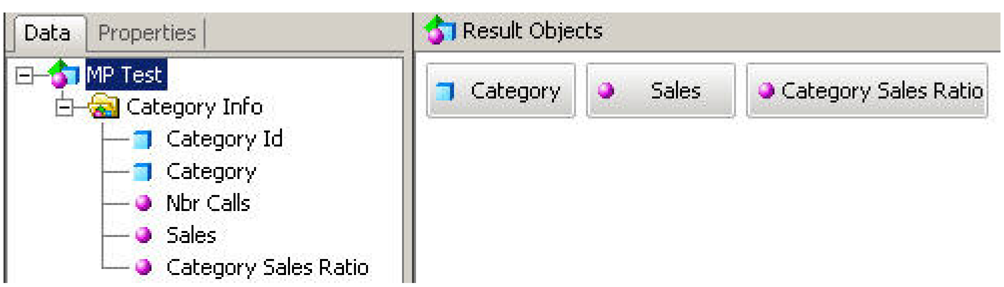


Figure 74: Sales Ratio used within query panel

The resulting SQL of the query is the joining of two SQL statements. One references the derived table. That SQL is generated as:

```
SELECT
  MP_T1_Cat.Category,
  min(Category_Sales_Ratio.Cat_Sales_Ratio)
FROM MP_T1_Cat,
  ( SELECT MP_T1_Sales.Cat_Id,
    (sum(MP_T1_Sales.Sales)/Company.All_Sales ) as Cat_Sales_Ratio
```

```
FROM MP_T1_Sales,
    ( SELECT sum(MP_T1_Sales.Sales) as All_Sales FROM MP_T1_Sales ) as
Company
GROUP BY MP_T1_Sales.Cat_Id, Company.All_Sales ) Category_Sales_Ratio
WHERE ( Category_Sales_Ratio.Cat_Id=MP_T1_Cat.Cat_Id )
GROUP BY MP_T1_Cat.Category
```

The first pass is shown in blue and the result set of this SELECT statement becomes known as *Company.* The purpose of *Company* is to calculate the total overall sales that is then labeled *All_Sales*. The second pass is highlighted in green. The result set of this SELECT becomes known as *Cat_Sales_Ratio*. It uses *All_Sales* from the first pass to calculate the sales ratio by category.

Once the results are returned no calculations are needed at the report level (Figure 75). Imagine if the report requirement was to only show the category and the sales ratio. Without the *Category Sales Ratio* object the report creator or ad hoc user would have to know to return the *Sales* object and then do a report calculation. Having the *Category Sales Ratio* object eliminates some of the guess work.

Category	Sales	Category Sales Ratio
Electronics	39,915	19.71%
Food	10,938	5.40%
Gifts	36,362	17.96%
Health & Beauty	11,707	5.78%
Household	88,774	43.84%
Kid's Korner	5,502	2.72%
Travel	9,289	4.59%

Figure 75: Query results containing Sales Ratio by Category

The measure *Category Sales Ratio* only has meaning when used with the *Category* (or *Category Id*) dimension object. Seeing the values of *Category Sales Ratio* without the category values is meaningless. Nonetheless nothing prevents the ad hoc user or report builder from only using *Category Sales Ratio* in the query panel. In addition, nothing prevents the report consumer from removing the *Category* column from the report during analysis. However, there is a setting within the Universe that can aid in preventing misinterpretation of results. The Properties tab of measure objects (Figure 76) contains the setting for *(Choose) how this measure will be projected when aggregated*. The values offered are: Average, Count, Max, Min, None, and Sum. This setting determines the action taken by report level aggregation engine after results have been returned by the database. This setting does not affect SQL generation or database aggregation. By default, the same function is employed as in the object's definition. In this case, since the object definition is `min(Category_Sales_Ratio.Cat_Sales_Ratio)`, the default setting for the projection aggregate will be *Min*. In order to maintain some semblance of reasonable results, the projection aggregate should be set to either *None* or *Sum*. The other settings can lead to very peculiar results if the ratio is viewed out of context. Even though *None* and *Sum* lead to very dissimilar results when the ratio is viewed without the category information, a case can be made for either setting.

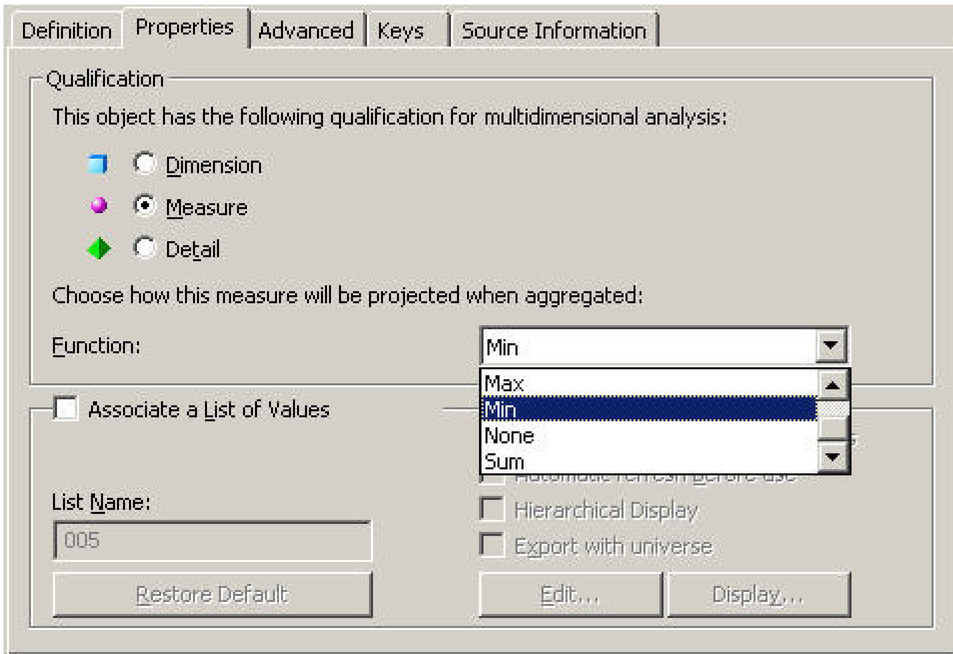


Figure 76: Properties tab for the Category Sales Ratio Object

In order to see the effects of the projection aggregate a new table is added to the schema. A *Region* table is added along with a region indicator to the sales table as noted in Figure 66. Since the region indicator exists only in the sales table, the region table is only part of the sales (*MP_T1_Sales*) context.

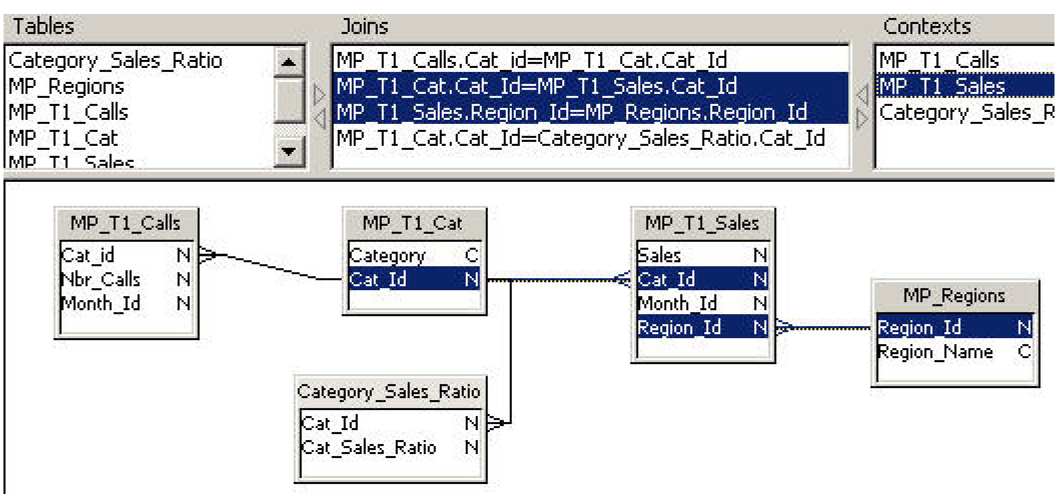


Figure 77: Universe with region table added

In all the projection examples which will follow, the query panel remains unchanged. The query is constructed as seen in Figure 78.

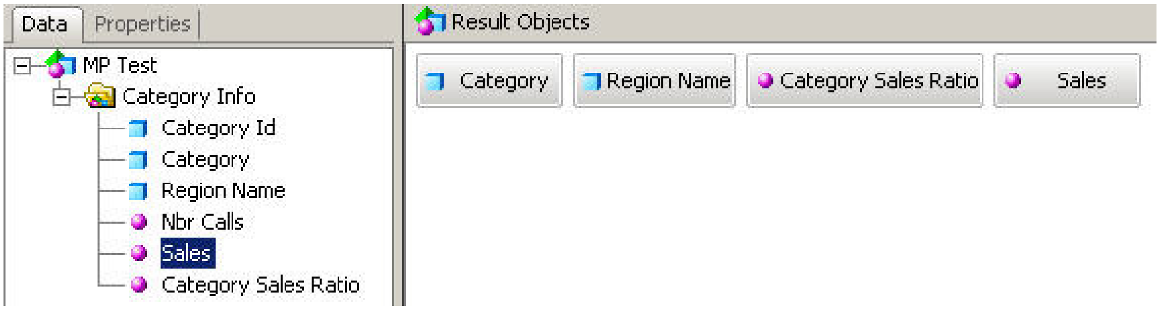


Figure 78: Query panel for aggregate projection examples

The initial results are shown in two blocks (Figure 79) as the sales region is not part of both contexts. The initial query results are not affected by the aggregate projection setting.

Category	Region Name	Sales
Electronics	Central	16,921
Electronics	East	22,994
Food	Central	10,938
Gifts	West	36,362
Health & Beauty	Central	11,707
Household	East	88,774
Kid's Korner	West	5,502
Travel	East	9,289

Category	Category Sales Ratio
Electronics	19.71 %
Food	5.40 %
Gifts	17.96 %
Health & Beauty	5.78 %
Household	43.84 %
Kid's Korner	2.72 %
Travel	4.59 %

Figure 79: Initial query results for aggregate projection examples

The *Category Sales Ratio* can be placed in the block containing the region. Since the category is also in the block the results remain correct (Figure 80) regardless of the projection aggregate.

Category	Region Name	Category Sales Ratio	Sales
Electronics	Central	19.71 %	16,921
Electronics	East	19.71 %	22,994
Food	Central	5.40 %	10,938
Gifts	West	17.96 %	36,362
Health & Beauty	Central	5.78 %	11,707
Household	East	43.84 %	88,774
Kid's Korner	West	2.72 %	5,502
Travel	East	4.59 %	9,289

Figure 80: Query results for aggregate projection examples

When the column category is removed the projection aggregate is used to recalculate or redistribute the *Category Sales Ratio*. The results when the projection aggregate is set to *Average* are shown in Figure 81. The figure of 14.29% is arrived at by summing all the ratios and dividing by seven (7). The sum of the ratios is 100% and there are 7 categories which yields an average of 14.29 (100 / 7 = 14.29).

Region Name	Category Sales Ratio	Sales
Central	14.29 %	39,566
East	14.29 %	121,057
West	14.29 %	41,864

Figure 81: Projection aggregate set to Average

The results when the projection aggregate is set to *Count* are shown in Figure 82. The figure of 700% is arrived at by counting the number of categories. There are seven (7) categories which yields a count of 7. Since the column is being formatted as a percent, 700% is displayed in the report.

Region Name	Category Sales Ratio	Sales
Central	700.00 %	39,566
East	700.00 %	121,057
West	700.00 %	41,864

Figure 82: Projection aggregate set to Count

The results when the projection aggregate is set to *Maximum* are shown in Figure 83. The *Category Sales Ratio* becomes 43.84% as this is the largest ratio of all categories. This is verified in Figure 69 for the “Household” category.

Region Name	Category Sales Ratio	Sales
Central	43.84 %	39,566
East	43.84 %	121,057
West	43.84 %	41,864

Figure 83: Projection aggregate set to Maximum

The results when the projection aggregate is set to *Minimum* are shown in Figure 84. The *Category Sales Ratio* becomes 2.72% as this is the smallest ratio of all categories. This is verified in Figure 69 for the category of "Kid's Korner".

Region Name	Category Sales Ratio	Sales
Central	2.72 %	39,566
East	2.72 %	121,057
West	2.72 %	41,864

Figure 84: Projection aggregate set to Minimum

The results when the projection aggregate is set to *Sum* are shown in Figure 85. The *Category Sales Ratio* becomes 100% as this is the sum of the ratios over all the categories. Without the *Category* column displayed, it can be argued that the *Category Sales Ratio* should be 100%. So the projection aggregate of *Sum* is a valid setting.

Region Name	Category Sales Ratio	Sales
Central	100.00 %	39,566
East	100.00 %	121,057
West	100.00 %	41,864

Figure 85: Projection aggregate set to Sum

The results when the projection aggregate is set to *None* are shown in Figure 86. The *Category Sales Ratio* is not recalculated or redistributed. Separate values for each category are maintained. In some aspects a setting of *None* causes the *Category Sales Ratio* to be treated as dimension once the category is removed. The original ratio values are preserved as entities and as such prevent *Sales* from being aggregated to the region level. So the projection aggregate of *None* can also be considered a proper setting.

Region Name	Category Sales Ratio	Sales
Central	19.71 %	16,921
Central	5.40 %	10,938
Central	5.78 %	11,707
East	19.71 %	22,994
East	43.84 %	88,774
East	4.59 %	9,289
West	17.96 %	36,362
West	2.72 %	5,502

Figure 86: Projection aggregate set to None

For measures (calculations) that are dependent upon a dimension, the projection aggregate should be set to either *Sum* or *None*. An argument can be made for either setting. The best setting for reporting environment should be determined in discussion with all parties involved. Once determined the same projection setting should be used in these situations. Consistency is important when explaining the end results to the report consumers.

How does this compare to use of a report variable? If region is added to the query of Figure 66 and the sales ratio is obtained from the calculation wizard (Figure 68) the results are shown in Figure 87. The ratio is not at the category level but at the region and category level.

Region Name	Category	Sales	Percentage
Central	Electronics	16,921	8.36%
Central	Food	10,938	5.40%
Central	Health & Beau	11,707	5.78%
East	Electronics	22,994	11.36%
East	Household	88,774	43.84%
East	Travel	9,289	4.59%
West	Gifts	36,362	17.96%
West	Kid's Korner	5,502	2.72%
		Percentage:	100.00%

Figure 87: Sales ratio from calculation wizard by Region by Category

Once *Category* is removed from the block, the sales ratio automatically recalculates to the *Region* level (Figure 88). This recalculation is done by the report engine not by the database. The report engine detects the context of the calculation by the dimensions used and recalculates accordingly.

Region Name	Sales	Percentage
Central	39,566	19.54%
East	121,057	59.79%
West	41,864	20.67%
	Percentage:	100.00%

Figure 88: Sales ratio from calculation wizard by Region

A quick comparison between report variables and Universe objects shows the advantages of each solution.

	Report Variable	Universe Object
Use in query panel which facilitates ease of use		X
Consistent definition, one version of truth		X
Recalculates according to dimensions used in block	X	
Does not require Universe changes for new calculations	X	
One place for maintenance if formula changes		X

Now one may ask how to obtain the SQL for the derived table. The best answer is to use a SQL generation tool like Business Objects. Desktop Intelligence lends itself to this task very well as changes to the Universe do not have to continually be exported to the repository to be available for use by Desktop Intelligence. Starting with a simple Universe (Figure 89) the derived table used in the previous example can be built. Since *MP_T1_Sales* has all the information required for the first SQL pass that is the only table included within the Universe.

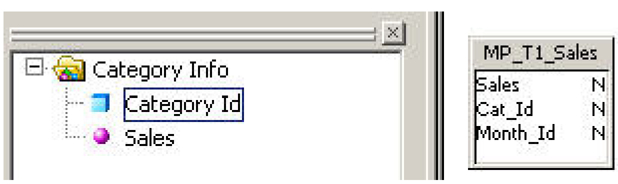


Figure 89: Starting Universe to build derived table

This Universe is then used within Desktop Intelligence to build the derived table. The first SQL pass or derived table needs to only return the overall sales. So that is the query that will be built.

Figure 90: Results objects required for first SQL pass

The SQL window, Figure 91, is then opened to copy the generated SQL and ensure that it resembles what was expected.

```
SELECT
  sum(MP_T1_Sales.Sales)
FROM
  MP_T1_Sales
```

Figure 91: Generated SQL for first SQL pass

The SQL is then copied to the clipboard so that it can later be pasted into a new derived table within Universe Designer. The query can also be executed so that the results (Figure 92) can be viewed and verified. This is the same results that a derived table would return.

Sales
202,487

Figure 92: Results from first SQL pass

Figure 93: Derived table for first SQL pass

Once pasted into a derived table (Figure 93), the only difference is that each calculated column must have a name. In this case the sum of sales is to be called *All_Sales*. Now it's time to create the second SQL pass. The second pass needs to return *All_Sales* and each category's sales ratio to the overall sales total (*All_Sales*), so a few new objects need to be created. The first object is for *All Sales*, Figure 94. No aggregation is needed so it is left as a dimension.

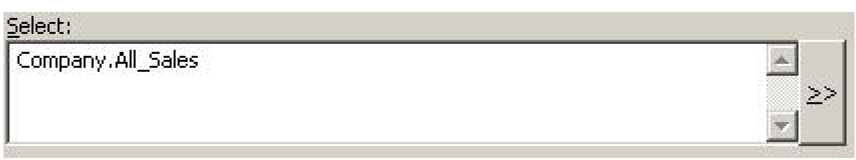


Figure 94: Object definition for All Sales

The second object is for the category's sales ratio to the overall total and will be known as *Sales Ratio*. Defined in Figure 95, two existing objects are used for the calculation.

Figure 95: Object definition for Sales Ratio

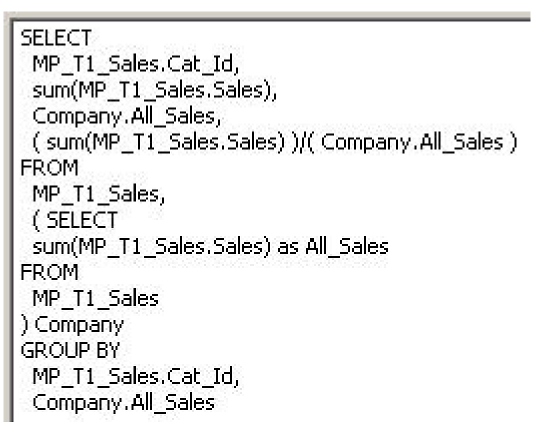
Now that the objects have been created the Universe has a new look as depicted in Figure 96. There is not a join from the derived table *Company* to the *MP_T1_Sales* table. If a join had been needed it would have been created as any other join between tables located within the database schema.

Figure 96: Universe after inclusion of first derived table

Now it is time to use Desktop Intelligence to create the SQL for the final derived table. For the second derived table the category id, total sales for that category, the overall sales total, and the category's sales ratio to the overall total is needed. So those objects are the ones used in the query (Figure 97).

Figure 97: Result objects for the second SQL pass

Again the SQL window, Figure 98, is opened to copy the generated SQL and ensure that it resembles what was expected. However when this attempted a Cartesian product warning is received (Figure 99). This is due to the *Company* derived table not being joined to any other table. In this case it is no cause for alarm.



```
SELECT
  MP_T1_Sales.Cat_Id,
  sum(MP_T1_Sales.Sales),
  Company.All_Sales,
  ( sum(MP_T1_Sales.Sales) )/( Company.All_Sales )
FROM
  MP_T1_Sales,
  ( SELECT
  sum(MP_T1_Sales.Sales) as All_Sales
FROM
  MP_T1_Sales
) Company
GROUP BY
  MP_T1_Sales.Cat_Id,
  Company.All_Sales
```

Figure 98: Generated SQL for the second SQL pass

Figure 99*: Cartesian product warning*

As with the first pass, the SQL is copied to the clipboard so that it can later be pasted into a new derived table within Universe Designer. The query can also be executed so that the results (Figure 100) can be viewed and verified. This is the same results that a derived table would return.

Category Id	Sales	All Sales	Sales Ratio
1	39,915	202,487.00	19.71 %
2	10,938	202,487.00	5.40 %
3	36,362	202,487.00	17.96 %
4	11,707	202,487.00	5.78 %
5	88,774	202,487.00	43.84 %
6	5,502	202,487.00	2.72 %
7	9,289	202,487.00	4.59 %

Figure 100: Results from second SQL pass

As with any derived table each calculated column must be named. The category's sales ratio to the overall total will be named *Cat_Sales_Ratio*.

```
SELECT
  MP_T1_Sales.Cat_Id,
  Company.All_Sales,
  ( sum(MP_T1_Sales.Sales) )/( Company.All_Sales ) as Cat_Sales_Ratio
FROM
  MP_T1_Sales,
  ( SELECT
  sum(MP_T1_Sales.Sales) as All_Sales
FROM
  MP_T1_Sales
) Company
GROUP BY
  MP_T1_Sales.Cat_Id,
  Company.All_Sales
```

The derived table (Figure 101), that has just been created with Desktop Intelligence bears a very strong resemblance to the one (Figure 98) used in the previous example. There is not any material difference between the two, only a few formatting and name changes.

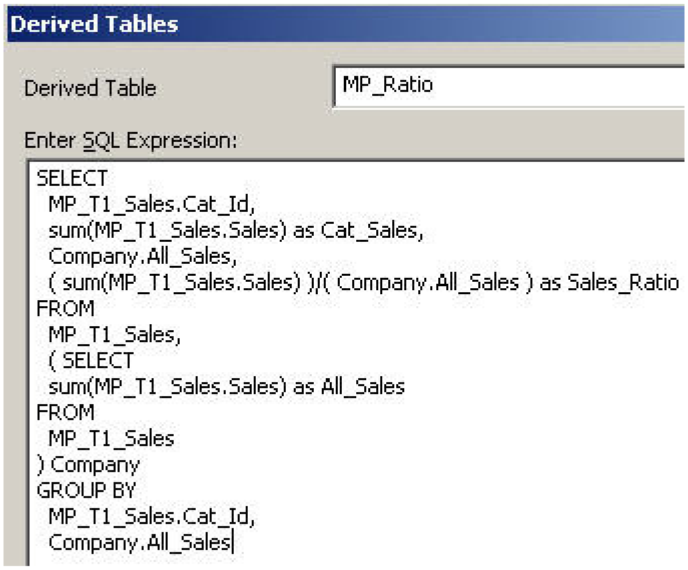


Figure 101: Derived table for second SQL pass

Using Desktop Intelligence or other SQL generation tool can be extremely useful when creating derived tables. Granted this has been a simple example, but derived tables can become quite intricate as desired calculations become more complex.

Scenario 4: Semi-additive Measures

The next scenario concerns semi-additive measures. This is most difficult of the scenarios to address. Additive measures can be summed up across all dimensions and rolled up within a dimensional hierarchy. Semi-additive measures can be summed up across some dimensions but not all. Usually the time dimension is the exception. Other aggregate functions such as average, minimum, and maximum can be applied on semi-additive measures for all dimensions but not sum. Two good examples of semi-additive measures are ending inventory and account balances. Account balances are usually kept on a daily basis. So user may want to see the month-ending account balances for a specific month. The balances can be summed over branch location or account type but not over time. The balances should not be summed over days for example since the user has fixed the time dimension by requesting balances as of month end. The month ending balance is not the sum of the daily balances for the month (as much as we would all like it to be). The month ending balance is the daily balance at a fixed point in time. So the balance can be summed over some dimensions but not all. Two common categories of semi-additive measures are periodic and level. Account balances fall into the periodic category while inventory and headcount are examples that fall into the level category.

The method to solve this situation involves a few components which are derived tables, aggregate awareness, and the Query Drill option available only within Web Intelligence. Derived tables are created for each time period that is needed. If the user needs to view balances as of month end, quarter end, and year end then a derived table is created for each. The derived table extracts the balance for each account as of the specified time period. For example a derived table for month end will extract the account balance as of the last day of the month for each account. In some aspects this type of derived table can be viewed as an aggregate table. With this concept in mind the next step is to setup aggregate awareness. Continuing with the example, the day time dimension is would not be compatible with the month-end derived table while the day and month time dimensions would not be compatible with the quarter-end derived table. In this way if the user requests account balance along with month in the query then the month-end balances are returned. If the user than adds day to the query the daily balance would be returned. Query drill is a relatively new option within Web Intelligence that re-queries the database with each drill operation and appends the drill filter to the query. In this regard, if a report displays month along with the month-end balance, a drill on month to day will then re-query the database causing the daily balance to be displayed along with day.

A different set of data is required to show this example. The spreadsheet below (Figure 102) shows the balances of four (4) accounts loaded to the database. Three tables are created within the database to hold the data.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Date_ID	Date	Acct_ID	Acct_Nbr	Acct_Balance	Acct_ID	Acct_Nbr	Acct_Balance	Acct_ID	Acct_Nbr	Acct_Balance	Acct_ID	Acct_Nbr	Acct_Balance
2	38745	1/29/2006	101	100101	160139	102	100202	260139	103	100337	360139	104	104044	460139
3	38746	1/30/2006	101	100101	160130	102	100202	260130	103	100337	360130	104	104044	460130
4	38747	1/31/2006	101	100101	160131	102	100202	260131	103	100337	360131	104	104044	460131
5	38748	2/1/2006	101	100101	160201	102	100202	260201	103	100337	360201	104	104044	460201
6	38774	2/27/2006	101	100101	160227	102	100202	260227	103	100337	360227	104	104044	460227
7	38775	2/28/2006	101	100101	160228	102	100202	260228	103	100337	360228	104	104044	
8	38776	3/1/2006	101	100101	160301	102	100202	260301	103	100337	360301	104	104044	460301
9	38804	3/29/2006	101	100101	160329	102	100202	260329	103	100337	360329	104	104044	460329
10	38805	3/30/2006	101	100101	160330	102	100202	260330	103	100337	360330	104	104044	
11	38806	3/31/2006	101	100101	160331	102	100202	260331	103	100337	360331	104	104044	460331
12	38807	4/1/2006	101	100101	160401	102	100202	260401	103	100337	360401	104	104044	460401
13	38895	6/28/2006	101	100101	160628	102	100202	260628	103	100337	360628	104	104044	460628
14	38896	6/29/2006	101	100101	160629	102	100202	260629	103	100337	360629	104	104044	460629
15	38897	6/30/2006	101	100101	160630	102	100202	260630	103	100337	360630	104	104044	
16	38898	7/1/2006	101	100101	160701	102	100202	260701	103	100337	360701	104	104044	460701
17	38928	7/31/2006	101	100101	160731	102	100202	260731	103	100337	360731	104	104044	460731
18	38989	9/30/2006	101	100101	160930	102	100202	260930	103	100337	360930	104	104044	460930
19	39080	12/30/2006	101	100101	161230	102	100202	261230	103	100337	361230	104	104044	461230
20	39081	12/31/2006	101	100101	161231	102	100202	261231	103	100337	361231	104	104044	

Figure 102: Account balance data

The table MP_T2_Account (Figure 103) contains balances for accounts for various days of the year. Each account exists only one time for each date. It represents the ending balance for that account on that day.

Date_ID	Acct_ID	Acct_Balance
38745	101	160139
38745	102	260139
38745	103	360139
38745	104	460139
38746	104	460130
38746	103	360130
38746	102	260130
38746	101	160130
38747	101	160131
38747	102	260131
38747	103	360131
38747	104	460131
38748	104	460201

Figure 103: MP_T2_Account sample data

The table MP_T2_Account_Info table (Figure 104) contains base information for each account. Only one row exists for each account.

Acct_ID	Acct_Nbr	Acct_Open_Date	Acct_Status
101	100101	4/26/2005	0
102	100202	10/19/2003	1
103	100337	3/1/2005	0
104	104044	8/30/2004	2

Figure 104: MP_T2_Acct_Info data

The MP_T2_Calendar contains the calendar information for each date. Some of the relevant columns are shown in Figure 105.

Calendar_Date	Day_Of_Week	Day_Of_Month	Day_Of_Calendar	Month_Of_Year	Quarter_Of_Year	Year_Of_Calendar
1/1/2006	1	1	38717	1	1	2006
1/2/2006	2	2	38718	1	1	2006
1/3/2006	3	3	38719	1	1	2006
1/4/2006	4	4	38720	1	1	2006
1/5/2006	5	5	38721	1	1	2006
1/6/2006	6	6	38722	1	1	2006
1/7/2006	7	7	38723	1	1	2006
1/8/2006	1	8	38724	1	1	2006
1/9/2006	2	9	38725	1	1	2006
1/10/2006	3	10	38726	1	1	2006
1/11/2006	4	11	38727	1	1	2006
1/12/2006	5	12	38728	1	1	2006

Figure 105: MP_T2_Calendar sample data. Not all columns shown.

The initial Universe, Figure 106, is created to aide in the creation of derived tables.

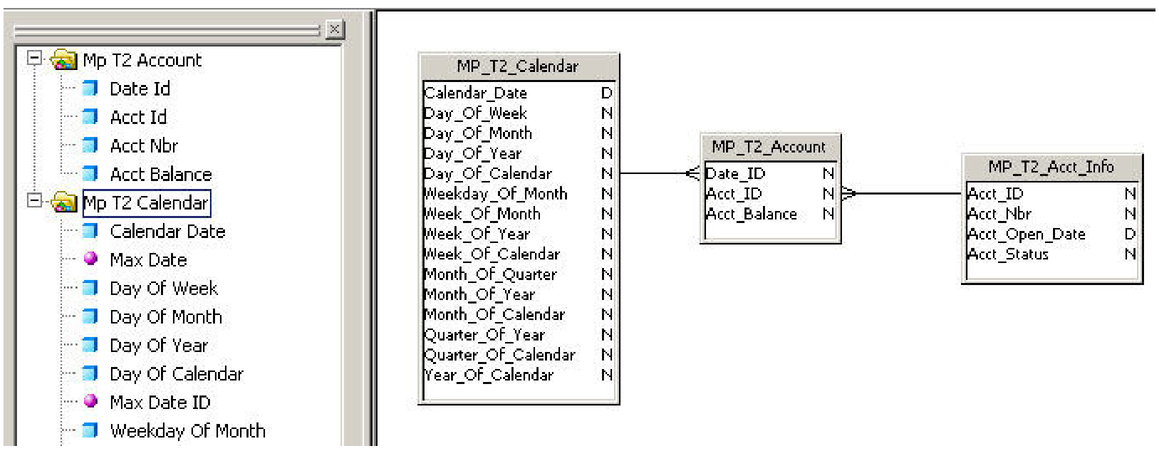


Figure 106: Initial Universe

A few measure objects are created. The definition of *Max Date ID* is shown in Figure 107.

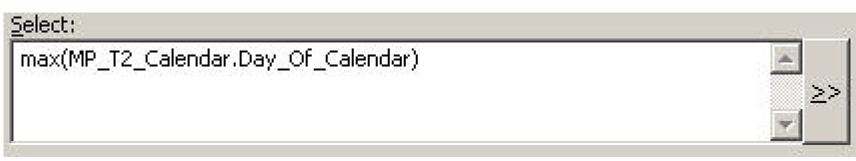


Figure 107: Definition of Max Date ID

The definition of *Max Date* is shown in Figure 108 the *Max Date* is used for descriptive purposes within the derived table.

Figure 108*: Definition of Max Date*

A derived table is needed to hold the month ending balance for each account. The first step is to create a list of ending date for each month. Using the Universe from Figure 106 within Desktop Intelligence, a query (Figure 109) is created that returns the maximum date for each month.

Figure 109: Result objects for month ending dates

The SQL generated for this query is shown in Figure 110.

```
SELECT
  MP_T2_Calendar.Month_Of_Calendar,
  max(MP_T2_Calendar.Day_Of_Calendar),
  max(MP_T2_Calendar.Calendar_Date)
FROM
  MP_T2_Calendar
GROUP BY
  MP_T2_Calendar.Month_Of_Calendar
```

Figure 110: SQL generated for month ending dates

A portion of the results is shown in Figure 111

Month Of Calendar	Max Date ID	Max Date
1273	38747	1/31/2006
1274	38775	2/28/2006
1275	38806	3/31/2006
1276	38836	4/30/2006
1277	38867	5/31/2006
1278	38897	6/30/2006
1279	38928	7/31/2006
1280	38959	8/31/2006

Figure 111: Sample of the month ending dates

The SQL from Figure 86 is then used to create a derived table (Figure 112).

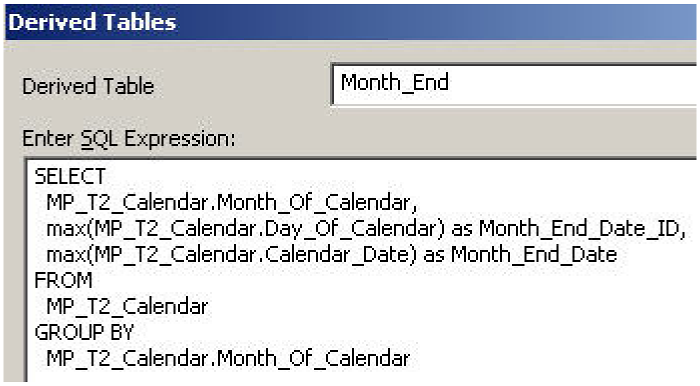


Figure 112: Creation of the month ending dates derived table

The new derived table, *Month End*, is integrated into the Universe as shown in Figure 113.

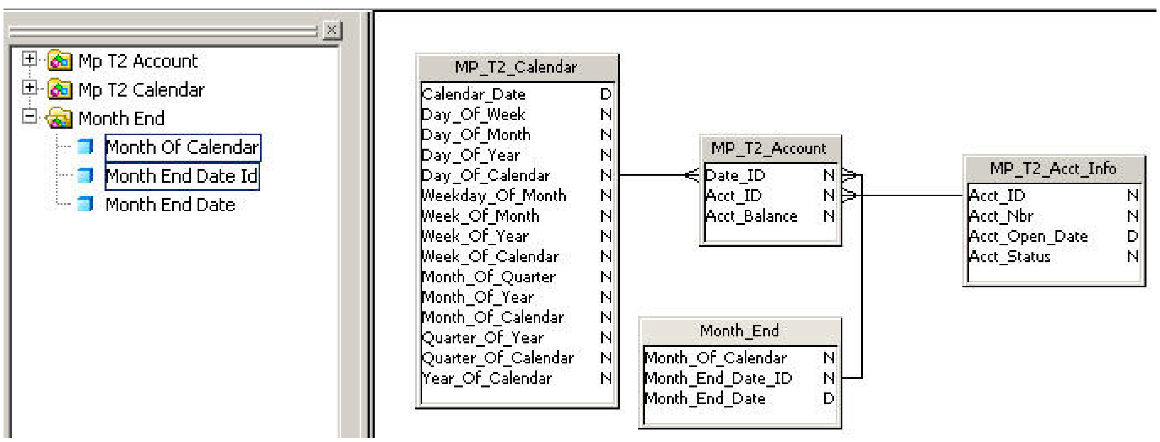


Figure 113: Create the joins and objects for the month ending dates table in the Universe

A query (Figure 114) is then created to retrieve the balance for each account on the month ending date.

Figure 114: Results objects for month ending balances

A sample of the results is shown in Figure 115. However there is a problem. Account Id 104 does not have an ending balance for February. The reason is that the last date that exists for 104 was on February 27, not February 28 the date that exists in the *Month_End* table.

Acct Id	Acct Nbr	Acct Balance	Month End Date	Month End Date Id
101	100101	160,131	1/31/2006	38747
101	100101	160,228	2/28/2006	38775
101	100101	160,331	3/31/2006	38806
101	100101	160,630	6/30/2006	38897
101	100101	160,731	7/31/2006	38928
103	100337	370,228	2/28/2007	39140
103	100337	370,331	3/31/2007	39171
103	100337	370,630	6/30/2007	39262
103	100337	370,731	7/31/2007	39293
103	100337	370,930	9/30/2007	39354
103	100337	371,231	12/31/2007	39446
104	104044	460,131	1/31/2006	38747
104	104044	460,331	3/31/2006	38806

Figure 115: Month ending balances by account

What is needed is the maximum date per month for each account. The query from Figure 109 is replaced with the query from Figure 116.

Figure 116: Query for month ending dates by account

The results are shown in Figure 117. Account ID of 104 now has entry for February.

Acct Id	Month Of Calend	Max Date	Max Date ID
101	1273	1/31/2006	38747
101	1274	2/28/2006	38775
101	1275	3/31/2006	38806
101	1276	4/1/2006	38807
103	1285	1/31/2007	39112
103	1286	2/28/2007	39140
103	1287	3/31/2007	39171
103	1288	4/1/2007	39172
103	1290	6/30/2007	39262
103	1291	7/31/2007	39293
103	1293	9/30/2007	39354
103	1296	12/31/2007	39446
104	1273	1/31/2006	38747
104	1274	2/27/2006	38774
104	1275	3/31/2006	38806

Figure 117: Sample results of month ending dates by account

The SQL generated by the query in Figure 116 is then used to create a derived table, *Acct_Month_End* as shown in Figure 118.

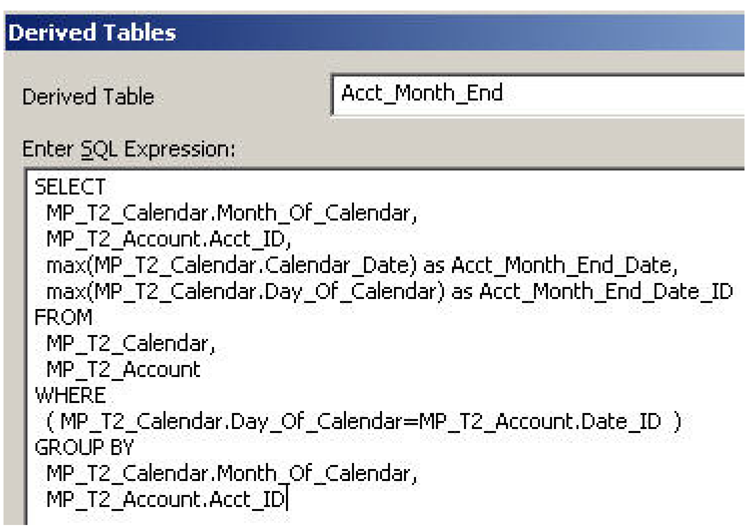


Figure 118: Derived table SQL for Acct_Month_End

There is one remaining issue with the derived table. In Figure 117 even though account id 104 now appears with a February date, it is not as of the month ending date of February. The next step is to correct this. The derived table created in Figure 118 is still used along with the *Month_End* derived table (Figure

112) but joined to the Universe in a different way (Figure 119). The table is now joined to the calendar table instead of the account.

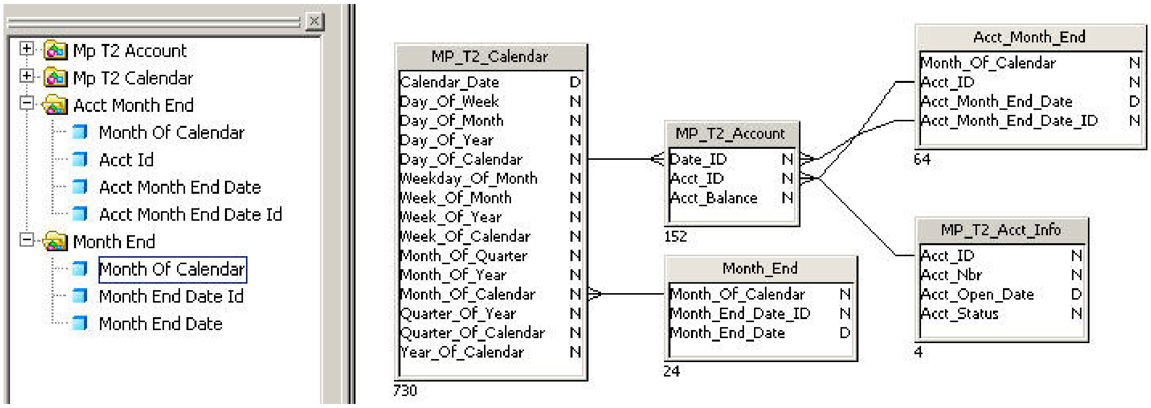


Figure 119: Both derived tables used in Universe

A new query is then built (Figure 120). The *Month of Calendar* object is from the *Acct_Month_End* table which forces the last balance for the month to be retrieved from *MP_T2_Account*. All other objects are pulled from either the *MP2_T2_Account* or *Month_End* tables.

Figure 120: Result objects for last balance available for month with month ending date

Now the results are in line with what is desired as seen in Figure 121. All dates appear as the actual month end, even account if of 104. The balance for account 104 is as of 2/27/2006, the last entry for that account in February 2006.

Month End Date	Acct Id	Acct Balance	Month Of Calendar	Month End Date Id
1/31/2006	101	160131	1273	38747
1/31/2006	102	260131	1273	38747
1/31/2006	103	360131	1273	38747
1/31/2006	104	460131	1273	38747
2/28/2006	101	160228	1274	38775
2/28/2006	102	260228	1274	38775
2/28/2006	103	360228	1274	38775
2/28/2006	104	460227	1274	38775
3/31/2006	101	160331	1275	38806

Figure 121: Results showing last balance available for month with month ending date

The generated SQL results in:

```
SELECT
  MP_T2_Account.Acct_ID,
  MP_T2_Account.Acct_Balance,
  Month_End.Month_End_Date,
  Acct_Month_End.Month_Of_Calendar,
  Month_End.Month_End_Date_ID
FROM
  ( SELECT
  MP_T2_Calendar.Month_Of_Calendar,
  max(MP_T2_Calendar.Day_Of_Calendar) as Month_End_Date_ID,
  max(MP_T2_Calendar.Calendar_Date) as Month_End_Date
FROM
  MP_T2_Calendar
GROUP BY
  MP_T2_Calendar.Month_Of_Calendar) Month_End INNER JOIN MP_T2_Calendar
ON (MP_T2_Calendar.Month_Of_Calendar=Month_End.Month_Of_Calendar)
   INNER JOIN MP_T2_Account ON
(MP_T2_Calendar.Day_Of_Calendar=MP_T2_Account.Date_ID)
   INNER JOIN ( SELECT
  MP_T2_Calendar.Month_Of_Calendar,
  MP_T2_Account.Acct_ID,
  max(MP_T2_Calendar.Calendar_Date) as Acct_Month_End_Date,
  max(MP_T2_Calendar.Day_Of_Calendar) as Acct_Month_End_Date_ID
FROM
  MP_T2_Calendar, MP_T2_Account
WHERE
  ( MP_T2_Calendar.Day_Of_Calendar=MP_T2_Account.Date_ID  )
GROUP BY
  MP_T2_Calendar.Month_Of_Calendar, MP_T2_Account.Acct_ID)
  Acct_Month_End ON(Acct_Month_End.Acct_ID=MP_T2_Account.Acct_ID and
  Acct_Month_End.Acct_Month_End_Date_ID=MP_T2_Account.Date_ID)
```

This SQL is now used to create the final derived table in the actual reporting Universe. The intent of showing the trials and tribulations of this derived table is to demonstrate that the creation of advanced derived tables is an iterative process. Be sure to verify the results at each step to ensure that expected results are being attained. Is the current derived table exactly what is needed in this situation? Possibly not. If you look at the results (Figure 122) only months with a balance entry are being returned. In this case there are no entries for May 2006. Example data was created for this exercise. It could very well be that at least one entry is made for every account every month in a real world situation. But we are going to proceed with the derived table as it now stands.

4/30/2006	103	360401	1276	38836
4/30/2006	104	460401	1276	38836
6/30/2006	101	160630	1278	38897
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 122: No results for May for any account

Finally we return to the actual reporting Universe. The SQL generated by the query in Figure 123 is used to create a derived table. The column containing the account balance, *Acct_ME_Balance*, is renamed to better reflect its actual content.

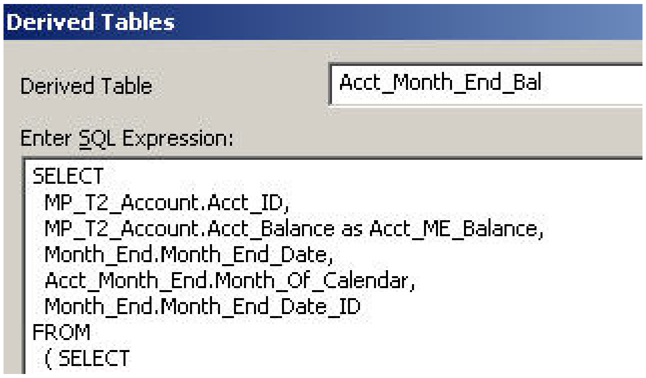


Figure 123: Derived table to return the month ending balance for each account

This derived table, *Acct_Month_End_Bal*, is now made part of the Universe as an additional fact table.

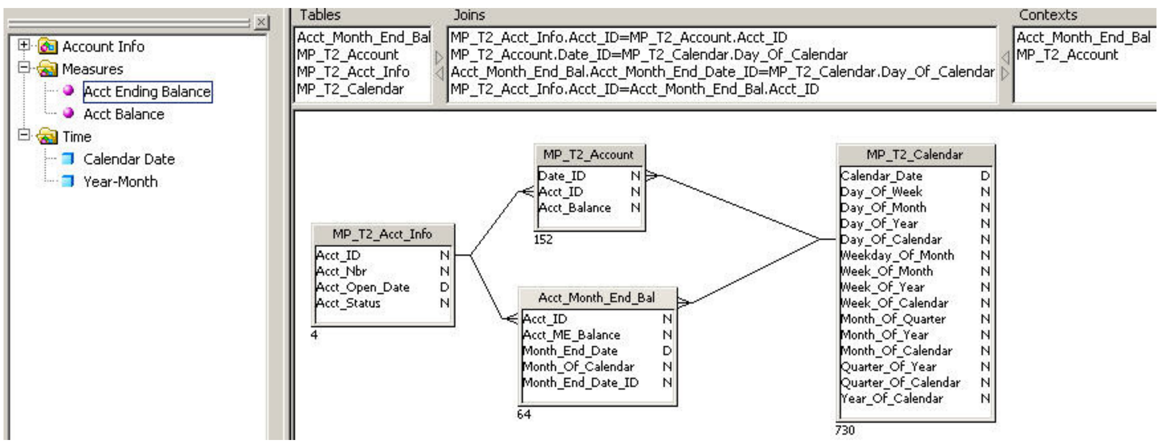


Figure 124: Universe with additional derived table (Acct_Month_End_Bal), aliases, contexts, and objects.

A few objects of interest have been created. The measure object, *Acct Ending Balance*, is defined as aggregate aware (Figure 125). The intent is that the object returns the ending balance on an account either as the account balance as of the end of a month or as of the end of a day. In that case aggregate navigation needs to be set up.

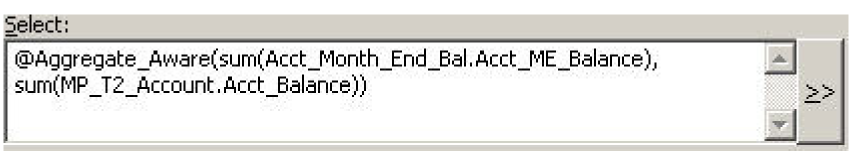


Figure 125: Definition of the Acct_Ending_Balance object

Aggregate navigation is set such that the *Acct_Month_End_Bal* should not be used in conjunction with the *Calendar Date* object (Figure 126). If the *Calendar Date* object is needed on the report, the ending account balance needs to be returned by day which is held in the *MP_T2_Account* table.

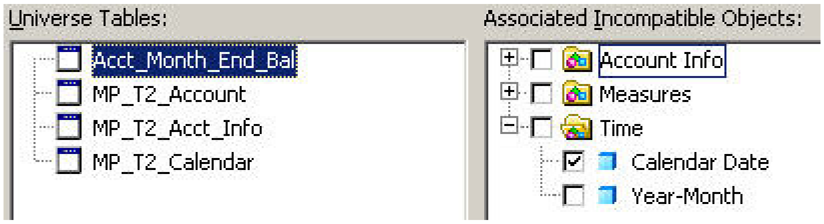


Figure 126: Aggregate Navigation for monthly and daily balances

The testing will be done in Web Intelligence in order to utilize the Query Drill feature. In the query panel, Figure 127, four objects will be returned. In case one has not noticed, the daily balances follow a pattern that will make it easier to verify the results. All balances contain six digits. The first digit is the account id, either 1, 2, 3, or 4. The next five digits represent the year (one digit – 6 or 7), month (two digits), and day (2 digits). So if the account id is represented as A, then the pattern will be seen as *AYMMDD*.

Figure 127: First test of semi-additive measure

A portion of the results is shown in Figure 128. It's interesting to note that the ending balance for account 104044 for February 2006 is 460227 which indicates that the balance occurred on the 27th while the ending balances in February 2006 on the other accounts indicate that they occurred on the 28th.

Year-Month	Acct Id	Acct Nbr	Acct Ending Balance
2006-01	101	100101	160,131
2006-01	102	100202	260,131
2006-01	103	100337	360,131
2006-01	104	104044	460,131
2006-02	101	100101	160,228
2006-02	102	100202	260,228
2006-02	103	100337	360,228
2006-02	104	104044	460,227
2006-03	101	100101	160,331

Figure 128: Sample results from first semi-additive query

Indeed, if the SQL (Figure 129) is viewed the ending balance was retrieved from the derived table, *Acct_Month_End_Bal*.

```
SELECT
 cast(MP_T2_Calendar.Year_Of_Calendar as char(4)) + '-' + case when MP_T2_Calendar.M
onth_Of_Year > 9 then cast(MP_T2_Calendar.Month_Of_Year as char(2)) else '0' + cast(M
P_T2_Calendar.Month_Of_Year as char(1)) end,
 MP_T2_Acct_Info.Acct_ID,
 MP_T2_Acct_Info.Acct_Nbr,
 sum(Acct_Month_End_Bal.Acct_ME_Balance)
FROM
```

Figure 129*: Portion of generated SQL*

Now it's time to time to test the drill functionality. The first set is to enable the Query Drill option (Figure 130). It is found on the Document Properties window.

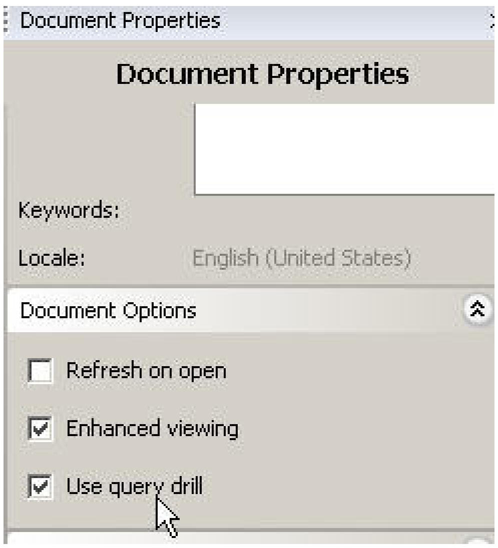


Figure 130: Enable Query Drill on document

Next, on the document we drill into the *Year-Month* column on the 2006-02 value. As desired the daily values are now returned as shown in Figure 131.

Calendar D:	Acct Id	Acct Nbr	Acct Ending Balance
2/1/06	101	100101	160,201
2/1/06	102	100202	260,201
2/1/06	103	100337	360,201
2/1/06	104	104044	460,201
2/27/06	101	100101	160,227
2/27/06	102	100202	260,227
2/27/06	103	100337	360,227
2/27/06	104	104044	460,227
2/28/06	101	100101	160,228

Figure 131: Drill into daily ending balances

So this must mean the account ending balance is no longer being retrieved from the derived table, *Acct_Month_End_Bal*, but from the actual account balances table, *MP_T2_Account*. Going back to the query panel and viewing the SQL (Figure 132) one will see that this is indeed what has happened. Why is this? The SQL also shows that the column *MP_T2_Calendar.Calendar_Date* is now being returned from the query. This is the *Calendar Date* object which has been defined as incompatible with the derived table (Figure 126). So the aggregate aware function has been invoked on the *Acct Ending Balance* object.

```
SELECT
  cast(MP_T2_Calendar.Year_Of_Calendar as char(4)) + '-' + case when MP_T2_Calendar.M
onth_Of_Year > 9 then cast(MP_T2_Calendar.Month_Of_Year as char(2)) else '0' + cast(M
P_T2_Calendar.Month_Of_Year as char(1)) end,
  MP_T2_Acct_Info.Acct_ID,
  MP_T2_Acct_Info.Acct_Nbr,
   sum(MP_T2_Account.Acct_Balance),
  MP_T2_Calendar.Calendar_Date
FROM
```

Figure 132: Generated SQL from Query Drill action

So navigation across the non-additive dimension has been a success. Now it's time to try to an additive dimension in order to meet the full definition of a semi-additive measure. In the query panel two dimensions, one additive and one non-additive are requested.

Figure 133: Result objects for semi-additive measure

The results, Figure 134, seem to look promising. Account ids 1 and 3 have a status of 0, account id 2 has a status of 1, and account id 4 has a status of 2. Keeping in mind that the balances are in a format of *AYMMDD* the results are correct.

Acct Status	0	1	2
Year-Month	Acct Ending Balance	Acct Ending Balance	Acct Ending Balance
2006-01	520,262	260,131	460,131
2006-02	520,456	260,228	460,227
2006-03	520,662	260,331	460,331
2006-04	520,802	260,401	460,401
2006-06	521,260	260,630	460,629

Figure 134: Semi-additive measure across one additive and one non-additive dimensions

Viewing the SQL, the aggregate sum function was used against the month ending balances table.

```
SELECT
 cast(MP_T2_Calendar.Year_Of_Calendar as char(4)) + '-' + case when MP_T2_Calendar.M
onth_Of_Year > 9 then cast(MP_T2_Calendar.Month_Of_Year as char(2)) else '0' + cast(M
P_T2_Calendar.Month_Of_Year as char(1)) end,
 MP_T2_Acct_Info.Acct_Status,
 sum(Acct_Month_End_Bal.Acct_ME_Balance)
FROM
```

Figure 135: Generated SQL for Semi-additive measure across one additive and one non-additive dimensions

After setting the *Query Drill* option, we again drill into the *Year-Month* value of *2006-02*. The results, Figure 136, are again correct.

Acct Status	0	1	2
Calendar	Acct Ending Balance	Acct Ending Balance	Acct Ending Balance
2/1/06	520,402	260,201	460,201
2/27/06	520,454	260,227	460,227
2/28/06	520,456	260,228	

Figure 136: Drill operation into a semi-additive measure

Viewing the SQL the same changes as before have occurred. The balances are now returned from the table *MP_T2_Account* and the *MP_T2_Calendar.Calendar_Date* has been appended to the query. This allows the balances to be summed across the *Account Status* dimension but not the time dimension.

```
SELECT
 cast(MP_T2_Calendar.Year_Of_Calendar as char(4)) + '-' + case when MP_T2_Calendar.M
onth_Of_Year > 9 then cast(MP_T2_Calendar.Month_Of_Year as char(2)) else '0' + cast(M
P_T2_Calendar.Month_Of_Year as char(1)) end,
 MP_T2_Acct_Info.Acct_Status,
  sum(MP_T2_Account.Acct_Balance),
 MP_T2_Calendar.Calendar_Date
FROM
```

Figure 137: Portion of the generated SQL after Query Drill operation

The next obvious question is how hard is it to add the next level of quarter ending balances? The answer is, not as hard as the first level. Copy the SQL text of the month end derived table, *Acct_Month_End_Bal*, into Notepad or something similar. Next apply a *Replace All* operation which substitutes *Quarter* for *Month* and you are 99% done. The last step is to rename the balance column from *Acct_ME_Balance* to *Acct_QE_Balance*. It may not always be this simple but it is common to be able to modify an existing derived table to create the table for the next level. Now create the join and contexts for the quarter end derived table to complete the Universe structure (Figure 138).

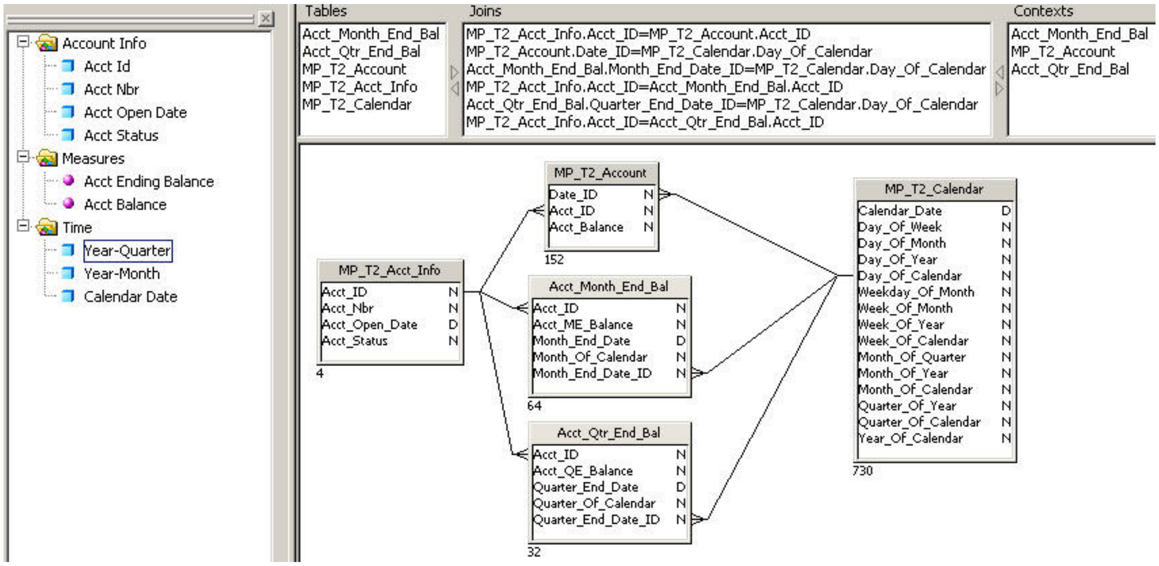


Figure 138: Universe structure with derived table for quarter ending balances

Now a new object, *Year-Quarter,* has to be created to display the year and quarter. Its definition is shown in Figure 139.

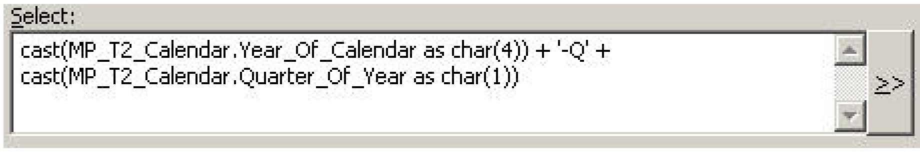


Figure 139: Definition of Year-Quarter object

The *Acct Ending Balance* object needs to be modified to make use of the new derived table. Its new definition is shown in Figure 140.

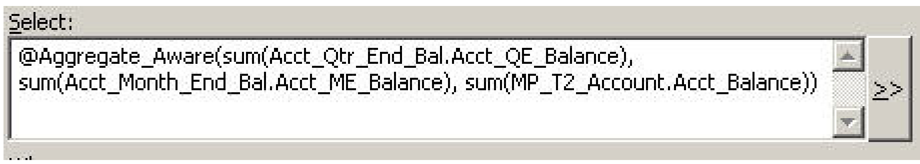


Figure 140: New definition of Acct Ending Balance to include quarter end

Be sure to update the time hierarchy with *Year-Quarter* as shown in Figure 141.

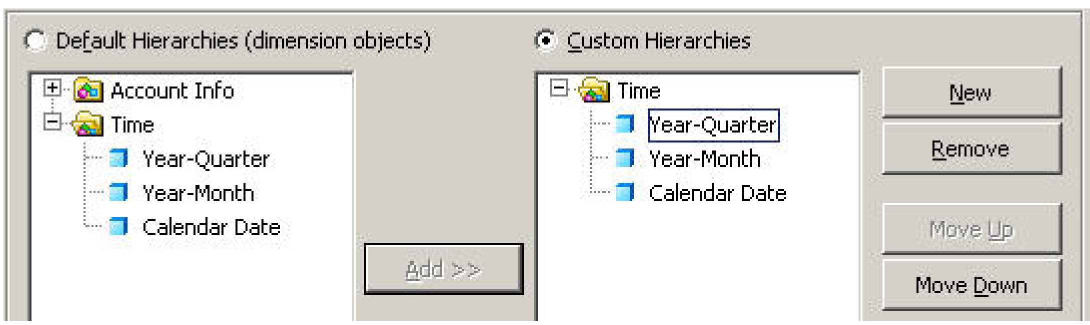


Figure 141: Add the Year-Quarter object into Time hierarchy

Aggregate Navigation should also be updated to reflect the proper use of the *Acct_Qtr_End_Bal* table with the time dimensions. The balances for quarter end should not be used whenever the *Year-Month* or the *Calendar Date* are referenced in a query. The incompatibility has been changed to reflect this in Figure 142.

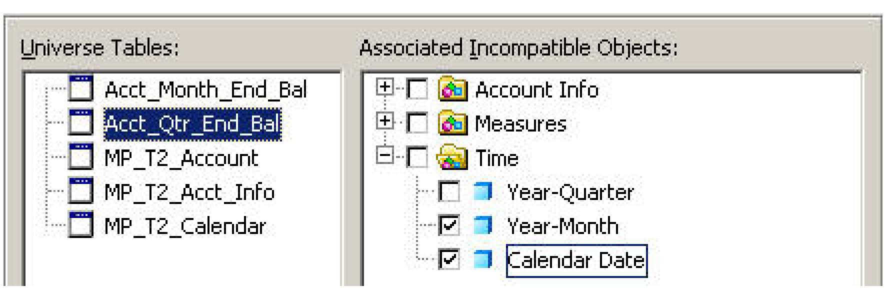


Figure 142: Aggregate Navigation with quarter end balances

Believe it or not, it's now ready to test. Build the query to show ending balances for each account (Figure 143).

Figure 143: Results objects to test quarter end balances by account

A quick look at the generated SQL (Figure 144) verifies that the quarter end derived table will be used.

```
SELECT
  cast(MP_T2_Calendar.Year_Of_Calendar as char(4)) + '-Q' + cast(MP_T2_Calendar.Quarter_Of_Year as char(1)),
  MP_T2_Acct_Info.Acct_ID,
  MP_T2_Acct_Info.Acct_Nbr,
  sum(Acct_Qtr_End_Bal.Acct_QE_Balance)
FROM
```

Figure 144: Generated SQL for the quarter end balances

The results look correct (Figure 145). A good check is to take a look at account if 104. Since the account balances are in the format of *AYMMDD* the quarter end balance for 2006-Q2 on account 104044 occurred on June 29, 2006. All other accounts the balance occurred on the actual quarter ending date of June 30, 2006.

Year-Quarter	Acct Id	Acct Nbr	Acct Ending Balance
2006-Q1	101	100101	160,331
2006-Q1	102	100202	260,331
2006-Q1	103	100337	360,331
2006-Q1	104	104044	460,331
2006-Q2	101	100101	160,630
2006-Q2	102	100202	260,630
2006-Q2	103	100337	360,630
2006-Q2	104	104044	460,629

Figure 145: Results showing quarter ending balances

A drill operation from quarter to month should invoke the month end balances derived table as long as *Query Drill* option has been enabled. The drill operation into 2006-Q2 depicted in Figure 146 confirms that the month end balances have now been accessed.

Year-Montl	Acct Id	Acct Nbr	Acct Ending Balance
2006-04	101	100101	160,401
2006-04	102	100202	260,401
2006-04	103	100337	360,401
2006-04	104	104044	460,401
2006-06	101	100101	160,630
2006-06	102	100202	260,630
2006-06	103	100337	360,630
2006-06	104	104044	460,629

Figure 146: Sample results for drill on 2006-Q2

Now a drill operation into *2006-06* results in Figure 147. These results confirm that the daily balances are being accessed.

Calendar D	Acct Id	Acct Nbr	Acct Ending Balance
6/28/06	101	100101	160,628
6/28/06	102	100202	260,628
6/28/06	103	100337	360,628
6/28/06	104	104044	460,628
6/29/06	101	100101	160,629
6/29/06	102	100202	260,629
6/29/06	103	100337	360,629
6/29/06	104	104044	460,629
6/30/06	101	100101	160,630
6/30/06	102	100202	260,630
6/30/06	103	100337	360,630

Figure 147: Sample results for drill on 2006-06

For the second round of tests the account will not be one of the results objects (Figure 148).

Figure 148: Result object for quarter end balances by account status

The initial results look correct. Remember that account ids of 1 and 3 have a status of 0, account id of 2 has a status of 1, and account id of has a status of 2.

Acct Status	0	1	2
Year-Quarter	Acct Ending Balance	Acct Ending Balance	Acct Ending Balance
2006-Q1	520,662	260,331	460,331
2006-Q2	521,260	260,630	460,629
2006-Q3	521,860	260,930	460,930
2006-Q4	522,462	261,231	461,230
2007-Q1	540,662	270,331	470,331
2007-Q2	541,260	270,630	470,630
2007-Q3	541,860	270,930	470,930
2007-Q4	542,462	271,231	471,231

Figure 149: Results of quarter ending balances by status.

The results of a drill operation into *2006-Q2* are shown in Figure 150. These results prove that the ending balances are now being retrieved from the month end derived tables instead of the quarter end table.

Acct Status	0	1	2
Year-Montł	Acct Ending Balance	Acct Ending Balance	Acct Ending Balance
2006-04	520,802	260,401	460,401
2006-06	521,260	260,630	460,629

Figure 150: Month ending balances by status after a drill into 2006-Q2

And the last step of a drilling into *2006-06* is shown in Figure 151. Now the balances are being retrieved from the daily balance table instead of the month end derived table.

Acct Status	0	1	2
Calendar D	Acct Ending Balance	Acct Ending Balance	Acct Ending Balance
6/28/06	521,256	260,628	460,628
6/29/06	521,258	260,629	460,629
6/30/06	521,260	260,630	

Figure 151*: Daily balance by status after drill operation into 2006-06*

At this point the Universe has a semi-additive measure, *Acct Ending Balance*, that is not additive over the time dimension but is over all other dimensions. To create a year ending balance level requires the same techniques that have been discussed already.

Creating semi-additive measures is the most difficult multi-pass scenario to satisfy. Because Desktop Intelligence does not support the *Query Drill* feature, use of semi-additive measures should only be used for drilling within Web Intelligence. Any report level aggregation should be avoided when a semi-additive measure is present. The report aggregation engine has no concept of semi-additive measures. The objects depicted in Figure 152 can be used to highlight the issue with report aggregation.

Figure 152: Result objects to highlight report aggregation issue

Because the lowest time level object requested is *Year-Month* the account ending balances are returned from the month ending balance table. Therefore the initial results shown in Figure 153 are correct.

	Acct Status	0	1	2
Year-Quart	Year-Month	Acct Ending Balance	Acct Ending Balance	Acct Ending Balance
2006-Q1	2006-01	520,262	260,131	460,131
2006-Q1	2006-02	520,456	260,228	460,227
2006-Q1	2006-03	520,662	260,331	460,331
2006-Q2	2006-04	520,802	260,401	460,401
2006-Q2	2006-06	521,260	260,630	460,629
2006-Q3	2006-07	521,462	260,731	460,731
2006-Q3	2006-09	521,860	260,930	460,930

Figure 153: Sample results of report aggregation issue after initial query

The problem occurs when the *Year-Month* column is removed from the report. The *Year-Month* is still part of the query but is not being displayed in the report. When the *Year_Month* object is removed from the report, the report aggregation takes over which causes incorrect results (Figure 154). The query has not been re-executed. Therefore the *Acct Ending Balance* is still based upon values in the month end balance table.

Acct Status	0	1	2
Year-Quart	Acct Ending Balance	Acct Ending Balance	Acct Ending Balance
2006-Q1	1,561,380	780,690	1,380,689
2006-Q2	1,042,062	521,031	921,030
2006-Q3	1,043,322	521,661	921,661
2006-Q4	522,462	261,231	461,230
2007-Q1	1,621,380	810,690	1,410,690
2007-Q2	1,082,062	541,031	941,031
2007-Q3	1,083,322	541,661	941,661
2007-Q4	542,462	271,231	471,231

Figure 154: Incorrect results dues to report aggregation

An additional issue of which one must be aware is that using derived tables such as these may cause full table scans. Full table scans by themselves are not an issue. But if such occurs against a large table performance can degrade sharply. Demonstrating this concept with the data sets encountered within the corporate demos or within a custom POC with customer provided data, in most cases, will not be a problem. However in an onsite POC or production implementation work closely with the DBA to scale down the impact of any full table scans on large tables. Two possible solutions in these situations are:

1. Embed prompts within the derived tables to reduce the number of rows being scanned in the detail table. Any filtering to reduce the number of rows required to be scanned will improve performance.
2. Have the DBA create an aggregate table or view to replicate the derived table. In the semi-additive example one could very well expect a month-end aggregate table to already be in place.

Scenario 5: Analyzing Data Subsets

The last multi-pass scenario is extracting a subset of data for further analysis in the same query. It is not uncommon for database schemas to contain status update records; as the status of an account changes, a new status record is inserted. A similar illustration is accounting systems in which transactions are posted against accounts. In both it may be necessary to extract a subset of data prior to doing analysis. For status records, only accounts that have reached a specific status contained in the last update record may be required for analysis. Similarly for transactions, only the most recent transactions for each account may be needed for further examination.

For this scenario a change in database tables is needed. The example is transaction based in which the most recent transaction for each account is to be analyzed. The following tables are part of the example schema (Figure 155):

accts	Base record for all accounts, regardless of type
CAL	System calendar table
CHANNEL_DESCR	Source of the transaction, description of channel group
CHANNEL_XREF	Combines channel numbers into channel groups
checking_acct	Contains based information for all checking accounts
checking_tran	Transactions processed against checking accounts
customer_name	Customer information

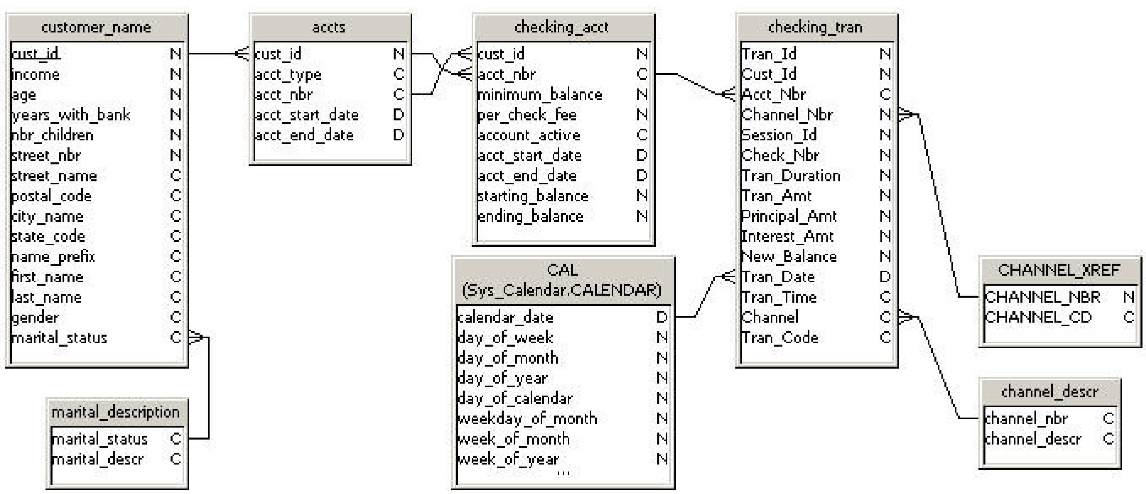


Figure 155: Initial universe for checking account transactional history

The goal is to provide the most recent transaction for each account for further analysis. The analysis may consist of the transaction source as to discovering if a trend exists in customers beginning to utilize a new transaction method over the historical trend. Or the end need may be to look at recent customer transaction experience in terms of transaction duration. In either case the first step if to isolate the last transaction for each account. Once this is accomplished then the rest of the analysis can continue.

The traditional approach is to actually extract the transactions that the criteria, in this case the most recent for each account, and then proceed with the analysis. So as usual, the first step is to create a work table.

```
CREATE TABLE LAST_TRANS_BY_ACCT
   (Acct_Nbr char(16), Tran_Date date)
```

Next the most recent transaction date for each account is extracted into the table.

```
INSERT INTO LAST_TRANS_BY_ACCT (Acct_Nbr, Tran_Date)
   SELECT Acct_Nbr, max(Tran_Date) from checking_tran
   GROUP BY Acct_Nbr
```

A second temporary table is needed to hold both the historical and recent transactions counts by channel.

```
CREATE TABLE TRANS_COUNTS
   (CHANNEL CHAR(1), HIST_CNT INTEGER, RECENT_CNT INTEGER)
```

Now the transaction counts for each timeframe need to be loaded into the temporary table before any analysis can be done. Recent customer behavior is obtained and inserted into the temporary table by using the last transaction date by account.

```
INSERT INTO TRANS_COUNTS (CHANNEL, HIST_CNT, RECENT_CNT)
   SELECT   a.CHANNEL, 0, COUNT(a.TRAN_ID)
     FROM   checking_tran a, LAST_TRANS_BY_ACCT t
    WHERE   a.acct_nbr = t.Acct_Nbr
      AND   a.tran_date = t.Tran_Date
   GROUP BY a.channel
```

The information in Figure 156 is inserted into TRANS_COUNTS.

	channel_descr	Count(Tran_Id)
1	ACH	127
2	Branch	202
3	Check	100
4	Electronic	90
5	Internet	37
6	Other	60
7	Paper	343
8	Wire	79

Figure 156: Transaction count by channel for only the most recent transaction(s) of each account

So that a comparison against historical customer behavior can be done, the counts
are also added to the temporary table..

```
INSERT INTO TRANS_COUNTS (CHANNEL, HIST_CNT, RECENT_CNT)
   SELECT  a.CHANNEL, COUNT(a.TRAN_ID), 0
     FROM  checking_tran a
 GROUP BY  a.channel
```

The historical counts in Figure 157 are inserted into TRANS_COUNTS.

	channel_descr	Count(Tran_Id)
1	ACH	9972
2	Branch	20317
3	Check	10714
4	Electronic	9037
5	Internet	4496
6	Other	6427
7	Paper	43443
8	Wire	8583

Figure 157: Transaction count by channel for all transactions

To compare historical counts against the most recent activity by account, the combined results are extracted from the temporary table. These results are shown in Figure 157.

```
SELECT c.channel_descr, sum(t.HIST_CNT), sum( t.RECENT_CNT)
FROM channel_descr c, TRANS_COUNTS t
WHERE c.channel_nbr = t.channel
GROUP BY c.channel_descr
ORDER BY c.channel_descr
```

	channel_descr	Sum(HIST_CNT)	Sum(RECENT_CNT)
1	ACH	9972	127
2	Branch	20317	202
3	Check	10714	100
4	Electronic	9037	90
5	Internet	4496	37
6	Other	6427	60
7	Paper	43443	343
8	Wire	8583	79

Figure 158: Comparison results from temporary table showing historical and recent counts

Comparing recent customer behavior to historical behavior by using Figure 158, the initial belief is that there has been an increase in the use of ACH transactions (direct deposit, etc.) from 8.83% to 12.2% of the transactions. Further analysis would reveal that the increase resulted from a corresponding decrease in the use of paper transactions, 38.44% to 33.04%. Even without calculating the percentages the traditionalist approach has created two temporary work tables, three passes to insert data into these tables, and a final pass to display the results. Business Objects can simplify this process in number of ways, building upon each other by utilizing techniques already discussed.

The first step is to construct a derived table that identifies the most recent transaction for each account. The derived table will return the last date for which transactions occurred for each account (Figure 159).

Figure 159: Derived table which returns last transaction date for each account

The derived table is then joined to the transaction history table based upon date and account number. The join is shown below and the altered universe is shown in Figure 160.

```
checking_tran.Acct_Nbr=LAST_TRANS.Acct_Nbr AND
checking_tran.Tran_Date=LAST_TRANS.last_date
```

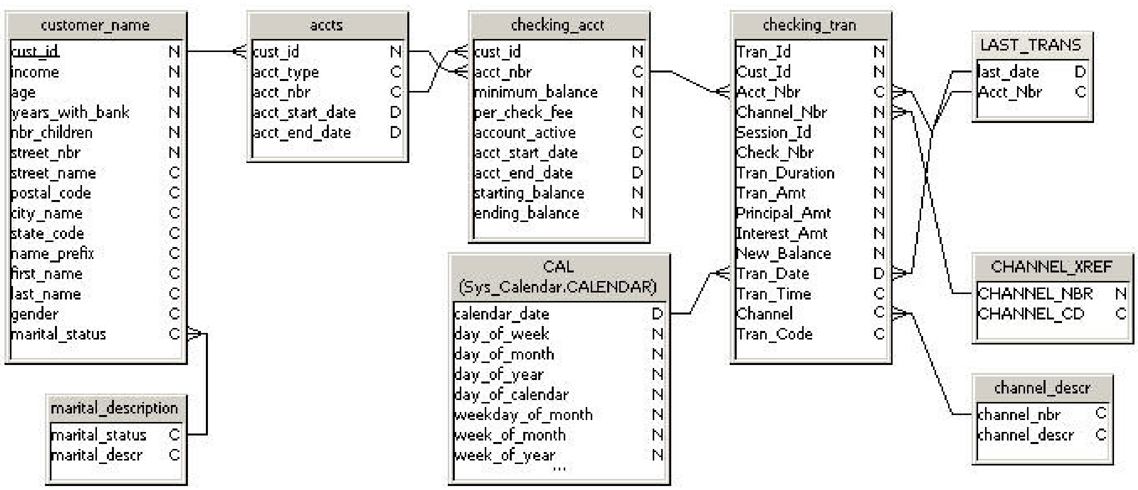


Figure 160: Derived table, LAST_TRANS, added to universe

Next a filter is constructed that takes advantage of the derived table (Figure 161). The *Where* clause for the filter is the same as the same join syntax.

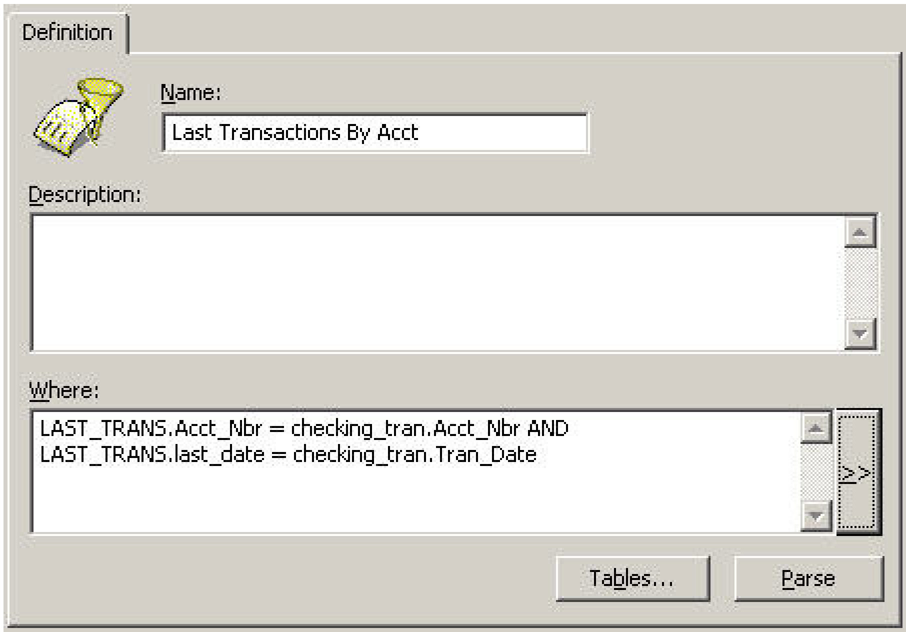


Figure 161: Filter to invoke the derived table LAST_TRANS. Used to return only the most recent transaction(s) for each account.

Now it's time to verify that the new filter does what is intended. First, a query is built to return all transactions by account (Figure 162).

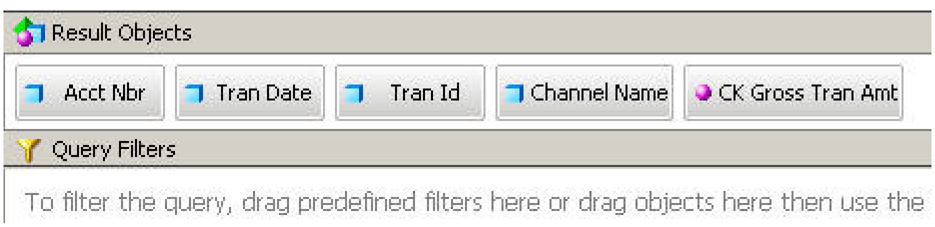


Figure 162: Query to return transactions by account

The generated SQL in Figure 163 is rather simple.

```
SELECT
  checking_acct.acct_nbr,
  checking_tran.Tran_Date,
  checking_tran.Tran_Id,
  channel_descr.channel_descr,
  sum (checking_tran.Tran_Amt)
FROM
  checking_tran,
  checking_acct,
  channel_descr
WHERE
  ( checking_acct.acct_nbr=checking_tran.Acct_Nbr  )
  AND  ( checking_tran.Channel=channel_descr.channel_nbr  )
GROUP BY
  1,
  2,
  3,
  4
```

Figure 163: Generated SQL to return transactions by account

Many rows are returned for each account as no limitation was placed on the query as seen in Figure 164.

Acct Nbr	Tran Date	Tran Id	Channel Name	CK Gross Tran Amt
0000000013624862	1/4/95	1	Paper	$38.70
0000000013624862	1/11/95	2	Branch	$379.28
0000000013624862	4/8/95	7	Paper	$67.32
0000000013624862	4/19/95	8	Check	$0.00
0000000013624862	7/14/95	12	Electronic	$47.61
0000000013624862	7/25/95	13	Internet	$0.00
0000000013624862	8/13/95	15	Paper	$18.30
0000000013624862	9/12/95	17	Paper	$123.76

Figure 164: Query results for transactions by account

The existing query is now modified to use the new filter.

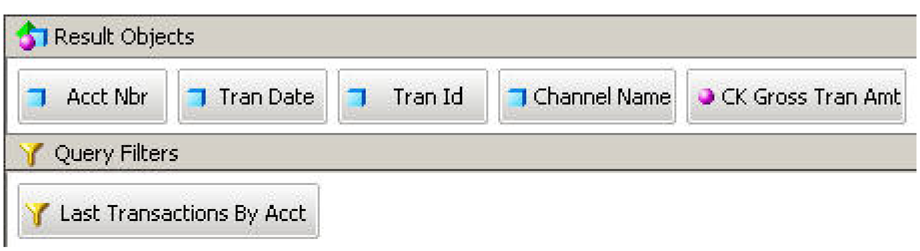


Figure 165: Query to return most recent transactions by account

Now the generated SQL is a bit more complex (Figure 166). By joining the transaction table to the derived table, the transactions will now be filtered by what is retrieved by the derived table.

```
SELECT
  checking_acct.acct_nbr,
  checking_tran.Tran_Date,
  checking_tran.Tran_Id,
  channel_descr.channel_descr,
  sum (checking_tran.Tran_Amt)
FROM
  checking_tran,
  checking_acct,
  (
  select max(tran_date) as last_date, acct_nbr from checking_tran group by acct_nbr
  ) LAST_TRANS,
  channel_descr
WHERE
  ( checking_acct.acct_nbr=checking_tran.Acct_Nbr  )
  AND  ( checking_tran.Acct_Nbr=LAST_TRANS.Acct_Nbr and checking_tran.Tran_Date=LAST_TRANS.last_date  )
  AND  ( checking_tran.Channel=channel_descr.channel_nbr  )
  AND
  ( LAST_TRANS.Acct_Nbr = checking_tran.Acct_Nbr AND LAST_TRANS.last_date = checking_tran.Tran_Date  )
GROUP BY
  1,
  2,
  3,
  4
```

Figure 166: Generated SQL to return most recent transactions by account

The results are indeed filtered, with only the most recent transactions for each account being shown. Notice that in Figure 167 transactions are displayed for different accounts and they occurred on different days.

Acct Nbr	Tran Date	Tran Id	Channel Name	CK Gross Tran Amt
0000000013624862	12/17/95	24	Other	$15.54
0000000013624892	12/23/95	161	Wire	$0.00
0000000013624982	12/31/95	56	Paper	$191.11
0000000013625002	12/25/95	18	Electronic	$691.07
0000000013625032	12/23/95	132	Paper	$19.56
0000000013625512	12/29/95	78	Paper	$228.89
0000000013626052	12/23/95	81	ACH	$17.99

Figure 167: Query results for most recent transactions by account

The first step is to build a query to return the number of transactions for each channel as shown in Figure 168.

Figure 168: Query to return transaction count by channel

The generated SQL is as expected (Figure 169).

```
SELECT
  channel_descr.channel_descr,
  count(checking_tran.Tran_Id)
FROM
  checking_tran,
  channel_descr
WHERE
  ( checking_tran.Channel=channel_descr.channel_nbr  )
GROUP BY
  1
```

Figure 169: Generated SQL to return transaction count by channel

The results returned by the query, Figure 170, are the same as those returned by the manual SQL, Figure 157.

Channel Name	Trans Count
ACH	9,972
Branch	20,317
Check	10,714
Electronic	9,037
Internet	4,496
Other	6,427
Paper	43,443
Wire	8,583

Figure 170: Results of transaction count by channel

Returning to the query panel the first step is to duplicate the existing query. Once this is done the newly duplicated query is modified to include the recent transaction filter, Figure 171.

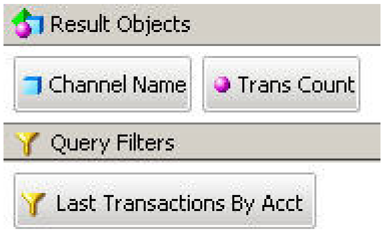


Figure 171: New query to return the count of the most recent transactions for each account by channel

The generated SQL, Figure 172, includes the join to the derived table which will result in only the most recent transactions being including in the count.

```
SELECT
  channel_descr.channel_descr,
  count(checking_tran.Tran_Id)
FROM
  checking_tran,
  (
  select max(tran_date) as last_date, acct_nbr from checking_tran group by acct_nbr
  ) LAST_TRANS,
  channel_descr
WHERE
  ( checking_tran.Acct_Nbr=LAST_TRANS.Acct_Nbr and checking_tran.Tran_Date=LAST_TRANS.last_date  )
  AND  ( checking_tran.Channel=channel_descr.channel_nbr  )
  AND
  ( LAST_TRANS.Acct_Nbr = checking_tran.Acct_Nbr AND LAST_TRANS.last_date = checking_tran.Tran_Date  )
GROUP BY
  1
```

Figure 172: Generated SQL to return the count of the most recent transactions for each account by channel

The results are added as a new block to the existing report. The results for the count of the most recent transactions for each account by channel, Figure 173, do correspond to the same totals obtained from manually entering the SQL, Figure 156.

Channel Name	Trans Count
ACH	127
Branch	202
Check	100
Electronic	90
Internet	37
Other	60
Paper	343
Wire	79

Figure 173: Query results for the count of the most recent transactions for each account by channel

To identify each block a row is inserted above each block and an appropriate title. Then using the calculation wizard available in the Web Intelligence tool bar, a percentage is added to each column (Figure 174).

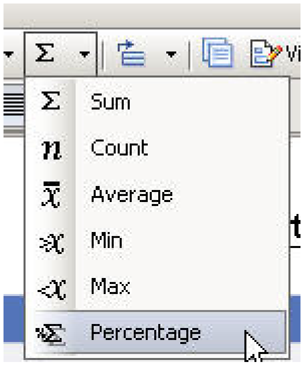


Figure 174: Web Intelligence calculation wizard

Using the final results in Figure 175 it is easy to see that paper transactions have dropped due to an increase in automated transactions (ACH).

Historical			Most Recent		
Channel Name	Trans Count	Percentage	Channel Name	Trans Count	Percentage
ACH	9,972	8.83%	ACH	127	12.24%
Branch	20,317	17.98%	Branch	202	19.46%
Check	10,714	9.48%	Check	100	9.63%
Electronic	9,037	8.00%	Electronic	90	8.67%
Internet	4,496	3.98%	Internet	37	3.56%
Other	6,427	5.69%	Other	60	5.78%
Paper	43,443	38.45%	Paper	343	33.04%
Wire	8,583	7.60%	Wire	79	7.61%
	Percentage:	100.00%		Percentage:	100.00%

Figure 175: Final results comparing transactions by channel

To obtain these results two very similar queries were created. Then calculations were created at the report level to obtain percentage of total transactions. Further formatting was required to bring the blocks together to present the consolidated view shown in Figure 176.

	Historical		Most Recent	
Channel Name	Trans Count	Percentage	Trans Count	Percentage
ACH	9,972	8.83%	127	12.24%
Branch	20,317	17.98%	202	19.46%
Check	10,714	9.48%	100	9.63%
Electronic	9,037	8.00%	90	8.67%
Internet	4,496	3.98%	37	3.56%
Other	6,427	5.69%	60	5.78%
Paper	43,443	38.45%	343	33.04%
Wire	8,583	7.60%	79	7.61%
	Percentage:	100.00%		100.00%

Figure 176: Final formatted results to compare transaction count

The next question is, is there anything to be done to reduce the work required by the report developer? The problems of multiple queries, additional formatting, and usage of report variables seem very familiar. Using techniques outlined previously a more refined solution is possible.

The first step is to alias the transaction table as *last_checking_tran* and then include it as part of the universe. The *LAST_TRANS* derived table used to filter for the most recent transactions by account is only joined to the new alias. This is the only difference to the technique covered in the varying grains of measurement section of this paper. In that section the same dimension tables were joined to all alias tables. In this situation *LAST_TRANS* is being used as a filtering table and not a dimension table. All other dimension tables, such as *CAL* and *channel_descr*, are joined to the new alias. Once this is done the contexts are regenerated using the wizard within Designer. Once these steps are completed the universe resembles Figure 177

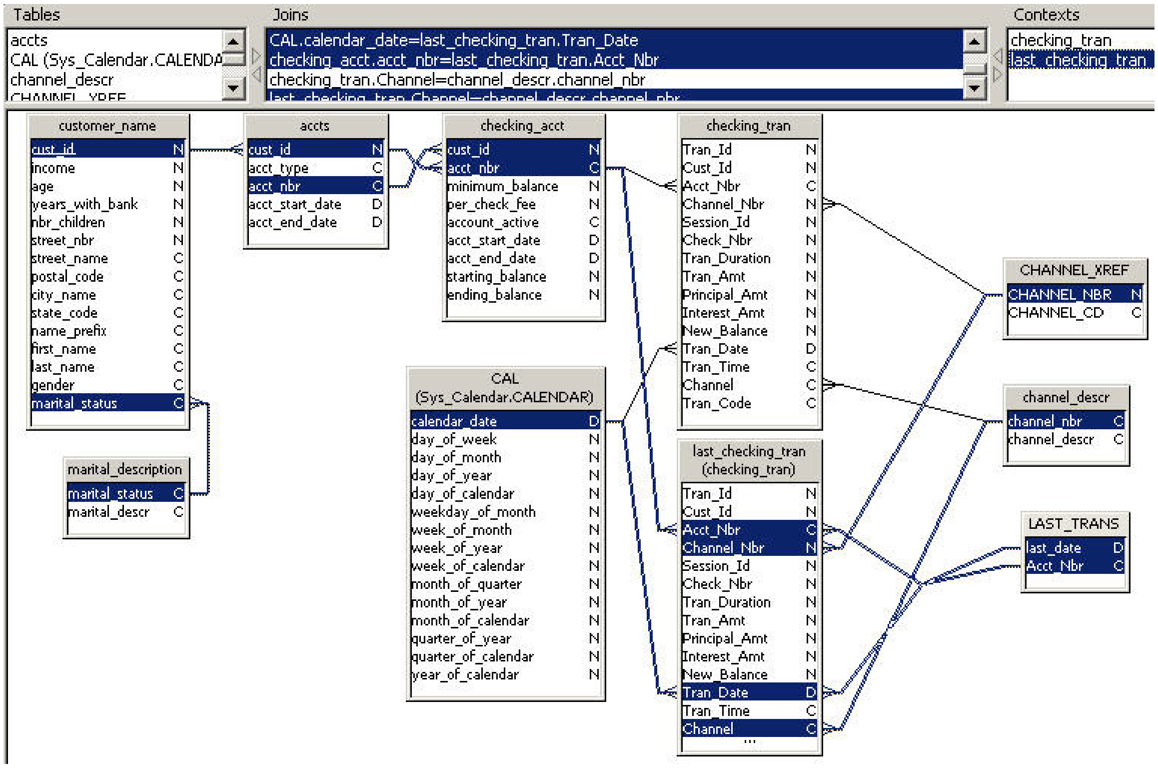


Figure 177: Universe with alias of transaction table added

An object is created for transaction count based upon the most recent transactions by account (Figure 178). This new measure object sources the new alias table and contains a *Where* clause to enforce the filter for most recent transactions.

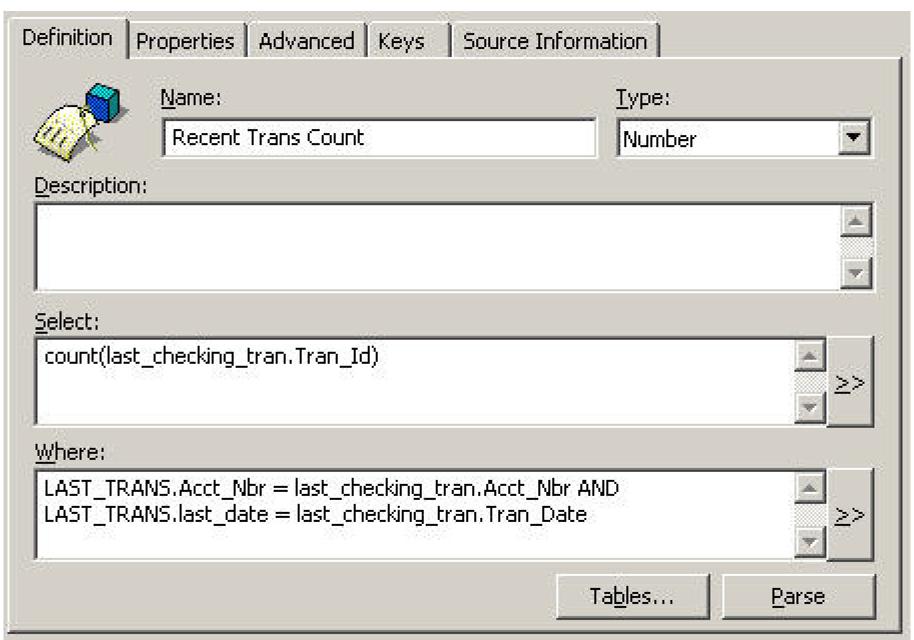


Figure 178: New measure object to count only the recent transactions by account

Once these changes are in place the query creation is a lot easier. A single query is created without any filters (Figure 179).

Figure 179: Query panel using universe having transaction alias

Due to each measure coming from a different context, multiple SELECT statements are generated. The first SELECT, Figure 180, returns the total transaction count by channel. The second SELECT returns the same transaction count but based only on the most recent transaction(s) for each account (Figure 181). As before, if *JOIN_BY_SQL* is set to *Yes* then only one SELECT is generated. Each of the previously separate SELECTs will become a derived table within a single SELECT statement.

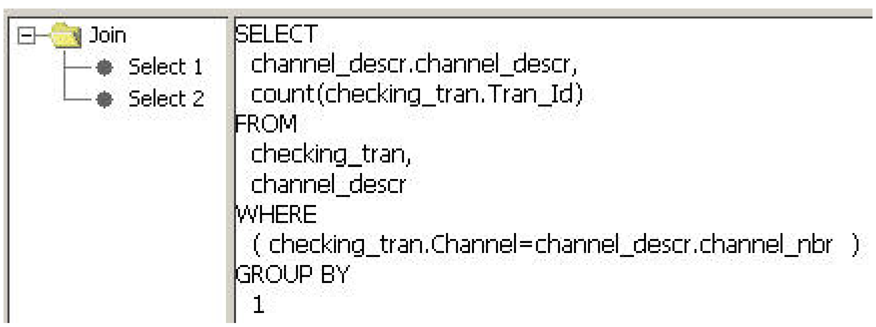


Figure 180: Generated SQL to return transaction count by channel

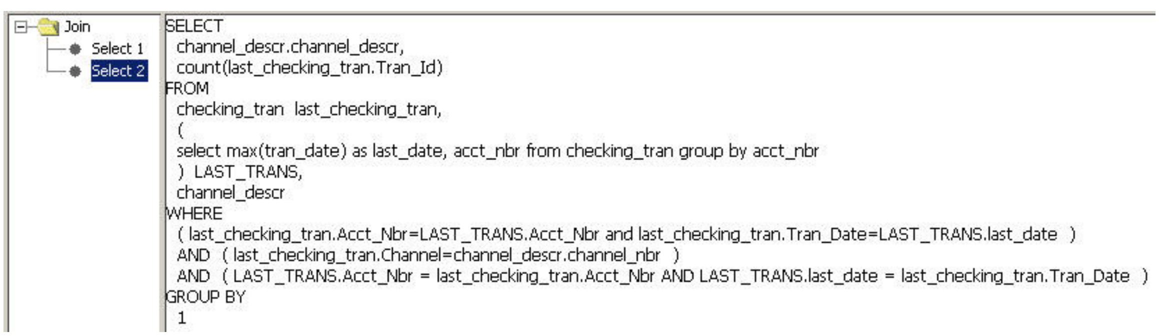


Figure 181: Generated SQL to return the count of the most recent transactions for each account by channel

As only one query was created within the query panel, only one report block is created by Web Intelligence, Figure 182

.

Channel Name	Trans Count	Recent Trans Count
ACH	9,972	127
Branch	20,317	202
Check	10,714	100
Electronic	9,037	90
Internet	4,496	37
Other	6,427	60
Paper	43,443	343
Wire	8,583	79

Figure 182: Initial results returned from single query

The after selecting each measure column, the calculation wizard on the tool bar can be used to add the appropriate percentages. The final result is shown in Figure 183.

Channel Name	Trans Count	Percentage	Recent Trans Count	Percentage
ACH	9,972	8.83%	127	12.24%
Branch	20,317	17.98%	202	19.46%
Check	10,714	9.48%	100	9.63%
Electronic	9,037	8.00%	90	8.67%
Internet	4,496	3.98%	37	3.56%
Other	6,427	5.69%	60	5.78%
Paper	43,443	38.45%	343	33.04%
Wire	8,583	7.60%	79	7.61%
	Percentage:	100.00%		100.00%

Figure 183: Final formatted results returned from single query

The next step would be to add the percentages to the universe. As this procedure has been illustrated in earlier sections of this paper there is not a need to repeat it here. The methodology has begun to repeat itself.

When situations arise that require analysis on a subset of data using a derived table as a filtering mechanism may be a solution. To simplify the solution the same techniques used in varying grains of measurement and using end results as part of calculation can be employed.

Xcelsius and Multi-pass SQL

Up to this point the multi-pass SQL discussion has been focused on Universe use within Web Intelligence and to a lesser extent Desktop Intelligence. The major difference between these two products' multi-pass SQL capabilities is the *Query Drill* document property available on a Web Intelligence document. This property allows semi-additive measure solution to also be used within drilling scenarios. Most of the techniques also apply whenever Crystal Reports utilizes a Universe as a data source. Keep in mind that Crystal Reports does not have the ability to combine the multiple result sets, one for each generated SQL statement, returned by the database Since many of the multi-pass solutions lead to the generation of multiple SQL statements, the Universe parameter *JOIN_BY_SQL* needs to be set to *Yes.*. With this setting the multiple SQL statements are submitted to the database as one statement and the database returns one result set. With this setting in place, Crystal Reports using the Universe as its data source should be able to take advantage of the multi-pass scenarios discussed.

Query as a Web Service, QaaWS, also utilizes Universes via a query panel. More like Web Intelligence and not Crystal Reports, QaaWS does have the ability to combine multiple result sets therefore the *JOIN_BY_SQL* parameter can be *Yes* or *No*. QaaWS creates a web service which returns data but has no useful data display capabilities on its own. For data visualization QaaWS is often paired with Crystal Xcelsius. Xcelsius can access data in many ways in addition to web services. But since the focal point of this paper has been primarily on Universe development, the combination of QaaWS and Xcelsius will be highlighted.

Xcelsius consumes the data returned from QaaWS. Crystal Xcelsius offers a different paradigm from the reporting tools already reviewed. Xcelsius does offer the ability to temporarily store query results, use previous query results as input to later queries, and post query data manipulation. Having a Universe which has implemented the multi-pass techniques eases the development of Xcelsius models. However Xcelsius is designed to for data visualization. For optimal performance its queries should be specific and return at most a few hundred rows of data. This is not to imply that Xcelsius can not be used to return detail level data. As seen in Figure 184, Xcelsius can effectively be used to show detail level data. Through a series of user interactions granular detail data is being retrieved without having to return an excessive number of rows of data. The drill actions of the user effectively limit the scope of the query.

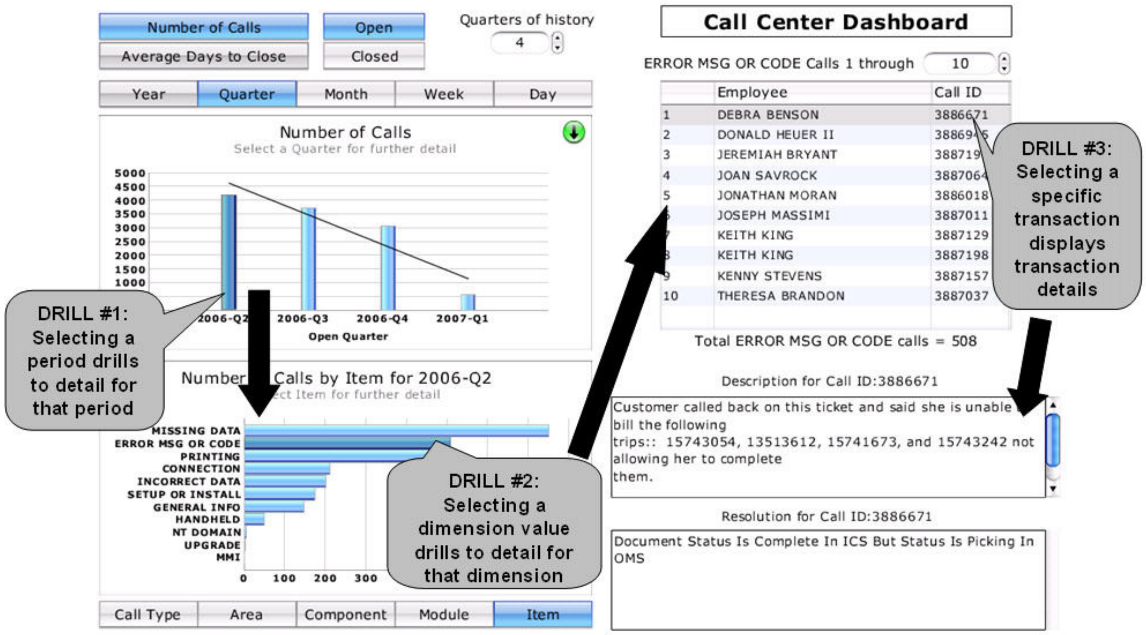


Figure 184: Xcelsius retrieving detail level data

For these reasons use of Xcelsius is very different from the reporting products. For simplicity each of the four multi-pass situations will be reviewed.

<u>Sharing dimensions across multiple fact tables</u>

The Universe approach used to handle this situation is considered best practice for Universe design. There is seldom a reason to bypass the use the contexts and/or the enablement of the Universe parameter setting for *Multiple SQL statements for each measure* (Figure 13). Unlike Crystal Reports, the setting of the *JOIN_BY_SQL* parameter is irrelevant. QaaWS is able to manage multiple SQL generation as long as the multiple result sets are *joined* and not *synchronized*. The most common situation in which result sets are synchronized occurs when all requested dimensions are not available in all contexts utilized in the query. In these situations QaaWS will return the following error message: “The query you built has incompatible objects. Make sure the objects are compatible.” Whenever result sets have to be synchronized, the setting of *JOIN_BY_SQL* is ignored. As QaaWS queries for Xcelsius tend to be very specific, usually involving one or two dimensions at a time (Figure 185), the likelihood of encountering synchronized queries is low. The Xcelsius example in Figure 184 allows the end user to select the desired dimension for the query. In this model, the technique limits the query to one time period, as indicated by 2006-Q2, and one other dimension such as *Item*. So this model effectively limits the query to one measure (number of calls) and one dimension (item) in the select clause and one other dimension (time, 2006-Q2) in the where clause.

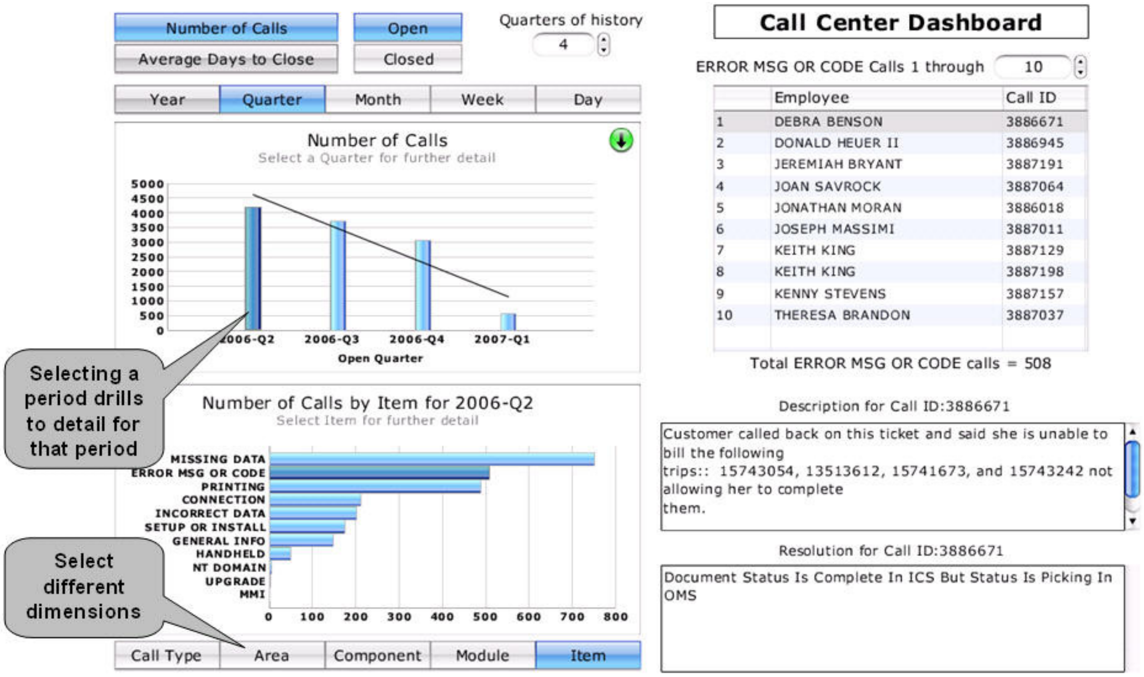


Figure 185: Xcelsius model drill into measures as of end of quarter

An alternative is to manually split the query into one query per fact table. In order to prevent too many queries from executing at once the queries should be executed serially and not in parallel. Each result set would then be stored within the Xcelsius model. The measures returned would need to be synchronized with the returned dimensions using the Excel functions of match, index, etc. Due to the complexity of this alternative, utilizing proper Universe design is preferred.

<u>Mixing the grains of measurement in the same query</u>

Often an Xcelsius model will offer the user the opportunity to indicate which time span to be shown (Figure 186). It is common for the end user to be presented with a set of radio buttons as a selector. For example, the buttons may offer current year, current month, and last year as options. There are really two solutions to this.

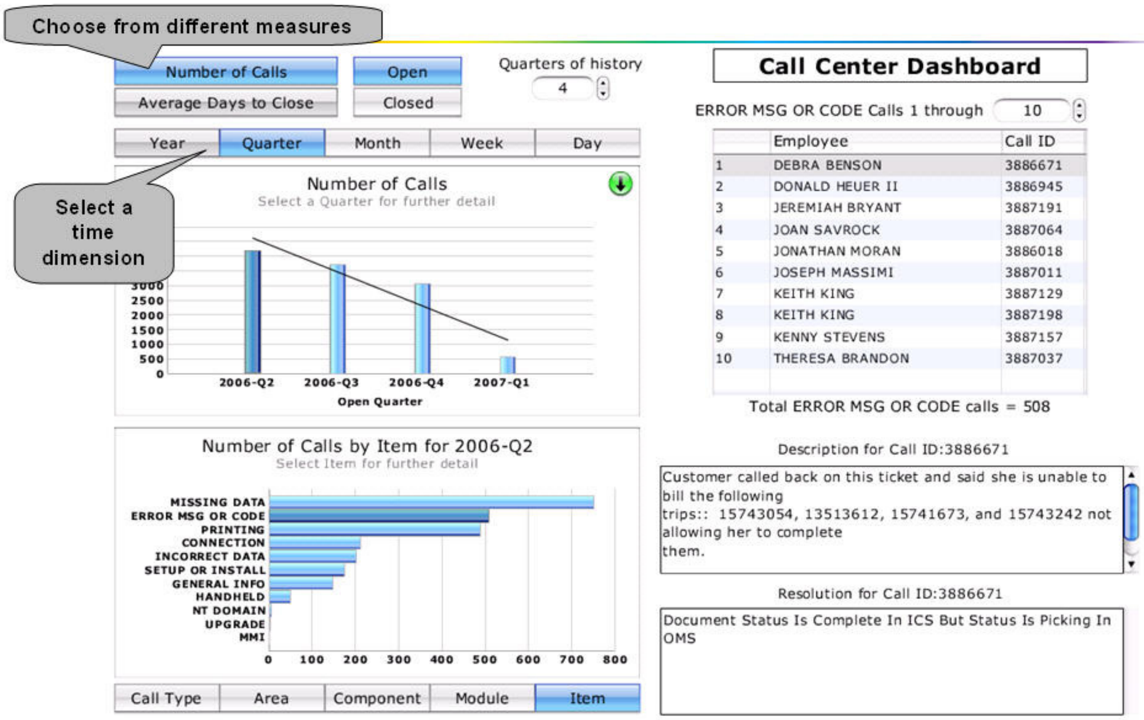


Figure 186: Xcelsius model allowing selection of timeframe for a measure

The first is for Xcelsius to execute a query every time the user alters the desired timeframe. For example the model may initially show the number of sales calls made for the current year. If the user then elects to see the sales calls for the current month, a new query is then executed. This option effectively restricts the query to a single timeframe eliminating the need for multi-pass SQL. The downside to this option is that if the user then elects to return to the current year view that query has to be re-executed.

The alternative is to have all possible timeframes returned when the model is loaded. The results are then cached within the model and displayed according to the user selection. Continuing with the example above, the number of sales calls for the current year, current month, and last year are retrieved and stored within the model when the model is initially loaded. As the user alters the view from current year to current month to last year no additional queries are execute. The proper timeframe is obtained from the results of the initial query. As there is not any database wait time as the timeframe is altered, response time tends to be quicker but it may take a bit longer for the model to initially load. One must examine the expected use of the model to make a proper decision on this classic tradeoff.

The remaining question is how to retrieve the multiple timeframes at model load time. The two options are to execute a separate QaaWS for each timeframe or a single QaaWS that utilizes a Universe built with techniques already discussed. Having several separate, specific QaaWS may lead to reuse of those web services but then there are many web services that have to be maintained.

Conversely having a single web service to maintain is nice but one may not be able to reuse it as often as desired. Another classic tradeoff encountered.

Defined calculations that require an end result as one of its factors
Calculations created within Xcelsius are subject to the operations available within Excel. Since retrieved data is stored within memory of an Xcelsius model, calculations can be based upon the end results of the query. For example, if a ratio of sales calls by category against the total number of sales calls is needed, the web service can return sales calls by category. Once returned these totals can be summed within the model to obtain the total number of sales calls which is then used against the totals by category to calculate the desired ratio. Similar to creating variables in Web Intelligence or Desktop Intelligence, this option also has the same issues. There is no reuse possible between Xcelsius models and this methodology leaves open the possibility of two different but valid calculations being named the same.

Need of semi-additive measures
User interaction with Xcelsius allows specific queries to be executed. When semi-additive measures are part of the visualization the user must specify the desired timeframe as shown in Figure 185. The selection of the timeframe can then be used execute the proper query.

Analyzing a subset of data
The Xeclsius model can be presented to allow the end user to specify various filters to add to a query. The user may elect to have one query return all the results without any filtering. For the comparison query, options may allow the use to choose from a wide array of choices. For example, in Figure 186 the user has the option to choose from *Open* or *Closed* cases. Thus, options can be presented to the end user that manipulate the query on the fly.

The question of dealing with multi-pass SQL depends as much on its definition as it does the capabilities of Business Objects. Business Objects can definitely return the results that necessitate the business need for multi-pass SQL. Before the advent of query and reporting tools, hand coded SQL was the answer for retrieving any information from a database. When hand coding SQL, the solution was to store intermediate results into temporary tables and pass the database again. Once all database passes were complete, the final result was obtained using the information accumulated in the temporary tables. The advanced query techniques employed by Business Objects circumvents this traditional approach. However, thinking in terms of the traditional approach can often aid in the development of derived tables and how to use them. The burden of solving a multi-pass situation shifts from being 100% database driven as in the hand coding scenario to a shared solution between the database and the query engine. The degree of the shift depends upon universe design, parameter settings, and the reporting requirements. As in other issues, the focus should be on solving the business problem and not the technique. Within Business Objects the multi-pass SQL problem can be solved without employing temporary tables as a traditionalist may expect. And usually results in a very well perform solution.

Appendix A – BOE Release Dependency

All the information and examples contained within this document are based upon Business Objects Enterprise XIr2. The service pack (SP) level is at SP2 or higher. At this time Business Objects Enterprise 3.0 has just been released. BOE 3.0 has new features that impact multi-pass SQL implementation. A revision of this paper is currently planned. None of the features in BOE 3.0 render the techniques reviewed obsolete. The additional features ease multi-pass SQL implementation and provide new techniques that can simplify the process.

Footnotes

[1] http://www.dbmsmag.com/9702d05.html
Features for Query Tools
DBMS - February 1997
Ralph Kimball

[2] http://www.olapreport.com/Architectures.htm
Nigel Pendse,. Last updated on June 27, 2006.

[3] http://kimballgroup.com/html/designtipsPDF/DesignTips2001/KimballDT29Graceful.pdf
Kimball Design Tip #29: Graceful Modifications To Existing Fact and Dimension Tables

[4] http://whitepapers.zdnet.co.uk/0,1000000651,260002629p,00.htm
Derived Tables and the NCR-MicroStrategy Decision Support System
DM Review
November 1999