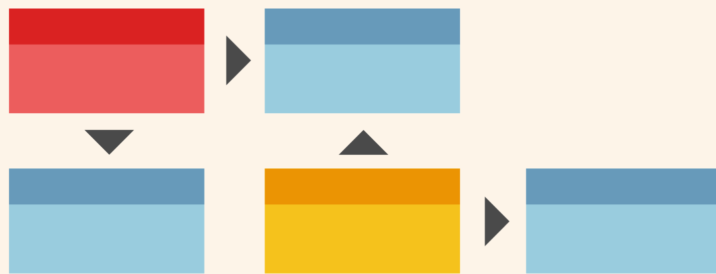
The Many-to-Many Revolution



ADVANCED DIMENSIONAL MODELING WITH
MICROSOFT SQL SERVER ANALYSIS SERVICES

produced by

The Many-to-Many Revolution

Advanced dimensional modeling with Microsoft SQL Server Analysis Services

Author: Marco Russo, Alberto Ferrari

Published: Version 2.0 Revision 1 – October 10, 2011

Contact: marco.russo@sqlbi.com and alberto.ferrari@sqlbi.com – www.sqlbi.com

Summary: This paper describes how to leverage the many-to-many dimension relationships, a feature that debuted available with Analysis Services 2005 and is now available by using DAX in the new BISM Tabular available in Analysis Services "Denali". After introducing the main concepts, the paper discusses various implementation techniques in the form of design patterns: for each model, there is a description of a business scenario that could benefit from the model, followed by an explanation of its implementation.

A separate download (available on http://www.sqlbi.com/manytomany.aspx) contains SQL Server database and Analysis Services projects with the same sample data used in this paper. BISM Tabular examples are available as PowerPivot for Excel workbooks.

Acknowledgments: we would like to thank the many peer reviewers that helped us to improve this document: Bryan Batchelder, Chris Webb, Darren Gosbell, Greg Galloway, Jon Axon, Mark Hill, Peter Koller, Sanjay Nayyar, Scott Barrett, Teo Lachev, Grant Paisley, Javier Guillén.

I would also like to thank T.K. Anand, Marin Bezic, Marius Dumitru, Jeffrey Wang, Ashvini Sharma and Akshai Mirchandani who answered to our questions.

Table of Contents

INTRODUCTION 5

MULTIDIMENSIONAL MODELS 7

- A Note about Visual totals 7
- A Note about Naming Convention & Cube Design Best Practices 7
- A Note about UDM and BISM acronyms 7

Classical many-to-many relationship 8

- Business scenario 8
- Implementation 8

Cascading many-to-many relationship 13

- Business scenario 13
- Implementation 16

Survey 24

- Business scenario 25
- Implementation 25

Distinct Count 33

- Business scenario 33
- Implementation 34
- Performance 45

Multiple groups 48

- Business scenario 48
- Implementation 49

Cross-Time 54

- Business scenario 54
- Implementation 55

Transition Matrix 63

- Business scenario 63
- Implementation 65

Multiple Parent/Child Hierarchies 70

- Business scenario 70
- Implementation 72

Hierarchy Reclassification with Unary Operator 80

- Business scenario 80
- Implementation 82
  - Handling of the unary operator 83
  - Using SQL to expand the expressions 85
  - Building the model 86

Considerations about Multidimensional Models 94

- Links 94

TABULAR MODELS 96

- A Note about UDM and BISM acronyms 96
- Modeling Patterns with many-to-many 96

Classical many-to-many Relationship 98

Business scenario 98
BISM Implementation 98
Denali Implementation 103
Performance Analysis 103

CASCADING MANY-TO-MANY RELATIONSHIPS 104

Business scenario 105
BISM Implementation 107

SURVEY 112

Business Scenario 112
BISM Implementation 114
Denali Implementation 119
Performance Analysis 120

MULTIPLE GROUPS 122

TRANSITION MATRIX 124

Transition Matrix with Snapshot Table 126

SNAPSHOT TABLE IN THE SLOWLY CHANGING DIMENSION SCENARIO 127

SNAPSHOT TABLE IN THE HISTORICAL ATTRIBUTE TRACKING SCENARIO 130

TRANSITION MATRIX WITH CALCULATED COLUMNS 133

BASKET ANALYSIS 138

Denali Implementation 143

CONSIDERATIONS ABOUT MULTIDIMENSIONAL MODELS 147

Links 147

Introduction

SQL Server Analysis Services introduced modeling many-to-many relationships between dimensions in version 2005. At a first glance, we may tend to underestimate the importance of this feature: after all, many other OLAP engines do not offer many-to-many relationships. Yet, this lack did not limit their adoption and, apparently, only a few businesses really require it.

The SQL Server Analysis Services version that will be released in 2012 (currently codenamed “Denali”) will introduce a new modeling (BISM, Business Intelligence Semantic Model) choice that is called “BISM Tabular” and will rename the former UDM (Unified Dimensional Model) to “BISM Multidimensional”.

UDM/BISM Multidimensional models can leverage many-to-many relationships helping you to present data from different perspectives that are not feasible with a traditional star schema. This opened a brand new world of opportunities that transcends the limits of traditional OLAP. At the same time, while BISM Tabular will not directly support many-to-many relationships between tables, you will be able to express such relationships by using DAX formulas.

The DAX language can be used in PowerPivot for Excel, which basically is SSAS running in process inside Excel and provides a good method to prototype complex cubes, learn the DAX language and experiment with the modeling features we are describing here.

In this paper, we will explore many different uses of many-to-many relationships in both BISM Multidimensional and BISM Tabular, in order to give us more choices to model effectively business needs, including:

- Classical many-to-many
- Cascading many-to-many
- Survey
- Distinct Count
- Multiple Groups
- Cross-Time
- Transition Matrix
- Multiple Hierarchies
- Hierarchy Reclassification with unary operator
- Basket Analysis

The paper will first present the BISM Multidimensional models, and then the BISM Tabular models. Most of the models correspond to the BISM Multidimensional and one is unique to BISM Tabular (Basket Analysis). You can read these two sections of the paper independently.

Although you do not have to, we recommend reading the models in the presented order, because often one builds upon a previous model and we have arranged them in order of complexity.

It is fundamental to understand how many-to-many relationships work within Analysis Services (in both BISM Multidimensional and BISM Tabular) in order to use them for different purposes: minor implementation details such as the relationships between dimensions and measure groups could have major design repercussions since small changes may lead to different results and confusion to the end users. The theory of chaos applies wonderfully to the usage of many-to-many relationships.

Each model has a brief introduction, followed by a business scenario that may benefit from its use and an explanation of its implementation. Each model uses only the minimal set of dimensions that are necessary to explain the concept behind it and a small dataset that demonstrates the underlying behavior.

Multidimensional Models

This section presents the many-to-many relationships applied to BISM Multidimensional / UDM models.

A Note about Visual totals

There is an important warning about the use of the VisualTotals MDX function and the corresponding OLAP feature. Visual Totals apply only to one level at a time with many-to-many dimensions. In the Links section, you will find a link to a document written by Richard Tkachuk that explains this limitation

A Note about Naming Convention & Cube Design Best Practices

The examples in this document do not follow best practices in naming convention and cube design for Analysis Services. Just to make an example, we kept the prefix Dim and Fact to the entities in Analysis Services and we directly map SQL Server tables, instead of using views defined on the database in order to decouple the two layers. These and other suggestions are included in the SQLBI Methodology paper and in the Expert Cube Development with Microsoft SQL Server 2008 Analysis Services book (see links at the end of the document).

A Note about UDM and BISM acronyms

In 2011, Microsoft announced that BISM (Business Intelligence Semantic Model) is the new acronym that groups all the models that you can create with Analysis Services. UDM (Unified Dimensional Model) is the name that, since SQL Server 2005, identified a multidimensional model for Analysis Services. Starting with SQL Server "Denali", which will probably be released in 2012, there will be two types of modeling possible in Analysis Services: BISM Multidimensional, which corresponds to the previous UDM, and BISM Tabular, which relies on a new storage engine (Vertipaq) and requires a different approach to model many-to-many relationships. In this part of the paper, we only consider the BISM Multidimensional model and we used the previous UDM acronym, because all the techniques described here can be used with any version of UDM since SQL Server Analysis Services 2005.

Classical many-to-many relationship

This common situation may benefit from many-to-many relationships. We will analyze a situation where we have a very simple many-to-many relationship between two dimensions. Even if the case is very simple, it is still useful to analyze it in detail because, later, those details will get more complex and we need to understand them very well.

Business scenario

Here is a typical business scenario drawn from the bank business: we have a fact table that describes a measure (in this case an account balance taken at a given point of time) for a given entity (a bank account) that can be joined to many members of another dimension (a joint account owned by several customers).

Those of you familiar with the "classical" multidimensional model can already see the difficulty. It is not easy to describe the non-aggregability of measures joined to dimensions with a many-to-many relationship. In this case, each bank account can have one or more owners and each owner can have one or more accounts but we should not add the cash of an owner to his/her joint owners.

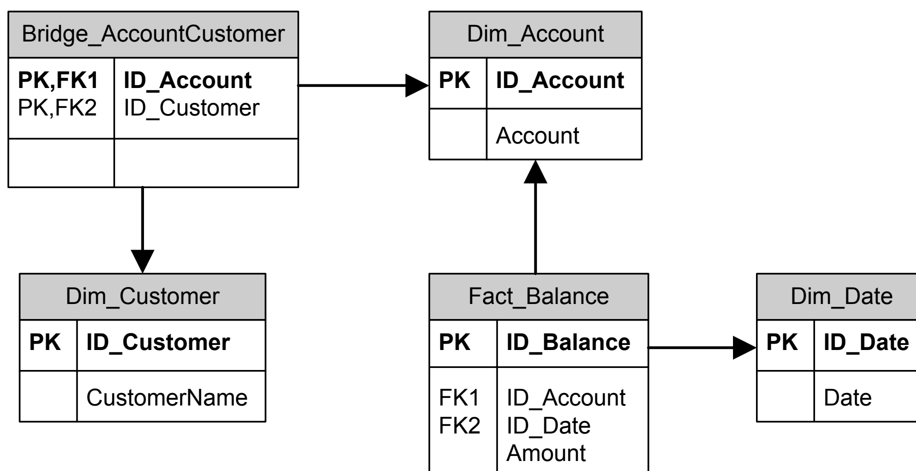


Figure 1 – Classical many-to-many diagram

There are many other scenarios where the classical many-to-many relationship appears. Because it is the simplest of all many-to-many relationships, we use this example to describe in detail how many-to-many relationships work in Analysis Services. We do not want to describe all the typical situations where we can use the classical many-to-many relationship.

Implementation

The important thing to remember is that we use a many-to-many relationship to correlate dimensions. In OLAP cubes, the relationship is always between facts and dimensions. We need to introduce an intermediate fact table that defines the many-to-many relationship between the dimensions. This fact table will join to both dimensions and act as a bridge between them. Typically, this "special" fact table has no measures.

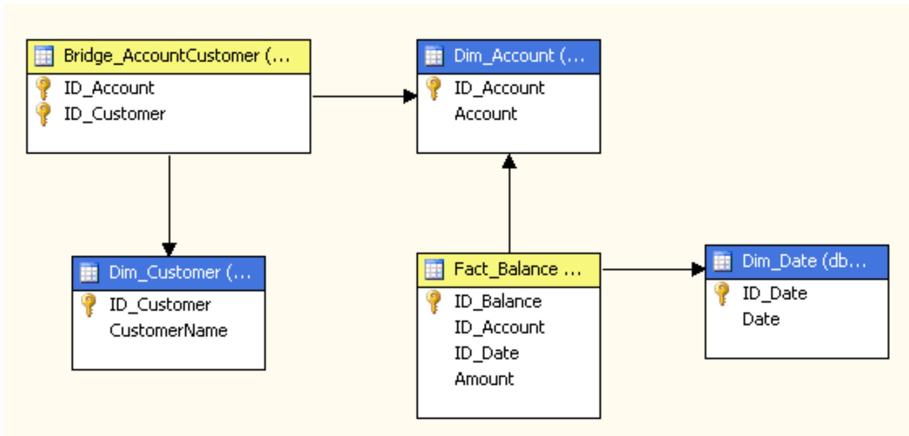


Figure 2 – Relational model with many-to-many relationship

When you define relationships between the dimensions and the measure groups, you specify that Dim Customer is joined to Fact Balance through the Bridge Account Customer measure group (as defined by the selected item in Figure 3).

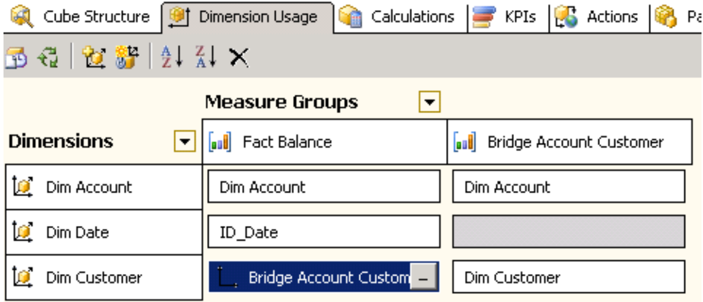


Figure 3 – UDM with many-to-many relationship

Please note that Figure 3 shows the results of the “auto build” feature of the Cube wizard: the wizard does a good job in this case detecting that the many-to-many relationship between Fact Balance and Dim Customer is via the Bridge table. In subsequent models, we will take these relationships one-step further by adding other dimension-measure group relationships manually. When the going is get tough, the auto build feature will be of no help, only our brain will be useful.

We can further describe the many-to-many relationship using the “Define Relationship” dialog box (Figure 4) that is displayed when you click on the button contained in the intersecting cell (the highlighted cell in Figure 3). This dialog box is available for every combination between dimensions and measure groups and it allows you to select the type of relationship.

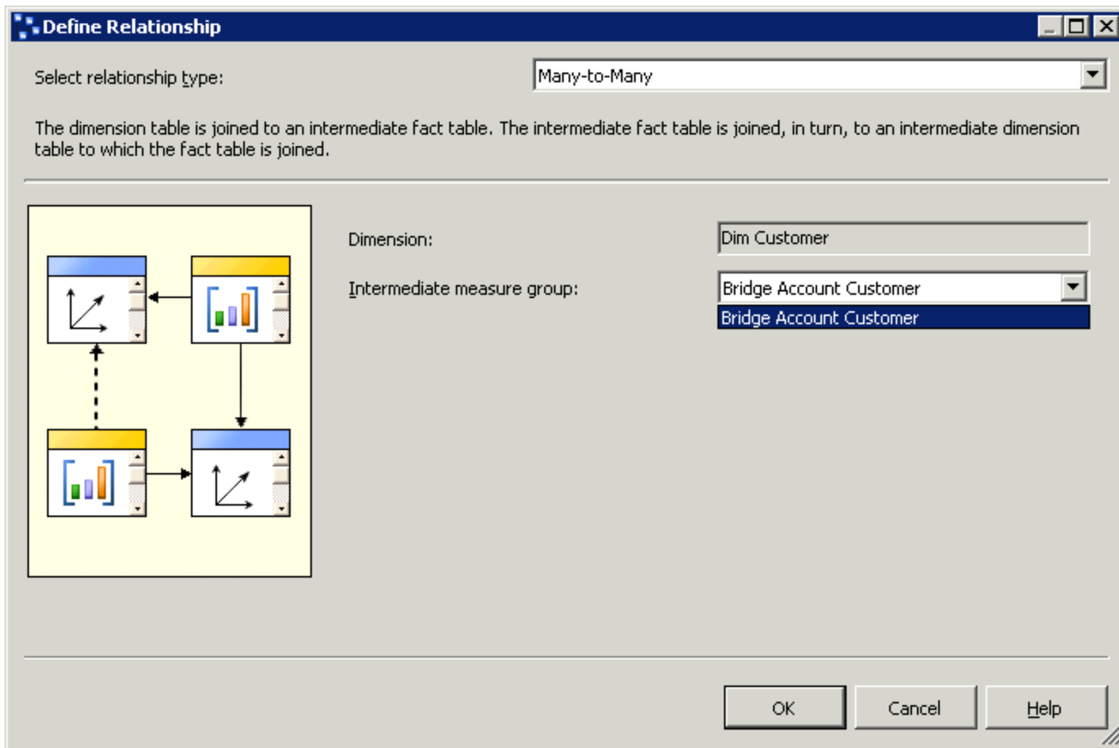


Figure 4 – Many-to-many relationship dialog box

When we define the relationship, we can cross the two dimensions and see the results shown in Figure 5.

We created four customers (Luke, Mark, Paul and Robert) and six accounts in the test tables. Each account joins to one or more customers (the account name is a concatenation of the account holders) and the balance for each account is always 100 at the date used by the query, as you can see in Figure 5.

	A	B	C	D	E	F
1	Date	2005-12-31 00:00:00				
2						
3	**Amount**	**Customer Name**				
4	**Account**	**Luke**	**Mark**	**Paul**	**Robert**	**Grand Total**
5	Luke	100				100
6	Mark		100			100
7	Mark-Paul		100	100		100
8	Mark-Robert		100		100	100
9	Paul			100		100
10	Robert				100	100
11	**Grand Total**	**100**	**300**	**200**	**200**	**600**

Figure 5 – Many-to-many relationship result

For each customer, we can identify the accounts that he owns and, for each account, we can see the balance repeated for each owner. What is interesting is that the total for each account (row) is always 100 (Grand Total row) and the balance for all accounts is 600 (100 * 6), instead of the sum of the customers balance, which would be 800 but doing that would sum the same account multiple times. This is the first example of the great power of the many to many dimensional modeling tools.

Figure 6 better highlights the particular aggregability of Amount and Count in respect of Dim Customer. You can easily see that they are not simply summed up but that the many-to-many relationship is working making the total of the column different from the sum of all the rows.

	A	B	C
1	Date	2005-12-31 00:00:00	
2			
3		Values	
4	Row Labels	Amount	Fact Balance Count
5	Luke	100	1
6	Mark	300	3
7	Paul	200	2
8	Robert	200	2
9	**Grand Total**	**600**	**6**

Figure 6 – Measures aggregation by Customer Name

We could obtain the same result without many-to-many relationships but only with some stunts and tradeoffs in terms of processing time or query performance (compared to results we can obtain with Analysis Services and many-to-many relationships).

Now, let us make some consideration about the count measure that is available in the Bridge Account Customer measure group. We said that – normally – the bridge table does not contain measures. Nevertheless, we are interested in looking at what happens when we use them.

Even if it seems to be very similar to the Fact Balance Count measure, it has an important difference that we can observe by querying different data. Let us look at data related to Jan-06 in Figure 7.

	A	B	C	D	E	F
1	Date	2006-01-31 00:00:00				
2						
3	Amount	Customer Name				
4	Account	Luke	Mark	Paul	Robert	Grand Total
5	Luke	105				105
6	Mark		105			105
7	Mark-Robert		105		105	105
8	Paul			105		105
9	Robert				105	105
10	**Grand Total**	**105**	**210**	**105**	**210**	**525**

Figure 7 – Account Mark-Paul is missing in Jan-06 data

Here you can see that the balance for the account Mark-Paul is missing from the query results and for this reason we do not have a corresponding row. This will have consequences in the measures exposed by the Fact Balance Count and Bridge Account Customer measure groups.

	A	B	C	D	E	F	G
1		Column Labels					
2		2005-12-31 00:00:00		2006-01-31 00:00:00		Total Fact Balance Count	Total Bridge Account Customer Count
3	Row Labels	Fact Balance Count	Bridge Account Customer Count	Fact Balance Count	Bridge Account Customer Count		
4	Luke	1	1	1	1	2	1
5	Mark	3	3	2	3	5	3
6	Paul	2	2	1	2	3	2
7	Robert	2	2	2	2	4	2
8	**Grand Total**	**6**	**8**	**5**	**8**	**11**	**8**

Figure 8 – Counts with no relationship between Dim Date and Bridge

Figure 8 shows query results for the two different count measures.

- The Fact Balance Count will count rows in the Fact Balance measure group: in this query, it represents how many balances are present for each customer within a given period. Since each

account has only one balance for each month, it could also be mistaken for the number of accounts that a customer has, but the Grand Total proves that this assumption is incorrect.

- The Bridge Account Customer Count measure provides always the number of accounts for each customer correctly. We obtain this value by directly counting the number of rows in the Bridge Account Customer measure group. However, this number is time invariant from a date, because its measure group has no relationship with the time dimension (Dim Date).

If we add the relationship between the Bridge Account Customers measure group and the Date dimension, we can see values that are more interesting. We do that by stating that Bridge Account Customers relates to Dim Date through Fact Balance. What we are doing is simply reversing the relationship (see Figure 9).

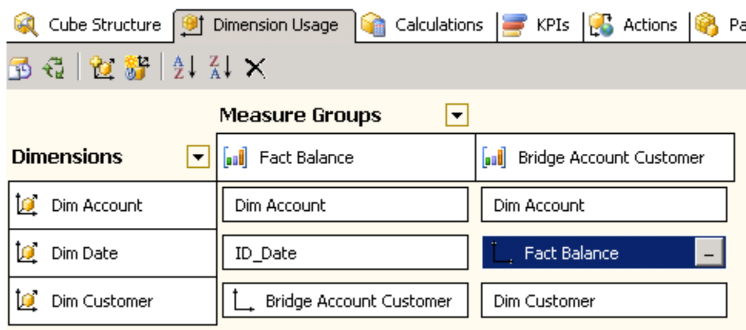


Figure 9 – Many-to-many relationship between Dim Date and Bridge Account Customer

The result is that now the Date dimension influences the values reported by the bridge tables, we can see it in Figure 10.

	Column Labels					
	2005-12-31 00:00:00		2006-01-31 00:00:00		Total Fact Balance Count	Total Bridge Account Customer Count
Row Labels	Fact Balance Count	Bridge Account Customer Count	Fact Balance Count	Bridge Account Customer Count		
Luke	1	1	1	1	2	1
Mark	3	3	2	2	5	3
Paul	2	2	1	1	3	2
Robert	2	2	2	2	4	2
Grand Total	6	8	5	6	11	8

Figure 10 – Counts with many-to-many relationship between Dim Date and Bridge Account Customer

Numbers have changed somewhat (changes are highlighted). Now the correct interpretation of the Bridge Account Customer Count measure is that it represents the number of combinations between customers and accounts having at least one balance in the considered period.

This explains the lower value in Jan-06 for Mark and Paul (a corresponding account balance is missing in that month) while the Grand Total is still the same (it includes both Dec-05 and Jan-06, so the account Mark-Paul has at least one balance).

We encourage you to experiment with your data using many-to-many relationships. This will really help you to understand the implications of having or not having a relationship between a dimension and a bridge measure group. It is only when you really master that concept at this simple level (only two measure groups involved) that you can go further with advanced dimensional modeling techniques, which leverage many-to-many relationships.

Cascading many-to-many relationship

When we apply the many-to-many relationship several times in a cube, we have to pay attention if there is a chain of many-to-many relationships. As we have seen in the classical many-to-many relationship scenario, dimensions that apparently do not relate to a bridge measure group could enhance the analytical capabilities of our model.

We will call the situation where there is a chain of many-to-many relationships a “cascading many-to-many relationship”.

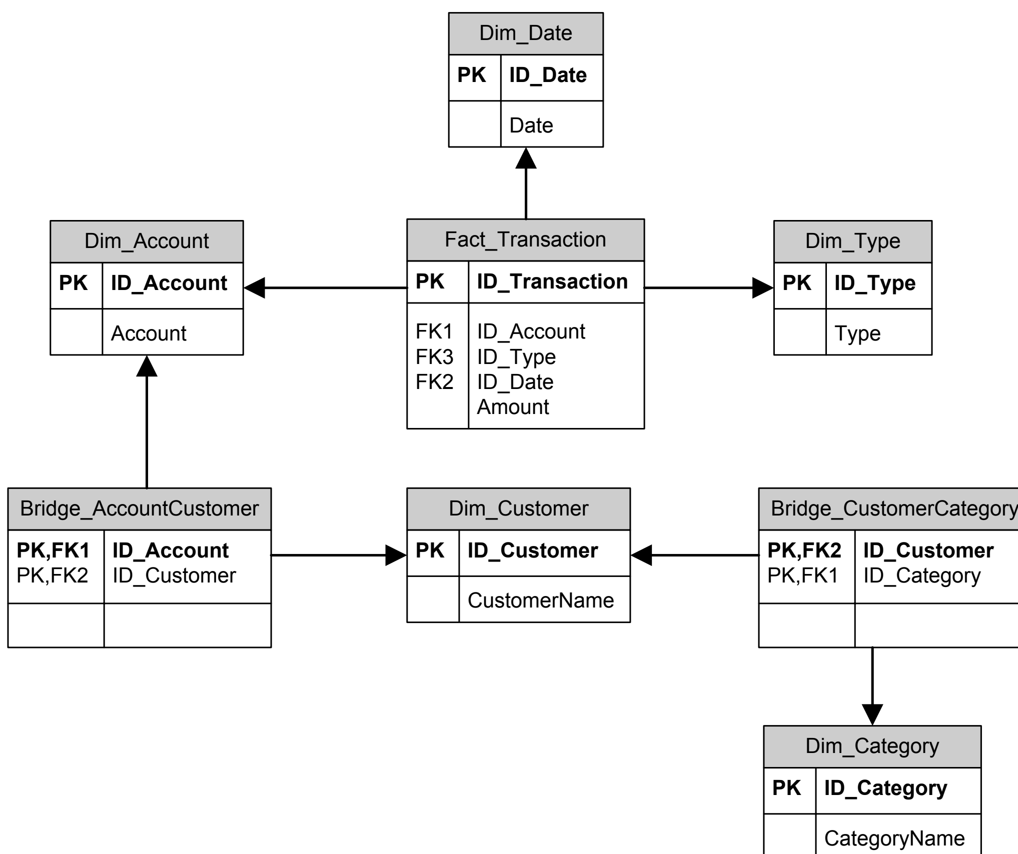


Figure 11 – Cascading many-to-many relationship diagram

In the picture, we can see that – in order to associate a category to a transaction – we need to traverse two many-to-many relationships: the first one from account to customer and the second one from customer to category.

Business scenario

A typical scenario is the case when a dimension far from the main fact table (a dimension that relates to one bridge fact table) is involved in an existing many-to-many relationship and has another many-to-many relationship with another dimension.

For example, consider this slightly modified bank account scenario, with a different fact that we want to consider:

- Account transactions: Transactions fact table related to Dim Date, Dim Account and Dim Type.
- Each account can have one or more owners (customers)
- Dim Account has a many-to-many relationship with Dim Customer
- **Each customer can be classified into one or more categories** Dim Customer has a many-to-many relationship with Dim Categories

Although we could have used the previous balance accounts scenario, the new schema adds the Dim Type dimension so we need to use the many-to-many relationship in a bidirectional way.

In order to understand the examples, we need to describe some of the data that we will use in our implementation. Table 1 shows the de-normalized fact table. Even if the Date dimension is not strictly necessary for this explanation, we will keep it in the model because it is a common dimension in a similar scenario and it is useful to see how it relates to the other dimensions.

Account	Type	Date	Amount
Mark	Cash deposit	20051130	1000.00
Paul	Cash deposit	20051130	1000.00
Robert	Cash deposit	20051130	1000.00
Luke	Salary	20051130	1000.00
Mark-Robert	Salary	20051130	1000.00
Mark-Paul	Cash deposit	20051130	1000.00
Mark	ATM withdrawal	20051205	-200.00
Robert	Credit card statement	20051210	-300.00
Paul	Credit card statement	20051215	-300.00
Luke	ATM withdrawal	20051215	-200.00

Table 1 – Fact table transaction data

The Type dimension is very important for our purposes: it describes the type of the transaction and it is useful to group transactions across other dimensions.

Let us see some kind of questions the user may ask based on this data:

- What is the salary/income for the “IT enthusiast” category?
- How many different transaction types involve the “Rally driver” category?
- What customer categories have ATM withdrawal transactions?

Within the fact table, there is not enough information to provide answers to those questions but all that we need is stored in tables (dimensions) reachable through the many-to-many relationships. We only have to create the correct relationships between dimensions. Table 2 contains the relationship existing between customers and categories in our sample data.

Customer	Category
Mark	IT enthusiast
Robert	IT enthusiast
Paul	Rally driver
Robert	Rally driver
Luke	Traveler
Mark	Traveler
Paul	Traveler
Robert	Traveler

Table 2 – Customers-categories relationship

Now, to give an answer to the first question (What is the salary/income for the "IT enthusiast" category?) we need an additional clarification.

- If we consider the accounts owned by only one person, then there are no customers belonging to the "IT enthusiast" category who get a salary income.
- If we consider joint accounts (e.g. Mark and Robert both own the same account), then their owners receive a salary income even if we do not know who is really earning money.

From Mark's perspective, he receives a salary income of 1000. On the other side, Robert gets a salary income of 1000 too! However, unfortunately for them, from the perspective of "IT enthusiast" category we cannot count the same salary income two times, so the "IT enthusiast" salary income is still 1000 and not 2000. The tough reality is that Mark and Robert have to share this single salary income, because we have no other way to know which of them is really receiving this income, because we recorded the transaction against their joint account.

This problem is very common in a bank environment: one of the possible SQL query solutions presented below demonstrates the difficulty of tackling this kind of problem using a generic query builder (see the subquery in the WHERE condition of the main SQL query).

```
SELECT SUM( ft.Amount ) AS Amount
FROM Fact_Transaction ft
INNER JOIN Dim_Type dt
   ON dt.ID_Type = ft.ID_Type
  AND dt.Type = 'Salary'
WHERE ID_Account IN (
    SELECT ID_Account
    FROM Factless_CustomerCategory fcc
    INNER JOIN Dim_Category dc
       ON dc.ID_Category = fcc.ID_Category
    INNER JOIN Factless_AccountCustomer ac
       ON ac.ID_Customer = fcc.ID_Customer
    WHERE CategoryName  = 'IT enthusiast'
)
```

For this reason, we would like to resolve similar questions with a pivot table.

Now let us consider the second question: How many different transaction types are used by the "Rally driver" category?

There are two customers belonging to the "Rally driver" category: Paul and Robert. These two customers own four accounts, which in our fact table get any transaction type other than "ATM withdrawal".

Therefore, the answer will be three transaction types: Cash deposit (for an amount of 3000), Salary (1000) and Credit card statement (-600.00). The SQL query construct could be very similar to the previous one.

```
SELECT COUNT( DISTINCT ft.ID_Type ) AS TransactionTypes
FROM Fact_Transaction ft
WHERE ID_Account IN (
SELECT ID_Account
FROM Factless_CustomerCategory fcc
INNER JOIN Dim_Category dc
   ON dc.ID_Category = fcc.ID_Category
INNER JOIN Factless_AccountCustomer ac
   ON ac.ID_Customer = fcc.ID_Customer
WHERE CategoryName  = 'Rally driver'
)
```

The third question (What customer categories have ATM withdrawal transactions?) requires a different approach: starting from a set of transactions (filtered by type) we need to get related customers and then related categories. In such a case a query builder could give us a working query, but it should be noted how potentially slow the query could be, because it could generate a large set of rows before applying the DISTINCT clause.

```
SELECT DISTINCT dc.CategoryName
FROM Fact_Transaction ft
INNER JOIN Dim_Type dt
   ON dt.ID_Type = ft.ID_Type
  AND dt.Type = 'ATM withdrawal'
INNER JOIN Factless_AccountCustomer fac
   ON fac.ID_Account = ft.ID_Account
INNER JOIN Factless_CustomerCategory fcc
   ON fcc.ID_Customer = fac.ID_Customer
INNER JOIN Dim_Category dc
   ON dc.ID_Category = fcc.ID_Category
```

We could optimize the SQL query but in a way that is difficult to obtain with a generic query builder. Even in this case, a working pivot table would be a dream that becomes reality for an end user.

Now we have enough requirements to design and test a multidimensional model that enables a pivot table to solve this kind of problems with a few clicks.

Implementation

Figure 12 shows the relational schema of our model: we have two bridge tables (or factless fact tables) that join two "cascading" many-to-many relationships, the first one between Dim Account and Dim Customer and the second one between Dim Customer and Dim Category.

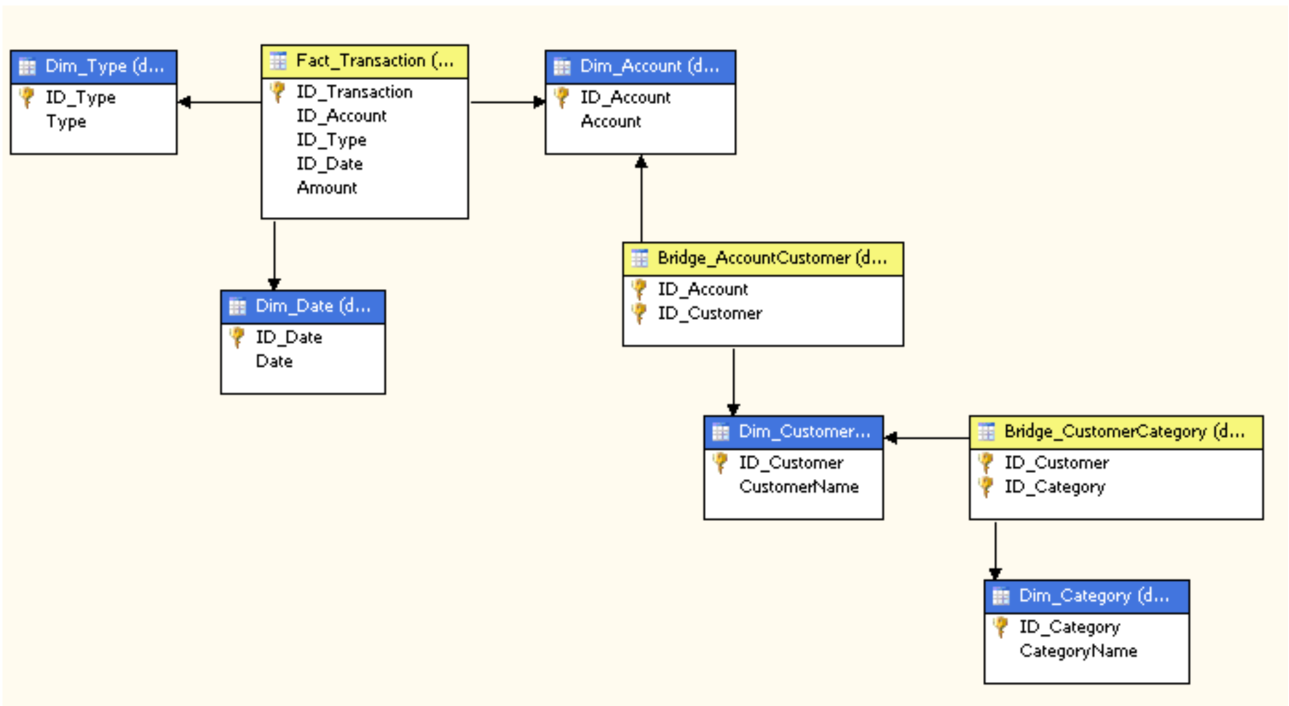


Figure 12 – Relational model with cascading many-to-many relationships

If we create the cube with the auto build feature of Cube Wizard, we end up with a model that correctly identifies dimension and fact tables. However, the problem of missing relationships between dimensions and measure groups we have already seen in the previous scenario is amplified here, as we can see in Figure 13. The wizard is not able to find cascading many-to-many relationships. A reason for this behavior is that defining all the many-to-many relationships could negatively affect performance.

Cube Structure | Dimension Usage | Calculations | KPIs | Actions | Partitions | Perspectives | T

Dimensions	Fact Transaction	Bridge Account Customer	Bridge Customer Category
Dim Type	Dim Type		
Dim Date	ID_Date		
Dim Account	Dim Account	Dim Account	Bridge Account Customer
Dim Customer	Bridge Account Customer	Dim Customer	Dim Customer
Dim Category			Dim Category

Figure 13 – Dimension relationship obtained by cube wizard/auto build feature

Unfortunately, the many gray boxes that are present in Figure 13 will produce meaningless results when we will query the dimension and measure group at corresponding coordinates. For example, as shown in Figure 14, we cannot see the amount of transactions for each customer category. Things are worse when we try to split the Amount measure by transaction type (see Figure 15).

	A	B
1	**Amount**	
2	**Category Name**	**Total**
3	IT enthusiast	5,000
4	Rally driver	5,000
5	Traveler	5,000
6	**Grand Total**	**5,000**

Figure 14 – Categories are not related to amount measure

	A	B	C	D	E	F
1	**Amount**	**Type**				
2	**Category Name**	**ATM withdrawal**	**Cash deposit**	**Credit card statement**	**Salary**	**Grand Total**
3	IT enthusiast	-400	4,000	-600	2,000	5,000
4	Rally driver	-400	4,000	-600	2,000	5,000
5	Traveler	-400	4,000	-600	2,000	5,000
6	**Grand Total**	**-400**	**4,000**	**-600**	**2,000**	**5,000**

Figure 15 – Categories still do not split amount measure

At this point, the problem seems to be the missing relationship between Dim Category and the Fact Transaction measure group. To define it, we can click on the button in the gray box of the Define Relationship dialog box, and then select the only available intermediate measure group once we have chosen the Many-to-Many relationship type (Figure 16 better summarize this selection).

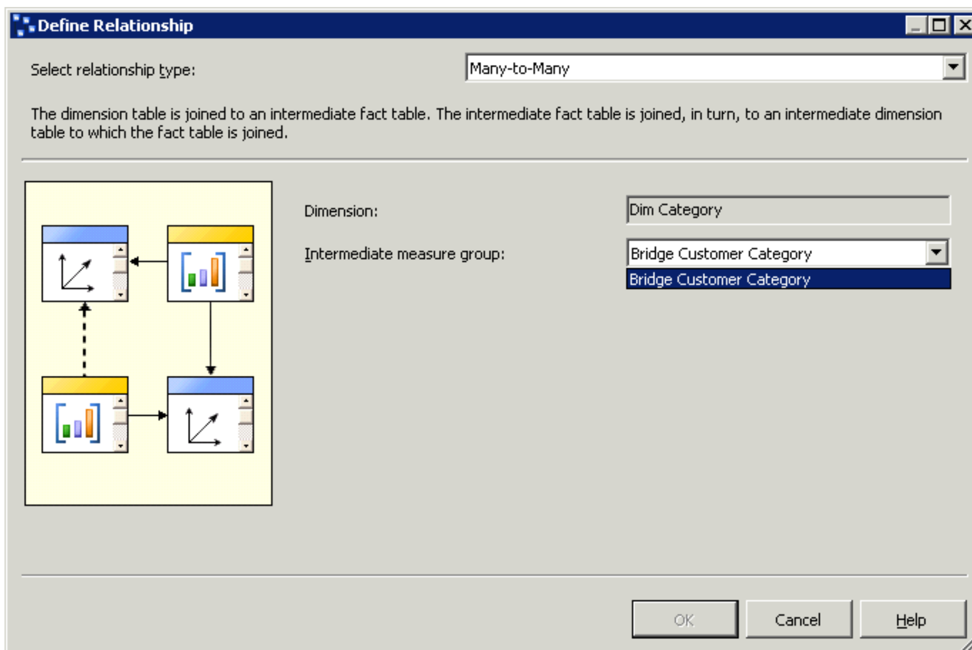


Figure 16 – Intermediate measure groups available for Dim Category

Now we can reprocess the cube, but the results will be the same and wrong as Figure 14 and Figure 15 show. Before claiming it is a bug of Analysis Services (it is not), look at the new dimension relationship summary in Figure 17. There are still a lot of gray boxes and the intermediate measure group between Dim Category and Fact Transaction is not the same as the one between Dim Customer and Fact Transaction (one is Bridge Customer Category and the other is Bridge Account Customer).

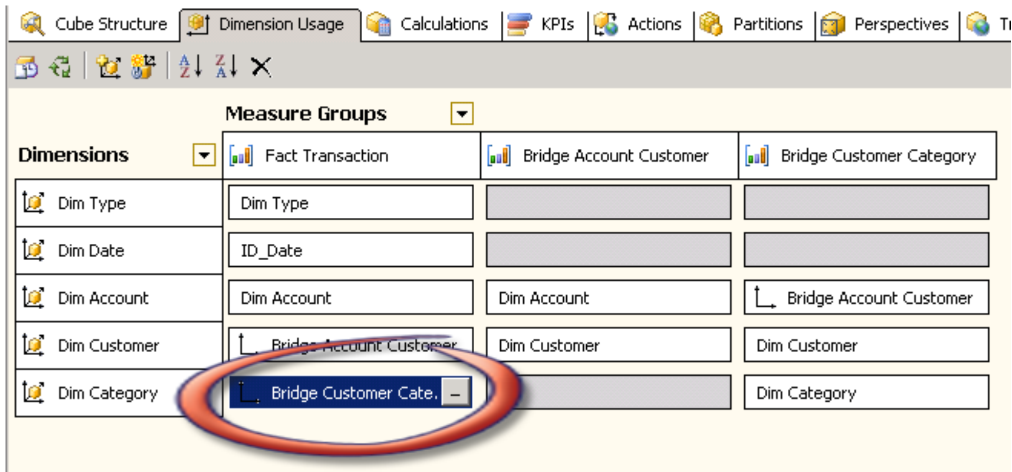


Figure 17 – Dimension relationship after Dim Category manual definition

To understand what is happening and why, you need to realize that Analysis Services entities, like dimensions and measure groups, are totally separated and disconnected from the underlying relational schema. Subsequently, Analysis Services has not enough information to relate correctly customer categories with account transactions. We told Analysis Services that a category is related to account transactions through the Bridge Customer Category measure group, but to go from a category to a transaction we need to get all the customers for that category (Bridge Customer Category) and then all the accounts for this set of customers (through Bridge Account Customer). Now the problem should be clear: we have not informed Analysis Services about the relationship between Dim Category and Bridge Account Customer. For this reason, it is still a gray box. We can fill this void by clicking on the … button: this time our dialog box shows up two possible intermediate measure groups (Figure 18).

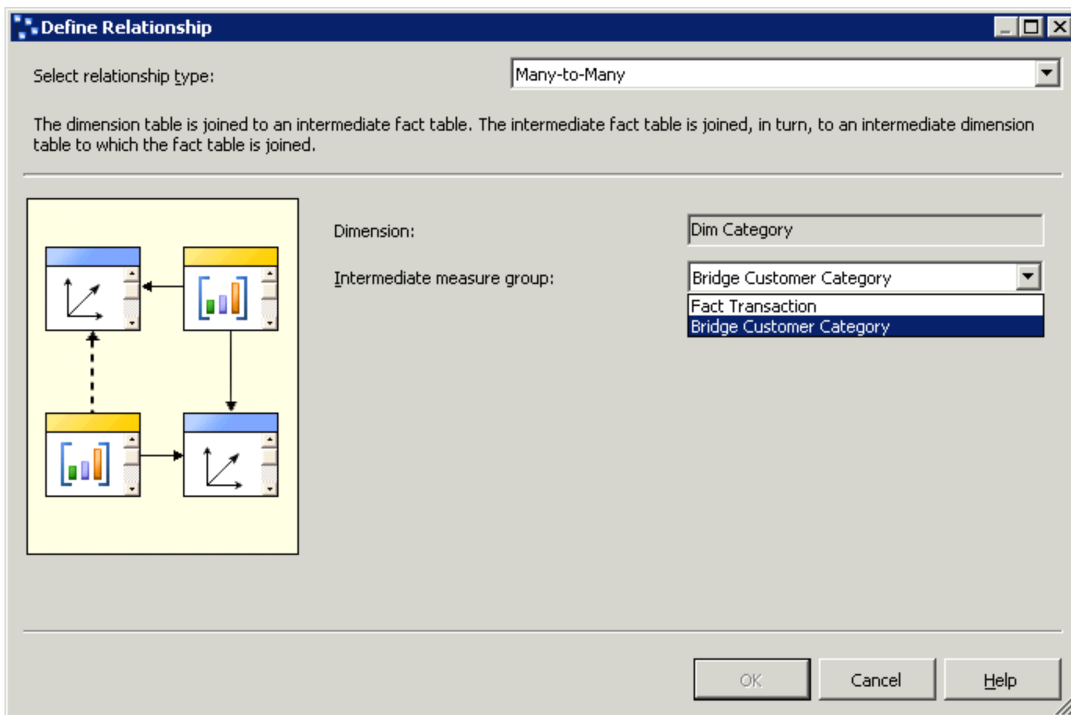


Figure 18 – Difficult choice for Dim Category intermediate measure group

We need to choose Bridge Customer Category as the intermediate measure group, because this is the only possible bridge fact table that we traverse walking from Dim Customer to Bridge Account Customer into

the relational schema (Figure 12). However, why does the “Intermediate measure group” dropdown include the Fact Transaction as a possible intermediate measure group? Simply because we previously defined a (wrong) relationship between Dim Category and Fact Transaction using Bridge Customer Category as the intermediate measure group (review Figure 16). If we would return to the stage immediately after the Cube Wizard, we would have seen only one choice (the right one) defining a many-to-many relationship between Dim Category and Bridge Account Customer.

At this point, we still need to correct the relationship between Dim Category and Fact Transaction: it has to be Bridge Account Customer instead of Bridge Customer Category that we chose previously. Now, if we redefine this relationship, the dropdown lists both choices, because Dim Category has many relationships with other measure groups. The resulting dimension usage schema is summarized in Figure 19.

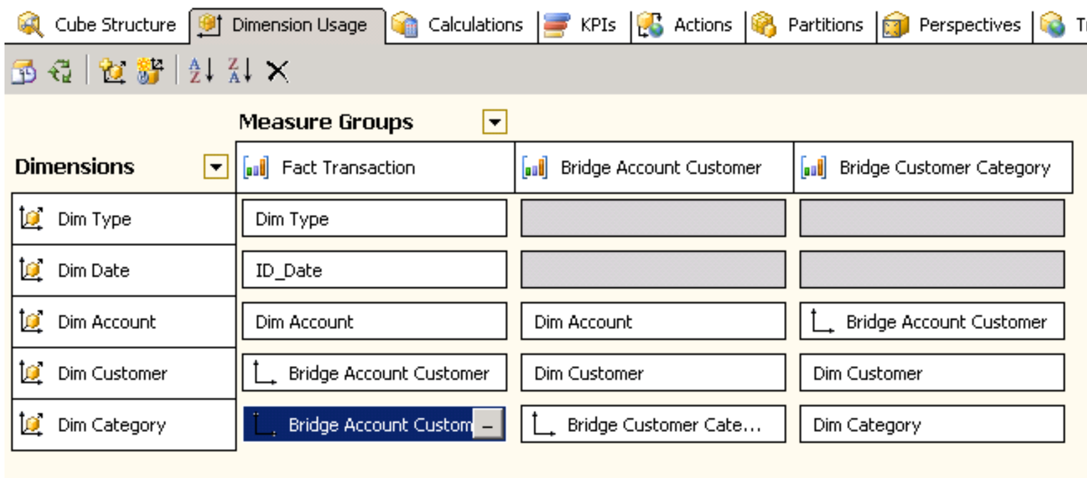


Figure 19 – Correct Dim Category many-to-many relationship assignments

To verify that this is correct, we can retry the queries that failed in Figure 14 and Figure 15. This time we get the correct numbers, as we can see in Figure 20 and Figure 21.

	A	B
1	**Amount**	
2	**Category Name**	**Total**
3	IT enthusiast	3,500
4	Rally driver	3,400
5	Traveler	5,000
6	**Grand Total**	**5,000**

Figure 20 – Categories are correctly related to amount measure

	A	B	C	D	E	F
1	**Amount**	**Type**				
2	**Category Name**	**ATM withdrawal**	**Cash deposit**	**Credit card statement**	**Salary**	**Grand Total**
3	IT enthusiast	-200	3,000	-300	1,000	3,500
4	Rally driver		3,000	-600	1,000	3,400
5	Traveler	-400	4,000	-600	2,000	5,000
6	**Grand Total**	**-400**	**4,000**	**-600**	**2,000**	**5,000**

Figure 21 – Categories correctly split amount measure

Now, prepare a cup of your favorite coffee and remember well what you are learning here: it will save you a lot of time when your favorite cube gets several cascading many-to-many relationships. The concept of

cascading many-to-many relationships is a fundamental one on which we will build the rest of the models described in this book.

We use the relationship between a dimension and a measure group to tell Analysis Services how to relate dimension members to fact measures. When the relationship is regular, it is simple. When the relationship is many-to-many**, the intermediate measure group must refer to a measure group that contains a valid relationship with a dimension that relates via a regular relationship to the target measure group.** This should explain why choices were good or bad in our previous examples. In this last example, the Bridge Account Customer measure group had to be used to relate Dim Category to the Fact Transaction measure group. This latter measure group is the only one that has a dimension (i.e. Account) that relates directly to the Fact Transaction measure group.

Official documentation explains this concept in terms of granularity, which is formally correct but much less intuitive. In other words, when you define a many-to-many relationship between a measure group and a dimension, you have to choose the intermediate measure group (bridge table) that is nearest to the measure group, considering all the possible measure groups that you can cross going from the measure group to the considered dimension.

We think that the Define Relationship dialog could be both clearer and smarter and maybe it will be in the future. For example, it could filter out the choices that are probably wrong. Unfortunately, in several release cycles of Analysis Services Microsoft has not prioritized such a feature.

We should still check if we meet all other business requirements:

- Figure 22 shows the right answers to the second question (How many different transaction types are used by the “Rally driver” category?).

- Figure 23 answers correctly to the third question (What categories of customers have ATM withdrawal transactions?).

Note that, in figure 37, the Grand Total row is not the sum of previous rows and that it is coherent with the nature of the many-to-many relationship.

	A	B
1	Category Name	Rally driver
2		
3	**Amount**	
4	**Type**	**Total**
5	Cash deposit	3,000
6	Credit card statement	-600
7	Salary	1,000
8	**Grand Total**	**3,400**

Figure 22 – Transaction types for Rally driver category

	A	B
1	Type	ATM withdrawal
2		
3	**Amount**	
4	**Category Name**	**Total**
5	IT enthusiast	-200
6	Traveler	-400
7	**Grand Total**	**-400**

Figure 23 – Category of customers who did ATM withdrawals

At this point, we should determine if the remaining gray boxes could still lead to issues with other queries. In fact, if we are interested in the count measure produced by the bridge table, they definitely do. For example, if for whatever reason you would choose to address the third question using the Bridge Customer Category Count measure instead of the Amount measure (it is not such a useful number, but you should not ask whether a number is useful while it is wrong), you would obtain the strange result of Figure 24.

	A	B
1	Type	ATM withdrawal
2		
3	**Bridge Account Customer Count**	
4	**Category Name**	**Total**
5	IT enthusiast	5
6	Rally driver	4
7	Traveler	8
8	**Grand Total**	**8**

Figure 24 – Wrong results using Factless Customer Category Count measure

Numbers aside (in this query the measure should represent the number of customers for each category that made at least one ATM withdrawal transaction, but it does not), the category list is wrong. The reason should be obvious: there are no valid relationships between Dim Type and Bridge Customer Category measure group, which contains the measure we used in our query. At this point, we must choose between making this measure invisible or fixing this measure. The second approach is better if, in the future, we might expand the UDM: more defined relationships will make the cube easier to explain. However, the first approach is easier to maintain and could result in faster queries, but you should remember to hide such a meaningless measure from the end user. For the sake of completeness for this section, we will proceed by completing the dimension relationship schema, removing all the "no relationship" gray cells.

Cube Structure | Dimension Usage | Calculations | KPIs | Actions | Partitions | Perspectives | T

Dimensions	Fact Transaction	Bridge Account Customer	Bridge Customer Category
Dim Type	Dim Type	Fact Transaction	Bridge Account Customer
Dim Date	ID_Date	Fact Transaction	Bridge Account Customer
Dim Account	Dim Account	Dim Account	Bridge Account Customer
Dim Customer	Bridge Account Customer	Dim Customer	Dim Customer
Dim Category	Bridge Account Customer	Bridge Customer Cate...	Dim Category

Figure 25 – Complete cube model for cascading many-to-many relationships

In Figure 25 we finalized the UDM dimension usage by defining relationships between all dimensions and all measure groups. Oftentimes, all the many-to-many relationships (all the cells) of a dimension usage column point to the same intermediate measure group. This is common because only the measure groups based on a true bridge fact table have different intermediate measure groups for different dimensions, e.g. the Bridge Account Customer measure group.

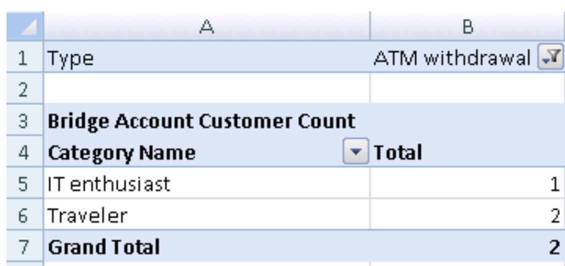
We worked on complex UDMs that have different intermediate measure groups for different dimensions linked with many-to-many relationships. Sometimes, it happens even for "standard" measure groups containing real fact measures (and not only a Count measure as in the case of a bridge table). Once you

understand how to choose the correct intermediate measure group for a dimension, you should be able to handle similar situations.

As we already pointed out, removing all the "gray cells" in the dimension usage matrix is not necessarily a "best practice" that you should follow in all cases. Maintaining all these relationships in an evolving cube (it is normal to add dimensions and measure groups over time in real life) could be extremely difficult and error-prone. Do it only when it is necessary. Even in this paper, there are scenarios that do not require a complete dimension usage matrix.

A simple rule of thumb: if we want to make visible any measure derived by an intermediate measure group (corresponding to a bridge table), we will have to define dimensions relationships for all intermediate measure groups that are traversed in order to connect the measure to other interesting dimensions, even if the visible measure is only a row count (the only measure you should get from a real bridge table).

Now we can get the right answer for the third question (What customer categories have ATM withdrawal transactions?) even with the Bridge Customer Category Count measure, as we can see in Figure 26.



	A	B
1	Type	ATM withdrawal
2		
3	**Bridge Account Customer Count**	
4	**Category Name**	**Total**
5	IT enthusiast	1
6	Traveler	2
7	**Grand Total**	**2**

Figure 26 – Right results using Factless Customer Category Count measure

Once you have mastered cascading many-to-many relationships, you will definitely have gained the ability to create richer multidimensional models, such as the ones that follow.

Survey

The survey scenario is a common example of a more general case where we have many attributes associated with a case (one customer, one product, and so on). We want to normalize the model because we do not want to change the UDM each time we add a new attribute to data (e.g. adding a new dimension or changing an existing one).

The common scenario is a questionnaire consisting of questions that have predefined answers with both simple and multiple choices. The naive solution is to define a fact table and three dimensions:

- Dim Questions with the questions.
- Dim Answers for the answers provided by customers
- Dim Customer for the customer who answered a specific question

The fact table will contain a value indicating the exact answer from the customer, in the case of multiple choices.

Because we do not need to analyze questions without answers, a better solution is to have only one table for both questions and answers. This will reduce the number of dimensions without having any influence on the expressivity of the model and will make the complete solution simpler to both navigate and create.

The star schema model (one fact table with answers joined with a questions/answers dimension and a case dimension) is fully queryable using SQL.

However, once we move to UDM things become harder: while it is very simple to compare different answers to the same question, it could be very difficult to correlate frequency counts of answers to more than one question. For example, if we have a question asking for sports practiced (multiple choices) and another one asking for job performed, probably we would like to know what pattern of statistical relationships – if any – exist between the two corresponding sets of answers.

The normal way to model it is having two different attributes (or dimensions) that users can combine on rows and columns of a pivot table. Unfortunately, having an attribute for each question is not very flexible. Moreover, we will have to change the star schema to accommodate having a single row in the fact table for each case. This makes it very difficult to handle any multiple-choice question.

Instead, we can change our perspective and leverage many-to-many relationships. We can build a finite number (as many as we want) of questions/answers dimensions, duplicating many times the original one and providing the user with a number of "filter" dimensions that can be crossed into a pivot table or can be used to filter data that, for each case, satisfy defined conditions for different questions.

This is the first time that we are duplicating data from the relational model in order to accommodate the needs of UDM, We shall see in subsequent chapters that the same technique will be useful quite often, every time we will need to make a table behave like both a dimension and bridge table. Using views, we can duplicate tables as many times as we need without having to worry about space optimization.

Remember that the survey scenario is usable in many similar circumstances: classification of product characteristics (for instance "tagging") and basket analysis are just two among many examples of applications of this technique.

BUSINESS SCENARIO

Let us explore the survey scenario in more detail. Data was loaded into the star schema shown in Figure 27. Dim_QuestionsAnswers contains both questions and answers. We could have defined two independent dimensions (resulting in a snowflake schema) but it is a choice we do not recommend for two reasons: the first is the maintenance cost to update surrogate keys, the second is that there is no reason to query questions without answers (typically, you will make visible only a hierarchy Question-Answer on the UDM).

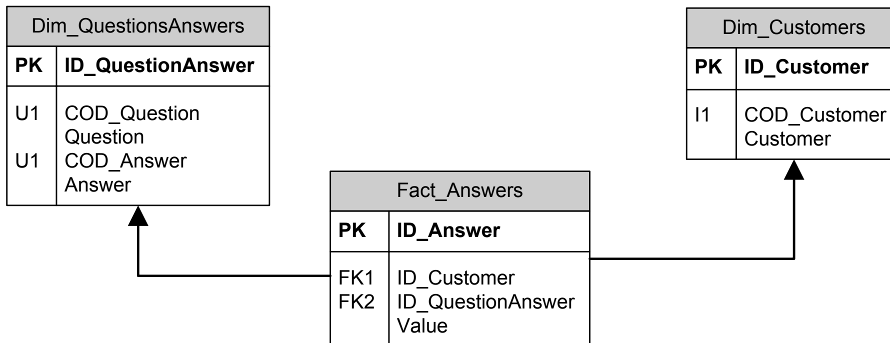


Figure 27 – Relational Survey star schema

Our users want to query this model in a PivotTable and want to avoid writing even a single row of MDX. A typical query could be "How many customers play both soccer and hockey?" We can calculate it using an SQL solution with COUNT(*) expression, while a more correct one could be COUNT(DISTINCT ID_Customer) in a more general case (useful if you add more complex filter conditions).

```
SELECT COUNT (*)
FROM Fact_Answers a1
INNER JOIN Dim_QuestionsAnswers q1
   ON q1.ID_QuestionAnswer = a1.ID_QuestionAnswer
INNER JOIN Fact_Answers a2
   ON a2.ID_Customer = a1.ID_Customer
INNER JOIN Dim_QuestionsAnswers q2
   ON q2.ID_QuestionAnswer = a2.ID_QuestionAnswer
WHERE q1.Answer = 'Soccer'
  AND q2.Answer = 'Hockey'
```

Adding more conditions would require new INNER JOINs (two for each condition) to the query. For this and other reasons, it would be very difficult to get a parameterized query that automatically solves this problem.

Moreover, we want to be able to change surveys in the future, keeping them compatible with existing data and queries (at least for identical questions that use the same answers). One day we could add more questions and answers, without requiring a cube or dimension full process, allowing incremental updates of any entity.

IMPLEMENTATION

To implement a cube based on the star schema shown in Figure 27 we define a single QuestionsAnswers dimension (see Figure 28). In this way, the user can filter rows of Fact_Answers table (or cells of the derived cube). However, we do not want to calculate the number of answers. Instead, we want to filter customers

that satisfy a given condition, then filter customers that satisfy another condition and, at the end, we need to get the intersection between these two sets of customers.

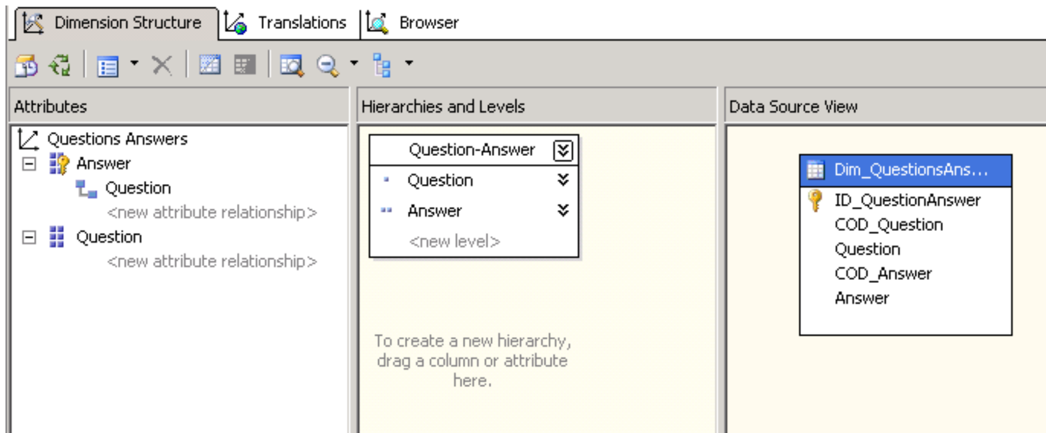


Figure 28 – Dimension QuestionsAnswers

We need to design a dimensional model that uses the same QuestionsAnswers dimension several times, allowing us to combine different answers and questions for the same customer. We will call the resulting dimensions "Filter 1", "Filter 2", and so on. To make an analogy, this approach is similar to defining aliases in a SQL query whenever you want to refer to the same table multiple times with different filter and/or join conditions.

Users will be able to select any combination of those dimensions and filter on them. This will result in a query that applies all the filters (logical AND). However, the AND condition will be applied only to those fact rows that belong to the same customer. Note that we seek to evaluate the number of customers who have specific characteristics based on the survey: In this case, our main fact table, in the cube, is not the Fact_Answers fact table, but Dim_Customers itself!

To model our cube, we need to relate each customer to all answers for that customer, as we would de-normalize the Fact_Answers fact table to have a column for each QuestionsAnswers member. From a practical point of view, there is a many-to-many relationship between Customers and each QuestionsAnswers dimensions (renamed to "Filter n") we added to the cube. In order to do that, we use the Fact_Answers fact table as the bridge fact table of a many-to-many relationship, and we use the Dim_Customers dimension table as a fact table (to get the customers count).

Each "Filter" dimension will use the same physical bridge table to reference the QuestionsAnswers dimension. It is convenient to define a logical view (named query) into the Data Source View (DSV) to create N different measures groups in the cube (each one has to be related to a different table)[1]. Here, N is the number of "Filter" dimensions we have chosen.

The bridge table is repeated in the DSV defining several named queries with the same query inside. In this way, we can use the Cube Editor for this model: normally, Visual Studio editor would not allow you to create many different measure groups based on the same fact table, unless you define a Distinct Count measure. Alternatively, you could manually define different measure groups related to the same fact table by modifying the cube XML definition using a text editor.

11 The careful reader should scream and ask why we are using a named query instead of a database view for these filter dimensions. The answer is that these filter dimension belong to the completely private area of the cube. They represent a technical means needed to create the UDM model. As there is no reason to share this information with anybody else, a named query hidden in the project is a good place for Always remember that you should not take any single hint in this book as it is, you always have to think and, if you believe something can be done better in your case overriding our hints, you will be welcome to do it, as we normally do. Your brain will work much better than any set of predefined rules.

The source of the named query is very simple:

```
SELECT
    ID_Answer,
    ID_Customer,
    ID_QuestionAnswer,
    Value
FROM Fact_Answers
```

In the example, we will use three “Filter” dimensions. Therefore, we need three aliases for the Fact_Answers fact table. We defined a named query view for each one instead of using the real fact table. Figure 29 shows the resulting DSV.

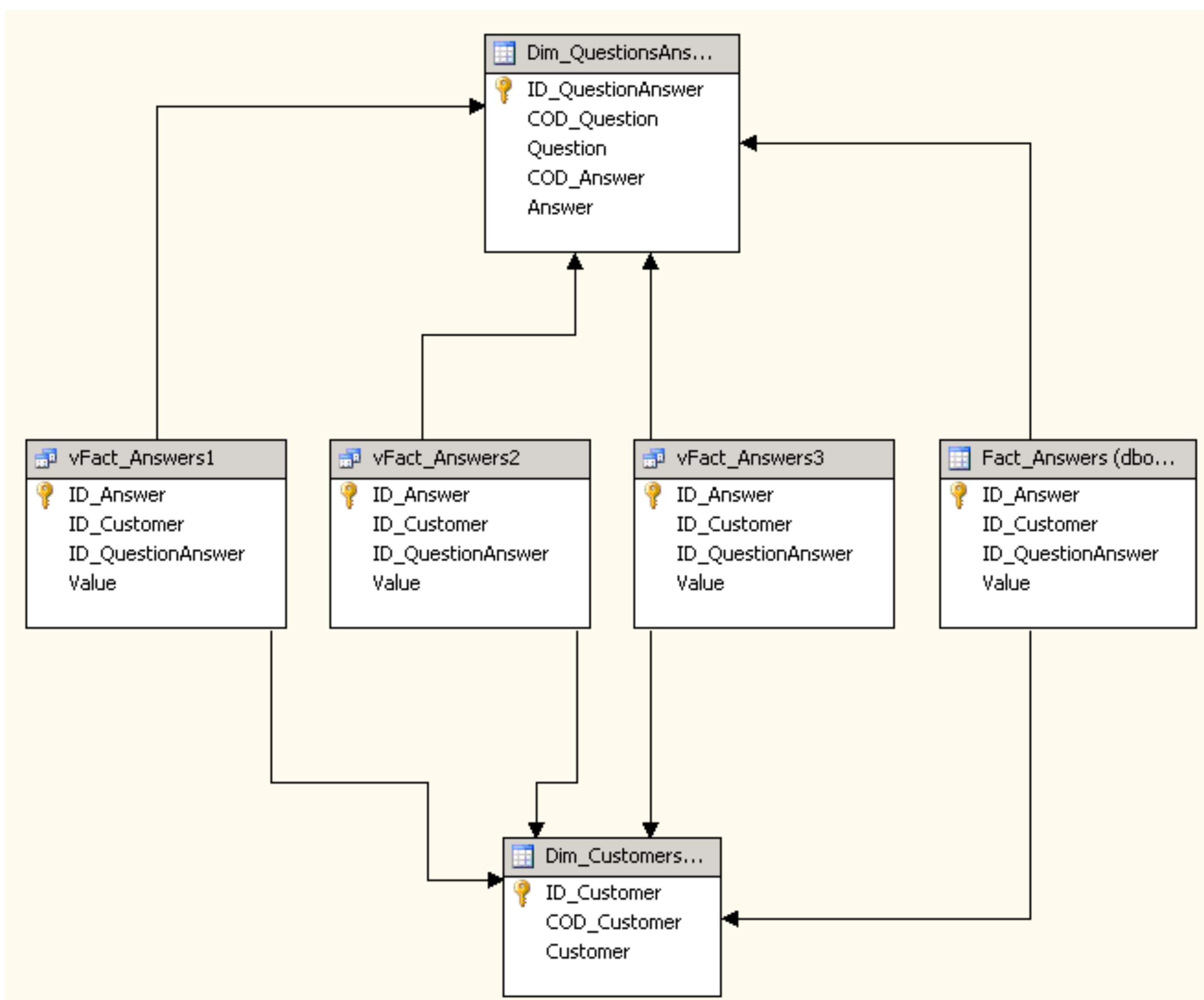


Figure 29 – Survey model Data Source View

We can use the Cube Wizard to start the cube modeling. After the first two steps (accept the defaults) we come to the Identify Fact and Dimension Tables step. We need to change the suggested selection as shown in Figure 30. We use Dim_Customers as Fact and Dimension and we excluded the Fact_Answers table (instead, we will use the named queries based on the vFact_Answers views).

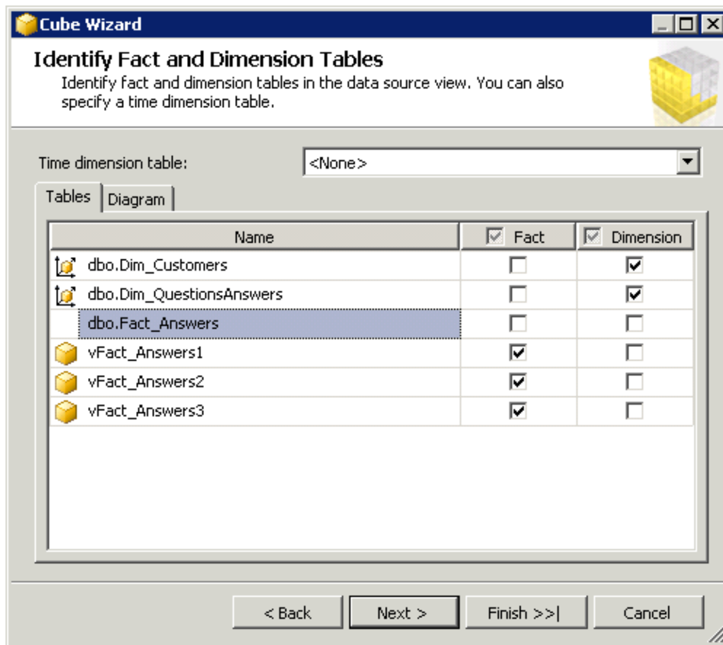


Figure 30 – Cube Wizard selection for Survey Cube

In the next step (Review Shared Dimensions), we choose all dimensions from the “available dimensions” list. In the Select Measures step that follows, we make a lifting to default measure names, as shown in Figure 31.

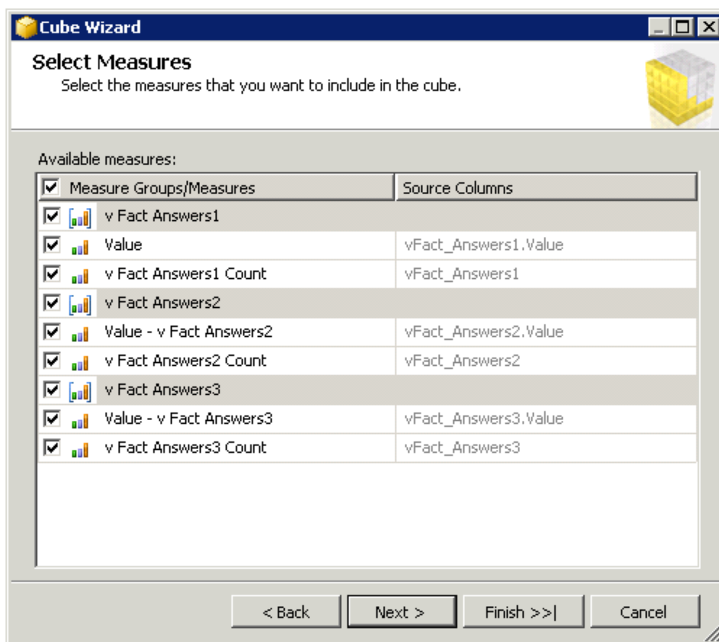


Figure 31 – Esthetic changes to measure names

We accept the defaults in the steps that follow and we name the cube Survey. Once we complete the Cube Wizard, the Cube Designer opens and shows the resulting cube structure (see Figure 32). Each AnswersN measure group will contain data needed to build three different Filter dimensions based on Questions Answers dimension.

We need to add “role-playing dimensions” to the cube to build the three Filter dimensions (shown in the Dimension pane in Figure 32).

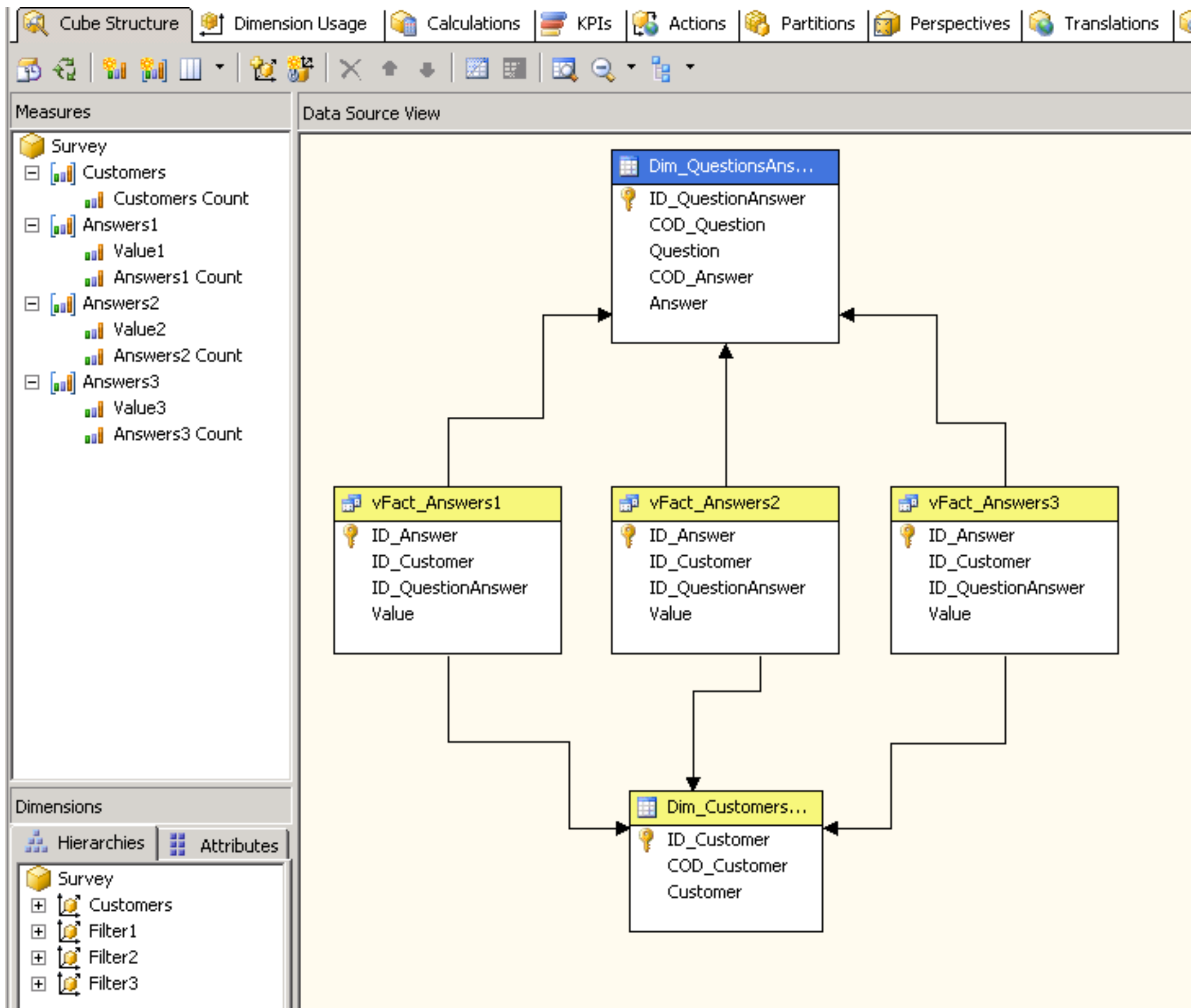


Figure 32 – Resulting Survey Cube Structure

To add a dimension we can use the Dimension Usage tab and click on the Add Cube Dimension button. We add the Questions Answers dimension three times and we rename them to “FilterN”, where N is a progressive number to distinguish the filter dimension (in this case ranging from one to three). We will rename the original Questions Answers dimension to Filter1.

As we learned in previous scenarios, we have to set up useful relationships between dimensions and measures groups. Figure 33 shows the relationships we need. If you consider the Customer Measure Group only, you realize that we have a fact dimension (Dim_Customers) related many times to Questions Answers (used three times as a role-playing dimension) through a different bridge fact table each time.

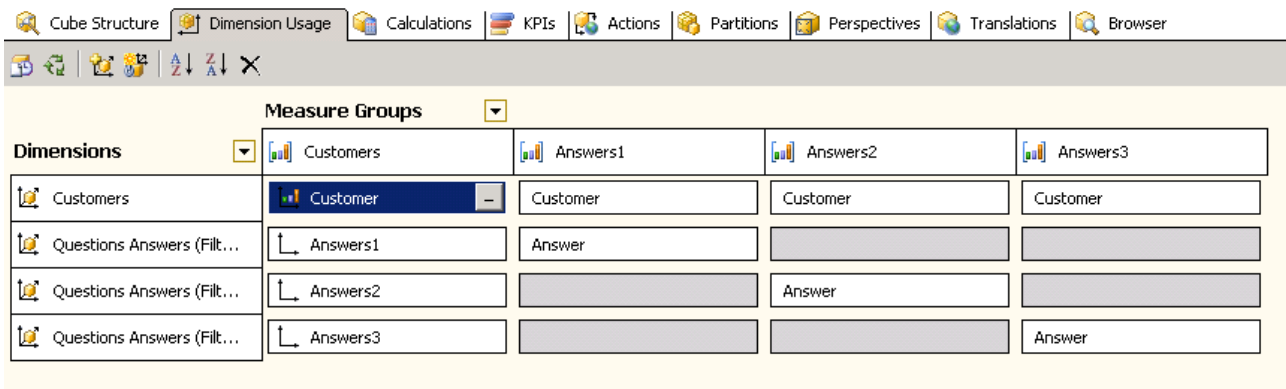


Figure 33 – Dimension Usage for Survey Cube

Before analyzing the results on a pivot table, look at sample data used in the test. We have four customers (Bill, Elisabeth, John and Mark) and a survey for each customer. Possible questions and answers of the survey are shown in Table 3.

Question	Answer
Sports	Tennis
Sports	Golf
Sports	Soccer
Sports	Hockey
Job	Employee
Job	Student
Job	Designer
Age	Age

Table 3 – Dim_QuestionsAnswers data

The survey data is visible in Table 4.

Customer	Question	Answer
Bill	Age	28
Bill	Job	Designer
Bill	Sports	Hockey
Bill	Sports	Soccer
Elisabeth	Age	31
Elisabeth	Job	Designer
Elisabeth	Sports	Golf
Elisabeth	Sports	Tennis
John	Age	29
John	Job	Student
John	Sports	Soccer
Mark	Age	30
Mark	Job	Employee
Mark	Sports	Golf
Mark	Sports	Soccer
Mark	Sports	Tennis

Table 4 – vFact_AnswersN data

Please note that only Bill plays both Soccer and Hockey. This will be useful for the next considerations. Now, we can process the cube and see if it works as expected.

In Figure 34, we put a Filter dimension on rows and another Filter dimension on columns and we selected only the answer Hockey for rows and only the answer Soccer for columns, because we wanted to limit the results to a specific case.

Answers1 Count	Column Labels		
	⊟ Sports	Sports Total	Grand Total
Row Labels	Soccer		
⊟ Sports	1	1	1
Hockey	1	1	1
Grand Total	1	1	1

Figure 34 – Query between members of the same dimension

We can also intersect more answers and questions into the same pivot table report. Figure 35 shows that many customers play two sports and what they are, what is the relationship between jobs and sports, and so on. There is a certain data redundancy because the data is mirrored diagonally from top-left to bottom-right. This kind of analysis is bidirectional and the order of answers provided by customers is unimportant.

Customers Count	Column Labels								
	⊟ Sports				⊟ Job			⊟ Age	Grand Total
Row Labels	Golf	Hockey	Soccer	Tennis	Designer	Employee	Student	Age	
⊟ Sports	2	1	3	2	2	1	1	4	4
Golf	2		1	2	1	1		2	2
Hockey		1	1		1			1	1
Soccer	1	1	3	1	1	1	1	3	3
Tennis	2		1	2	1	1		2	2
⊟ Job	2	1	3	2	2	1	1	4	4
Designer	1	1	1	1	2			2	2
Employee	1		1	1		1		1	1
Student			1				1	1	1
⊟ Age	2	1	3	2	2	1	1	4	4
Age	2	1	3	2	2	1	1	4	4
Grand Total	2	1	3	2	2	1	1	4	4

Figure 35 – Cross selection between members of the same dimension

The final touch is to query who the customers with specific characteristics are. In Figure 36, we double clicked on the intersection between column Golf and row Tennis to get a drill through and we get the people who play both golf and tennis. You can check in Table 4 that the result is correct.

	A	B
1	Data returned for Customers Count, Sports - Tennis, Sports - Golf (First 1000 rows).	
2		
3	[Customers].[$Customers.Customer]	[Customers].[Customers Count]
4	Mark	1
5	Elisabeth	1
6		
7		

Figure 36 – Drillthrough on Golf-Tennis cell

It is possible to use the Survey model for many scenarios that present similar challenges. For example, we could apply the same technique to alarms and/or diagnostics generated on items (customers, cars). Another scenario is the analysis of cross-sell opportunities. There are many data mining models to do that but, sometimes, a graphical output helps to visualize all of the relationships between specific items: the pivot table is the simplest way to obtain it.

Distinct Count

Distinct count measures are very useful and commonly required. Unfortunately, Analysis Services implementation of distinct count is very resource-intensive. The algorithm used to process a distinct count measure queries the source data using an ORDER BY clause. For this reason, a separate measure group is required for each distinct count measure (SSAS generates a query for each partition/measure group). This technique requires a long processing time and places strains on the source RDBMS when the cube is fully processed (assuming no incremental update). Moreover, SSAS has a relatively slow response time when the end user queries the distinct count measure due to the specific method of querying the distinct count measure.

Looking at data in a creative way, instead of using the UDM native distinct count support, we can build an alternative model based on many-to-many relationships that produces the same results but with potentially faster processing times and equivalent or even faster response times.

As established by the whitepaper

> http://www.microsoft.com/downloads/details.aspx?FamilyID=3494E712-C90B-4A4E-AD45-01009C15C665&displaylang=en

the performance of many-to-many is directly related to the row count in the intermediate measure group. So replacing a distinct count measure with a works best when you are dealing with low cardinality distinct counts (e.g. distinct employee count, rather than distinct web sessions count). Moreover, if you have several distinct count measures over the same fact table, leveraging the many-to-many distinct count technique will save you from having to process the multiple copies of that measure group.

The usage of many-to-many relationships is particularly advantageous when you want to build a distinct count on a slowly changing dimension (SCD) dimension.

Business scenario

Marketing analysis often requires distinct count measures for customers and products sold. These measures are important to evaluate averages as sales for distinct customer, sales for distinct product, and so on.

For simplicity, we define a relational schema with only two dimensions: Date and Customers. To describe better the changing set of attributes related to it, we created the Customers dimension as a slowly changing dimension (SCD). We show the relational model in Figure 37. For the sake of simplicity, dimensions here have only the essential attributes. A real model would have many more attributes that would justify the presence of a Type II SCD for Customers.

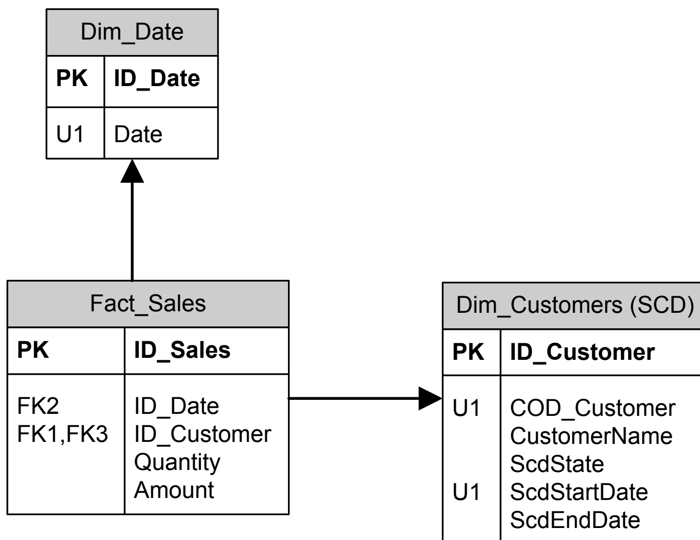


Figure 37 – Relational model with slowly changing dimension (SCD) Type II

As we have an SCD dimension, we need a distinct count of customers applied to the COD_Customer attribute and not to the ID_Customer surrogate key. We will analyze several possible implementations that provide the desired results, considering both performance and impact on the relational and multidimensional models.

Implementation

We would like to introduce a simpler model than the one based on the Customers SCD, because it is important to understand how a many-to-many relationship works when we use it to obtain a value equivalent to a distinct count measure.

In order to do that, we will consider the simpler relational model illustrated in Figure 38: Dim_Customers is a Type I SCD. You will notice that we removed all the detail not required for the example.

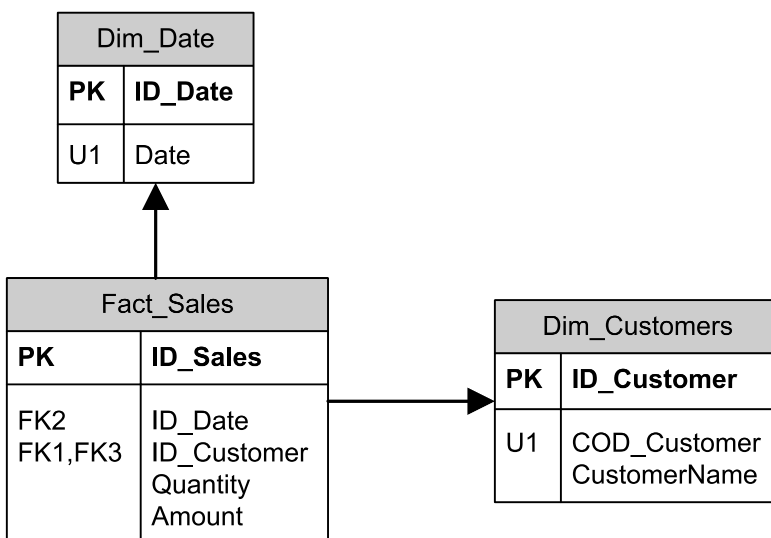


Figure 38 – Relational model without SCD (or SCD Type I)

We can easily build a cube with the two dimensions and standard measures (Sum of Quantity, Sum of Amount and Fact Sales Count). As you can see in Figure 39, we added a Year attribute to the Date dimension (calculated as YEAR(Date)) and a Distinct Count of ID_Customer (we called it Customers Distinct Count in the Distinct Customers measure group).

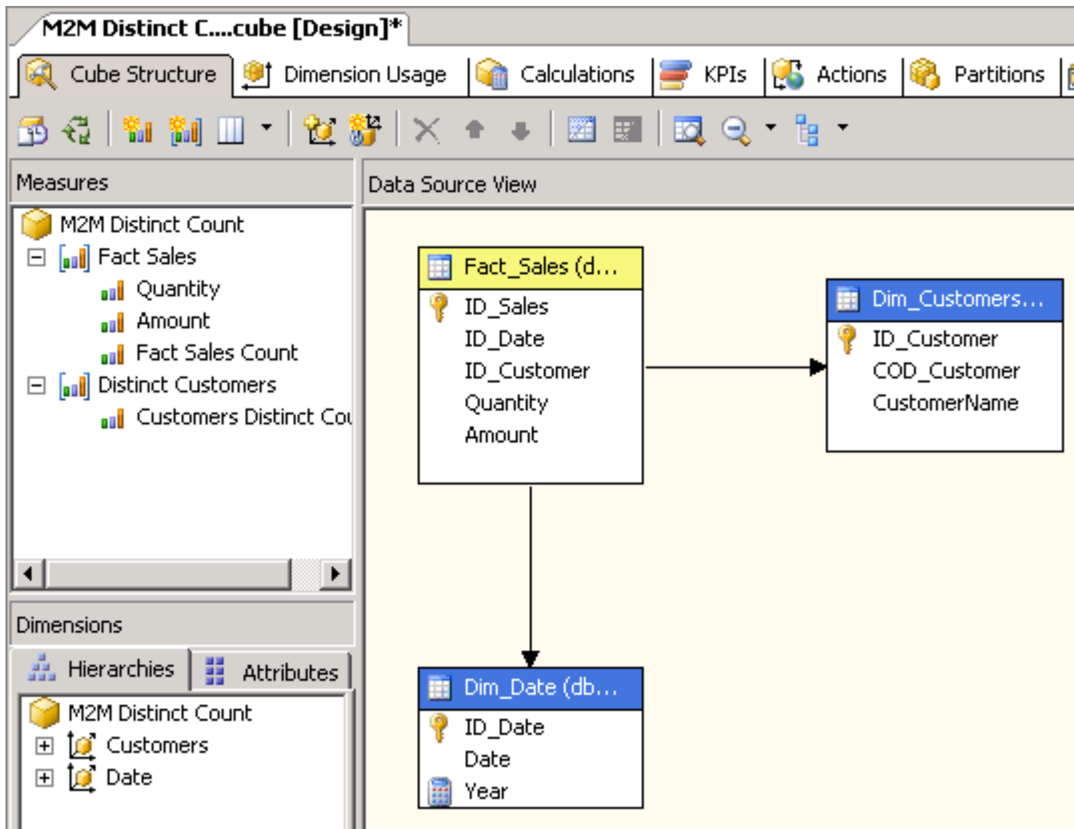


Figure 39 – Regular distinct count cube model

In Figure 39, you can look at sample data loaded into the data mart. Note that Dim_Customers has nine customers, numbered from Customer 1 to Customer 9.

Date	Customer	Quantity	Amount
01/01/2006	Customer 5	20	495.67
01/01/2006	Customer 5	3	6458.27
01/01/2006	Customer 6	7	7330.54
02/01/2006	Customer 3	28	2201.90
02/01/2006	Customer 5	25	911.05
06/01/2006	Customer 9	5	6342.61
07/01/2006	Customer 6	20	5437.42
10/01/2006	Customer 1	1	1084.56
10/01/2006	Customer 6	2	1000.29
10/01/2006	Customer 9	20	9319.23

Table 5 – Fact_Sales sample data

Figure 40 shows the pivot table results. We have only five distinct customers who made 10 sale transactions. The pivot table shows also the numbers at the day level (lowest grain) of the date dimension.

	A	B	C
1		Values	
2	Row Labels	Customers Distinct Count	Fact Sales Count
3	2006-01-01 00:00:00	2	3
4	2006-01-02 00:00:00	2	2
5	2006-01-06 00:00:00	1	1
6	2006-01-07 00:00:00	1	1
7	2006-01-10 00:00:00	3	3
8	**Grand Total**	**5**	**10**

Figure 40 – Regular distinct count results

Now, we can add a measure that counts the number of rows in Dim_Customers and then comparing the results. We configure the New Measure dialog box as shown in Figure 41.

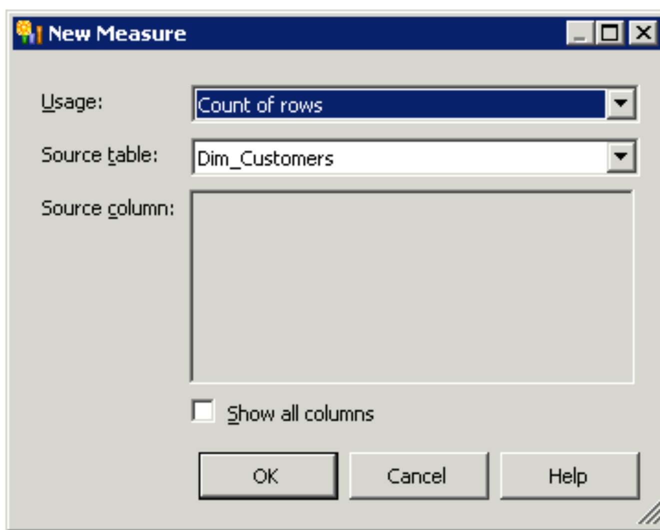


Figure 41 – New Measure based on Count of Rows of Dim_Customers

Figure 42 shows the updated cube structure after renaming the measure.

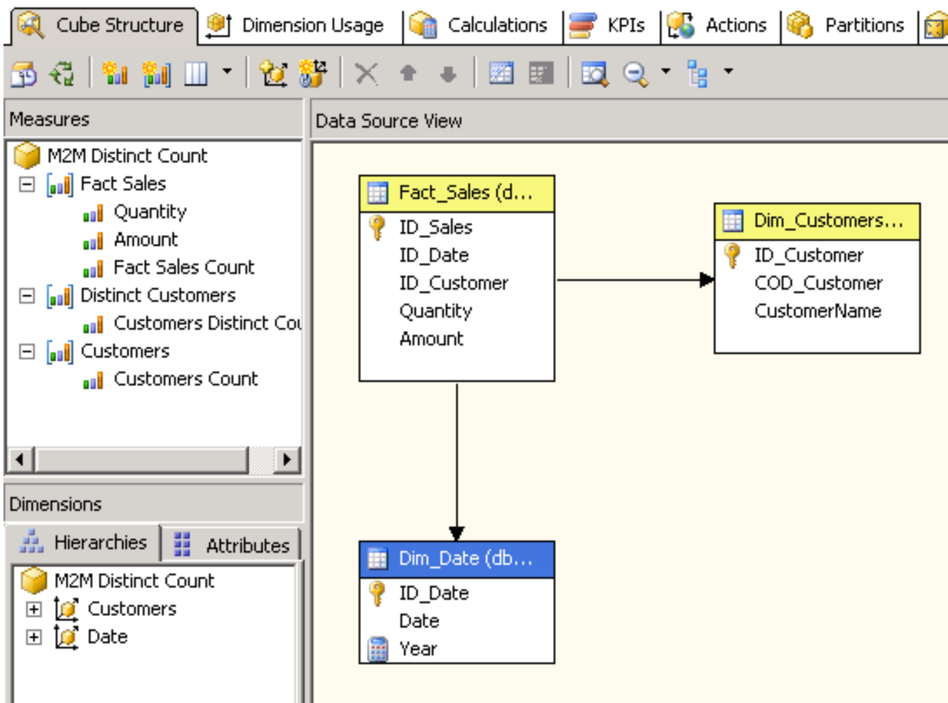


Figure 42 – Customers Count added to cube model

At this point, we need to define a relationship between the Customers measure group and the cube dimensions: if we did not, the report would show the total row count of all rows in Dim_Customers for any query we will do. To avoid this, we use the Dimension Usage dialog to set up a many-to-many relationship with the Date dimension using Fact Sales as the intermediate measure group (see Figure 43).

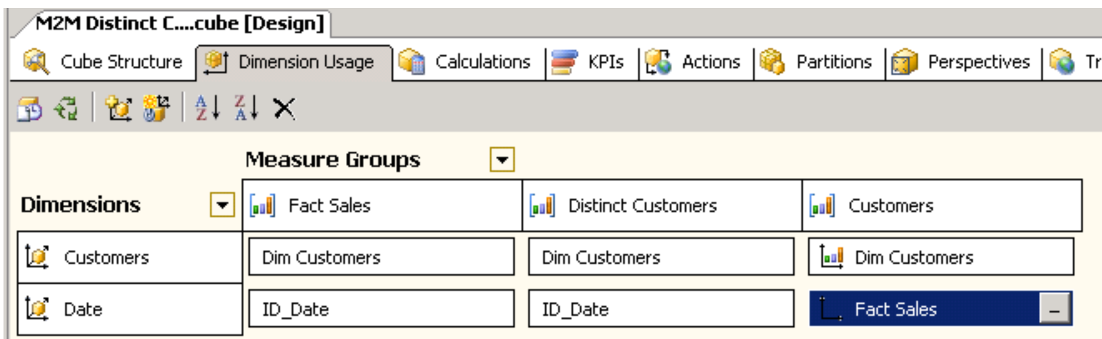


Figure 43 – Many-to-Many relationship between Customers and Date

Now, we can compare the Customer Count produced by the many-to-many relationship with the Customers Distinct Count obtained with the regular Distinct Count aggregate function. As Figure 44 shows, the numbers are the same regardless of the selected date, but the Grand Totals are different. The reason is that in absence of a Date selection there is no need to apply a filter on Customers based on the many-to-many relationship with the Date dimension. Therefore, we have a value of 5 for Customers Distinct Count and 9 for Customers Count.

	A	B	C	D
1		Values		
2	Row Labels	Customers Distinct Count	Fact Sales Count	Customers Count
3	2006-01-01 00:00:00	2	3	2
4	2006-01-02 00:00:00	2	2	2
5	2006-01-06 00:00:00	1	1	1
6	2006-01-07 00:00:00	1	1	1
7	2006-01-10 00:00:00	3	3	3
8	**Grand Total**	**5**	**10**	**9**

Figure 44 – Customers Count compared to Customers Distinct Count

You might think that the Customer Count column is useless because it is not consistent with the Customers Distinct Count measure. However, most of the time, a query includes a selection of an involved dimension. If we use the Year attribute instead of the Date attribute, we see the interesting data in Figure 45.

	A	B	C	D
1		Values		
2	Row Labels	Customers Distinct Count	Fact Sales Count	Customers Count
3	2006	5	10	5
4	**Grand Total**	**5**	**10**	**9**

Figure 45 – Use of Year attribute instead of Date attribute

The Year 2006 is exactly what we are interested in. If you consider that we usually need to count the customer only if he did at least one sale transaction overall (we assume that a customer is not a prospect), then it should be reasonable to expect that the Customers Count measure is in practice the same as the Customers Distinct Count measure.

	A	B	C
1	Data returned for 'Customers' ([Measures].[Fact Sales Count],[Date].[Year].&[2006]).		
2			
3	[$Customers].[COD Customer]	[$Customers].[Customer Name]	
4	C03	Customer 3	
5	C05	Customer 5	
6	C06	Customer 6	
7	C05	Customer 5	
8	C09	Customer 9	
9	C01	Customer 1	
10	C06	Customer 6	
11	C09	Customer 9	
12	C06	Customer 6	

Figure 46 – Customers drillthrough for standard distinct count measure

Some modelers might favor the use of many-to-many relationships to define a distinct count measure just for a simple feature you obtain as a side effect. If we define a drillthrough action (named Customers in our case) to get the list of customers behind a given cell, we will get the results shown in Figure 46 after drilling through the Customers Distinct Count measure for 2006. In comparison, Figure 47 shows the same drillthrough results for the Customers Count measure for 2006.

Here, we obtain the list of distinct customers while this is not the case with the Customers Distinct Count measure (see Figure 46 again). If you use a distinct count measure, consider a distinct filter on the drillthrough results to eliminate duplicated customers. This is not necessary with a many-to-many

relationship. The reason for this behavior is that the drillthrough action on the Customers Distinct Count measure will return the list of transactions made by those customers. Instead of this, the drillthrough action on the Customer Count measure will return the list of customers filtered from the bridge table of our many-to-many relationship.

	A	B	C
1	Data returned for 'Customers' ([Measures].[Customers Count],[Date].[Year].&[2006]).		
2			
3	[$Customers].[COD Customer]	[$Customers].[Customer Name]	
4	C01	Customer 1	
5	C03	Customer 3	
6	C05	Customer 5	
7	C06	Customer 6	
8	C09	Customer 9	

Figure 47 – Customers drillthrough for Customers Count (obtained by many-to-many relationship)

We are ready to introduce the slowly changing dimension in this scenario. When evaluating the distinct count of customers in a Type II SCD who have made a transaction, we cannot rely on the distinct count of the customer surrogate key in the fact dimension.

Three feasible solutions are as follows:

- **Solution A**. Create a unique customer dimension: this means duplicating the customer dimension, at least for the most recent version of each member
- **Solution B**. Create a Distinct Count measure on the application key of the customer dimension: the measure is defined into a measure group that has similar relationship to the one we just used to evaluate the customer count measure through a many-to-many relationship
- **Solution C**. Define a solution that is similar to solution B, substituting the distinct count measure with another count measure derived from a many-to-many relationship

Each one of these solutions has its positive and negative aspects. To test all of these cases, we need to modify our data. Table 6 shows that Customer 6 has two versions (it changed on 05/01/2006). For this reason, we have still 9 customers but 10 different rows in Dim_Customers, and we have 5 different customers who made transactions but 6 different customer surrogate keys referenced in the fact table.

Date	Customer	Quantity	Amount
01/01/2006	Customer 5	20	495.67
01/01/2006	Customer 5	3	6458.27
01/01/2006	Customer 6 v1	7	7330.54
02/01/2006	Customer 3	28	2201.90
02/01/2006	Customer 5	25	911.05
06/01/2006	Customer 9	5	6342.61
07/01/2006	Customer 6 v2	20	5437.42
10/01/2006	Customer 1	1	1084.56
10/01/2006	Customer 6 v2	2	1000.29
10/01/2006	Customer 9	20	9319.23

Table 6 – Fact_Sales SCD sample data

Figure 48 shows the new Data Source View. It uses additional views that simulate what we could have achieved by modifying the relational schema of our Data Mart. When to use views against materialized tables is another topic by itself, which we have to evaluate considering the processing time, the number of distinct count measures and the complexity of existing ETL processes (we should modify them if we change the data mart schema). The view vFact_Sales_Unique adds the COD_Customer at the fact table level, which is necessary to implement Solution A. Solution B does not need any new elements. To implement Solution C we have to add two views: vDim_CustomersUnique simulates a customer dimension containing only a unique row for customers (without changing attributes); vCustomersScd simulates a bridge table that joins each unique customer member (vDim_CustomersUnique) with its versions (Dim_Customers).

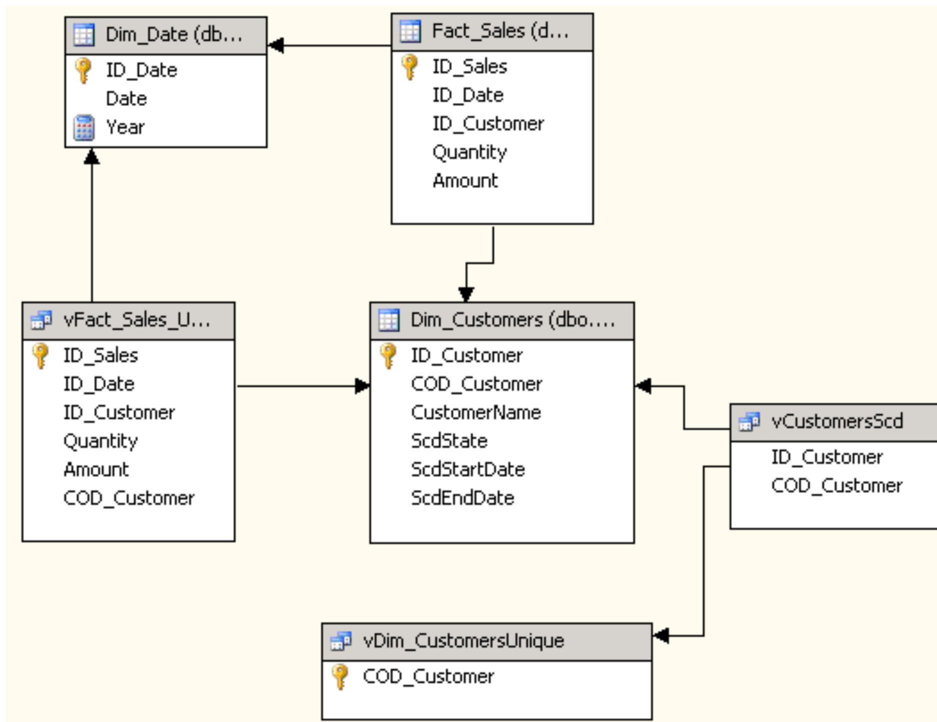


Figure 48 – Data Source View to implement different distinct count strategies

In the simplest scenario (Solution A), we create a unique customer id at the fact table level and then define a distinct count measure on it. Figure 49 shows that we could have used vFact_Sales_Unique view to build both Fact Sales measure group measures and the A Count measure on the A Customers measure group.

However, there is no benefit on doing so. A distinct count measure needs a dedicated measure group (A Customers) that SSAS will process with a separated query to the fact table. In this case, we want to limit the join between Fact_Sales and Dim_Customers only for the COD_Customer distinct count evaluation. From this point of view, we could eliminate the other measures (Quantity and Amount) from vFact_Sales_Unique. This is only an aesthetic touch without improvements on the performance side, but it makes a lot of sense from the maintenance point of view and in order not to confuse people with two copies of the same table.

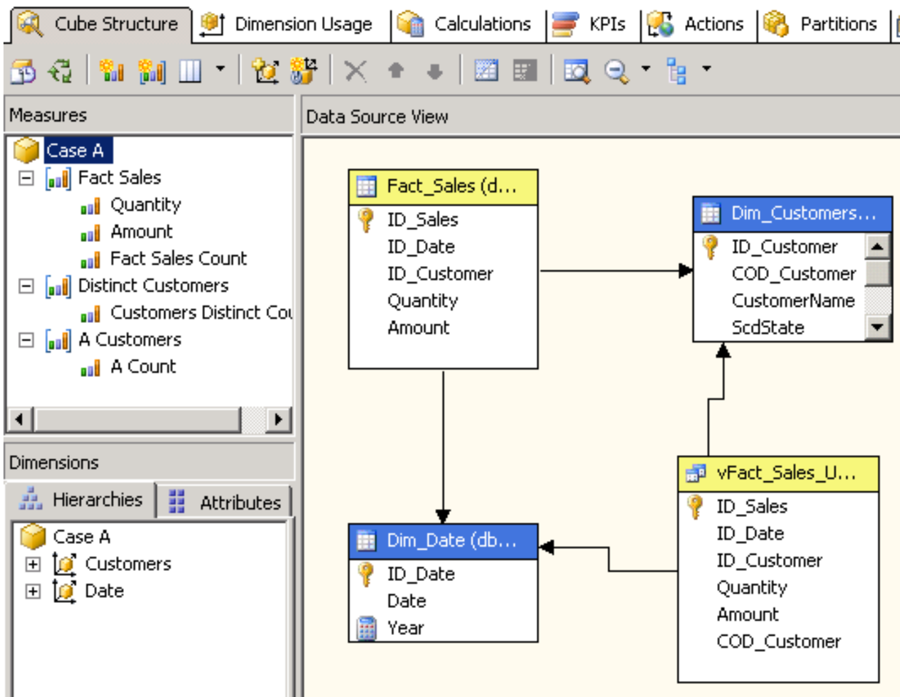


Figure 49 – Case A with standard distinct count measure on fact table

Once we created the A Customers measure group, we need to relate it to cube dimensions, as shown in Figure 50. The relationships are very simple and identical to those with other measure groups.

Dimensions	Fact Sales	Distinct Customers	A Customers
Customers	Dim Customers	Dim Customers	Dim Customers
Date	ID_Date	ID_Date	ID_Date

Figure 50 – Case A Dimension Usage

We can look at results obtained with A Count measure (Figure 51). The Customers Distinct Count measure is 6 for 2006 because it counts the number of rows in Dim_Customers; we have two versions for Customer 6 (v1 and v2) so it is counted twice here. The new A Count measure has the right number of 5 and it is the number we want to see.

	Values				
Row Labels	**Quantity**	**Amount**	**Fact Sales Count**	**Customers Distinct Count**	**A Count**
2006	131	40581.54	10	6	5
Grand Total	**131**	**40581.54**	**10**	**6**	**5**

Figure 51 – Case A results

Although we have solved the business problem, we could face some performance issues, which we will discuss further in the Performance section. However, it is necessary to note something here:

- SSAS will obtain A Distinct Count measure through an ORDER BY query that uses the measure expression as the key to sort.
- The application key we are using to evaluate the distinct count could be a long string. We have to handle it in the cube, even if it is not interesting to the end user.
- We used a view to avoid duplicating customer dimension data in a Customers Unique dimension, but this view contains a join and will be queried using an ORDER BY clause. This could be very heavy on large fact tables and large dimensions.
- Distinct Count measures on Analysis Services 2005 are not very scalable when the size of data grows. Starting from 2008 forward, the algorithm has been improved but it still requires a careful planning of the partitions to provide fast parallel computation of distinct counts.

Solution B still uses a distinct measure, but this time we do not use a view. Instead, we rely on the UDM fact dimension feature. Figure 52 shows that the B Count measure on a distinct count of the COD_Customer field in Dim_Customers table (that is used both as a dimension and as a fact table).

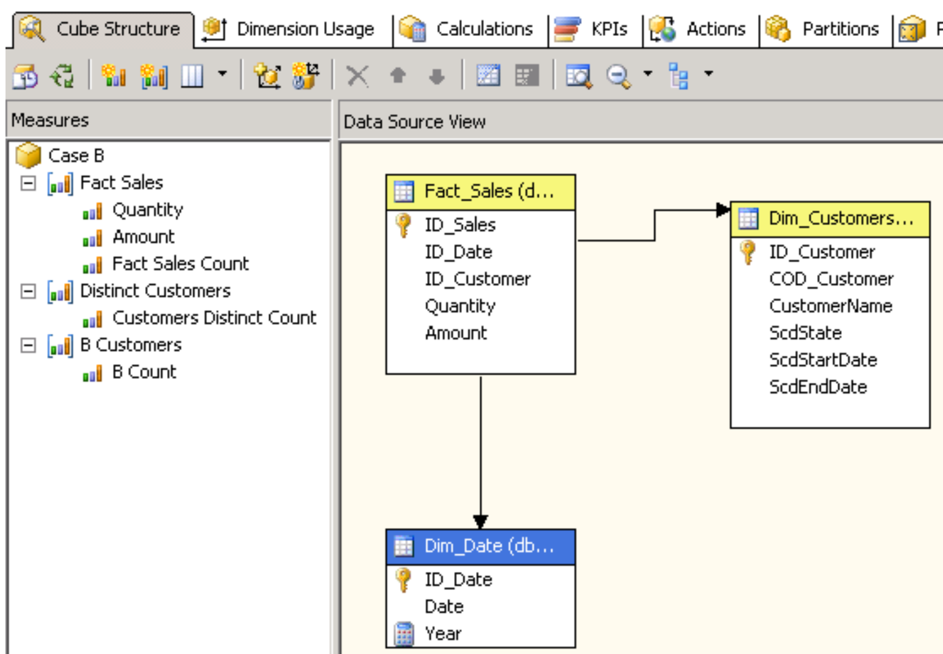


Figure 52 – Case B with distinct count measure on customer dimension

The B Customers measure group has a direct relationship with the Customers dimension (the relationship type is “Fact”) and a many-to-many relationship with Date dimension via the Fact Sales measure group. Apparently, this is a strange relationship because a row in Dim_Customers as fact table has a one-to-one relationship with Dim_Customers as Customers dimension (it is the same table!). However, the reality is that each customer can be related to many dates and each date can be related to many customers, and Fact_Sales defines exactly this relationship. Figure 53 shows the resulting Dimension Usage.

Dimensions	Fact Sales	Distinct Customers	B Customers
Customers	Dim Customers	Dim Customers	Dim Customers
Date	ID_Date	ID_Date	Fact Sales

Figure 53 – Case B Dimension Usage

At the end, we have similar results to those obtained with Solution A: Figure 54 shows the B count results. The only difference is that when there is no filter on Date dimension (the Grand Total row) the B count shows the overall number of unique customers instead of considering only the customers who made at least one transaction. We already discussed this in the context of the previous scenario when we did not have a slowly changing dimension for Customers.

Row Labels	Quantity	Amount	Fact Sales Count	Customers Distinct Count	B Count
2006	131	40581.54	10	6	5
Grand Total	**131**	**40581.54**	**10**	**6**	**9**

Figure 54 – Case B results

What is the biggest difference between Solution A and Solution B? In Solution A, we had to build a view (or a persisted dimension table) to link the unique customers dimension to the Fact_Sales table. In Solution B, we do not need it. SSAS makes the processing query only against the cardinality of Dim_Customers table and not against the cardinality of the more populated Fact_Sales table. This might result in significantly better performances.

In Solution C, we apply the lesson we learned at the beginning of this chapter, when we used a many-to-many relationship to get the same results of a distinct count measure. In this way, we will remove the need for a distinct count measure and related implications.

Figure 55 shows that the model becomes relatively more complex. We need to build a fact dimension (vDim_CustomersUnique) where the number of rows equals the number of unique Customers we have. Unfortunately, we cannot extend the model we previously defined for Solution B because the fact dimension we used (Dim_Customers) cannot serve as an intermediate measure group in a many-to-many relationship. For this reason, we created a view (vCustomersScd) that serves as a bridge table between Dim_Customers and vDim_CustomersUnique.

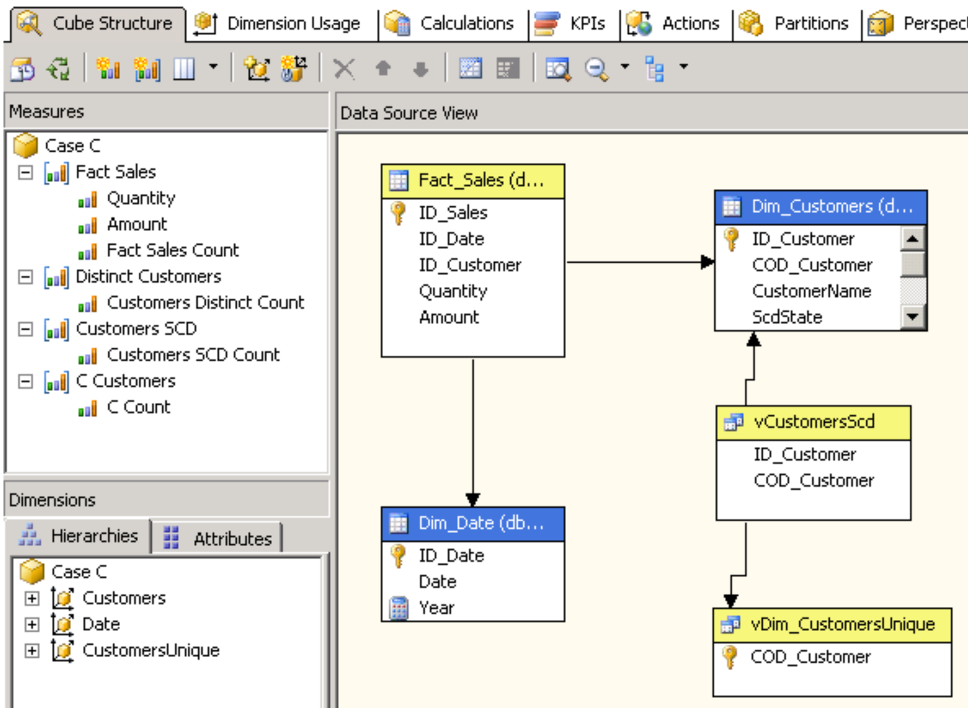


Figure 55 – Case C with distinct count measure by many-to-many relationship

The Customers SCD measure group has a row for each row in Dim_Customers, with a one-to-one relationship. The C Customers measure group has a row for each unique customer. To define a relationship between these two measure groups, it is necessary to have a dimension shared by both measure groups. This role is fulfilled by the CustomersUnique dimension, which has the same cardinality as C Customers. While we can identify a one-to-many relationship between the Customers SCD and C Customers measure group, a better approach is to leverage the UDM many-to-many relationship.

The Customers SCD measure group plays a very important role linking the C Customers measure group with all the other measure groups of a cube. Figure 56 shows the Dimension Usage setup required to implement case C.

Dimensions	Measure Groups: Fact Sales	Distinct Customers	Customers SCD	C Customers
Customers	Dim Customers	Dim Customers	Dim Customers	Customers SCD
Date	ID_Date	ID_Date	Fact Sales	Customers SCD
CustomersUnique	Customers SCD	Customers SCD	Customer Code	Customer Code

Figure 56 – Case C Dimension Usage

The CustomersUnique dimension plays an important role in the definition of the correct relationship between measure groups. Nevertheless, its content may not be useful for end user reporting. For this reason, it is often convenient to hide this dimension from end users.

Another interesting aspect is that the CustomersUnique dimension has the customer application key as a primary key of the dimension. If the application key (COD_Customer, in this case) is very long, it could become a potential performance bottleneck and it will consume more space for data storage. In real project, we often use a persistent dimension table instead of a view, just to get a surrogate key (of type

integer) instead of the large application key (more than 20 characters) we get from the OLTP. Figure 57 shows the results.

	A	B	C	D	E	F	G
1		Values					
2	Row Labels	Quantity	Amount	Fact Sales Count	Customers Distinct Count	Customers SCD Count	C Count
3	2006	131	40581.54	10	6	6	5
4	**Grand Total**	**131**	**40581.54**	**10**	**6**	**10**	**9**

Figure 57 – Case C results

While the Customer SCD Count measure is not useful, the C Count behaves exactly as the B Count measure.

As we said before, it is interesting to consider the implementation of the distinct count measures with many-to-many relationships in order to gain performance improvements. In the next pages, we will look at performances.

Performance

There are two main observable differences when we query a cube that has distinct count measures obtained with different methods:

- Usage of more processors and I/O
- Effectiveness in caching the query results

To understand performance impact, we have to understand how Analysis Services resolves queries for these kinds of measures.

A "classical" distinct count measure works in this way:

- At processing time, SSAS adds an ORDER BY clause to the query sent to the relational engine for each cube partition to order data by the distinct count measure expression. In the best-case scenario, you do not have joins between the fact table and other tables, but when you have millions of rows the ORDER BY clause could be very slow and it may require many resources (memory and disk for temporary table).

  Note that this affects the performance of the relational engine. The processing time of distinct count measures could be very long. However, it can be improved using incremental updates rather than processing the entire cube (FullProcess option).

  This explains why the Distinct Count measure needs a separate measure group in UDM.

  Our hypothesis is that SSAS dedicates an index for a distinct count measure and the correct order of items is necessary to use a memory-efficient algorithm.

- At query time, SSAS makes a sequential scan of the distinct count partition for each query involving a distinct count measure. Query response time depends on both the number of rows in the fact table and the number of different distinct values of the measure.

For some reason, SSAS cannot entirely cache the query and a subsequent query containing a distinct count measure requires more or less the same time (probably the time improvement depends from the elimination of disk I/O with all necessary data already in server memory).

Even a full measure group optimization (building 100% of possible aggregation) does not improve significantly this type of queries.

A measure involving a many-to-many relationship works as follows:

- At processing time, SSAS reads the bridge table used by the many-to-many relationship in no particular order (like a regular fact table). There is no pressure on the relational engine even with millions of rows. Nevertheless, as a many-to-many relationship relates members of two different dimensions, it should be rare to have more than 10 million rows to process.

- At query time, SSAS reads the fact table into memory to evaluate the many-to-many relationship through the bridge measure group. It does so mainly to join the two measure groups at query time and it needs to join them at the lowest level of each dimension (common to both measure groups). Aggregations at the proper granularity level might improve performance, reducing the number of rows considered, according with the slicers used in a query.

  The engine does a hash join for this purpose (unlike the SQL Server query engine, Analysis Services does not have multiple join algorithms to choose from). The hash join does a lookup on the bridge measure group (or to one of its aggregation if possible, in order to reduce the workload), builds a hash index on it, scans the fact measure group and combines the two results together.

  As you can imagine, this operation requires enough virtual memory to load and evaluate the datasets. The resolution of the join can exhausts the two gigabytes addressable memory space in a 32-bit system. A 64-bit system does not exhaust the virtual memory, but it is important that enough physical RAM is available to avoid paging of memory. If the memory is enough, the first query may be very slow, while in a two gigabytes user memory address space a fact table with 100 million rows joined to an intermediate measure group with 1 million rows could fail the query exhausting the address space of the Analysis Services process. It could be very slow the first time, but subsequent queries are very fast (immediate response) because Analysis Services caches very well previous results.
  Consider that the critical condition for the memory usage is a combination of sizes of the two tables, a small bridge table consumes less memory than a large one applied to the same 100 million rows fact table. You can apply the techniques described in the "Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques" whitepaper (see Links section).

Unfortunately, these two techniques to calculate a distinct count measure (the "classic" one and the one based on many-to-many dimension relationships) have both some shortcomings. If we could warm up the cache after cube processing (for example by executing a scheduled MDX query), users would probably favor the performance of a distinct count measure based on many-to-many relationships. That is because each time the end user changes a selection or a filter with the "classic" model, the user will experience performance degradation. Consequently, interactive reports typically run faster with the many-to-many relationship technique. The performance degradation associated with the "classic" distinct count model is a minor issue with static reports, especially with Reporting Services cached reports.

The real problem with using many-to-many relationship is the limit of fact table rows you can query. We should evaluate carefully the use of many-to-many relationships when you have intermediate measure groups getting data from fact table with more than 1 million of rows. Refer to many-to-many optimization whitepaper in the Links section.

Finally, please consider we compared the classical Distinct Count measure without a particular optimization. You should look at the Analysis Services Distinct Count Optimization white paper (see Links section) in order to know how to optimize the regular distinct count measure. It could be expensive at processing time, but it can improve performance at query time more than you can to with the many-to-many relationship approach.

Multiple groups

BI users tend to love freedom; this is a fact of reality. Among all the other kinds of freedom they like, a very frequent one is the ability to group items from dimensions in diverse and unpredictable ways, in order to have their specific way of looking at dimension members.

We build attributes to help them in aggregating dimension members but, in a typical cube dimension, we define attributes at the data warehouse's design stage. Adding an attribute is an operation that requires changes in all layers of the BI solution.

While a rigid design is good for performance optimization, this is a severe limitation for end users like marketing analysts, who try to jump over these limits by extracting data from the data warehouse, working with them offline. They need to make custom groups of dimension elements based on some characteristics that they did not known before and that probably will change over time.

There are many examples of this situation, but we can generalize it by assuming that a user may want to group some dimension members together, associating them with a group name. Moreover, a single dimension member might belong to several groups.

The "Multiple Groups" model we are going to introduce has an interesting characteristic: we will base it on a fixed relational and multidimensional schema, and the user will be able to define new groups using a data driven methodology. Moreover, groups are immediately available to all clients and a new group can be added by only reprocessing a small measure group (corresponding to the bridge table for a many-to-many relationship), giving the opportunity to create solutions that enable a user to create custom groups on the fly.

Business scenario

Typically, sales analysis involves the creation of specific groups of customer and product dimension members. These groups can be based on events (e.g. who has been included in a mail campaign), on profiling analysis (e.g. could be the result of a manual segmentation or a data mining clustering model) or on other arbitrary data.

The classical approach for custom grouping is to define a table for each type of group, with a field for each group attribute and a field for customer key. The table will contain a row for each customer that belongs to each group.

For example, imagine that we need to segment customers with some profile and want to track customers who received mailing offers for our products: Figure 58 shows a canonical solution that uses a separate table for each kind of group.

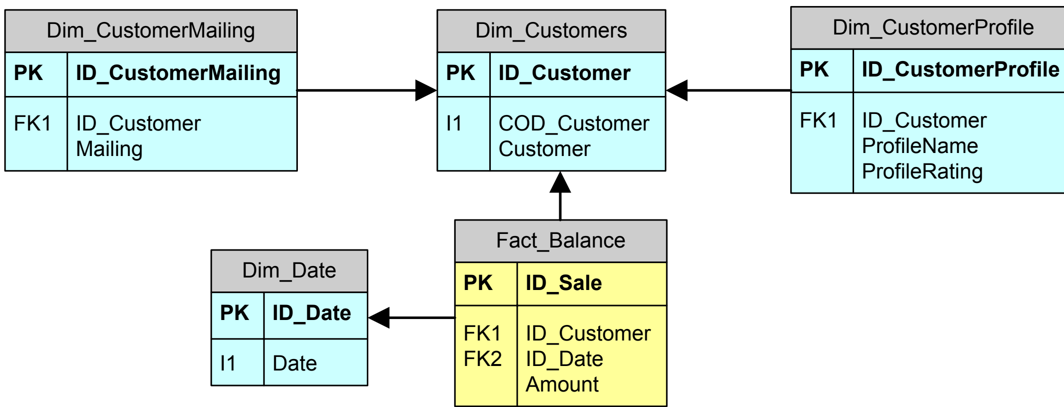


Figure 58 – Multiple grouping made with a table for each kind of group

We could implement a corresponding UDM with the Customer dimension related to CustomerProfile and CustomerMailing dimensions with two different many-to-many relationships. The key point here is that if a customer could belong to more than one group, we need to go for many-to-many relationships.

At this point, a more normalized and UDM-friendly way to handle this scenario is shown in Figure 59.

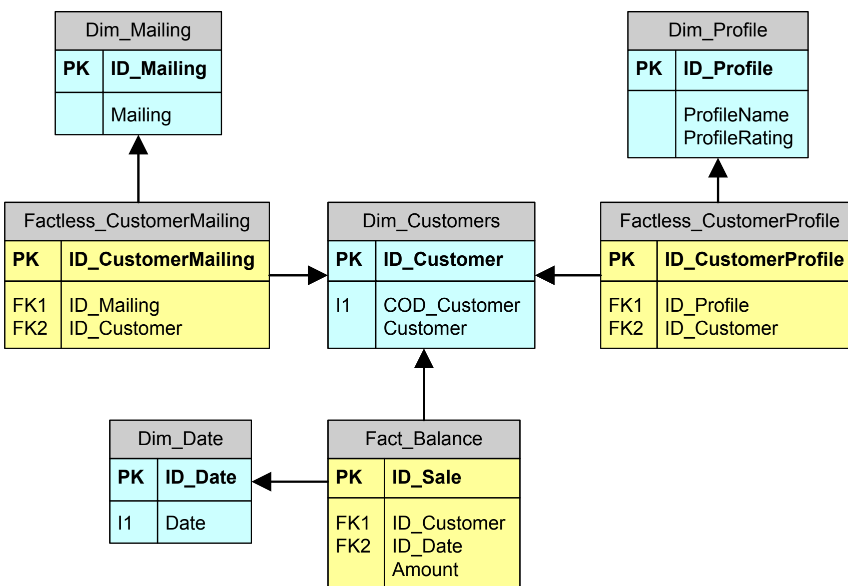


Figure 59 – Multiple grouping with explicit many-to-many relationships

This model allows us to use a single "group" dimension table for any kind of grouping, but it does not give us enough flexibility: if a new group requires a new table in the data warehouse, it will also require changes to the ETL processes and UDM.

Implementation

A potential weakness of the model (see Figure 59) resides in the customer-profiling requirement. In the real world, we can have many profiles, but for each profile, a customer can have only one rating (or no rating at all). Unfortunately, we cannot implement this requirement with a constraint in the relational database. One way to implement this level of control on the model shown in Figure 58 would be a unique index on the ProfileName and ID_Customer fields. However, data integrity is out of our scope here. After all, a data mart

has to be loaded with correct data and we will delegate this check responsibility to the ETL pipeline, but we will see that this note will be important in our final considerations for this scenario.

If we consider the whole scenario, we can identify these requirements:

- A customer can belong to many groups
- A group can have many customers
- A group can have a characteristic name and a "value" textual attribute (see Figure 60).

Please note that in the next implementation COD_GroupName and COD_GroupValue fields are application keys that we will use to implement grouping.

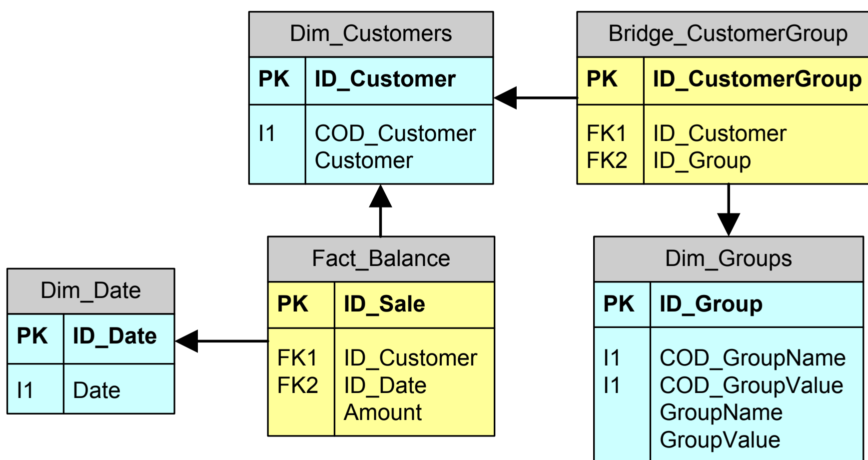


Figure 60 – Multiple grouping with a generic flexible model

Sometimes we can use the group name as a sort of group category and the group value as the real group name. Other times, we can use the group value for segmenting the group population. Table 7 shows both variants. The Mailing group name identifies a category of mailings and a customer could belong to any (even all) of the possible groups defined by Group Value (in this example, Promo Spring and Promo Fall are two mailings we have made to two different and partially overlapping groups of customers). The Profile group name identifies a single group where each customer must belong to only one of the possible group values: Retail, Affluent, Private or Corporate.

Group Name	Group Value
Mailing	Promo Spring
Mailing	Promo Fall
Profile	Retail
Profile	Affluent
Profile	Private
Profile	Corporate

Table 7 – Groups dimension sample data

The interesting part is that adding a new group does not require any structural change in the model. For example, a new Promo Winter mailing needs only a new record in the Dim_Groups table and a correct population of the Bridge CustomerGroup table: given a new ID_Group, it is only necessary to get a list of ID_Customer to do this population.

We can create the cube with the auto build feature of the Cube Wizard. The resulting model would correctly identify dimension and fact tables but, as we have seen before, we have to define manually some of the missing relationships between dimensions and measure groups.

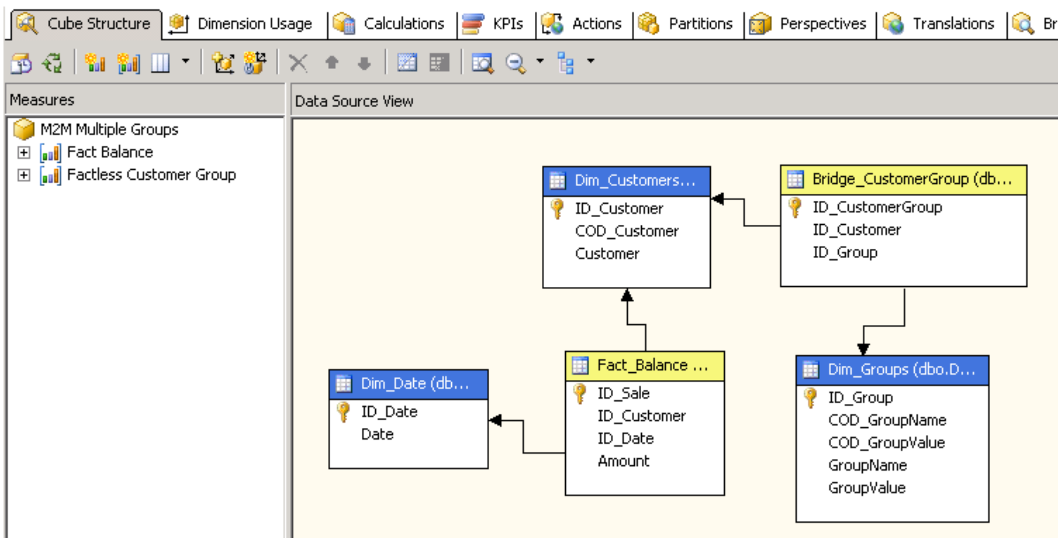


Figure 61 – Cube structure for multiple grouping

As you can see in Figure 61, we have two measure groups for a total of three measures.

- Fact Balance Count is the number of rows for the Fact Balance table.
- Bridge Customer Group Count is the number of customers for selected group(s). From another point of view, it is also the number of groups to which a customer belongs, depending on which dimension you are using to slice or filter data.

If users do not need to analyze a group population, you can hide the Bridge Customer Group Count measure. Otherwise, it would be a good idea renaming it to a more meaningful name.

Dimensions	Measure Groups: Fact Balance	Bridge Customer Group
Dim Customers	Dim Customers	Dim Customers
Dim Date	ID_Date	
Dim Groups	Bridge Customer Group	Dim Groups

Figure 62 – Cube wizard dimension usage results for multiple grouping

Figure 62 shows that in this case only the Date dimension has to be related to Bridge Customer Group. We can fill the gray cell with another many-to-many relationship (the first one was created by the wizard), as shown in Figure 63.

Dimensions	Measure Groups: Fact Balance	Bridge Customer Group
Dim Customers	Dim Customers	Dim Customers
Dim Date	ID_Date	Fact Balance
Dim Groups	Bridge Customer Group	Dim Groups

Figure 63 – Completed dimension usage for multiple grouping

Figure 64 shows a sample report using this model. The filter is set on a specific date (remember that the Balance Amount is a measure that cannot be summed over time), while the Group Name and Group Value dimensions are placed on the rows.

The two mailing groups (Promo Fall and Promo Spring) partially contain same members. In fact, Total row for Mailing group name is less than the sum of each single group value row. We have a different situation for Profile group name. A customer should belong to only one of the possible child group values, which is the case with our sample data.

	A	B	C	D
1	Date	2005-12-31 00:00:00		
2				
3		Values		
4	**Row Labels**	**Amount**	**Fact Balance Count**	**Bridge Customer Group Count**
5	**Mailing**	**400**	**4**	**5**
6	Promo Fall	300	3	3
7	Promo Spring	200	2	2
8	**Profile**	**400**	**4**	**4**
9	Corporate	200	2	2
10	Private	200	2	2
11	**Grand Total**	**400**	**4**	**9**

Figure 64 – Sample query for multiple groups

Having analyzed the Balance Amount measure, we can apply the same considerations for the Fact Balance Count measure. Typically, it is used as the denominator to get an average amount balance instead of the total balance (Sum aggregation). It is important to note that Fact Balance Count could be lower than Bridge Customer Group Count, even for a single Group Value row. This happens when at least one customer associated with the group has no registered balance for the chosen date.

A further consideration about Bridge Customer Group Count measure is that it is aggregated as a regular measure and it has not to be confused with the number of *different* customers belonging to a group. This is particularly important when you are considering the total for a Group Name, grouping all its Group Value children: this is another good reason to hide this measure from end users.

While it could be possible adding other attributes to the Groups dimension, you have to be very careful in doing so. If you want a generic way to group items of a dimension, it is important to leave the group dimension design as generic as possible. Adding an attribute used only with some specific groups would be a bad way to make things easy to use and to read.

A final consideration is about the overall performance. From a query standpoint, it is not possible to define aggregations at a group level for the Fact Balance measure group (like any other many-to-many relationships, it has to be evaluated at query time). Nevertheless, in our experience the query response

time could still be acceptable for many real-world scenarios. Most important, this query-time calculation has a very positive impact on the processing-time. If you need to add data to form a new group, it is necessary to process only Dim Groups dimension and Bridge Customer Group measure group and these processes can be done incrementally! For this reason, we suggest that you to consider this scenario even for on-the-fly modifications of custom groups made by end users, without relying on client-based solutions.

Remember that we need ETL processes to update and process group-related structures. The end user should not be able to manipulate the Dim_Groups dimension, because this might lead to inconsistent data.

Cross-Time

Almost all measures in a data warehouse are time-dependent. The classical star schema has a fact table that contains numeric measures and many dimension tables that define the grain of any single measure. This is a good model (especially if you build an OLAP cube on it) for analyzing sales over a given period. Nevertheless, it does not show how the distribution of dimension attributes changes over time. For this reason, Kimball's advice is to define a separate fact table that take "snapshots" of dimension state over time.

However, snapshot fact tables might not satisfy all reporting needs. For example, it is hard to query for the change of an attribute distribution between two dates. We can leverage the many-to-many relationship feature in order to solve the problem. We will call "cross-time" the technique that combines "time snapshots" and many-to-many relationships to enhance analysis capabilities inside client tools.

Business scenario

While we can apply the cross-time technique to any slowly changing dimension, we will use as a typical scenario one that involves the customer dimension. Customer attributes change over time and SCD tracks the history changes. However, it is not easy to analyze the SCD changes without a two-step operation that will require first the selection of a set of customers with certain attributes at a specific date and then the usage of this selection to query data and see measures or attribute values on a different date.

Typically, the existing star schema may look like the one illustrated in Figure 65. Here, we have a fact table with meaningful measures (in this case Balance is a non-additive measure over Time), a date dimension and a Customer SCD Type II dimension.

Please note that we have created a snowflake schema for customers because the application key is already in normal form in the Dim_CustomerUnique table. This model also makes it easier to model distinct count measures as we have seen before.

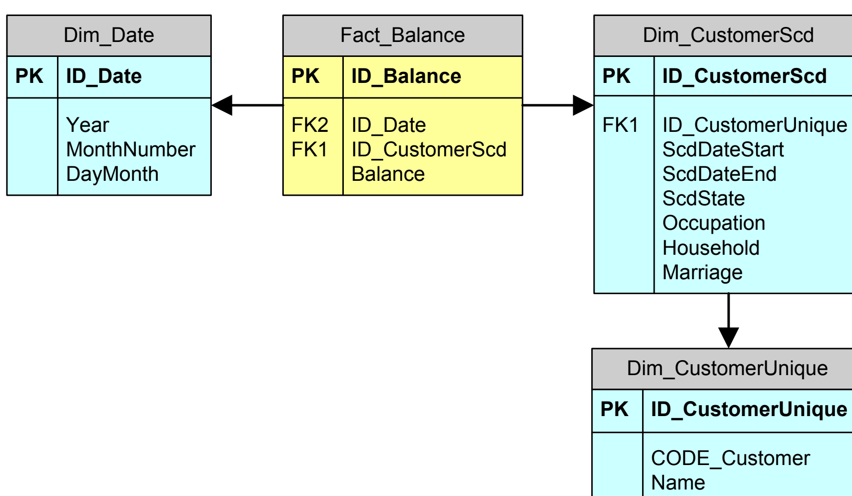


Figure 65 – Relational star schema with unique dimension

Our users may need to analyze how customers have changed occupation from January to December 2005.

This first question can be answered by the SQL query shown below, which is not so easy to build with a query builder.

```
SELECT
  c1.Occupation AS JanuaryOccupation,
  c2.Occupation AS DecemberOccupation,
  COUNT(*) AS Customers
FROM Dim_CustomerScd c1
INNER JOIN Dim_CustomerScd c2
   ON c2.ID_CustomerUnique = c1.ID_CustomerUnique
  AND '20051201' >= c2.ScdDateStart
  AND ('20051201' <= c2.ScdDateEnd OR c2.ScdDateEnd IS NULL)
WHERE '20050101' >= c1.ScdDateStart
  AND ('20050101' <= c1.ScdDateEnd OR c1.ScdDateEnd IS NULL)
GROUP BY c1.Occupation, c2.Occupation
```

Moreover, changing the analyzed attribute requires a different syntax since it could not be parameterized.

What will happen if the end user wants to analyze balances for year 2005 for customers who had a Student occupation in January, irrespective of their current occupation? To answer this second question, we can use SQL again, but this might not be practical to analyze if the end users use OLAP browsers, such as pivot table or pivot chart. Moreover, the end user writing a SQL query might mismatch one of the joins getting wrong results.

```
SELECT b.ID_Date, SUM( b.Balance )
FROM Fact_Balance b
INNER JOIN Dim_CustomerScd c
   ON c.ID_CustomerScd = b.ID_CustomerScd
INNER JOIN Dim_CustomerUnique u
   ON u.ID_CustomerUnique = c.ID_CustomerUnique
INNER JOIN Dim_CustomerScd cj
   ON cj.ID_CustomerUnique = u.ID_CustomerUnique
WHERE cj.Occupation = 'Student'
  AND '20050101' >= cj.ScdDateStart
  AND ('20050101' <= cj.ScdDateEnd
       OR cj.ScdDateEnd IS NULL)
GROUP BY b.ID_Date
```

Ideally, we would like to track differences in attributes between different snapshots and relate standard measures (like balance in our sample) behavior over time for a group of customers at a given point of time.

Implementation

ScdDateStart and ScdDateEnd do not give us an easy way to get the set of valid customers at a certain date period, especially if we want to do it from a pivot table report. The classical solution is to de-normalize the schema.

We could get a complete denormalization of customer attributes by making a snapshot table for them. Since we already have a SCD Type II customer dimension, we can shortcut the implementation process by making a snapshot table that only stores the relationship between a date period and a customer version. In this way, we can obtain the previously discussed complete attribute snapshot table by only joining two tables (this solution saves disk space and execution time).

The tricky part is to make it possible for the user to select all the customer versions that had a particular attribute value for a particular date period. We will use many-to-many dimensions to model the solution.

To relate different versions of the same customer, we store the unique customer identification in a snapshot table too. Figure 66 illustrates the resulting relational model where we named the snapshot table as Bridge_CustomerSnapshot. This table will be the bridge table that relates different Date and Customer dimensions.

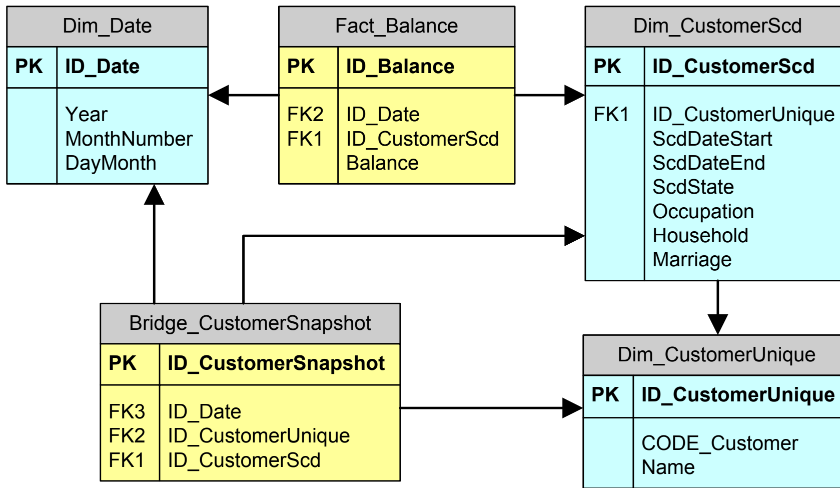


Figure 66 – Relational schema enable to cross-time analysis

We need to make a minor change to the data source view (Figure 67) by adding the vBridge_CustomerScd view. As we have already seen for Distinct Count scenario, we cannot use a fact dimension as an intermediate measure group in a many-to-many relationship.

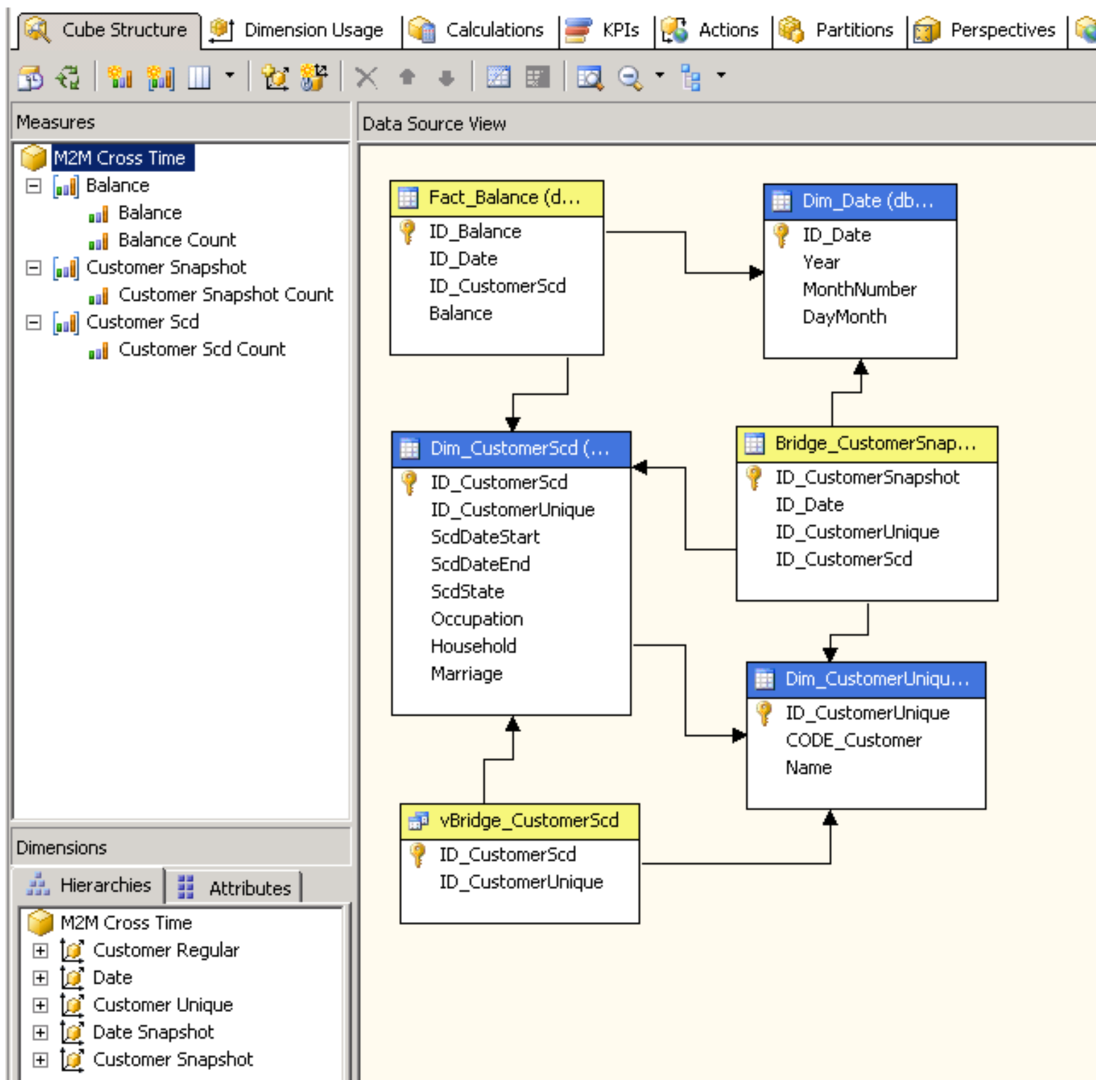


Figure 67 – Cross-Time Data Source View and cube structure

We will build this first cube using role-playing dimensions:

- Date dimension represents the date for balance measure group
- Date Snapshot dimension represents the date for a particular snapshot (in the sample UDM, we used a monthly snapshot, but we could choose a different granularity).
- Customer Regular dimension references to the Balance measure group (it is the "regular" dimension for transactional measures) and thus is indirectly related to the Date dimension.
- Customer Snapshot dimension relates to Date Snapshot and we will use it to select a group of customers for a particular snapshot.

The Dim_CustomerUnique dimension becomes the bridge between Customer Snapshot and Customer Scd measure groups. The former mirrors the snapshot information, while the latter supports the many-to-many relationship between Customer Regular and Customer Scd dimensions. Note that we can use Customer Unique dimension to query data for a particular customer given its application key.

Figure 68 shows the resulting dimension usage matrix. If we try to use the cube wizard, the dimension usage matrix will be very different. Once auto build completes, we will need to add role-playing dimensions

and to change many dimension usage settings. Instead, we suggest defining the dimension usage manually, to reduce the chance of forgetting to adjust some relationships.

Dimensions	Measure Groups: Balance	Customer Snapshot	Customer Scd
Customer Scd (Customer Regular)	ID Customer Scd	Customer Scd	ID Customer Scd
Date	ID_Date	Customer Scd	Balance
Customer Unique	Customer Regular	Dim Customer Unique	Dim Customer Unique
Date (Date Snapshot)	Customer Snapshot	ID_Date	Customer Snapshot
Customer Scd (Customer Snaps…	Customer Snapshot	ID Customer Scd	Customer Snapshot

Figure 68 – Dimension Usage for Cross-Time cube

Before querying the cube, it is important to make the purpose of the measures clear, because they are not very intuitive at first glance.

- Balance and Balance Count are regular measures coming from the Fact_Balance table. If we need an average balance, we should divide Balance by Balance Count.

- Customer Snapshot Count shows how many customers are present in the snapshot for the current date selection. When we select a single snapshot date, Customer Snapshot Count shows the number of customers for that snapshot at that date.

- Customer Scd Count might be not very useful to users because it shows the number of customer versions for the current selection. For example, if we choose a single customer, it will show the number of different versions stored in Dim_CustomerScd (a SCD Type II Customer Dimension) for that customer. To avoid confusion, it would be better to hide Customer Scd Count and Customer Snapshot Count.

Figure 69 show a sample report. We filtered data by January 2005 (dimension Date Snapshot in filter area), and we placed Year/Month of Date dimension on rows and Customer Snapshot Occupation attribute on columns. As you can see, we had three customers in January snapshot: one was a student and two were teachers.

	A	B	C	D	E	F	G
1	Date Snapshot.Year - Month Number	1					
2							
3		**Column Labels**					
4		**Student**		**Teacher**		**Total Balance**	**Total Balance Count**
5	**Row Labels**	**Balance**	**Balance Count**	**Balance**	**Balance Count**		
6	**⊟2005**	**1980**	**12**	**3960**	**24**	**5940**	**36**
7	1	110	1	220	2	330	3
8	2	120	1	240	2	360	3
9	3	130	1	260	2	390	3
10	4	140	1	280	2	420	3
11	5	150	1	300	2	450	3
12	6	160	1	320	2	480	3
13	7	170	1	340	2	510	3
14	8	180	1	360	2	540	3
15	9	190	1	380	2	570	3
16	10	200	1	400	2	600	3
17	11	210	1	420	2	630	3
18	12	220	1	440	2	660	3
19	**Grand Total**	**1980**	**12**	**3960**	**24**	**5940**	**36**

Figure 69 – Cross-Time results for snapshot customer occupation attribute

If we place the Customer Regular Occupation attribute on columns, we get the results shown in Figure 70. We can deduce that the student we had in January became an employee in March. If we had filtered by Student, we would have obtained the same result without the Teacher column.

	A	B	C	D	E	F	G	H	I
1	Date Snapshot.Year - Month Number	1							
2									
3		**Column Labels**							
4		**Employee**		**Student**		**Teacher**		**Total Balance**	**Total Balance Count**
5	**Row Labels**	**Balance**	**Balance Count**	**Balance**	**Balance Count**	**Balance**	**Balance Count**		
6	**⊟2005**	**1750**	**10**	**230**	**2**	**3960**	**24**	**5940**	**36**
7	1			110	1	220	2	330	3
8	2			120	1	240	2	360	3
9	3	130	1			260	2	390	3
10	4	140	1			280	2	420	3
11	5	150	1			300	2	450	3
12	6	160	1			320	2	480	3
13	7	170	1			340	2	510	3
14	8	180	1			360	2	540	3
15	9	190	1			380	2	570	3
16	10	200	1			400	2	600	3
17	11	210	1			420	2	630	3
18	12	220	1			440	2	660	3
19	**Grand Total**	**1750**	**10**	**230**	**2**	**3960**	**24**	**5940**	**36**

Figure 70 – Cross-Time results for regular customer occupation attribute

Now we have a powerful tool to produce reports that cross snapshots, "transactional" dimensions and attributes. This is very useful to satisfy our second objective (occupation change between January and December) previously described in the business scenario, but still does not completely satisfy the first one (balances for customers who were students in January). In fact, if we want to compare the January snapshot with the December customer data, we need to make a prior assumption that all customers in January have balances in December. If this is correct, we can author the report shown in Figure 71.

	A	B	C	D	E	F	G
1	Date Snapshot.Year - Month Number	1					
2	Date.Year - Month Number	12					
3							
4		**Column Labels**					
5		**Student**		**Teacher**		**Total Balance**	**Total Balance Count**
6	**Row Labels**	**Balance**	**Balance Count**	**Balance**	**Balance Count**		
7	Employee	220	1			220	1
8	Teacher			440	2	440	2
9	**Grand Total**	**220**	**1**	**440**	**2**	**660**	**3**

Figure 71 – Cross-Time results comparing a snapshot with a date

It seems all good, but it does not work well when we choose different months for the Date dimensions. In Figure 72, we chose Jan/Feb for Date Snapshot dimension and Nov/Dec for Date dimension. Now, it is difficult to understand how many customers we really have. We would have run into the same problem before, if only a customer had multiple balances for the same date. Therefore, dividing by the number of months selected is not a valid solution.

	A	B	C	D	E	F	G
1	Date Snapshot.Year - Month Number	(Multiple Items)					
2	Date.Year - Month Number	(Multiple Items)					
3							
4		**Column Labels**					
5		**Student**		**Teacher**		**Total Balance**	**Total Balance Count**
6	**Row Labels**	**Balance**	**Balance Count**	**Balance**	**Balance Count**		
7	Employee	430	2			430	2
8	Teacher			860	4	860	4
9	**Grand Total**	**430**	**2**	**860**	**4**	**1290**	**6**

Figure 72 – Cross-Time results comparing multiple periods

The last step to obtain the final model is the most complex one. We have to add a distinct count measure to get the right number of customers under any conditions. This is often the case when we have a SCD Type II dimension in our cube.

Figure 73 shows the new cube structure. While the Data Source View is unchanged, we have a new measure group (Customer Unique) that contains a new measure (Customers Distinct Count) that should be visible to end users. Dim_CustomerUnique is assumed also to be a fact table role other than its own dimension role.

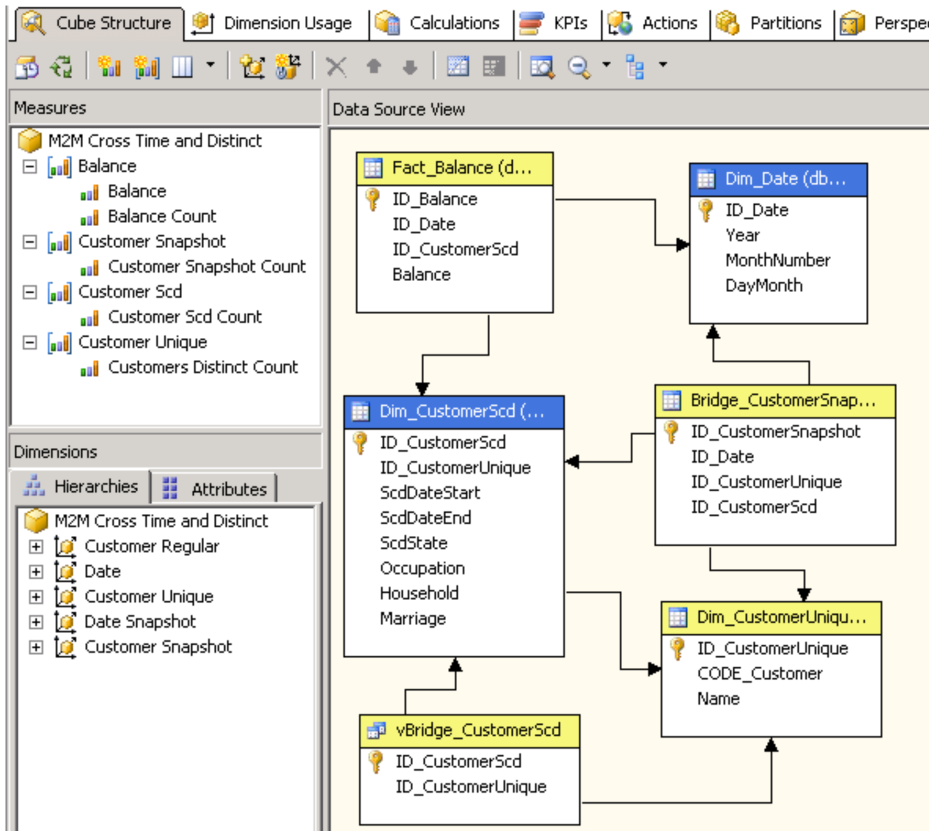


Figure 73 – Cross-Time Data Source View and cube structure (with distinct)

We need to relate the new measure group correctly to existing dimensions (see Figure 74). It is important to define all relationships between Customer Unique and all cube dimensions other than Customer Unique dimension as many-to-many relationships.

Dimensions	Measure Groups: Balance	Customer Snapshot	Customer Scd	Customer Unique
Customer Scd (Customer Regular)	ID Customer Scd	Customer Scd	ID Customer Scd	Balance
Date	ID_Date	Customer Scd	Balance	Balance
Customer Unique	Customer Regular	Dim Customer Unique	Dim Customer Unique	Dim Customer Unique
Date (Date Snapshot)	Customer Snapshot	ID_Date	Customer Snapshot	Customer Snapshot
Customer Scd (Customer Snaps...	Customer Snapshot	ID Customer Scd	Customer Snapshot	Customer Snapshot

Figure 74 – Dimension Usage for Cross Time cube (with distinct customers)

Once done, we can get the correct results for our first objective (see Figure 75), i.e. how customers have changed occupation from January to December 2005.

	A	B	C	D
1	Date Snapshot.Year - Month Number	(Multiple Items)		
2	Date.Year - Month Number	(Multiple Items)		
3				
4	**Customers Distinct Count**	**Column Labels**		
5	**Row Labels**	**Student**	**Teacher**	**Grand Total**
6	Employee	1		1
7	Teacher		2	2
8	**Grand Total**	**1**	**2**	**3**

Figure 75 – Cross-Time results comparing multiple periods with distinct count

This report still does not allow us to compare different snapshots, because we assume that we can treat the regular date dimension as another snapshot dimension. This could be a valid assumption because the balance fact table is a different kind of snapshot table. Nevertheless, this would not be the case if we had a sales or movements fact table instead of the one with balances. We will present a better model that meets these requirements in the next section titled Transition Matrix. However, the distinct count measure is often required by end users to ease the analysis of data when grouping more periods both in snapshot and/or in regular Date dimensions, especially when we have a more complex cube with other dimensions and relationships with other measure groups.

Transition Matrix

Has any user ever asked you for how many cases a given attribute has changed between two dates? A typical question with customer segmentation could be “How many customers classified as type A in 2004 have been converted to type B in 2005?”

In such cases, Kimball suggests that we consider a snapshot fact table. This is a good approach, but in the UDM we cannot relate the same time dimension twice to a fact table without having two different date columns. One possible solution could be to generate the Cartesian product of the time dimension, but this could be very expensive in terms of storage space and processing time.

The many-to-many relationship allows us to duplicate the time dimension in the UDM without duplication of storage. By the term “transition matrix”, we will refer to a model that makes it possible to analyze the state transition of analyzed elements between two dates even when results are displayed in a pivot table report.

Business scenario

In general, every time we have a fact repeated over time against the same dimension members, we could analyze how related attributes change over time for that dimension member. A good example of it is the credit rating of a customer. In this scenario, we have a monthly update of the customer ratings and we are interested in analyzing their changes. Figure 76 shows the relational schema for this data.

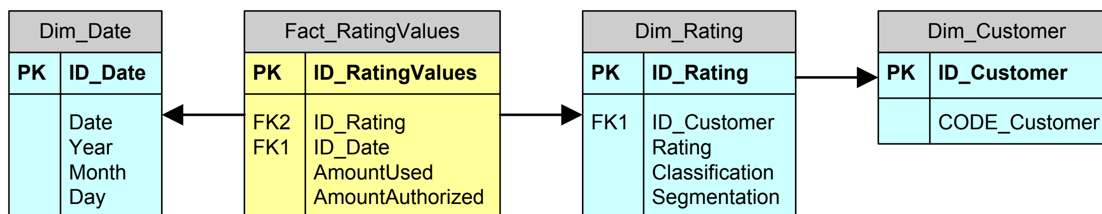


Figure 76 – Relational rating star schema

This star schema allows us to create a simple UDM, with Rating, Customer and Date dimensions. The fact table has two measures: the credit amount authorized and used for each customer on a specific date.

One potential source of confusion is that we would expect a regular star schema to have two completely independent dimensions for Rating and Customer, while our Customer dimension is a referenced dimension (a sort of a snowflake schema design) that relates to the fact table through the Rating dimension. However, there are good reasons to go for this design, as we will see next.

Fact_RatingValues is a sort of snapshot fact table with periodic (probably monthly) updates. For each snapshot, we have current amounts (AmountUsed and AmountAuthorized) and a related rating. Chances are that we will not update the customer rating frequently and regularly. We could have an original rating table with two date fields (DateStart and DateEnd) indicating the validity period of a certain rating (it ends when rating changes for any reason).

Similarly, we could create Dim_Rating with the same criteria, making it a Type II SCD. For the sake of simplicity, we do not have SCD canonical fields (DateStart, DateEnd, Current flag), because they are not relevant for the examples.

It is important to note that the ETL must ensure the consistency between those SCD dates and the Fact_RatingValues snapshot, because it contains the only date used by our analysis. Table 8 shows the history of rating changes used in our scenario. For the sake of simplicity again, AmountUsed and AmountAuthorized values will be respectively 50 and 100 for all Fact_RatingValues rows.

DateStart	DateEnd	Customer	Rating	Classification	Segmentation
01/01/2005	15/04/2005	Frank	AAB	Class A	Retail
16/04/2005	21/06/2005	Frank	AAB	Class A	Affluent
22/06/2005		Frank	AAC	Class A	Affluent
01/01/2005	14/03/2005	Mark	AAA	Class A	Private
15/03/2005	08/05/2005	Mark	AAA	Class A	Affluent
09/05/2005		Mark	AAB	Class A	Affluent
01/01/2005	11/02/2005	Paul	AAA	Class A	Private
12/02/2005	17/05/2005	Paul	AAB	Class B	Private
18/05/2005		Paul	AAB	Class C	Private

Table 8 – Ratings used for sample data in transition matrix model

Rating, Classification and Segmentation are independent attributes that are customer-related and might change over time. Our goal is to analyze the customer changes from one attribute state to another over time. The typical question we would like to answer is “How many customers with rating AAA have been downgraded to AAB from January to June?”

This scenario is very similar to the survey model we have discussed before. The main difference is that we have two classes of dimensions to “duplicate” in query (Date and Rating) as we can see in the following code with a possible SQL solution.

```
SELECT COUNT(*)
FROM Fact_RatingValues rv1
INNER JOIN Dim_Rating r1
   ON r1.ID_Rating = rv1.ID_Rating
INNER JOIN Dim_Date d1
   ON d1.ID_Date = rv1.ID_Date
INNER JOIN Dim_Rating r2
   ON r1.ID_Customer = r2.ID_Customer
INNER JOIN Fact_RatingValues rv2
   ON r2.ID_Rating = rv2.ID_Rating
INNER JOIN Dim_Date d2
   ON d2.ID_Date = rv2.ID_Date
WHERE d1.ID_Date = ‘20050131’
  AND d2.ID_Date = ‘20050630’
  AND r1.Rating = ‘AAA’
  AND r2.Rating = ‘AAB’
```

Another common request could be to produce a transition matrix of ratings from January to June: the following SQL statement shows the PIVOT SQL query that generates the results in Table 9.

```
SELECT
  RatingJanuary AS Rating,
  [AAA], [AAB], [AAC]
FROM (
    SELECT
       r1.Rating AS RatingJanuary,
       r2.Rating AS RatingJune
    FROM Fact_RatingValues rv1
    INNER JOIN Dim_Rating r1
       ON r1.ID_Rating = rv1.ID_Rating
    INNER JOIN Dim_Date d1
       ON d1.ID_Date = rv1.ID_Date
    INNER JOIN Dim_Rating r2
       ON r1.ID_Customer = r2.ID_Customer
    INNER JOIN Fact_RatingValues rv2
       ON r2.ID_Rating = rv2.ID_Rating
    INNER JOIN Dim_Date d2
       ON d2.ID_Date = rv2.ID_Date
    WHERE d1.ID_Date = '20050131'
      AND d2.ID_Date = '20050630'
) AS Ratings
PIVOT (
  COUNT(RatingJune)
  FOR RatingJune IN ( AAA, AAB, AAC)
) AS PivotTable
```

Rating	AAA	AAB	AAC
AAA	0	2	0
AAB	0	0	1

Table 9 – Transition matrix with SQL

Each row represents a rating in January and each column shows a rating in June. We can say that two customers who were rated AAA in January (Mark and Paul) have been declassified to AAB in June (even if the date of the rating change could be anywhere between these two dates). Moreover, the customer who was rated AAB in January (Frank) has been declassified to AAC in June. Tough times for credit ratings!

As before, SQL solutions are not very flexible or easy to build for an end user. A pivot table that can generate the results shown in Table 9 would be greatly appreciated.

Implementation

We will implement a UDM with three dimensions: Date, Customer and Rating. These dimensions correspond to the dimension tables shown in Figure 76.

The cube needs to duplicate some dimensions: we need two Date dimensions and two Rating dimensions. The end user would select two dates to see two different ratings, one for each selected date. We will use again the technique discussed in the Survey model, where we used role-playing dimensions. Then we will change relationships with the measure groups by using views representing different and independent bridge tables, although they duplicate the same original data.

As in the Survey model, we need to maintain the relationship between the selected attributes (ratings in this case) of the same customer. For this reason, we create two named views that we will use as bridge tables between the Customer and the Rating dimensions.

```
SELECT
  ID_Rating,
  ID_Customer
FROM Dim_Rating
```

These views use Dim_Rating, which is already a sort of bridge table, because it relates rating and customers.

As we have two independent measure groups (fact tables) between Customer and Rating dimensions, we also need two independent measure groups between Rating and Date dimensions. Before going further, look at Figure 77 to get a complete picture of the model we are going to build.

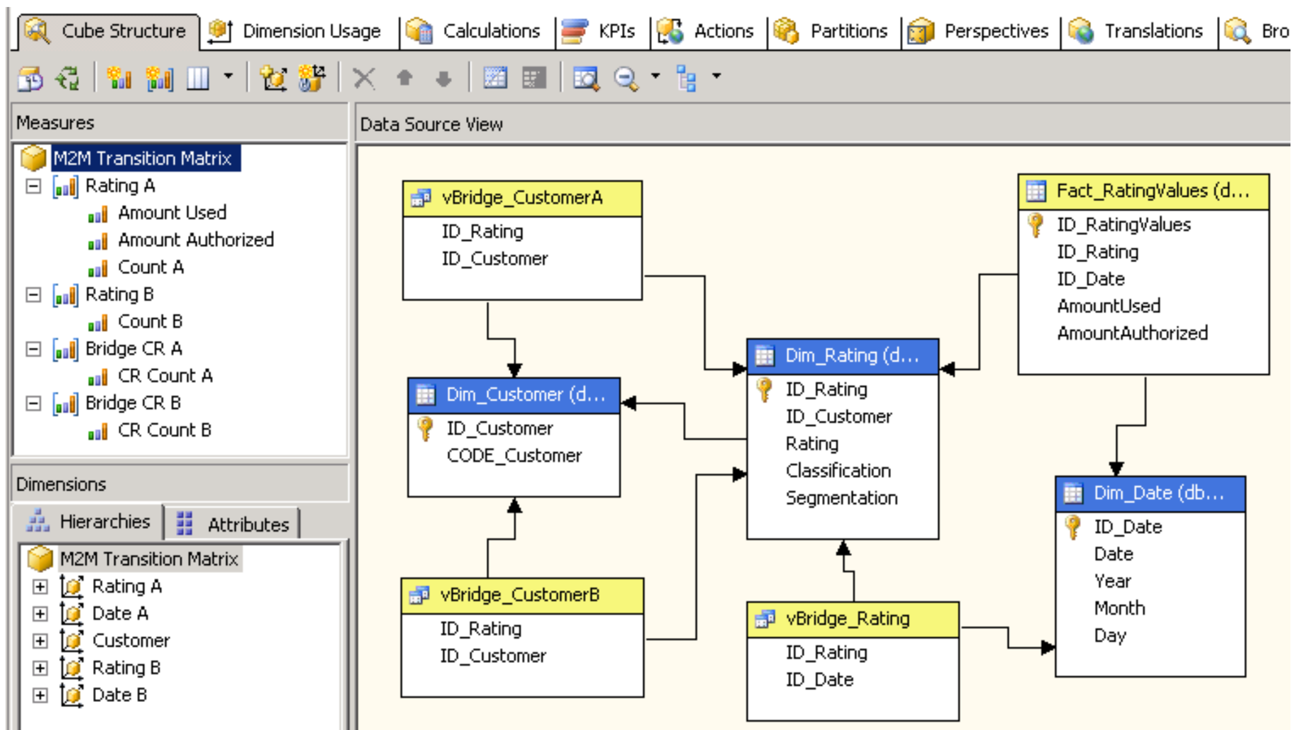


Figure 77 – Transition Matrix cube structure

While we already have Fact_RatingValues fact table to relate Rating and Date, we need another named view (vBridge_Rating, defined in side box) to get an independent alternative relationship between these dimensions.

```
SELECT
  ID_Rating,
  ID_Date
FROM Fact_RatingValues
```

We do not need to duplicate the other Fact_RatingValues measures (AmountUsed and AmountAuthorized).

The two views discussed above are vBridge_CustomerA and vBridge_CustomerB. They serve as a source for the Bridge CR A and Bridge CR B measure groups. These measure groups contain only one measure (row count) that will be useful to define the required many-to-many relationships. We need Count A and Count B measures, used in Rating A and Rating B measure groups, to make many-to-many relationships work.

Why do these views, measure groups and dimensions have suffixes A and B? To understand this, look at Figure 78.

	Measure Groups			
Dimensions	Rating A	Rating B	Bridge CR A	Bridge CR B
Rating (Rating A)	Dim Rating	Bridge CR B	Dim Rating	Bridge CR A
Date (Date A)	ID_Date	Bridge CR B	Rating A	Bridge CR A
Customer	Rating A	Rating B	Customer Code	Customer Code
Rating (Rating B)	Bridge CR A	Dim Rating	Bridge CR B	Dim Rating
Date (Date B)	Bridge CR A	ID_Date	Bridge CR B	Rating B

Figure 78 – Transition Matrix dimension usage

We can figure A and B as two axes in our transition matrix: if A is the starting point (i.e. the rating on a certain date), then B is the ending one (the rating on another date). The same suffix is used in measure groups and role-playing dimensions. The Rating A measure group has a direct relationship with the Rating A and Date A dimensions. It also has a referenced dimension relationship with Customer through the Rating A dimension and has many-to-many relationships with the Rating B and Date B dimensions. The opposite is true for the Rating B measure group (it has a regular relationship with B-suffixed dimensions and many-to-many relationships with A-suffixed dimensions).

The other two measure groups work as links between the Rating and Customer dimension: it is very important to observe carefully the relationships used in this case. The bridge measure groups (Bridge CR *) have to connect the rating of a customer with all the dates in which this rating exists for that customer. Thus, Bridge CR A has a direct relationship with Rating A and Customer dimension, while the many-to-many relationship with Date A has to be defined through the Rating A measure group. Likewise, Bridge CR B goes directly to the Rating B and Customer dimensions and goes to Date B through the Rating B measure group.

All relationships between a measure group and dimensions with a different suffix must be many-to-many relationships using a Factless CR measure group as an intermediate one. The rating measure groups use a Factless CR with the same suffix, while Factless measure groups use the other Factless CR measure group as intermediate one. If you are starting to get confused, remember the rule of thumb we suggested for dealing with cascading many-to-many scenarios: we have to choose the intermediate measure group that is nearest to the measure group [we are starting from], considering all the possible measure groups that we can cross going from the measure group to the dimension of interest. If we look at the data source view in Figure 77 following the links between dimensions and measure groups, it should be easy to apply this rule.

As we have seen in Figure 77, only the Rating A measure group has Amount-* measures: if we want to name measures more specifically than using A and B suffixes (e.g. with “Start” end “End”), then it could be a good idea to duplicate the Amount-* measures in both measure groups, just to give more flexibility to the end user.

Here is a potential source of confusion when querying the cube. When we use role-playing dimensions, we can change the dimension name but not internal hierarchies and/or attributes names. If the OLAP browser does not show the dimension name near the attribute, the end user will have to pay attention to which attributes he is using in order to understand the meaning of results: this is (sadly) the case for Excel. You have to duplicate the physical dimension instead of using the role-playing dimensions if you want to change hierarchies and/or attribute names between these twin dimensions.

Figure 79 shows the resulting report with Rating A on columns, Customer and Date A (level Months) on rows and the Amount Authorized measure in the data area. This report shows the underlying data at the maximum granularity. We can see all snapshots available for all customers, where the amount is displayed in the column corresponding to the rating that customer has on the month for the matching row.

	A	B	C	D	E
1	**Amount Authorized**				
2		**AAA**	**AAB**	**AAC**	**Grand Total**
3	**Mark**				
4	1	100			100
5	2	100			100
6	3	100			100
7	4	100			100
8	5		100		100
9	6		100		100
10	**Paul**				
11	1	100			100
12	2		100		100
13	3		100		100
14	4		100		100
15	5		100		100
16	6		100		100
17	**Frank**				
18	1		100		100
19	2		100		100
20	3		100		100
21	4		100		100
22	5		100		100
23	6			100	100
24	**Grand Total**	**500**	**1200**	**100**	**1800**

Figure 79 – Amount Authorized by Customer, Month and Rating

So far, we have not used the B-suffixed dimensions. We will do so in the following example.

In Figure 80, we get all the answers for our initial business scenario. The filter area contains January (selected from the Date A dimension) and the columns contains Rating attribute from the Rating A dimension. The rows are a Cartesian product of the months and ratings available from the Date B and Rating B dimensions. For each month, there is a row for each possible rating. The meaning of each data cell is "how many customers with [column rating] in January have [row rating] in [row month name]". Obviously, January rows have the same rating as shown in the corresponding columns. We can see that June has two original AAA customers who turned into AAB and one AAB who turned into AAC: this is the same data presented in Table 9, except that the meaning of the rows and columns is inverted.

Date A.Year - Month - Date	1	
Count A		
	AAA	**AAB**
1		
AAA	2	
AAB		1
2		
AAA	1	
AAB	1	1
3		
AAA	1	
AAB	1	1
4		
AAA	1	
AAB	1	1
5		
AAB	2	1
6		
AAB	2	
AAC		1
Grand Total	**2**	**1**

Figure 80 – Transition Matrix of Ratings between January and other months

Besides allowing us to use a pivot table for transition matrix queries, our model also simplifies direct queries: the next listing shows the MDX query that produces the same result shown in Table 9, but it has a syntax that is much easier to write and understand than the SQL one we used before.

```
SELECT
  [Rating B].Rating.Rating ON 0,
  NON EMPTY [Rating A].Rating.Rating ON 1
FROM [M2M Transition Matrix]
WHERE (
  [Date A].[Month].[1],
  [Date B].[Month].[6],
  Measures.[Count A] )
```

The only difference is that the query returns null values instead of zeros.

Multiple Parent/Child Hierarchies

Parent-child dimensions are a useful feature of Analysis Services. We can use them to model hierarchical and fast changing dimensions like sales or employee organizations. A very annoying limitation of this feature is that you can define only one parent-child hierarchy in a dimension. In the real world, this can be an issue.

For example, in the middle of a company reorganization, someone might need to analyze alternatively the present with the eyes of the past (actual data for previous organization hierarchy) and the past with the eyes of the present (past data for actual organization hierarchy).

The “multiple hierarchies” is a model that uses the many-to-many relationship feature: it allows us to assign a single leaf member to many different logical parent-child hierarchies, by using a single physical parent-child hierarchy.

Business scenario

As usual, let us explore a business scenario that applies to many different situations. In this case, we have a branch organization and we want to show it under different hierarchies: one is the regional hierarchy, which has an impact on logistical and technical aspects; the other is the market hierarchy, responsible for customer market analysis. Table 10 shows us the hierarchies associated with each company branch.

Branch	Regional Hierarchy	Market Hierarchy
NY01	North West	Corporate / Enterprise
NY02	North West / New York	Corporate / Small
NY03	North West / New York	Private
BO01	North West	Corporate / Small
BO02	North West / Boston	Corporate / Small
BO03	North West / Boston	Private

Table 10 – Branch Hierarchies

Oftentimes, we may face the issue that the hierarchies change over time, both in the number of hierarchies and in the hierarchy’s data. As usual, we do not want to change the relational and dimensional schemas each time we need to modify these logical views.

The relational model to work around this issue is relatively simple: Figure 81 shows that we have a Dim_Branch dimension table that contains real branches and a Dim_BranchHierarchies dimension table that associates each Dim_Branch row to many different hierarchies. As always, the ETL process must maintain data integrity in these tables.

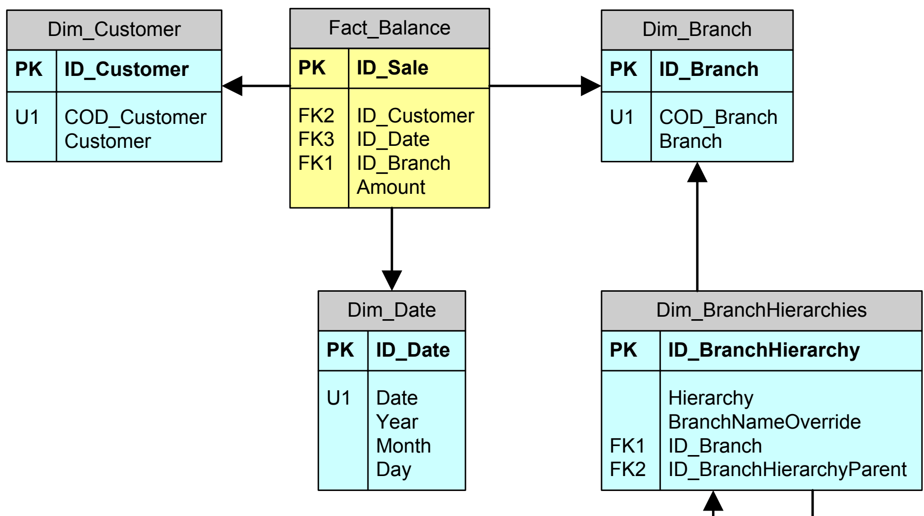


Figure 81 – Relational schema for Multiple Hierarchies

The Dim_BranchHierarchies table defines different hierarchies for the same branch. We can obtain a single parent-child hierarchy by filtering the rows in Dim_BranchHierarchies for a specific Hierarchy attribute value.

A Dim_Branch row should exist only once for each hierarchy and a row in Dim_BranchHierarchies must have a reference to an ID_Branch only for the leaf members of the hierarchy. Intermediate nodes might not have a corresponding ID_Branch: in this case, it is necessary to define a value for BranchNameOverride, which is the name to give to the node. If we do not define BranchNameOverride, the node/leaf name is the branch name referenced by ID_Branch.

To understand better how this model is populated, we can look at the dimensions members. Table 11 shows the Dim_Branch data. As explained, we have only branches that have corresponding measures in Fact_Balance and it is not necessary to have rows for the intermediate nodes of a hierarchy.

ID_Branch	COD_Branch	Branch
1	NY01	NY01
2	NY02	NY02
3	NY03	NY03
4	BO01	BO01
5	BO02	BO02
6	BO03	BO03

Table 11 – Dim_Branch content

The two hierarchies are in Dim_BranchHierarchies, as shown in Table 12. We use the field BranchNameOverride only for intermediate hierarchy nodes, even if it could be useful to rename a leaf name for a particular hierarchy too. When its value is NULL then we use the branch name referenced by ID_Branch as the name of the item in the hierarchy.

ID_Branch-Hierarchies	Hierarchy	BranchName-Override	ID_Branch	ID_Branch-HierarchyParent
1	Regional		1	7
2	Regional		2	8
3	Regional		3	8
4	Regional		4	7
5	Regional		5	9
6	Regional		6	9
7	Regional	North West		
8	Regional	New York		7
9	Regional	Boston		7
10	Market		1	17
11	Market		1	18
12	Market		2	19
13	Market		3	18
14	Market		4	18
15	Market		5	19
16	Market	Corporate		
17	Market	Enterprise		16
18	Market	Small		16
19	Market	Private		

Table 12 – Dim_BranchHierarchies content

To keep things simple, each branch has a single customer and a single balance amount for each date and the balance amount is the same for all customers on that date (Table 12 shows the same balance amount of 100 for January).

Our goal is to query a UDM that has many parent-child hierarchies for the Branch dimension. The query results should automatically show only the members of the selected hierarchy.

Implementation

The relational model shown in Figure 81 is close to the envisioned UDM. However, Analysis Services does not directly support the relationship that exists between Dim_Branch and Dim_BranchHierarchies, because we have a one-to-many relationship while only many-to-one relationships can be implemented through referenced dimensions.

We already solved an analogous problem in the Multiple Groups scenario using a many-to-many relationship through the expanded model shown in Figure 82. We define a view that corresponds to Bridge_BranchHierarchies in our model.

```
SELECT
    ID_BranchHierarchy,
    ID_Branch
FROM Dim_BranchHierarchies
WHERE ID_Branch IS NOT NULL
```

This query filters all rows that do not relate to a branch: these rows in our sample are all intermediate nodes in the hierarchies.

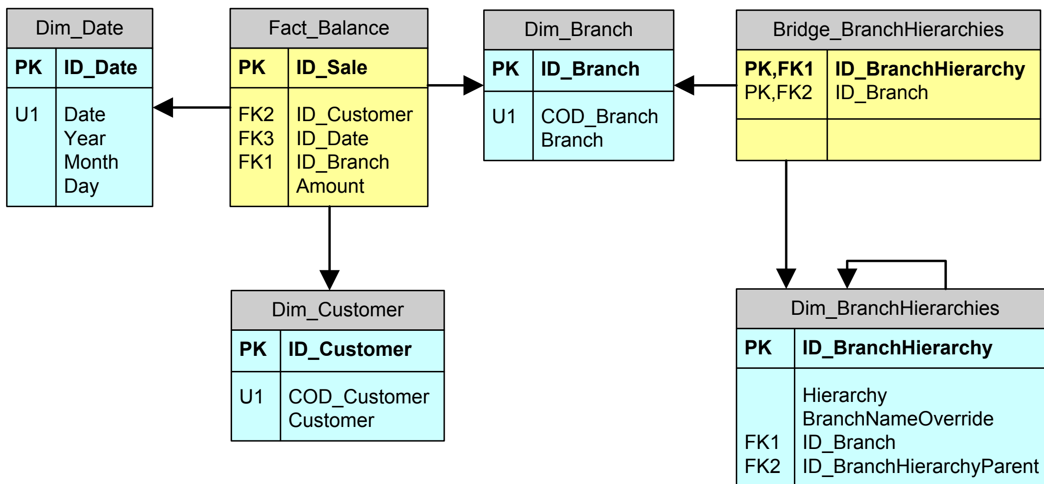


Figure 82 – Relational schema expanded for Multiple Hierarchies

We have four dimensions in our UDM: the only one that deserves more attention is the one based on Dim_BranchHierarchies. First, we create a named view called vDim_BranchHierarchies, which we will use as a source for the dimension.

```
SELECT
    h.ID_BranchHierarchy,
    h.ID_Branch,
    h.ID_BranchHierarchyParent,
    h.Hierarchy,
    COALESCE (h.BranchNameOverride, b.COD_Branch) AS COD_Branch,
    COALESCE (h.BranchNameOverride, b.Branch) AS Branch
FROM Dim_BranchHierarchies h
LEFT JOIN Dim_Branch b
  ON h.ID_Branch = b.ID_Branch
```

This is necessary because we have to define the name of all the nodes and leaves for all the hierarchies. Our relational model defines an override mechanism for the node name that has to be resolved for Analysis Services.

Then, we create the dimension Branch Hierarchies shown in Figure 83. It has a single parent-child hierarchy (Hierarchized Branch) and an attribute (Hierarchy) containing the names of the available hierarchies. Filtering by this attribute, users will be able to see only nodes and leaves of the desired hierarchy.

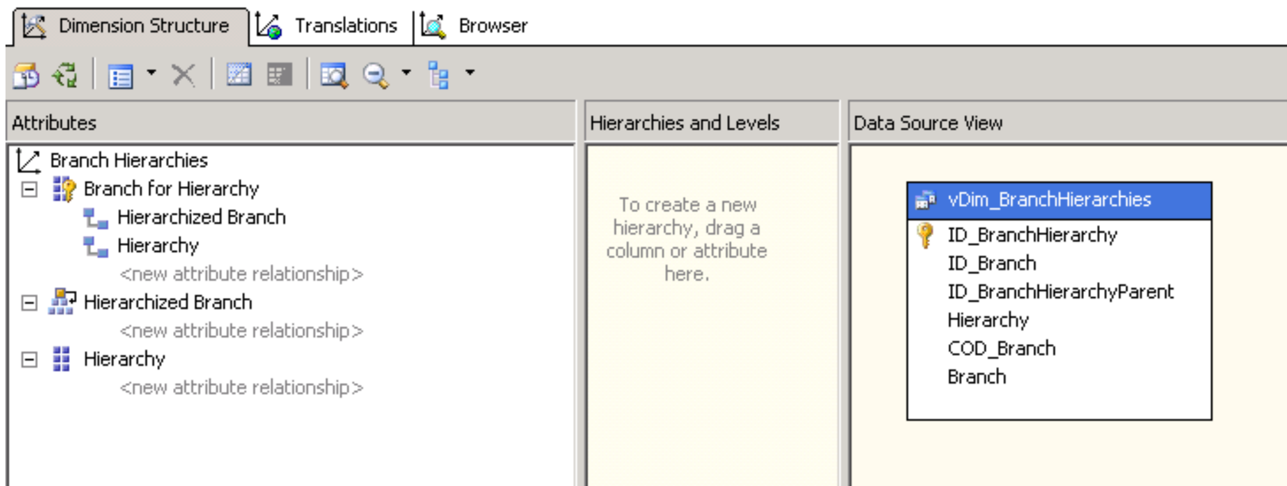


Figure 83 – Dimension Branch Hierarchies

It is important to understand what keys and names we use for the dimension attributes. Table 13 shows the most important properties used. Note that the InstanceSelection property for Hierarchy attribute should be set to MandatoryFilter (so that a client application that supports this property could suggest the selection of a member to end users).

Attribute Name	Hierarchy Visible	Usage	Key Column	Name Column
Branch for Hierarchy	False	Key	ID_BranchHierarchy	Branch
Hierarchized Branch	True	Parent	ID_BranchHierarchyParent	(none)
Hierarchy	True	Regular	Hierarchy	(none)

Table 13 – Dim_BranchHierarchies attributes properties definition

Figure 84 shows that by browsing the Hierarchized Branch hierarchy we can get all the hierarchies: Corporate and Private are top-level nodes for the Market hierarchy, while North West is the only top-level node for the Regional hierarchy. If we filter by the Hierarchy attribute, then we can only browse the nodes of the selected hierarchy.

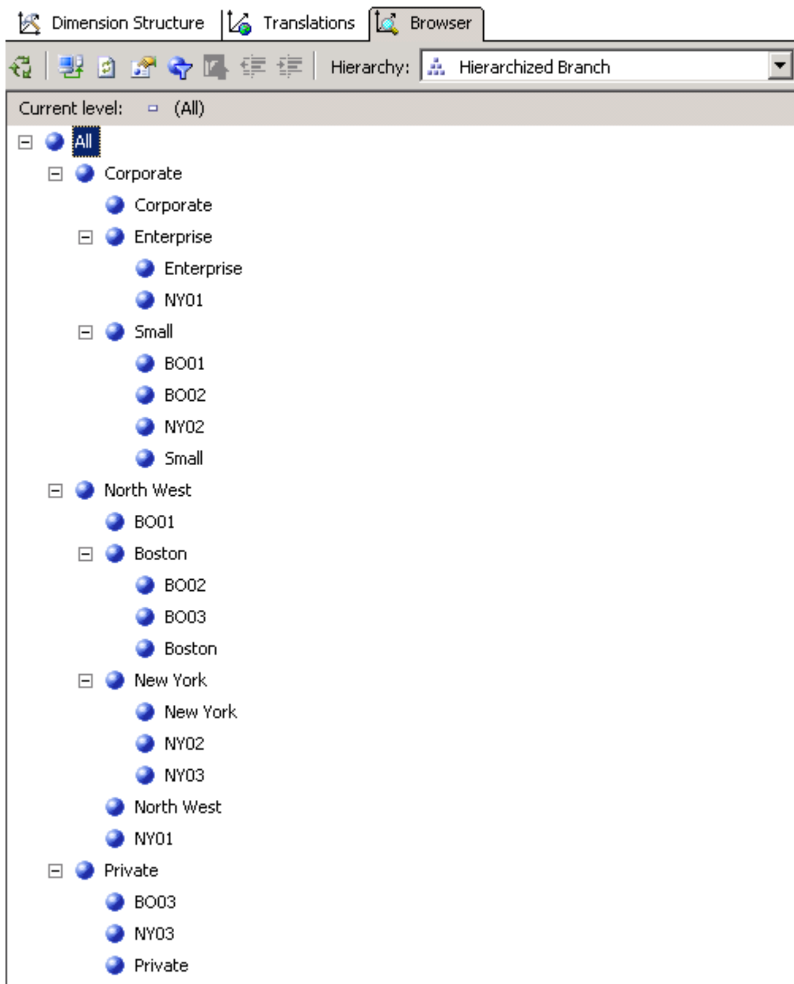


Figure 84 – Browse Branch Hierarchies dimension without filtering

The cube definition shown in Figure 85 has two measure groups, one for the facts (Fact Balance) and the other (Branch Hierarchies) for the many-to-many relationship between Branch and Branch Hierarchies dimensions.

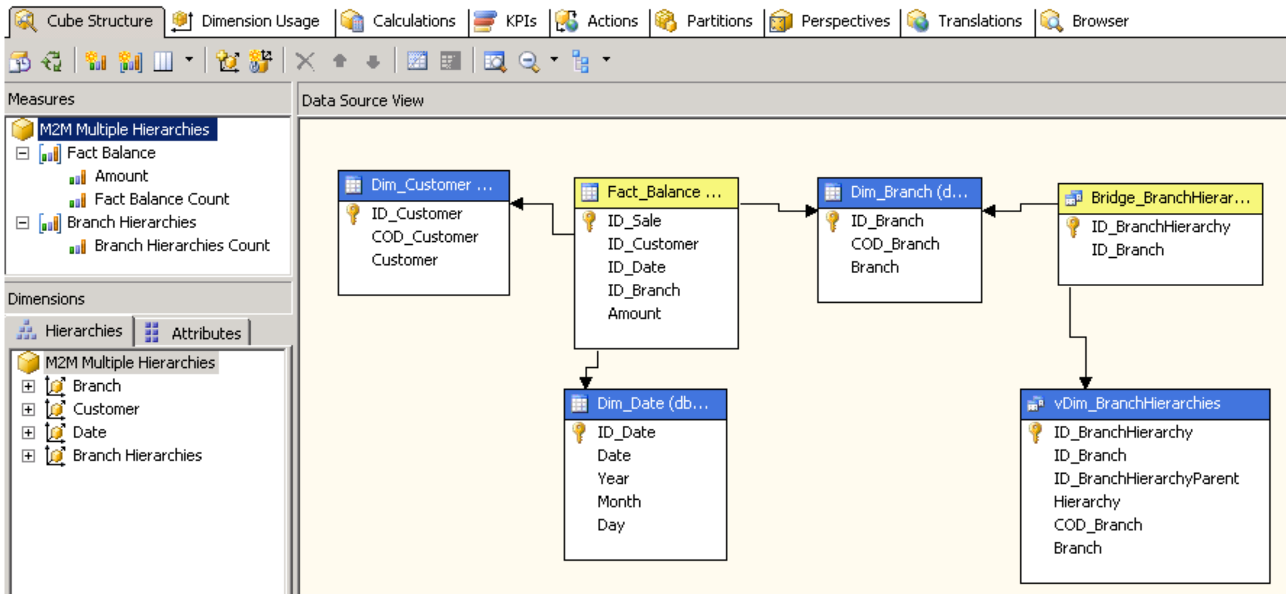


Figure 85 – Multiple Hierarchies Cube Structure

At this point, the dimension usage shown in Figure 86 should be easy to grasp. Since the Branch Hierarchies measure group has only a hidden measure, we do not need any relationship between this measure group and the Customer and Date dimensions. This is why the Branch Hierarchies measure group has only two regular relationships with Branch and Branch Hierarchies dimension. The Fact Balance measure group has a many-to-many relationship with the Branch Hierarchies dimension using the Branch Hierarchies measure group as intermediate.

Dimensions	Measure Groups: Fact Balance	Branch Hierarchies
Branch	Branch	Branch
Customer	Customer	
Date	Date	
Branch Hierarchies	Branch Hierarchies	Branch for Hierarchy

Figure 86 – Multiple Hierarchies Dimension Usage

Browsing the cube with a NON EMPTY clause give us the expected results. In Figure 87 we show three queries made to the cube.

	A	B	C	D	E	F	G	H	I	J
1	Year - Month - Date	1			Year - Month - Date	1			Year - Month - Date	1
2					Hierarchy	Market			Hierarchy	Regional
3	**Row Labels**	**Amount**								
4	**Market**	**600**			**Row Labels**	**Amount**			**Row Labels**	**Amount**
5	**Corporate**	**400**			**Corporate**	**400**			**North West**	**600**
6	Enterprise	100			**Enterprise**	**100**			**BO01**	**100**
7	NY01	100			NY01	100			**Boston**	**200**
8	Small	300			**Small**	**300**			BO02	100
9	BO01	100			BO01	100			BO03	100
10	BO02	100			BO02	100			**New York**	**200**
11	NY02	100			NY02	100			NY02	100
12	**Private**	**200**			**Private**	**200**			NY03	100
13	BO03	100			**BO03**	**100**			**NY01**	**100**
14	NY03	100			**NY03**	**100**			**Grand Total**	**600**
15	**Regional**	**600**			**Grand Total**	**600**				
16	**North West**	**600**								
17	BO01	100								
18	Boston	200								
19	BO02	100								
20	BO03	100								
21	New York	200								
22	NY02	100								
23	NY03	100								
24	NY01	100								
25	**Grand Total**	**600**								

Figure 87 – Sample queries using multiple hierarchies

The first query has Brand Hierarchy Name as the first row field and Branch Hierarchies immediately next. It is easy to see that the two hierarchies show the same data (the total is always 600) even if the same row (B001 for example) appears more than once.

Moving Brand Hierarchy Name to the filter area (i.e. the second and third queries), we are able to see the "multiple hierarchies" as one single hierarchy. The filtering for the Hierarchy Name does the magic making multiple hierarchies available in SSAS.

Using Hierarchy Name as a filter seems to be the best option; nevertheless, we have to be aware of client limitations. If, for example, we use Excel to navigate through the multiple hierarchies, we will soon discover that the OLAP browser is not smart enough to filter dimension browsing when we already put a filter on the table, as you can see in Figure 88.

Figure 88 – Excel does not filter dimension correctly

It is easy to see that Excel mixes the nodes from both hierarchies in the field selection window, forcing on the user a very non-intuitive experience. The problem resides in the client capabilities and not in the cube structure. Nevertheless, this is a problem and we need to face and solve it.

Note that if we define a subcube by applying a dimension filter in the area above pivot table using BIDS (see Figure 89), the Cube Browser will show the filtered members only, but this does not happen if you place the filter in the filter area, because of a different filtering implementation.

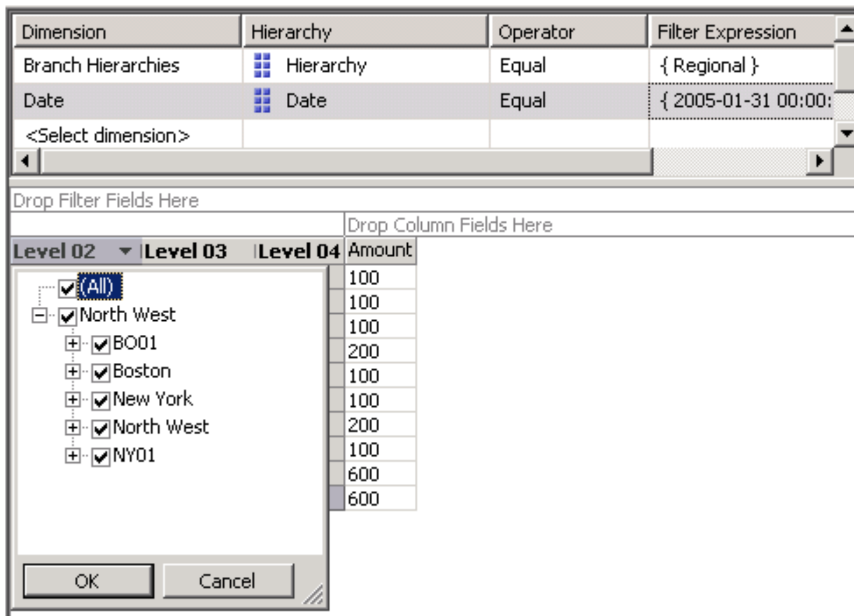


Figure 89 – Dimension filter lets filter dimension browsing too

In the case the OLAP browser does not support subcube filtering, or it may be too complex for the end user to use this feature, we could define a single top-level member for each hierarchy that corresponds to the name of the hierarchy. We always use this strategy when we need to facilitate the use of “legacy” clients, like Excel 2003, with this model.

The Multiple Hierarchies model allows the creation of a new hierarchy without the need to restructure either the relational or the dimensional schemas. In addition, changes only need an incremental update of the Branch Hierarchies dimension and measure group. It is interesting for such on-the-fly modifications as those suggested at the end of the Multiple Groups scenario: you could provide a user interface to create new hierarchies that become available on the server immediately for all users, without the need for a full process.

One very important note regards unary operators. When we define a parent/child hierarchy in SSAS, we have the option of choosing as an attribute as the unary operator that defines how SSAS aggregates the values on the parent/child hierarchy. You can choose to sum up some nodes, subtract others and so on. Clearly, we do not want to explain here how the unary operator feature works, if you need it then you already know it.

What is important is that, due to the internal architecture of SSAS, the unary operator does not work correctly when many-to-many relationships relate a fact table with the hierarchy we realized. This is true even for SQL Server 2008. We will show shortly a new and much more complex model that will let you define unary operators with multiple hierarchies and gives some more advantages over unary operator. Nevertheless, we always need to be aware that the model of simple multiple hierarchies and the unary operator are two concepts that cannot be mixed. If you need the unary operator, carefully read the next chapter “Hierarchy Reclassification with Unary Operator” to learn how to implement unary operator with multiple hierarchies.

Hierarchy Reclassification with Unary Operator

Balance reclassification is a very common need when we have a balance sheet that is useful for accounting, but we would like to see it under different hierarchies as each hierarchy gives us a different view of the same data and is good for different analyses. Balance reclassification is a variation of the Multiple Hierarchies model, but hides some subtle differences. It is a model more complex than multiple hierarchies are because, in balance sheets, we normally have to deal with the unary operator feature of Analysis Services. SSAS uses the unary operator to determine how a specific balance value should be aggregated on its parent. Unfortunately, unary operators cannot be used when you make calculation over a many-to-many relationship, which is used to build the multiple hierarchies model.

Business scenario

Suppose we have a customer with a solution that let him investigate on his balance sheet with excel worksheets like the one in Figure 90.

	A	B	C
1	**Amount**	**Column Labels**	
2	**Row Labels**	**2006**	**2007**
3	**Expenses**	**506.25**	**759.45**
4	**Office**	**168.75**	**253.15**
5	Employees	70.31	105.48
6	Licenses	56.25	84.38
7	Marketing	42.19	63.29
8	**SQL Server**	**168.75**	**253.15**
9	Employees	70.31	105.48
10	Licenses	56.25	84.38
11	Marketing	42.19	63.29
12	**Visual Studio**	**168.75**	**253.15**
13	Employees	70.31	105.48
14	Licenses	56.25	84.38
15	Marketing	42.19	63.29
16	**Revenues**	**750.00**	**1,125.00**
17	**Office**	**225.00**	**337.50**
18	Courses	125.00	187.50
19	Licenses	100.00	150.00
20	**SQL Server**	**225.00**	**337.50**
21	Licenses	100.00	150.00
22	Technical Support	125.00	187.50
23	**Visual Studio**	**300.00**	**450.00**
24	Courses	75.00	112.50
25	Licenses	100.00	150.00
26	Technical Support	125.00	187.50
27	**Grand Total**	**243.75**	**365.55**

Figure 90 – Standard Balance Sheet

The cube structure is straightforward and we will discuss it later. The important thing to note is that our user wants to see positive numbers for expenses and revenues but, on the grand total, he clearly wants to subtract expenses from revenues, in order to see his final profits.

This is the perfect situation where the unary operator feature is good. Moreover, the positive sign for the expenses at the leaf level is mandatory, because we use the same numbers somewhere for other analyses that do not include the balance sheet.

Now, our user wants to have more than one hierarchy to analyze his balance sheet. He recognizes that he might be interested in a different hierarchy that shows the same data but starts with the product, in order to detect profits on a product basis (SQL, Office, Visual Studio). Moreover, he might be interested in a third hierarchy that starts with the leaves of the standard one to detect the total of expenses and revenues for employees, licenses and so on. More generally, we, as good BI analysts, want to give him a tool that lets him create any kind of hierarchy, rearranging the leaves of the original hierarchy under a new structure that he can build by his own.

The final solution we want to build has these characteristics:

- Each leaf of the original balance sheet may be moved anywhere in the new hierarchy

- Users can move only leaves, the intermediate nodes of the original hierarchy have no data associated with them and so we cannot use them to build a new hierarchy. We might want to relax this rule by providing the capability to move intermediate nodes but, for the sake of simplicity, we adopt it.

- Each leaf might change its sign in the new hierarchy (SQL 2005 expenses are positives in the original hierarchy, but we might want to aggregate them as negative on the product hierarchy, while retaining the values in the fact table as positive).

- The new hierarchy does not need to include all the leaves of the original hierarchy. If we consider a much more complex balance sheet, it is obvious that some kinds of analysis do not need to look at all the leaves. Some analysis might concentrate on a subset of data, ignoring the others and still being a good analysis.

- The user might decide that many leaves in the original hierarchy should be aggregated under a single node in the new hierarchy. In the previous example, the new dimension might include a “Development products” node that condenses both SQL 2005 and Visual Studio trees, without going down to the single leaf level. This can be useful to produce a highly aggregated hierarchy that shows only top nodes quickly.

- All the hierarchies will be contained in a single dimension. We will use the Multiple Hierarchies pattern to maintain all the user-defined hierarchies into a single dimension. Clearly, we will extend it to fulfill our new needs.

As we will see, this model is a derivation of the Multiple Hierarchies model, but introduces several levels of complexity and it needs a much more intricate UDM model. We will need to create an adequate ETL phase and so we will go deep in the description of some of the ETL steps needed to build the model.

The most common scenario for Hierarchy Reclassification is that of the balance sheet that we will describe in the implementation. However, we can use the same model to create reclassification whenever we face a parent/child hierarchy.

Before the technical details, look at the most classical balance sheet database model in Figure 91.

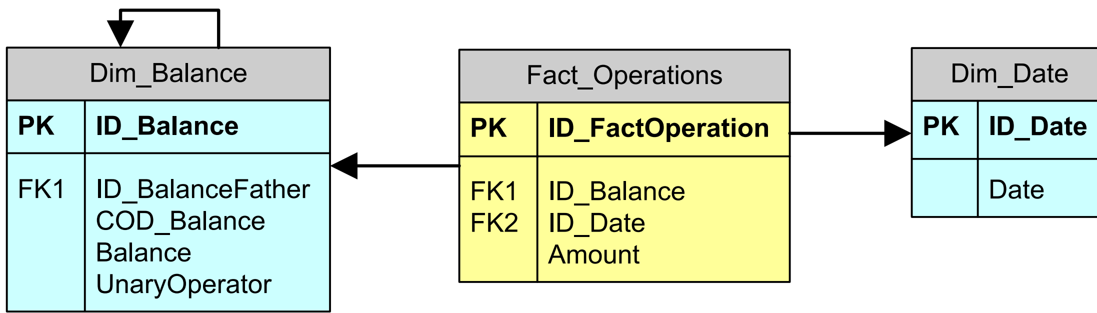


Figure 91 – Standard Balance Sheet model

We have a balance sheet with a parent/child hierarchy, the standard date dimension and some operations that belong to a specific balance node and that compose its total.

This simple model is useful for a single parent/child hierarchy. The UnaryOperator column in the Dim_Balance dimension is the field used as the unary operator in the parent/child specification.

Implementation

The implementation of the Balance Reclassification model needs to use a more complex structure, shown in Figure 92.

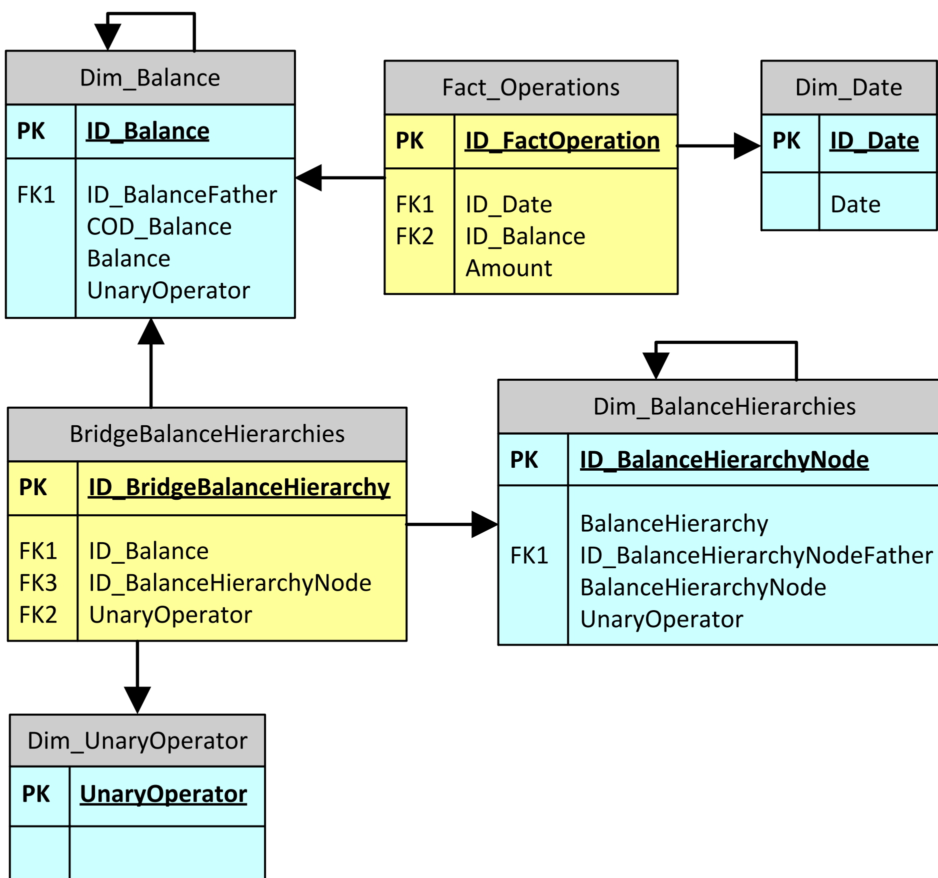


Figure 92 – Balance Reclassification Model

Let us review the structure in more detail:

- The bridge table acts as a bridge between the original balance sheet and the new dimension Dim_BalanceHierarchies that, following the Multiple Hierarchies model, will contain all the hierarchies separated by the BalanceHierarchy column.

- We will not use Dim_Balance's UnaryOperator column, we will use the column with the same name in Dim_BalanceHierarchies instead, with the same meaning but without the SSAS built-in mechanism to handle unary operators, which do not work with many-to-many relationships.

- As one of the requisites of the model is that some of the leaves in Dim_Balance appear under only one node of the new hierarchy, it is possible that many rows in the bridge table reference only a single row in Dim_BalanceHierarchies, for one single hierarchy. This could be useful in the Multiple Hierarchies pattern, but in this model it is not only an option, it is one of the fundamentals of the model. Keep it in mind, as it will be useful later.

- Moreover, we need a unary operator in the bridge table in order to define if the balance row will be summed to or subtracted from the total of the BalanceHierarchy node. The Dim_UnaryOperator dimension contains only two rows: one for SUM and one for SUBTRACT.

- The careful reader should remember that we already said that unary operators do not work when used with many to many relationships. In fact, the UnaryOperator columns of both Dim_Balance and Dim_BalanceHierarchies are not used by this multidimensional model. We have introduced them to make things clearer, but we will end up with a model that does not rely on the UnaryOperator feature of SSAS. Instead, the Dim_UnaryOperator dimension will take care of the interpretation of unary operators, with the help of some MDX code. However, in the demo code we generate the UnaryOperator in the BridgeBalanceHierarchies view starting from the Dim_BalanceHierarchies UnaryOperator, by leveraging on a SQL query that expand the original hierarchy into the complete list of accounts (leaf nodes) that have to be aggregated for each hierarchy node.

One problem that was still open in the Multiple Hierarchies model is the fact that some clients do not filter dimensions based on other filters. For instance, even if we make a filter for BalanceHierarchy name, the client will still let us choose among all the nodes in the Dim_BalanceHierarchy parent/child hierarchy. In this model, we will address the problem and show the final solution that uses the hierarchy names as root nodes of the complete hierarchy.

Handling of the unary operator

The first problem we have to face is the fact that we cannot rely on the unary operator feature of Analysis Services. Thus, we will need to handle the complexity of unary operator by ourselves.

We have defined a specific dimension (Dim_UnaryOperator) that will let us define, through the relationship with the bridge table, which operation to use for each of the nodes from Dim_Balance. Using some simple MDX code, we can divide values from the fact table in SUM and SUBTRACT, compute the difference between them and the show the result to the user. Therefore, if the ETL computes correctly the values in the bridge table, the handling of the unary operator seems straightforward.

The problem is that we cannot associate a unique operation to a node, when aggregated at the balance hierarchy level. To understand the problem, suppose that you have a balance composed of only four independent nodes (C, D, F, G) and that we want to create on it a structure (A,B,E) like this:

Hierarchy Node	Balance	Remarks
A		A is the root node, it contains (B – E)
+B	-C	B is an intermediate node, it contains D – C
	+D	
-E	+F	E is an intermediate node, it contains F - G
	-G	

Table 14 – Balance reclassification formulas

It is quite easy to see that the sign of G, for example, is not definable “a priori”. When G is aggregated to E it needs to have the MINUS sign (the complete expression of E is (F-G)) while, when it is aggregated to A, it will have a PLUS sign (the complete expression of A is (D-C-F+G)).

We cannot mark G with a specific sign. Its sign depends on the path that we are traversing in the tree to reach it. SSAS unary operator handles all that but, as we cannot use that feature, we need to find a UDM model that solves the problem.

It will be very helpful that the fact that a single row in the hierarchy dimension can reference many rows in the balance sheet dimension and each of those rows can have a sign. If we expand, for each node in the hierarchy, the complete expression that computes its value from the balance sheet, we will be able to define a specific sign of each balance row when aggregated to each row in the hierarchy. In other words, the sign is unique when we know the Balance Hierarchy node and the Balance node.

In the following table, we show, for each node in the hierarchy, the expanded expression based solely on balance rows.

Hierarchy Node	Expression	SUM	SUBTRACT
A	D-C-F+G	D G	C F
+B	D-C	D	C
-E	F-G	F	G

Table 15 – Balance reclassification formulas expanded

Now, even if the G node does not have a specific sign through all the hierarchy (it appears both on the SUM and SUBTRACT columns), it does have a specific sign for a single hierarchy node. If, with this structure, we want to compute the value of the A node, we can take the sum of D and G and subtract from it the sum of C and F. We will show the MDX code that does this computation later.

What is important to understand at this point is the fact that, due to the lack of the unary operator feature, we will need to use the bridge table to hold the complete expanded expression of each hierarchy node along with the sign of each balance row when aggregated to that specific hierarchy node. Moreover, the ETL phase of this model gets complicated. We will need to write some recursive code to traverse the hierarchy tree and define, for each node, the complete expression.

We are now computing, for each hierarchy node, the complete value. Therefore, we need to tell SSAS not to perform any computation by itself. By default, SSAS will sum up children to their father, causing each node to have a much higher value than expected.

There is a unary operator that will tell SSAS to ignore the children during computation of the father: it is the tilde character “~”. Therefore, to make the hierarchy work as we want, we will need to define the unary operator value for all the nodes in the parent/child hierarchy to tilde.

Using SQL to expand the expressions

The formula expansion ETL and the configuration phases of this model are complex. We want to take a closer look at code that handles the balance reclassification.

In the configuration, we have a table (Config_BridgeBalanceHierarchies) that contains the associations between hierarchies and balance accounts stating, for each hierarchical leaf node, the balance accounts and the unary operator that will be used for the aggregation of the balance account into the hierarchical one.

This table should contain, among all the other hierarchies, a standard hierarchy named "BALANCE SHEET" that is a perfect copy of the original balance sheet hierarchy. This hierarchy is very useful for some purposes:

- It gives the user the ability to use only one parent/child structure to analyze both the standard and the user-defined hierarchies. We will see that this is mandatory: in order to make the model work, we will need to remove the parent/child hierarchy from the solution, avoiding any interference between the SSAS unary operator and our model.

- It must show exactly the data that we can check against the original hierarchy. We will extensively use it to verify that both the ETL and SSAS computations are correct.

The ETL that creates the hierarchies is straightforward and we will not describe it. The only complex part is the expansion of the formulas needed to compute each hierarchy node using the balance sheet nodes. We will need to traverse the tree and evaluate the related formula for each node.

Traversing the tree using SQL is easy using recursive queries. Nevertheless, we need a more complex algorithm, as we want to compute the final formula for each node of the tree.

The trick is to perform a complete traversal of the tree considering each single node as the root of a subtree that need to be completely traversed and flattened, in order to be able to reconstruct, for each single node, the complete formula that computes its value. The following is the definition of the BridgeBalanceHierarchies view used for the Bridge Balance Hierarchies intermediate measure group. As we said, the relationship between leaf nodes in Dim_BalanceHierarchies dimension and accounts in Dim_Balance are defined by the Config_BridgeBalanceHierarchies table, which might contain a list of accounts for a single node in the hierarchy. This is common in balance reclassification. However, remember that relationships for intermediate nodes in the hierarchy will be generated by this view. Remember this when you populate the Config_BridgeBalanceHierarchies table, otherwise you might obtain duplicated rows in the result of this view.

```
WITH
CombineOperators (Sign1, Sign2, ResultSign) AS (
    SELECT '+', '+', '+'
    UNION ALL SELECT '+', '-', '-'
    UNION ALL SELECT '-', '+', '-'
    UNION ALL SELECT '-', '-', '+'
),
BalanceTreeTraverser AS (
    SELECT
        ID_BalanceHierarchyRoot = B.ID_BalanceHierarchyNode,
        ID_BalanceHierarchyNode = B.ID_BalanceHierarchyNode,
        UnaryOperator           = B.UnaryOperator,
        BalanceHierarchyNode    = B.BalanceHierarchyNode
    FROM Dim_BalanceHierarchies B
    UNION ALL
    SELECT
        ID_BalanceHierarchyRoot = T.ID_BalanceHierarchyRoot,
        ID_BalanceHierarchyNode = B.ID_BalanceHierarchyNode,
        UnaryOperator           = (SELECT Cast (ResultSign AS CHAR (1))
                                    FROM CombineOperators
                                    WHERE Sign1 = B.UnaryOperator AND Sign2 = T.UnaryOperator),
        BalanceHierarchyNode    = B.BalanceHierarchyNode
    FROM
        BalanceTreeTraverser T
        INNER JOIN Dim_BalanceHierarchies B
           ON B.ID_BalanceHierarchyNodeFather = T.ID_BalanceHierarchyNode
          AND B.ID_BalanceHierarchyNodeFather <> B.ID_BalanceHierarchyNode),
ReconstructHierarchy AS (
    SELECT
        T.ID_BalanceHierarchyRoot,
        T.ID_BalanceHierarchyNode,
        B.ID_Balance,
        BalanceHierarchyNode      = T.BalanceHierarchyNode,
        UnaryOperator             = (SELECT CAST (ResultSign As Char (1))
                                      FROM CombineOperators
                                      WHERE Sign1 = B.UnaryOperator And Sign2 =
T.UnaryOperator),
        OB.COD_Balance,
        OB.Balance
    FROM
        BalanceTreeTraverser T
        LEFT OUTER JOIN Config_BridgeBalanceHierarchies B
            ON B.ID_BalanceHierarchyNode = T.ID_BalanceHierarchyNode
        LEFT OUTER JOIN Dim_Balance OB
            ON OB.ID_Balance = B.ID_Balance
        WHERE Balance IS NOT NULL
)
SELECT
    ID_BalanceHierarchyRoot,
    ID_Balance,
    ID_UnaryOperator
FROM
    ReconstructHierarchy
    LEFT OUTER JOIN DIM_UnaryOperator
        ON ReconstructHierarchy.UnaryOperator = Dim_UnaryOperator.UnaryOperator
```

The query seems complex but, in reality, it is a simple tree traversal query with the peculiarity of the ID_BalanceHierarchyRoot node that will identify, for each expanded expression, the starting node. We will use the results of this query in the BridgeBalanceHierarchies view, in order to link each node with the expanded expression that computes it. Take some time to understand the query before going on with the model, because this query is building the most complex table of the model itself.

Building the model

Using the tables described, the resulting UDM model is the one shown in Figure 93.

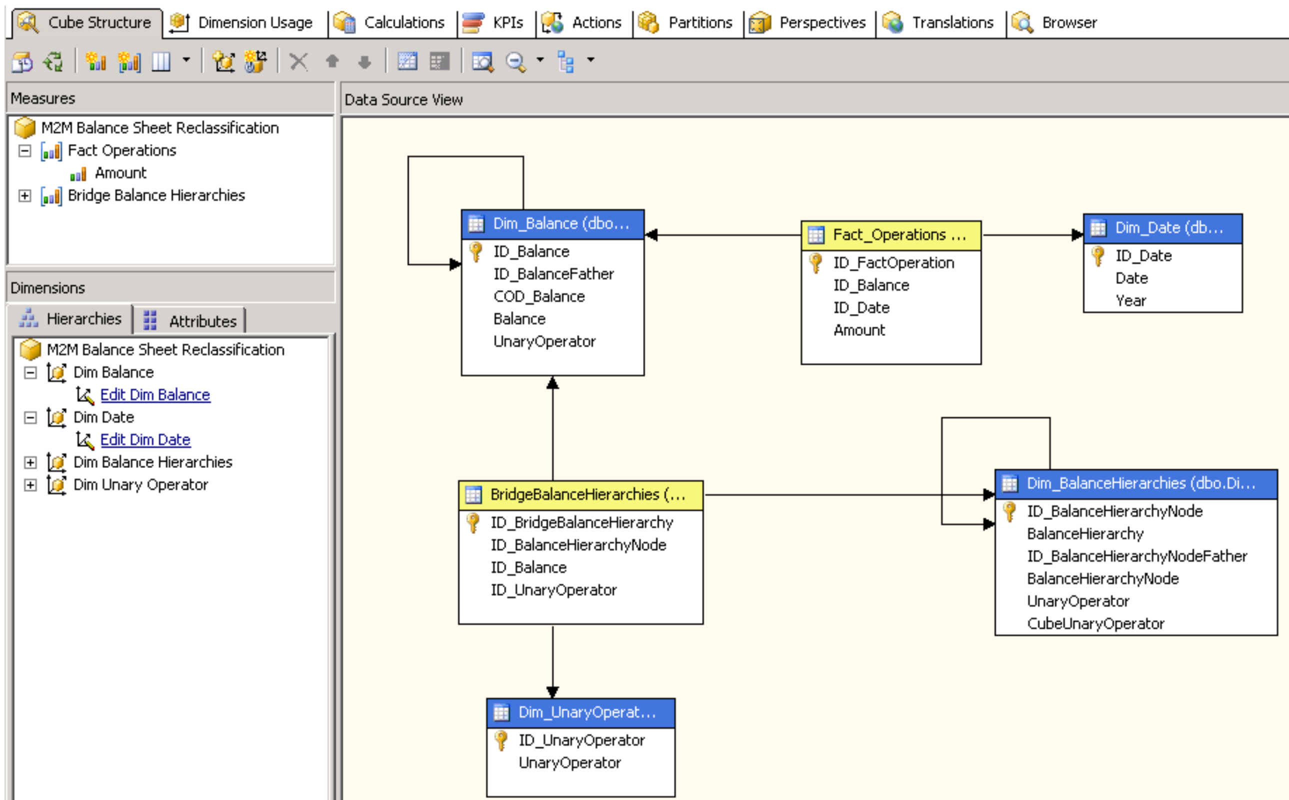


Figure 93 – Balance Reclassification UDM Model

This model is very similar to that of the Multiple Hierarchies. The main differences are:

- The Dim Unary Operator is used to divide between nodes to sum or subtract from the total
- The bridge table contains the complete expression that computes the total of each node in the Dim Balance Hierarchies.

The dimension usage pane is straightforward. We use the bridge table to link the fact table to both the balance hierarchy dimension and the unary operator. There is nothing strange here, just some many-to-many relationships like you can see in Figure 94.

	Measure Groups	
Dimensions	Fact Operations	Bridge Balance Hierarchies
Dim Balance	Dim Balance	Dim Balance
Dim Date	Dim Date	Fact Operations
Dim Balance Hierarc…	Bridge Balance Hierarc…	Dim Balance Hierarchies
Dim Unary Operator	Bridge Balance Hierarc…	Dim Unary Operator

Figure 94 – Balance Reclassification UDM Model

The last operation that we need to do is to set the unary operator for all the nodes in the Dim Balance Hierarchies dimension to "~". We have done it adding a column (CubeUnaryOperator) that evaluates to ~ for each node in the multiple hierarchies' dimension.

We are now able to query the cube analyzing the multiple hierarchies. In Figure 95 we put a filter on the year, used the Dim Balance Hierarchies on the row and put the Dim Unary Operator on the columns.

	A	B	C	D
1	Year	2007		
2				
3	**Amount**	**Column Labels**		
4	**Row Labels**	-	+	**Grand Total**
5	**BALANCE SHEET**	**759.45**	**1,125.00**	**1,884.45**
6	**Expenses**	**759.45**		**759.45**
7	Office		253.15	253.15
8	Employees		105.48	105.48
9	Licenses		84.38	84.38
10	Marketing		63.29	63.29
11	SQL Server		253.15	253.15
12	Visual Studio		253.15	253.15
13	**Revenues**		**1,125.00**	**1,125.00**
14	Office		337.50	337.50
15	Courses		187.50	187.50
16	Licenses		150.00	150.00
17	SQL Server		337.50	337.50
18	Licenses		150.00	150.00
19	Technical Support		187.50	187.50
20	Visual Studio		450.00	450.00
21	**GENERAL**	**759.45**	**1,125.00**	**1,884.45**
22	**Courses and Technical Support**		**675.00**	**675.00**
23	**Employees**	**316.44**		**316.44**
24	**Licenses Bought**	**253.14**		**253.14**
25	**Licenses sold**		**450.00**	**450.00**
26	**Marketing**	**189.87**		**189.87**
27	**PRODUCTS**	**759.45**	**1,125.00**	**1,884.45**
28	**Office**	**253.15**	**337.50**	**590.65**
29	Expenses	253.15		253.15
30	Revenues		337.50	337.50
31	**SQL Server**	**253.15**	**337.50**	**590.65**
32	Expenses	253.15		253.15
33	Revenues		337.50	337.50
34	**Visual Studio**	**253.15**	**450.00**	**703.15**
35	Expenses	253.15		253.15
36	Revenues		450.00	450.00

Figure 95 – Excel Query with the Unary Operator

Everything seems good. We are now able to show all the hierarchies on one single excel worksheet. We have a simple problem to solve that is relevant to the grand total: it always computes the sum of its children. A simple MDX Scope will solve the problem, as you can see in Figure 96.

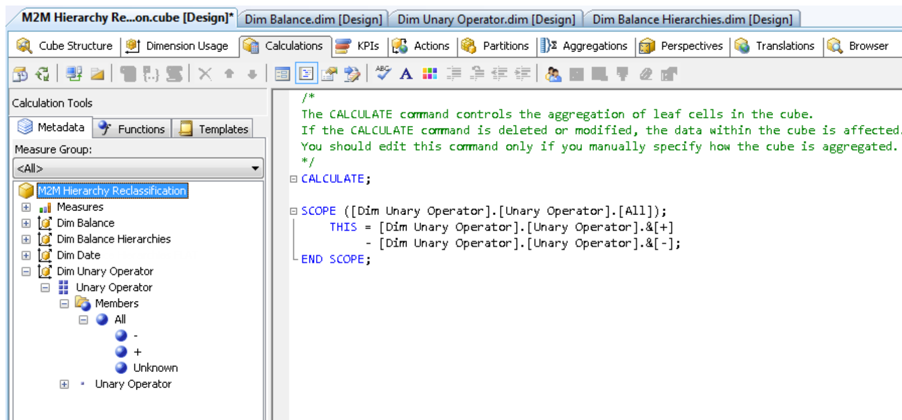


Figure 96 – MDX code to compute the correct total with unary operators

With the MDX code, the result is exactly what we wanted to see. Please note that we have SCOPEd on the Dim Unary Operator dimension. It might be better, depending on your specific needs, to use the SCOPE by filtering the descendants of the parent/child hierarchy in order to use the unary operator only when the hierarchy is selected, avoiding any confusion to the user. In Figure 97 you can see the result of the SCOPE calculation just defined.

	A	B	C	D
1	Year	2007		
2				
3	**Amount**	**Column Labels**		
4	**Row Labels**	-	+	**Grand Total**
5	**BALANCE SHEET**	**759.45**	**1,125.00**	**365.55**
6	**Expenses**	**759.45**		**-759.45**
7	Office		253.15	253.15
8	Employees		105.48	105.48
9	Licenses		84.38	84.38
10	Marketing		63.29	63.29
11	SQL Server		253.15	253.15
12	Visual Studio		253.15	253.15
13	**Revenues**		**1,125.00**	**1,125.00**
14	Office		337.50	337.50
15	Courses		187.50	187.50
16	Licenses		150.00	150.00
17	SQL Server		337.50	337.50
18	Licenses		150.00	150.00
19	Technical Support		187.50	187.50
20	Visual Studio		450.00	450.00
21	**GENERAL**	**759.45**	**1,125.00**	**365.55**
22	**Courses and Technical Support**		**675.00**	**675.00**
23	**Employees**	**316.44**		**-316.44**
24	**Licenses Bought**	**253.14**		**-253.14**
25	**Licenses sold**		**450.00**	**450.00**
26	**Marketing**	**189.87**		**-189.87**
27	**PRODUCTS**	**759.45**	**1,125.00**	**365.55**
28	**Office**	**253.15**	**337.50**	**84.35**
29	Expenses	253.15		-253.15
30	Revenues		337.50	337.50
31	**SQL Server**	**253.15**	**337.50**	**84.35**
32	Expenses	253.15		-253.15
33	Revenues		337.50	337.50
34	**Visual Studio**	**253.15**	**450.00**	**196.85**
35	Expenses	253.15		-253.15
36	Revenues		450.00	450.00

Figure 97 – Excel Query after MDX code to compute operators

One big advantage of having a single parent/child dimension that has the hierarchy names as first nodes is that the selection of a specific hierarchy is very easy with excel. If you filter on the Dim Balance Hierarchies dimension, you will get a very user-friendly interface like shown in Figure 98.

Figure 98 – Filtering the hierarchies is very easy

The work is not finished yet. You might have noticed that there is no grand total for the rows in any of the queries we have shown up to now. This is correct. The many hierarchies cannot be mixed up because of their values representing completely different concepts. However, even if it is correct, it brings a very annoying problem: the “All” member of the Dim Balance Hierarchies evaluates to NULL. In other words, there is no “All” member for this dimension.

The problem of not having an “All” member in the dimension is that, in order to be able to make any query on the cube without the Dim Balance Hierarchies dimension, we will have to mark the dimension hierarchy with the attribute “IsAggregatable” to FALSE. This will make the computation working, but it is a viable solution only for very small BI solutions. Otherwise, SSAS will need to scan the dimension for each query,

even for those that do not contain it. With a medium sized solution, this is definitely a bad situation and the user will be very unhappy about performances.

Fortunately, there is a good solution to the problem even if it involves some tricks to fool SSAS and make it work the way we want it to do. We will tell SSAS that there is no relation at all between the fact table and the Dim Balance Hierarchies dimension. In this way, SSAS will compute all the queries ignoring the non-aggregatable dimension. However, doing this, we will have no relationships between the Dim Balance Hierarchies and the fact table: in this way, we are losing the ability to use the dimension. We will solve this issue by creating a new dimension that we will call “Balance Hierarchies FLAT”, which will contain exactly the same nodes as Dim Balance Hierarchies but will not contain any parent/child hierarchy. The name “FLAT” comes from the fact that this dimension is identical to its parent/child source but has no parent/child hierarchies. What is the advantage of the flat child against the hierarchical one? The flat dimension is aggregatable, while the hierarchical one is not.

After the creation of the new dimension, we need to change the dimension usage by removing the relationship with the Dim Balance Hierarchies and then by moving it to the flat dimension, as shown in Figure 99.

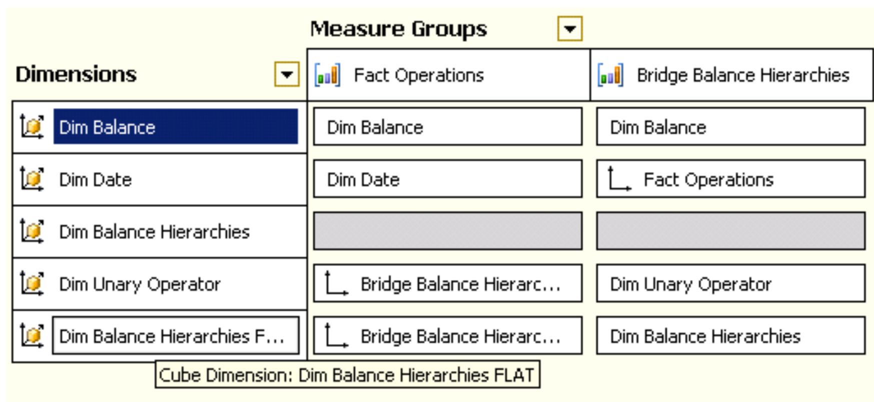


Figure 99 – Dimension Usage with the FLAT dimension

The queries with the flat dimension are very cryptic, because the dimension, as its name implies, is flat and does not contain any hierarchy, as you can see in Figure 100.

	A	B	C	D
1				
2	Year	2007		
3	Balance Hierarchy	GENERAL		
4				
5	**Amount**	**Column Labels**		
6	**Row Labels**	-	+	**Grand Total**
7	Courses and Technical Support		675.00	675.00
8	Employees	316.44		-316.44
9	GENERAL	759.45	1,125.00	365.55
10	Licenses Bought	253.14		-253.14
11	Licenses sold		450.00	450.00
12	Marketing	189.87		-189.87
13	**Grand Total**	**759.45**	**1,125.00**	**365.55**

Figure 100 – Dimension Usage with the FLAT dimension

However, in this way we are able to make computations working on an aggregatable dimension. The next step will be that of linking the aggregatable dimension to the hierarchical one.

MDX function “StrToMember” can assist us here. Since the flat and hierarchical dimensions are the same, they share the primary IDs. Therefore, we know that the value of a node in the flat hierarchy is the same as the value in the hierarchical one. We need to write some MDX code (like the one shown in Figure 101) that will redirect the queries made on the hierarchical dimension to the non-hierarchical one.

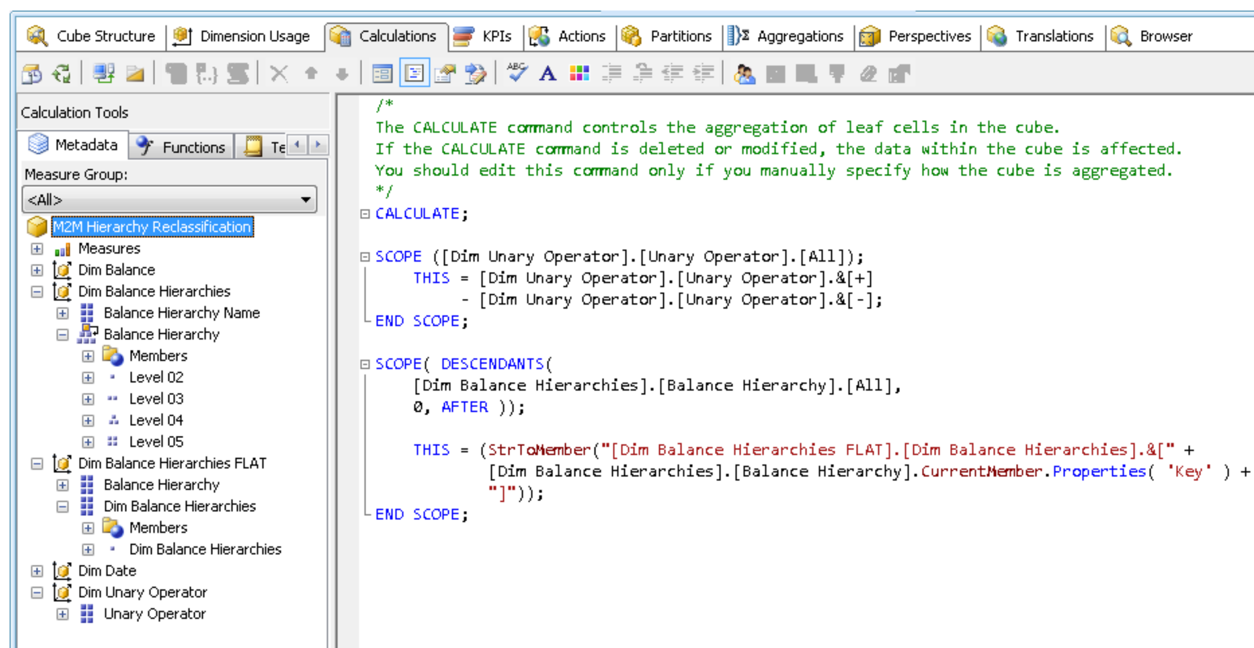


Figure 101 – MDX code for the redirection

This code will make the bridge between the flat and the hierarchical dimensions and is the final touch of this model. Now SSAS will show the values taken from the aggregatable dimension into the hierarchical one.

We might decide to hide the flat dimension from the model if it going to confuse the end user. Nevertheless, we need to consider that the flat dimension might come in handy when the user wants to select a single node from the hierarchy to slice it using other dimensions. It is often easier to select a node from a flat hierarchy (when the user knows exactly what he wants to search for).

One interesting question is “Is this solution equivalent to the usage of Unary Operators”? The answer is definitely no, there are some differences but they are not so easy to understand at first glance.

- Unary Operator in Analysis Services works at the aggregation level, while our solution does not influence aggregations. This means that if you analyze the aggregation of “all amount” using the standard parent/child hierarchy with the unary operator, you will get the total computed using the unary operator (in our example, you would get the difference between profits and expenses). For this reason, we will need to remove the unary operator from the original hierarchy, because we do not want it to interfere with our implementation of the unary operator.

- Our model is different from the same implementation with many dimensions each having a single parent/child hierarchy. For the same reason as before, if you develop many parent/child hierarchies with unary operators they will interfere each other giving you very strange values.

Moreover, having more than one parent/child hierarchy with unary operator will lead to very poor performances, because the different hierarchies will work on the same data, slowing down all queries.

- Our model does not handle operations apart from sum and subtraction. The model might be modified in order to use other operation, but it would become very complex and difficult to check. Nevertheless, in our experience, sum and subtraction are the most (if not the only) useful operations.

The Balance Reclassification model is not an easy one to implement. It requires attention and patience both during the ETL development and the cube design, but it gives to the users a very easy way of interact with the final solution. Moreover, as with all the complex models, it remains hard to apply until you deeply understand it. Afterwards, it will become a very nice tool to present to users with no secrets at all for you.

Considerations about Multidimensional Models

We examined several models that leverage on the UDM many-to-many relationships feature to solve real world business problems. There are many other uses and scenarios that we have not considered here. Many-to-many relationships opened a wide world of opportunities. I hope that the models presented in this paper will help you to approach this new revolutionary world of data analysis.

The most important technique to master with many-to-many relationships in Analysis Services is the use of cascading many-to-many relationships. The other skill you have to acquire is the ability to create a relational model that can easily be represented in a UDM, even if the relational model by itself cannot be easily queried by SQL. By leveraging the many-to-many relationships, you may need relatively simple MDX queries instead of rather complex SQL queries with one or more subqueries.

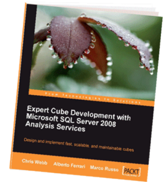
Unfortunately, several releases of Analysis Services have not improved the designer experience in creating a UDM with many-to-many relationships. It can be very difficult to add a measure group or a dimension to a UDM with several many-to-many relationships: many of the choices that are offered when you select the intermediate measure group do not make sense, a better order or a simple hint suggesting the most probable right choice would have been more helpful.

Performance analysis and scalability are areas that require a further study related to data volume, MDX queries and specific models. Recommendations presented in this document might refer to tests made in 2005/2006. Improved server memory and performance might have enlarged the limits to which the usage of many-to-many relationships can be used and should always be checked in the customer’s real scenario. In general, the flexibility provided by these models is much more important that possible degradation in performance. Without precise rules to anticipate performance issues as a result of using the many-to-many relationships, the most effective practice is to run tests measuring the response times with your own data.

Feedbacks are most welcome: write us at info@sqlbi.com!

Links

- http://www.sqlbi.com/articles/many2many/ : the project home page for this paper and related resources
- http://www.sqlbi.com: community dedicated to Business Intelligence with SQL Server
- http://sqlblog.com/blogs/marco_russo : blog of Marco Russo (author)
- http://sqlblog.com/blogs/alberto_ferrari : blog of Alberto Ferrari(author)
- http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=137 : Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques (best practices white paper by SQLCAT team)
- http://www.sqlserveranalysisservices.com/OLAPPapers/IntroToMMDimensionsV2.htm : introduction to Many-to-Many Dimensions by Richard Tkachuk (contains an explanation of Visual Total limitations)
- http://www.sqlbi.com/articles/sqlbi-methodology/ : the SQLBI Methodology paper for best practices in BI solution design



- http://www.amazon.com/gp/product/1847197221/?tag=se04-20 : the book "Expert Cube Development with Microsoft SQL Server 2008 Analysis Services" written by Marco Russo, Alberto Ferrari and Chris Webb
- http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=891 : Analysis Services Distinct Count Optimization

Tabular Models

This section presents the many-to-many relationships applied to BISM Tabular.

A Note about UDM and BISM acronyms

In 2011, Microsoft announced that BISM (Business Intelligence Semantic Model) is the new acronym that groups all the models that you can create with Analysis Services. While initially BISM was only referring to the models that relies on the new Vertipaq engine, there are now two choices of modeling in BISM: BISM Multidimensional, formerly known as UDM, and BISM Tabular, the one based on the Vertipaq engine. In this section, we will refer to BISM Tabular by only using the BISM acronym. In a future revision of this document, we will rename BISM to BISM Tabular and/or Tabular, in order to better point to the correct definition.

Modeling Patterns with many-to-many

In 2010, Microsoft introduced PowerPivot and the DAX programming language, along with the news that the same data-modeling engine would be used for the new BI semantic model, available with Microsoft SQL Server Denali. Both PowerPivot and BISM (BI Semantic Model) rely on the same core technology. Thus, from now on, we will refer to BISM as the data-modeling tool available in both.

BISM Tabular unexpectedly lacks the option to handle many-to-many relationships natively. Thus, if we just look at the surface, it seems that BISM is one step back in the past, removing the support for such a powerful data modeling option. Nevertheless, the DAX programming language can easily supersede this limitation since we will be able to use many-to-many relationships in BISM through some DAX coding. In the next chapters, we are going to look at several data models that can be implemented through the usage of the many-to-many pattern in DAX.

An advantage of Multidimensional models is the ability to setup many-to-many dimensions in a declarative way, which applies to all measures in a measure group. If a user does not filter or slice by a many-to-many dimension, then the many-to-many relationships do not affect performance and the query is not resolved through the bridge table. In the Tabular model, you must create one calculated measure per numeric column per many-to-many dimension. The number of calculations can quickly grow exponentially. Moreover, including many-to-many type calculations will affect query performance even when not filtering by these many-to-many dimensions in a Tabular model.

We added some performance measurements for many of the scenarios. When measuring the performance of a BI solution, it is not very interesting to check at milliseconds. Moreover, such a measurement would have turned into a too complex description, losing the focus on data modeling. Our main interest is in the user experience. In our opinion, a BI solution is well designed if the queries are answered in a matter of two or three seconds. If the user need to wait more to get an answer, then the interactive nature of PivotTable is broken. In some very cases, we decided that 10 seconds were the upper limit, but this happened only for very complex data models where the user is willing to wait some seconds to get an answer.

Thus, at the end of each scenario you will find some considerations about the optimal size of each involved table. All data for the tests has been generated with RedGate SQL Data Generator, using random values for most of the columns. We found this tool really invaluable to create test sets of different sizes and with different characteristics. At the end, having data available proved to be very useful because a data model is worthless if it brings poor performances. Having the ability to test performances with different data sets let us focus on the most complex scenarios where optimizations were still needed.

Classical many-to-many Relationship

We will analyze a situation where we have a very simple many-to-many relationship between two dimensions. Even if the case is very simple, it is still useful to analyze it in detail because, later, those details will get more complex and we need to understand them very well.

Business scenario

Here is a typical business scenario drawn from the bank business: we have a fact table that describes a measure (in this case banking transactions) for a given entity (a bank account) that can be joined to many members of another dimension (a joint account owned by several customers).

Those of you familiar with the “classical” multidimensional model can already see the difficulty. It is not easy to describe the non-aggregative nature of measures joined to dimensions with a many-to-many relationship. In this case, each bank account can have one or more owners and each owner can have one or more accounts but we should not add the cash of an owner to his/her joint owners.

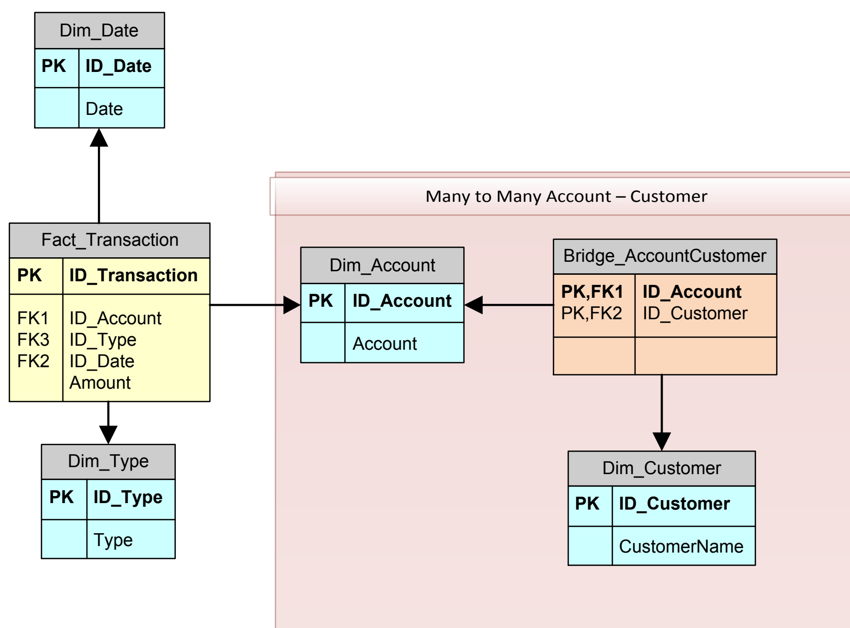


Figure 102 – Classical many-to-many diagram

There are many other scenarios where the classical many-to-many relationship appears. Because it is the simplest of all many-to-many relationships, we use this example to describe in detail how many-to-many relationships work in BISM.

BISM Implementation

The first thing to say about BISM is that many-to-many are simply not supported. This does not mean that we will not be able to make them work, otherwise this whitepaper would end here, but that in order to leverage many-to-many we will need to write some DAX code and, most important, understand what is going on under the cover in terms of evaluation contexts. This set of information will be invaluable with the following scenarios, where the going gets though.

If we start querying this data model with a PivotTable, the first result is, as expected, not working (see Figure 103).

Sum of Amount	Column Labels				
Row Labels	ATM withdrawal	Cash deposit	Credit card statement	Salary	Grand Total
Luke	-400	4000	-600	2000	5000
Mark	-400	4000	-600	2000	5000
Paul	-400	4000	-600	2000	5000
Robert	-400	4000	-600	2000	5000
Grand Total	**-400**	**4000**	**-600**	**2000**	**5000**

Figure 103 – Many-to-many showing wrong results

We can see that the same amount is repeated for all of the customers. On the other hand, the Dim_Type works fine since it is related directly to the fact table. Dim_Customer has only an indirect relationship with the Fact_Transaction table through the many-to-many structure built by Bridge_AccountCustomer.

In order to understand how to make many-to-many work, it is important to note a few things:

- If we filter Dim_Customer, the table Bridge_AccountCustomer is filtered too. Nevertheless, automatic filter propagation stops there due to the direction of the relationship. Thus, the Dim_Account table is not filtered.

- If we filter the Dim_Account table, the fact table is filtered too due to automatic propagation of the filter context through relationships.

The latter statement can be easily seen in Figure 104.

Sum of Amount	Column Labels				
Row Labels	ATM withdrawal	Cash deposit	Credit card statement	Salary	Grand Total
Luke	-200			1000	800
Mark	-200	1000			800
Mark-Paul		1000			1000
Mark-Robert				1000	1000
Paul		1000	-300		700
Robert		1000	-300		700
Grand Total	**-400**	**4000**	**-600**	**2000**	**5000**

Figure 104 – Direct relationship lead to correct results

Because Dim_Account has a direct relationship with the fact table, it works fine as a slicer for the transactions.

Thus, to solve our scenario it will be enough to filter the Dim_Account table by writing some code that takes the existing filter on Dim_Customer and builds a new filter, on Dim_Account, that shows only the accounts of the selected customers.

Now, how can we detect which accounts are tied to the selected customers? We need to check, for each account, if there is a row in the Bridge_AccountCustomer that is related to a selected customer. Instead of showing the final formula, let us build it step-by-step.

We can create a simple measure like this:

```
NumOfCustomers := COUNTROWS (Bridge_AccountCustomer)
```

When used in a PivotTable with the accounts on rows, it gives the interesting (and expected) result shown in Figure 105.

Row Labels	Sum of Amount	NumOfCustomers
Luke	800	1
Mark	800	1
Mark-Paul	1000	2
Mark-Robert	1000	2
Paul	700	1
Robert	700	1
Grand Total	**5000**	**8**

Figure 105 – COUNTROWS of the bridge table

The measure simply counts, for each account, the number of customers tied to the account. The grand total is clearly incorrect, because it counts a single customer for each account to which he belongs. This is not interesting because we are not going to use the formula to count the customers at the grand total level.

If we simply add a slicer to the previous PivotTable and select Mark, we will get the result shown in Figure 106.

CustomerName	
Luke	Mark
Paul	Robert

Row Labels	Sum of Amount	NumOfCustomers
Luke	800	
Mark	800	1
Mark-Paul	1000	1
Mark-Robert	1000	1
Paul	700	
Robert	700	
Grand Total	**5000**	**3**

Figure 106 – FILTERED COUNTROWS of the bridge table

Filtering the customer has no effect on the Amount, but the "number of customers" column has changed its values and is now showing "1" for the accounts that belong to Mark. The reason is that the Bridge_AccountCustomer is filtered by the slicer (the filter on customer is propagated to the bridge). Thus, none of the customers that are selected yields a value of zero.

At this point, the procedure should be pretty clear: to move the filter from the customer to the accounts, we need to detect all the accounts for which the NumOfCustomers measure has a value greater than zero.

The first formula might be like this:

```
AmountM2M :=

CALCULATE (
    SUM (Fact_Transaction[Amount]),
    FILTER (
        Dim_Account,
        [NumOfCustomers] > 0
    )
)
```

We create a filter context on Dim_Account containing only the accounts for which the NumOfCustomers is greater than zero. After having applied this filter, the Dim_Account table is filtered, showing only the accounts of selected customers and the SUM of amount yields a correct result, as it can be seen in the Figure 107.

CustomerName	
Luke	Mark
Paul	Robert

Row Labels	Sum of Amount	NumOfCustomers	AmountM2M
Luke	800		
Mark	800	1	800
Mark-Paul	1000	1	1000
Mark-Robert	1000	1	1000
Paul	700		
Robert	700		
Grand Total	**5000**	**3**	**2800**

Figure 107 – First version of amountm2m

The non-additive nature of many-to-many becomes evident if we add the customers on the rows and remove the filter. The result is illustrated in Figure 108.

Row Labels	NumOfCustomers	AmountM2M
⊟Luke	**1**	**800**
Luke	1	800
⊟Mark	**3**	**2800**
Mark	1	800
Mark-Paul	1	1000
Mark-Robert	1	1000
⊟Paul	**2**	**1700**
Mark-Paul	1	1000
Paul	1	700
⊟Robert	**2**	**1700**
Mark-Robert	1	1000
Robert	1	700
Grand Total	**8**	**5000**

Figure 108 – Final result of amountm2m

It is very easy to check that the grand total is not the sum of the single amounts, as it is expected from such a formula. At the grand total level, each account is counted only once, even if it appears under more than one customer.

Now, the final touch is to remove the helper measure NumOfCustomers, which is useful only to compute the AmountM2M measure, and incorporate its code inside a single formula. A first trial might be:

```
AmountM2M_Wrong :=

CALCULATE (
    SUM (Fact_Transaction[Amount]),
    FILTER (
        Dim_Account,
        COUNTROWS (Bridge_AccountCustomer) > 0
    )
)
```

If we use this formula in the PivotTable, we get the wrong result shown in Figure 109.

Row Labels	NumOfCustomers	AmountM2M	AmountM2M_Wrong
Luke	**1**	**800**	**5000**
Luke	1	800	800
Mark	**3**	**2800**	**5000**
Mark	1	800	800
Mark-Paul	1	1000	1000
Mark-Robert	1	1000	1000
Paul	**2**	**1700**	**5000**
Mark-Paul	1	1000	1000
Paul	1	700	700
Robert	**2**	**1700**	**5000**
Mark-Robert	1	1000	1000
Robert	1	700	700
Grand Total	**8**	**5000**	**5000**

Figure 109 – Missing a calculate leads to wrong results

At the leaf level the values are correct (because the filter from Account is working), whereas at the customer level (yellow cells) all values are incorrectly computed as 5000.

The reason is that, in order to filter the bridge table using the account, the account should be present in the filter context, so that existing relationship is considered. In our formula the account is iterated by the row context introduced by FILTER but is never pushed into a filter context. The previous formula worked because, during the FILTER iteration, we were calling a measure that, by definition, is computed as if it was automatically surrounded by a CALCULATE. CALCULATE transforms the row context into a filter context, making it follow relationships and filtering, in turn, the fact table.

In order to make our formula work, it is enough to use an explicit CALCULATE in place of the implicit one:

```
AmountM2M :=

CALCULATE (
    SUM (Fact_Transaction[Amount]),
    FILTER (
        Dim_Account,
        CALCULATE (COUNTROWS (Bridge_AccountCustomer) > 0)
    )
)
```

Please, take a look at the highlighted part of the formula and take the time necessary to understand it, because it is the core of any many-to-many formula we are going to write from now on.

The key is to move the filter from the farthest table to the nearest one using the bridge to check if the account rows should be made visible or not, depending on the selection on customers. The innermost CALCULATE is needed to convert the row context into a filter one so that the bridge table is filtered from two sides: the customers from the original filter context and the accounts from the iteration. When both filters are active, the number of rows visible in the bridge table is either zero or one: zero for accounts that should be hidden, one for accounts that should be visible.

Once you master this formula, all of the remaining scenarios will be affordable. If you do not fully understand it, then all of the following scenario will be impossible to understand because they all use the same pattern to make many-to-many relationships work in BISM.

Denali Implementation

In the previous paragraphs, we have shown the formula that works in DAX 1.0, the version of the language available in PowerPivot. In the new version of SQL Server codename “Denali”, there is a new way to author the same formula by means of using the new SUMMARIZE function.

```
AmountM2M :=

CALCULATE (
    SUM (Fact_Transaction[Amount]),
    SUMMARIZE (Bridge_AccountCustomer, Dim_Account[ID_Account])
)
```

The performance of this new formula are better, because we do not use an iterator, instead we ask DAX to returns us the values of DimAccount[ID_Account] which can be reached through the relationship existing from the bridge to Dim_Account. It is important to note that we need to use Dim_Account[ID_Account] and not Bridge_AccountCustomer[ID_Account] as the summarized column. The latter, in fact, will not filter the fact table.

Avoiding the usage of the iterator, Vertipaq is able to push the relationship down to the storage engine, providing better performance over big tables.

In this paper, we will use both types of formulas, depending on the context. We feel that the FILTER version is easier to understand, because it shows clearly the algorithm while the SUMMARIZE version is faster and more concise. Thus, for educational purposes, we will normally provide the description of the formulas with the FILTER version and provide the SUMMARIZE version of the same formula at the end, so that the reader can appreciate the difference and have both working formulas at hand.

Performance Analysis

From the performance point of view, it is interesting to note that this formula contains a single iteration, which is the one introduced by FILTER. This means that the time required to retrieve a value is dependent on three factors:

- Size of the transaction table
- Size of the account table
- Size of the bridge table

During our tests it turned out that the size of the transaction table is not very important, a 50 million rows table is computed pretty fast. The algorithm is much more sensible to the size of the account and of the bridge table. Moreover, normally the two tables have a similar cardinality because each account is linked to many customers, so they share the overall size.

Cascading many-to-many Relationships

When we apply the many-to-many relationship several times in a cube, we have to pay attention if there is a chain of many-to-many relationships. As we have seen in the classical many-to-many relationship scenario, dimensions that apparently do not relate to a bridge measure group could be meaningful and important for the enhancement of the analytical capabilities of our model.

We call the situation where there is a chain of many-to-many relationships a "cascading many-to-many relationship".

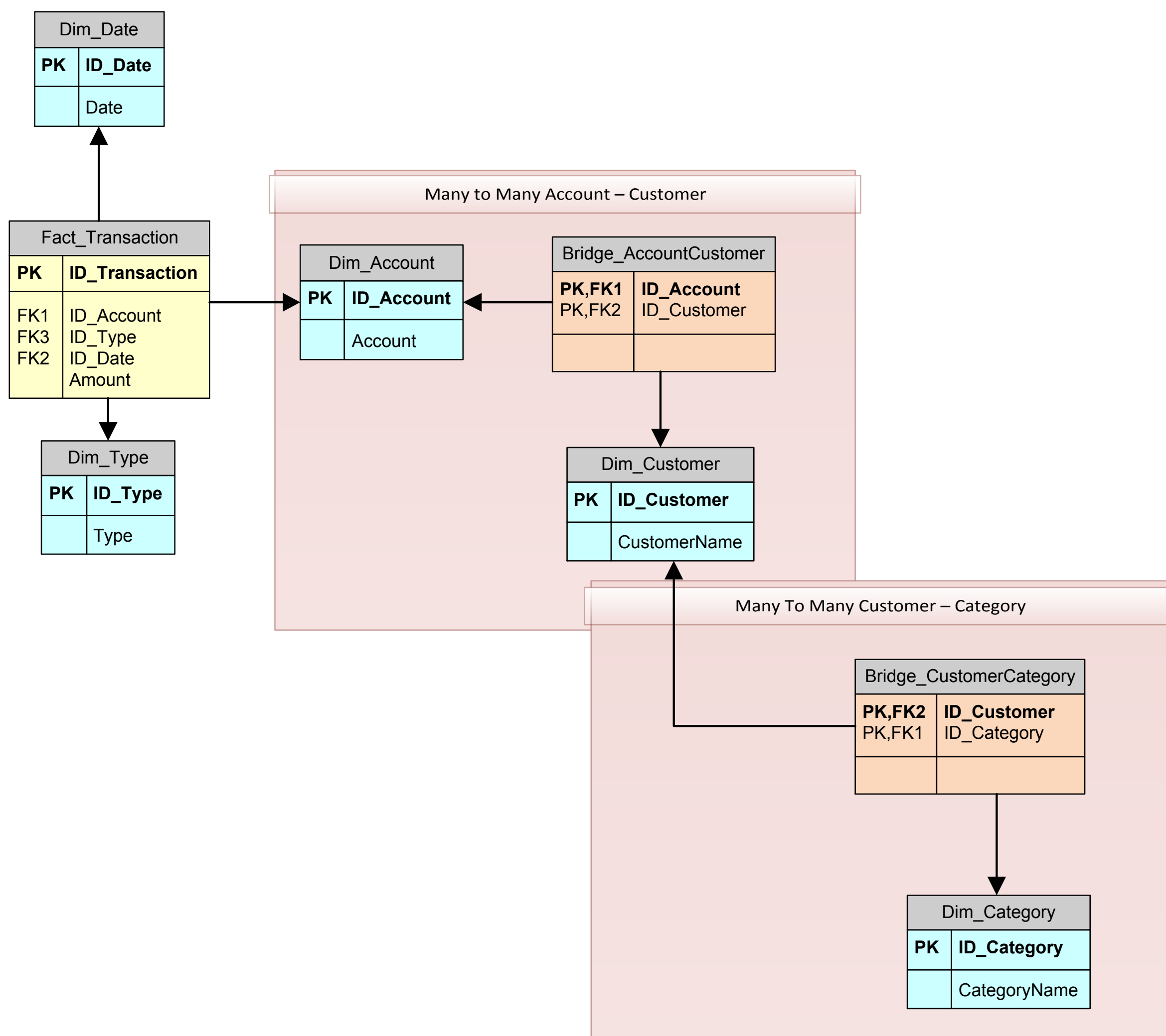


Figure 110 – Cascading many-to-many diagram

In the picture, we can see that – in order to associate a category to a transaction – we need to traverse two different many-to-many relationships: the first one from account to customer and the second one from customer to category. We say that the chain starts from the DimCategory and ends to Dim_Account, traversing two many-to-many relationships.

BUSINESS SCENARIO

A typical scenario is the case when a dimension far from the main fact table (a dimension that relates to one bridge fact table) is involved in an existing many-to-many relationship and has another many-to-many relationship with another dimension.

For example, consider bank account scenario shown in the previous picture. The main characteristics of the data model are:

- Account transactions: Transactions fact table related to Dim Date, Dim Account and Dim Type.
- Each account can have one or more owners (customers): Dim Account has a many-to-many relationship with Dim Customer through a bridge table.
- Each customer can be classified into one or more categories: Dim Customer has a many-to-many relationship with Dim Categories through another bridge table.

In order to understand the examples, we need to describe some of the data that we will use in our implementation. The next table shows the denormalized fact table. Even if the Date dimension is not strictly necessary for this explanation, we will keep it in the model because it is a common dimension in a similar scenario and it is useful to see how it relates to the other dimensions.

Account	Type	Date	Amount
Mark	Cash deposit	20051130	1000.00
Paul	Cash deposit	20051130	1000.00
Robert	Cash deposit	20051130	1000.00
Luke	Salary	20051130	1000.00
Mark-Robert	Salary	20051130	1000.00
Mark-Paul	Cash deposit	20051130	1000.00
Mark	ATM withdrawal	20051205	-200.00
Robert	Credit card statement	20051210	-300.00
Paul	Credit card statement	20051215	-300.00
Luke	ATM withdrawal	20051215	-200.00

Table 16 – Denormalized model for cascading many-to-many

The Type dimension is very important for our purposes: it describes the type of the transaction and it is useful to group transactions across other dimensions.

Let us see some kind of questions the user may ask upon these data:

- What is the salary/income for the “IT enthusiast” category?
- How many different transaction types involve the “Rally driver” category?
- What customer categories have ATM withdrawal transactions?

Within the fact table, there is not enough information to provide answers to those questions but all what we need is stored in tables (dimensions) reachable through the many-to-many relationships. We only have to create the correct relationships between dimensions. The following table contains the relationship existing between customers and categories in our sample data:

Customer	Category
Mark	IT enthusiast
Robert	IT enthusiast
Paul	Rally driver
Robert	Rally driver
Luke	Traveler
Mark	Traveler
Paul	Traveler
Robert	Traveler

Table 17 – Relationships between customers and categories

Now, to give an answer to the first question (What is the salary/income for the “IT enthusiast” category?) we need an additional clarification.

- If we consider the accounts owned by only one person, then there are no customers belonging to the “IT enthusiast” category who get a salary income.
- If we consider joint accounts (e.g. Mark and Robert both own the same account), then their owners receive a salary income even if we do not know who is really gaining money.

From Mark’s perspective, he receives a salary income of 1000. On the other side, Robert gets a salary income of 1000 too! However, unfortunately for them, from the perspective of “IT enthusiast” category we cannot count the same salary income two times, so the “IT enthusiast” salary income is still 1000 and not 2000. The tough reality is that Mark and Robert have to share this single salary income, because we have no other way to know which of them is really receiving this income, because we recorded the transaction against their joint account.

Before looking at the implementation, let us try to answer the previous three questions with a PivotTable. We will use only a PivotTable with some slicers to answer all of them.

- What is the salary/income for the “IT enthusiast” category?

  To solve this problem it is enough to select the type and category from the slicer and we get the result shown in Figure 111.

Year: 2005

Type: ATM withdrawal, Cash deposit, Credit card statement, Salary

CategoryName: IT enthusiast, Rally driver, Traveler

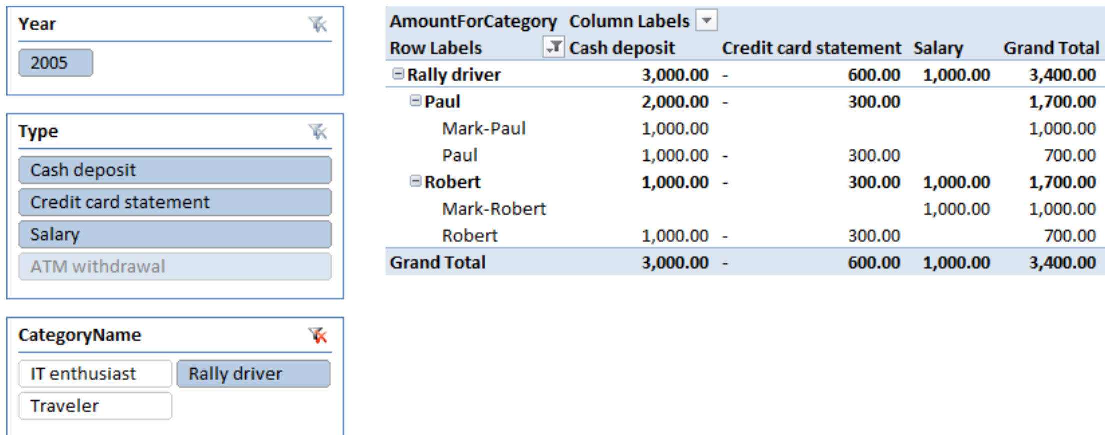
AmountForCategory	Column Labels	
Row Labels	Salary	Grand Total
IT enthusiast	1,000.00	1,000.00
Mark	1,000.00	1,000.00
Mark-Robert	1,000.00	1,000.00
Robert	1,000.00	1,000.00
Mark-Robert	1,000.00	1,000.00
Grand Total	1,000.00	1,000.00

Figure 111 – Cascading m2m example

In the previous figure it is evident that the value of 1,000.00 is shown for both Mark and Robert, because it belongs to the current account owned by both even if, at the category level, it is still 1,000.00.

- How many different transaction types involve the “Rally driver” category?

  Even in this case the question can be answered by easily changing the filters in the slicers, as you can see in Figure 112.



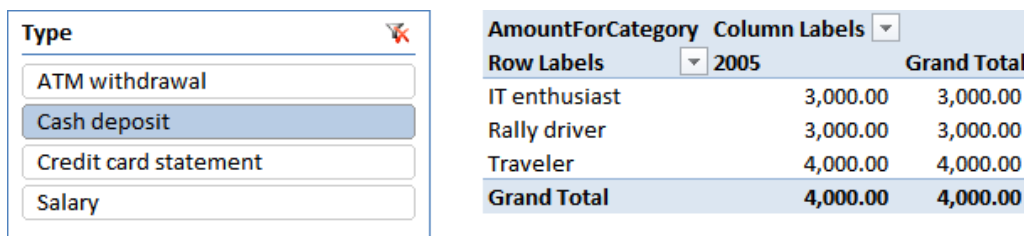
AmountForCategory	Column Labels			
Row Labels	Cash deposit	Credit card statement	Salary	Grand Total
Rally driver	3,000.00	- 600.00	1,000.00	3,400.00
Paul	2,000.00	- 300.00		1,700.00
Mark-Paul	1,000.00			1,000.00
Paul	1,000.00	- 300.00		700.00
Robert	1,000.00	- 300.00	1,000.00	1,700.00
Mark-Robert			1,000.00	1,000.00
Robert	1,000.00	- 300.00		700.00
Grand Total	3,000.00	- 600.00	1,000.00	3,400.00

Figure 112 – Cascading m2m example

The rally drivers are Paul and Robert and the transaction types are shown in the columns.

- What customer categories have Cash Deposit transactions?

  Selecting the type of transaction, we easily get the list of categories (IT enthusiast, Traveler and Rally Driver) like those you can see in Figure 113.



AmountForCategory	Column Labels	
Row Labels	2005	Grand Total
IT enthusiast	3,000.00	3,000.00
Rally driver	3,000.00	3,000.00
Traveler	4,000.00	4,000.00
Grand Total	4,000.00	4,000.00

Figure 113 – Cascading m2m example

BISM Implementation

In order to solve this scenario, we are going to develop a modified version of the formula we have already seen for the classical many-to-many relationship. The major modification needed is to take into account the cascading nature of the new relationship.

The original formula for many-to-many had the need to “push” a filter context from the table used to slice data into the table on the other side of the bridge relationship, in effect forcing the bridge table to get “double filtered” so to create a filter on the dimension that is directly related with the fact table. In the cascading many-to-many we will need to do the same process walking two steps instead of a single one.

For example, if the user selects a category, we will need to take the filter context of the DimCategory table and push it to a filter context on Dim_Account, traversing the two many-to-many defined by Bridge_CustomerCategory first and Bridge_CustomerAccount next.

With this description in mind, the formula is straightforward:

```
AmountForCategory =

CALCULATE (
    CALCULATE (
        SUM (Fact_Transaction[Amount]),
        FILTER (
            Dim_Account,
            CALCULATE (COUNTROWS (Bridge_AccountCustomer) > 0)
        )
    ),
    FILTER (
        Dim_Customer,
        CALCULATE (COUNTROWS (Bridge_CustomerCategory) > 0)
    )
)
```

It is composed by two nested CALCULATE. The outer one filters the customers based on the category, the inner one filters the accounts based on the customers (which are filtered by the previous CALCULATE). The formula makes evident the cascading nature of the relationship.

The complexity of this formula is clearly depending on the size of the bridge tables, because it is composed by two filters, each of which needs to iterate one of the two bridge tables. Because the two filter operations are carried on sequentially, the complexity should be related to the product of the size of the two bridge tables.

In Denali, the same formula can be written as

```
AmountForCategory =

CALCULATE (
    CALCULATE (
        SUM (Fact_Transaction[Amount]),
        SUMMARIZE (Bridge_AccountCustomer, Dim_Account[ID_Account])
    ),
    SUMMARIZE (Bridge_CustomerCategory, Dim_Customer[ID_Customer])
)
```

As usual, the Denali version is much more compact, even if its behavior is somehow less clear, at first glance.

During performance testing, we have noticed that the PivotTable is able to answer very quickly regardless of the size of the fact table. We tested 10 and 50 millions of rows in the Fact table and performance did not change significantly, leading to a good user experience. Things changed when we started to change the number of customers and, consequently, the size of the bridge tables. We have used the following parameters for data generation:

- Each customer has an average of 1.2 accounts (i.e. 12 accounts every 10 customers) and belongs to an average of 3.4 categories.

- We tested an increasing number of customers and detected that the optimal user experience is with 200.000 customers, which means 680.000 rows in the Bridge_CustomerCategory and 240.000 rows in the Bridge_AccountCustomer.

- In such a scenario, all of the queries return results in less than 2/3 seconds.

It is interesting to note that this model supports even much more complex calculations with a minimum effort. For example, if we want to compute the number of distinct accounts per category that have transactions in a period of time, we can write this formula:

```
DistinctAccounts =

CALCULATE (
    CALCULATE (
        COUNTROWS (DISTINCT (Fact_Transaction[ID_Account])),
        FILTER (
            Dim_Account,
            CALCULATE (COUNTROWS (Bridge_AccountCustomer) > 0)
        )
    ),
    FILTER (
        Dim_Customer,
        CALCULATE (COUNTROWS (Bridge_CustomerCategory) > 0)
    )
)
```

The only changed part is the innermost calculation, which now counts the number of distinct accounts. The time required for the computation of this formula is not significantly different than the simpler SUM, resulting in 4/5 seconds for the most complex queries (i.e. covering all of the categories and all of the transactions, sliced by year on the columns).

This pattern can be easily adapted with cascading relationships that need to traverse more than two steps. You should take care of selecting the correct ordering of filters, starting with the farthest one from the fact table and moving one step after each other in the direction of the fact table.

Nevertheless, before leaving this scenario, it is worth considering different modeling options that are available in BISM. Because in BISM many-to-many are not handled directly by the system, we are free to choose a different data model to reduce the number of steps in the computation. The idea is to flatten the cascading relationship using a single table that holds the complete chain of relationships between customers, accounts and categories. This new data model is clearly exposed in the Figure 114.

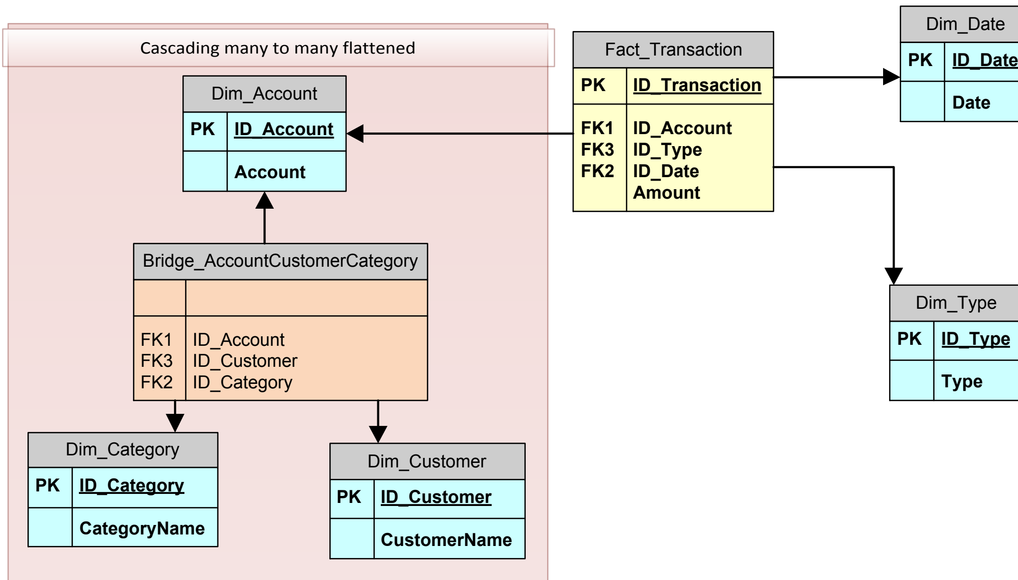


Figure 114 – Flattened cascading many-to-many diagram

In this data model, the scenario is that of a classical many-to-many relationship, the cascading structure is gone. We call this data model the "flattened cascading many-to-many". Such a data structure can be easily created using a SQL query like the following one:

```
SELECT
    AC.ID_Account,
    CC.ID_Customer,
    CC.ID_Category
FROM
    M2M_Cascading.Bridge_CustomerCategory CC
    INNER JOIN M2M_Cascading.Bridge_AccountCustomer AC
        ON CC.ID_Customer = AC.ID_Customer
```

This leads to the table shown in Figure 115, where we have put the denormalized names to make it clearer.

ID_Account	ID_Customer	ID_Category	Account	Customer	Category
1	1	1	Mark	Mark	IT enthusiast
1	1	3	Mark	Mark	Traveler
2	2	2	Paul	Paul	Rally driver
2	2	3	Paul	Paul	Traveler
3	3	1	Robert	Robert	IT enthusiast
3	3	2	Robert	Robert	Rally driver
3	3	3	Robert	Robert	Traveler
4	4	3	Luke	Luke	Traveler
5	1	1	Mark-Robert	Mark	IT enthusiast
5	1	3	Mark-Robert	Mark	Traveler
5	3	1	Mark-Robert	Robert	IT enthusiast
5	3	2	Mark-Robert	Robert	Rally driver
5	3	3	Mark-Robert	Robert	Traveler
6	1	1	Mark-Paul	Mark	IT enthusiast
6	1	3	Mark-Paul	Mark	Traveler
6	2	2	Mark-Paul	Paul	Rally driver
6	2	3	Mark-Paul	Paul	Traveler

Figure 115 – Flattened cascading table

You can see that, in the boxed area, the account Mark-Robert is repeated for each category to which Mark or Robert belong. Using such a structure the formula becomes easier to write, since it is identical to the classical many-to-many one:

```
AmountFlattened=

CALCULATE (
    SUM (Fact_Transaction[Amount]),
    FILTER (
        Dim_Account,
        CALCULATE (COUNTROWS (Bridge_AccountCustomerCategory)) > 0
    )
)
```

Moreover, this final formula runs faster than the previous one, which means that you will be able to handle bigger tables without compromising the user experience.

Thus, the cascading many-to-many model can be easily solved in BISM using either the cascading pattern or a flattened one. The latter gives better results in terms of simplicity of formulas and query speed.

Survey

The survey scenario is a common example of a more general case where we have many attributes associated with a case (one customer, one product, and so on). We want to normalize the model because we do not want to change the data model each time we add a new attribute (e.g. adding a new dimension or changing an existing one).

The common scenario is a questionnaire consisting of questions that have predefined answers with both simple and multiple choices. The classical relational solution is to define a fact table and three dimensions:

- Dim Questions with the questions.
- Dim Answers for the answers provided by customers
- Dim Customer for the customer who answered a specific question

The fact table will contain a value indicating the exact answer from the customer, in the case of multiple choices.

However, since we do not need to analyze questions without answers, a better solution is to have only one table for both questions and answers. This will reduce the number of dimensions without having any influence on the expressivity of the model and will make the complete solution simpler to both navigate and create. The star schema model (one fact table with answers joined with a questions/answers dimension and a case dimension) is fully queryable using SQL.

However, once we move to UDM or BISM things become harder: while it is very simple to compare different answers to the same question, it could be very difficult to correlate frequency counts of answers to more than one question. For example, if we have a question asking for sports practiced (multiple choices) and another one asking for job performed, probably we would like to know what pattern of statistical relationships – if any – exist between the two corresponding sets of answers.

The normal way to model it is having two different attributes (or dimensions) that users can combine on rows and columns of a pivot table. Unfortunately, having an attribute for each question is not very flexible and becomes a real problem as the number of questions grows over time. We will need to change the star schema to accommodate having a single row in the fact table for each case. This makes it very difficult to handle any multiple-choice question.

Instead, we can change our perspective and leverage many-to-many relationships. We can build a finite number (as many as we want) of questions/answers dimensions, duplicating many times the original one and providing the user with a number of “filter” dimensions that can be crossed into a pivot table or can be used to filter data that, for each case, satisfy defined conditions for different questions.

Business Scenario

Let us explore the survey scenario in more detail. Data is contained in the relational schema shown in Figure 116.

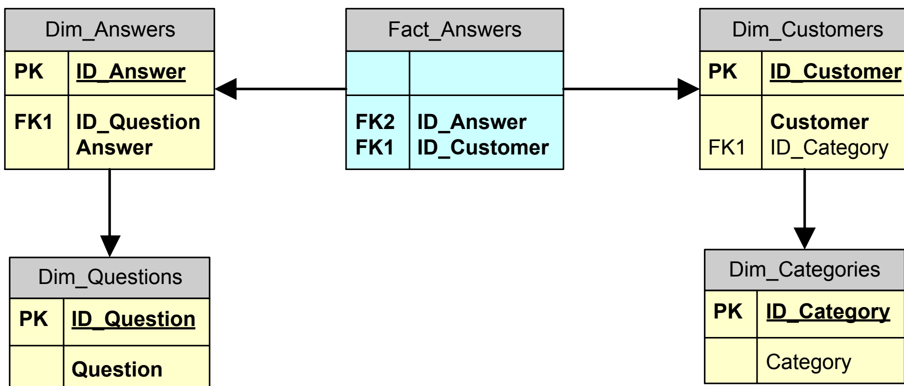


Figure 116 – Survey many-to-many diagram

Each customer belongs to a category, and gives several answers to the questionnaire. Each answer is then related to the question. Each customer can provide more than one answer to each question (i.e. multiple choices are supported by this data model).

This model is good to store the raw data but, from the analytical point of view, it is not very easy to query. Therefore, we are going to build a query model on top of this structure, which is composed by:

- Two Filter tables, which we will can Filter1 and Filter2. Each filter table is composed by joining Dim_Answers and Dim_Questions, merging them into a single entity that contains both answers and questions. Obviously, the query should be loaded twice in the data model to create the two filter dimensions.

- One Customer table, created by joining Customers and Categories and denormalizing the data structure.

- The fact table, as it is present in the relational model.

The analytical model is shown in Figure 117.

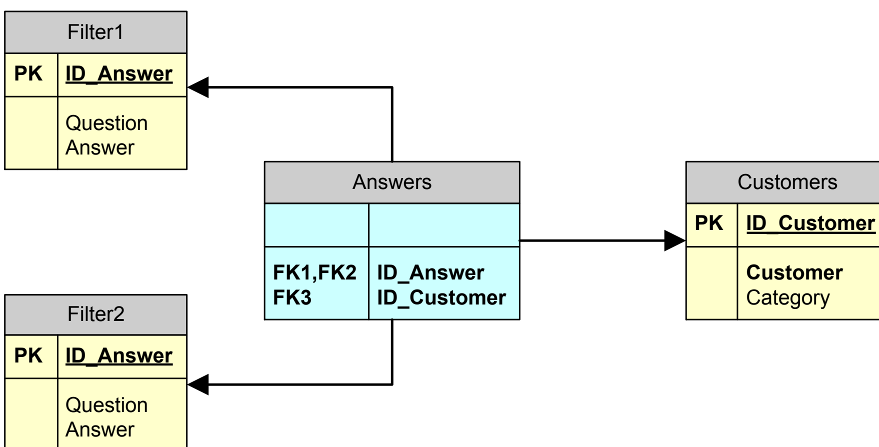


Figure 117 – Survey analytical diagram

This new dataset will be used in a PivotTable to perform cross queries. For example, we can filter a specific question in Filter1 and the see the profile of people who answered that question, like in the report shown in Figure 118.

Question 1
Gender
Job
Movies Preferences
Sport Practiced
Yearly Net Income

Question 2
Gender
Job
Movies Preferences
Sport Practiced
Yearly Net Income

CustomerCount	Column Labels	
Row Labels	Female	Male
Job		
Consultant	1	1
IT Pro	3	3
Teacher	1	2
Movies Preferences		
Cartoons	3	3
Comedy	3	2
Sport Practiced		
Baseball	2	4
Football	3	4
Karate	5	3
Tennis	2	1
Yearly Net Income		
< 10,000	3	
> 80,000	2	1
10,000 - 20,000	3	3
20,000 - 40,000	3	2
40,000 - 80,000	3	2

Figure 118 – Survey report example

In this specific report we are looking at job, movies preferences and other characteristics of all the people divided between Male and Female.

BISM Implementation

Before we dive into the DAX code, let us focus on the algorithm. In order to calculate the numbers we need to:

- Identify the customers that answered Male or Female to the Gender question, which is selected through Filter1. The answer of Filter1 is, in the example, on the columns.
- Check what those customers answered against Filter2 (i.e. Question2 and Answer2, on the rows in the example), compute the values and show them in the PivotTable.

Since we have a limitation in the relationship definition in BISM, we cannot create a relationship between ID_Answer in Answers and the two Filter tables, because the column ID_Answer can be used only for one relationship. Thus, in the BISM data model we are not going to leverage the relationships in the model. All of the relationships between Filter 1 and 2 and the Answers table will be emulated by DAX code.

Let us start with step 1, i.e. identify the customer that has given a specific answer to the question filtered by Filter1. The DAX code is not very hard to write:

```
CustomerCountFilter1 =

IF (
    COUNTROWS (VALUES (Filter1[ID_Answer])) = 1,
    CALCULATE (
        COUNTROWS (Customers),
        FILTER (
            Customers,
            CALCULATE (
                COUNTROWS (Answers),
                Answers[ID_Answer] = VALUES (Filter1[ID_Answer]))
            > 0
        )
    )
)
```

The initial IF is needed because the computation can be carried on if and only if the current filter context contains a single answer. If this is the case, we use a classical many-to-many formula with the simple addition of a filter to the Answers table that makes visible only the rows that are in relationship with the only answer selected in Filter1.

The Denali version of the same formula looks interesting too, because of the need to use CALCULATETABLE as one of the filter for the outermost CALCULATE:

```
=IF (
    COUNTROWS (VALUES (Filter1[ID_Answer])) = 1,
    CALCULATE (
        COUNTROWS (Customers),
        CALCULATETABLE (
            SUMMARIZE (Answers, Customers[ID_Customer]),
            Answers[ID_Answer] = VALUES (Filter1[ID_Answer])
        )
    )
)
```

The CALCULATETABLE is needed to emulate the relationship between Answers and Filter1. By modifying the data model adding an inactive relationship between Answers and Filter1, as in figure Figure 119, we can rely on USERELATIONSHIP and get a new formula.

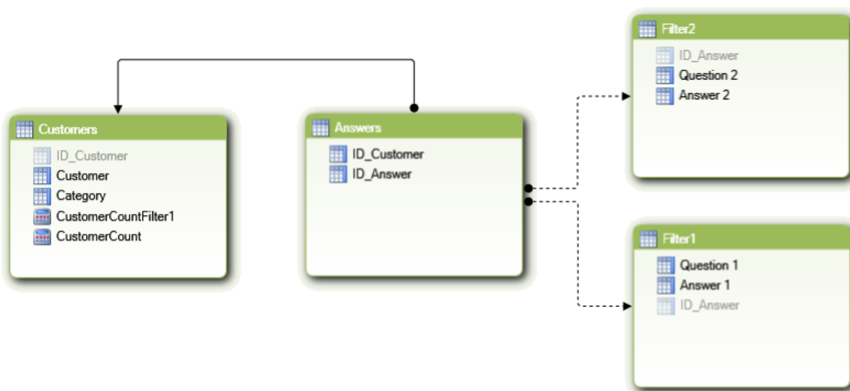


Figure 119 – Inactive relationships in the Survey data model

The formula with USERELATIONSHIP is clearer:

```
=IF (
    HASONEVALUE (Filter1[ID_Answer]),
    CALCULATE (
        COUNTROWS (Customers),
        CALCULATETABLE (
            SUMMARIZE (Answers, Customers[ID_Customer]),
            USERELATIONSHIP (Answers[ID_Answer], Filter1[ID_Answer])
        )
    )
)
```

In the formula, we have made use of HASONEVALUE too, which makes it easier to read,

This first formula, in any of its flavors, produces a report like the one in Figure 120.

➢ We will now continue the description on the 1.0 version of the formula because the Denali version is not working and will cause Excel to crash, due to some bug in the beta release of PowerPivot.
At the end of this chapter, we have added the Denali formula with some explanation about how it works but, at the Denali CTP3 release time, the only working solution is the 1.0 version of the formula.

Question 1
- Gender
- Job
- Movies Preferences
- Sport Practiced
- Yearly Net Income

Question 2
- Gender
- Job
- Movies Preferences
- Sport Practiced
- Yearly Net Income

Row Labels	CustomerCountFilter1
Gender	
Female	12
Alvin Benton	1
Derek Mills	1
Derrick Mills	1
Dewayne Forbes	1
Evan Russell	1
Helen Collier	1
Jimmie Hoover	1
Leonard Ritter	1
Rickey Navarro	1
Shane Anderson	1
Sheryl Hale	1
Stephen Ali	1
Male	7
Courtney Faulkner	1
Courtney Hubbard	1
Karen Pope	1
Leonard Ritter	1
Trisha Travis	1
Wendi Ross	1
Yesenia Compton	1

Figure 120 – First trial of survey formula

Now, the interesting part is that the COUNTROWS in the formula is evaluated in a filter context where only the customers that have answered to the question in Filter1 are visible. Thus, in that context, we can use a similar pattern to verify what those customers have answered to the question eventually filtered by Filter2.

It is time to look at the complete formula for the survey model:

```
CustomerCount =
IF (
    COUNTROWS (VALUES (Filter1[ID_Answer])) = 1 && COUNTROWS (VALUES (Filter2[ID_Answer])) =
1,
    CALCULATE (
        CALCULATE (
            COUNTROWS (Customers),
            FILTER (
                Customers,
                CALCULATE (
                    COUNTROWS (Answers),
                    Answers[ID_Answer] = VALUES (Filter2[ID_Answer]))
                > 0
            )
        ),
        FILTER (
            Customers,
            CALCULATE (
                COUNTROWS (Answers),
                Answers[ID_Answer] = VALUES (Filter1[ID_Answer]))
            > 0
        )
    )
)
```

You can easily see that the inner part of the formula (the highlighted one) follows the same pattern of the original one. The big difference is that this time the formula is evaluated in a filter context that is already filtered based on Filter 2 to show only the customer who answered a specific answer on Filter 1. In other words, both filters intersect each other. Moreover, the highlighted formula computes the number of customers who answered a specific question in Filter2, following the relationship from Answer to Filter2 using DAX code, exactly as we did for Filter1.

If we use this formula in a PivotTable, we get the interesting result shown in Figure 121.

Question 1
- Gender
- Job
- Movies Preferences
- Sport Practiced
- Yearly Net Income

Question 2
- Gender
- Job
- Movies Preferences
- Sport Practiced
- Yearly Net Income

CustomerCount	Column Labels	
Row Labels	Female	Male
Gender		
Female	12	1
Male	1	7
Job		
Consultant	1	1
IT Pro	3	3
Teacher	1	2
Movies Preferences		
Cartoons	3	3
Comedy	3	2
Sport Practiced		
Baseball	2	4
Football	3	4
Karate	5	3
Tennis	2	1
Yearly Net Income		
< 10,000	3	
> 80,000	2	1
10,000 - 20,000	3	3
20,000 - 40,000	3	2
40,000 - 80,000	3	2

Figure 121 – Final survey example

This report is interesting, because it shows, for each customer who answered to the Gender question, which other questions he answered (including the answers). There are a couple of things to note here, looking at the highlighted part of the figure:

- It seems that somebody has answered both Male and Female to the same question. If you look carefully at the previous figure, where all the customers are shown, you will easily check that "Leonard Ritter" is the guilty. This happens with random generated data, we do not need to worry about that but it is nice to see that the problem is evident in the report and can be addressed.

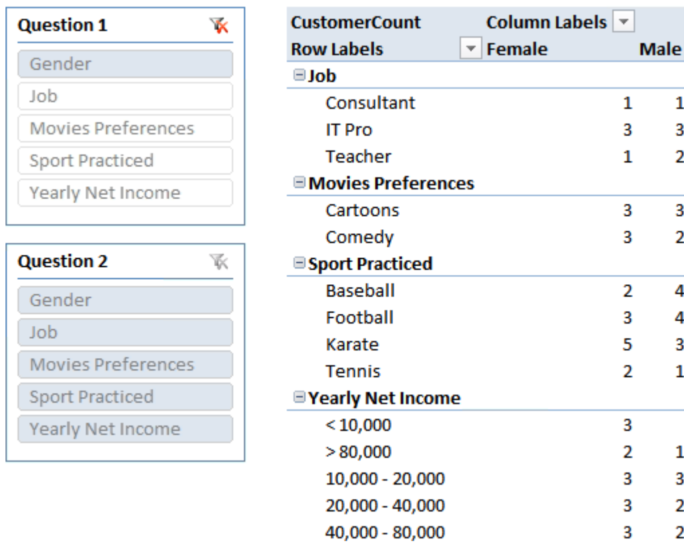
- The other point is that, because we have already filtered the Gender question, we might not be interested in looking again at the Gender question on the rows. We already know that we are looking at people that have answered Male or Female to the Gender question.

The latter issue is the most interesting one, because it has a very neat solution in DAX by means of using filter contexts. What we really want to ask is "Given the question in Filter1, shown what people have answered to other questions, I do not really mind the question I have already selected, only different ones".

The final formula looks like this:

```
CustomerCount =
=IF (
    COUNTROWS (VALUES (Filter1[ID_Answer])) = 1 && COUNTROWS (VALUES (Filter2[ID_Answer])) =
1,
    CALCULATE (
        CALCULATE (
            COUNTROWS (Customers),
            FILTER (
                Customers,
                CALCULATE (
                    COUNTROWS (Answers),
                    Answers[ID_Answer] = VALUES (Filter2[ID_Answer]))
                > 0
            )
        ),
        FILTER (
            Customers,
            CALCULATE (
                COUNTROWS (Answers),
                Answers[ID_Answer] = VALUES (Filter1[ID_Answer]))
            > 0
        ),
        Filter2[Question 2] <> VALUES (Filter1[Question 1])
    )
)
```

We have added a condition to the outer CALCULATE where we basically say that we are not interested, in the count, in the situation where the question in Filter2 is the same question already selected in Filter1. This simple condition removes the annoying duplicates. The final report is the one shown at the beginning of this section and shown again in Figure 122.

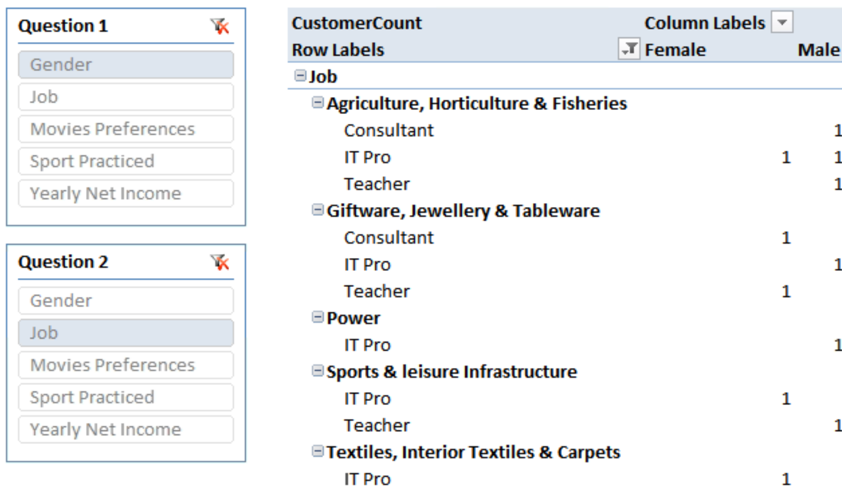


CustomerCount	Column Labels	
Row Labels	Female	Male
⊟Job		
Consultant	1	1
IT Pro	3	3
Teacher	1	2
⊟Movies Preferences		
Cartoons	3	3
Comedy	3	2
⊟Sport Practiced		
Baseball	2	4
Football	3	4
Karate	5	3
Tennis	2	1
⊟Yearly Net Income		
< 10,000	3	
> 80,000	2	1
10,000 - 20,000	3	3
20,000 - 40,000	3	2
40,000 - 80,000	3	2

Figure 122 – Survey with more checks leads to better results

Filter1 now selects the Gender question and the same question is no longer shown on the rows, because it would be useless to repeat the same information.

Clearly, the presence of other attributes in the customer table, like the Category, makes investigation of information even more interesting. The formula works fine even if we add more filters to the Customers table by means of selecting a specific category, as we show in the report in Figure 123, where the question in Filter2 has been fixed and the category has been added to the rows.



CustomerCount	Column Labels	
Row Labels	Female	Male
⊟Job		
⊟Agriculture, Horticulture & Fisheries		
Consultant		1
IT Pro	1	1
Teacher		1
⊟Giftware, Jewellery & Tableware		
Consultant	1	
IT Pro		1
Teacher	1	
⊟Power		
IT Pro		1
⊟Sports & leisure Infrastructure		
IT Pro	1	
Teacher		1
⊟Textiles, Interior Textiles & Carpets		
IT Pro	1	

Figure 123 – Adding other columns to the pivot table makes analysis more interesting

Denali Implementation

As we said during the description, the Denali version of the formula is not working in the release used to create this document. Nevertheless, we feel that it is important to show the complete formula so that you can try working on it when the next version of SQL Server will be available.

Here is the complete formula:

```
IF (
    HASONEVALUE (Filter1[ID_Answer]) && HASONEVALUE (Filter2[ID_Answer]),
    CALCULATE (
        CALCULATE (
            COUNTROWS (Customers),
            CALCULATETABLE (
                SUMMARIZE (Answers, Customers[ID_Customer]),
                USERELATIONSHIP (Answers[ID_Answer], Filter2[ID_Answer])
            )
        ),
        CALCULATETABLE (
            SUMMARIZE (Answers, Customers[ID_Customer]),
            USERELATIONSHIP (Answers[ID_Answer], Filter1[ID_Answer])
        ),
        Filter2[Question 2] <> VALUES (Filter1[Question 1])
    )
)
```

You can see that the formula is much more elegant and clear when compared with the previous version. Unfortunately, due to a bug in the beta release of the product, this formula will crash the engine. The problem seems to be related with USERELATIONSHIP and this slightly modified version of the same formula works fine:

```
IF (
    HASONEVALUE (Filter1[ID_Answer]) && HASONEVALUE (Filter2[ID_Answer]),
    CALCULATE (
        CALCULATE (
            COUNTROWS (Customers),
            CALCULATETABLE (
                SUMMARIZE (Answers, Customers[ID_Customer]),
                Answers[ID_Answer] = VALUES (Filter2[ID_Answer])
            )
        ),
        CALCULATETABLE (
            SUMMARIZE (Answers, Customers[ID_Customer]),
            USERELATIONSHIP (Answers[ID_Answer], Filter1[ID_Answer])
        ),
        Filter2[Question 2] <> VALUES (Filter1[Question 1])
    )
)
```

Simply avoiding the innermost USERELATIONSHIP makes the formula work even with CTP3 of Denali.

Performance Analysis

The formula in the Survey is pretty complex. Thus, we performed some tests in order to check how far it can be pushed with volumes of data. The main parameters are:

- Number of customers (dimension)
- Number of answers (fact table)
- Number of questions (dimension)

Data was generated randomly and we tested basically two different scenarios:

- **Few questions, with many customers and, obviously, many answers**

In this scenario we kept the number of questions very low (5 questions only, with 20 answers in total) and we increased the number of customers up to 1 million. The number of answers is always 5 times the number of customers, so that – on average – each customer provided 5 answers.

We used, as a test, the same report shown in the previous figures.

The performances are very good up to 100.000 customers (and 500.000 answers), because the report is rendered in less than 2 seconds. However, they become unacceptable at 1 million customers and 5 million answers, because the report is finished in 20 seconds, which is below our usability limit even if still reasonable for such amount of customers.

- **Many questions, with average customers and many answers**

  In this scenario, we increased the number of questions to 100, then 1,000 and finally 10,000. The number of customers has been fixed to 100,000 while the number of answers has been increased to 20 times the number of customers. The goal of this test was to determine the complexity of the formula in relation to the number of questions.

  Because the increasing number of questions would make the report unusable, we used a different report that filters both a question in Filter1 and a question in Filter2, in order to avoid a huge and increasing number of cells in the report.

  Using PowerPivot 1.0, 1000 questions is the limit and 100 is the "good number". As soon as the number of questions reaches 1,000, the performance of the PivotTable are pretty bad and at 10,000 it is no longer usable.

  The same test, made on version 2 of PowerPivot, shows that 10,000 questions is still a reasonable number. The report is rendered in a few seconds and provides a good experience.

Thus, the conclusion is that the data model and the formula are very strongly tied to the number of questions: increasing the number over 100 (10,000 in Denali) leads to poor performances. On the other hand, the number of customers and of answers can be pretty big, leading to a good experience even with 1 million of customers.

Multiple Groups

Users want to group items in many and unpredictable ways. For example, we might want to group all the customers who live in a specific city and have some characteristics in a group, give that group a name and then analyze the behavior of this group of customers. Even if we can leverage a PivotTable to perform all of this, a very useful feature is that of saving the group under a name, so that we can retrieve the selection very quickly.

A simple data model that fulfills this requirement is shown in Figure 124.

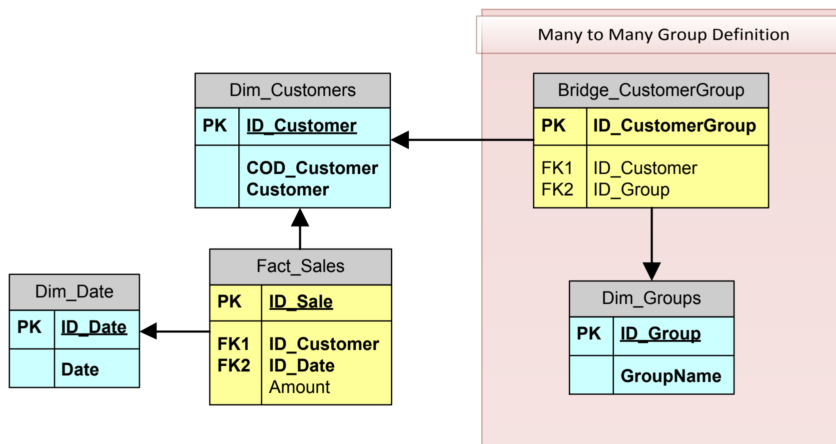


Figure 124 – Multiple groups diagram

By means of using a many-to-many relationship, we can group customers under groups. This pattern is identical to a classical many-to-many model. The interesting point is in the usage we are doing of the data model, not in the DAX formulas we are going to write. Moreover, focusing on the usage, we want to spend some time discussing how we can implement this pattern in both the server version of BISM (i.e. Tabular) or in the client one (i.e. PowerPivot).

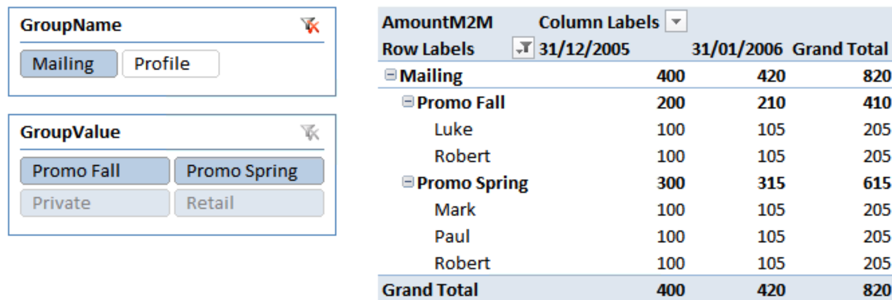
In fact, the multiple groups pattern is very useful in a self-service BI environment, where each user can define a custom grouping in a very flexible way. Unfortunately, since changing the group definition requires an update of the data in the model, this flexibility is somehow lost in a multi user environment where data resides on a server.

The formula for the AmountM2M is straightforward:

```
AmountM2M =

CALCULATE (
    SUM (Fact_Sales[Amount]),
    FILTER (
        Dim_Customers,
        CALCULATE (COUNTROWS (Bridge_CustomerGroup) > 0)
    )
)
```

With this basic formula, we get the desired result of custom grouping of customers, as you can see in Figure 125.



AmountM2M	Column Labels		
Row Labels	31/12/2005	31/01/2006	Grand Total
Mailing	400	420	820
Promo Fall	200	210	410
Luke	100	105	205
Robert	100	105	205
Promo Spring	300	315	615
Mark	100	105	205
Paul	100	105	205
Robert	100	105	205
Grand Total	400	420	820

Figure 125 – Excel custom grouping with slicers

It is worth noting that Excel 2010 slicers really shine with this data model since they make the filtering of groups very convenient.

Now, the interesting discussion about this data model is about how a user can update the bridge table containing the custom groupings. In a server driven environment, the table has to reside in an Analysis Services database. In such a scenario, IT needs to build some mechanism to let the user update the table and reprocess the bridge table on the server. This can be done with some coding and the usage of the AMO libraries.

If the model is built inside PowerPivot, then a much easier implementation can be done by using linked tables. By means of creating an Excel table that is then linked in PowerPivot, we can very easily update the groups without the need to make server trips. In such a scenario, it might be useful to denormalize the data model creating a bridge table that does not contain the customer and group keys. The bridge table can be created in a completely denormalized way like in the model shown in Figure 126.

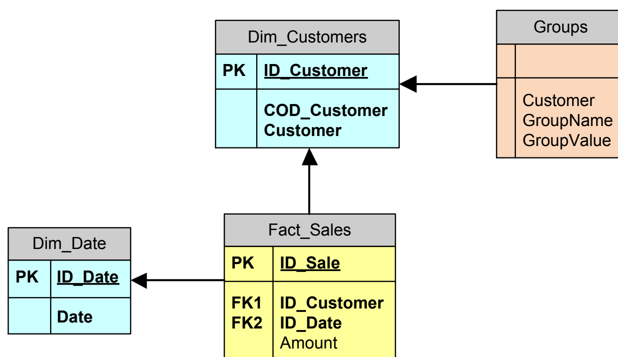


Figure 126 – Denormalized structure for multiple groups on PowerPivot

Because Vertipaq compresses data in a very good way, we can easily put the name of the customer, group and value directly inside the bridge table, letting the user load names in the Excel table instead of complex identifiers. Clearly, the formula needs to be updated to reflect the changes in the data model, but it is very easy to write.

We do not provide performance analysis for this data model for a couple of reasons:

- The group dimension is normally very small, so performance is not a big issue when many-to-many relationships are used in this type of scenario.
- The data model is identical to a classical many-to-many one. Thus, the same performance considerations apply here.

Transition Matrix

Transition Matrix is a very common scenario where we want to analyze the changes in a particular attribute of a table. A common example is for customer segmentation: “how many customers classified with rating A in 2010 have been classified type B in 2011”?

There are basically two different data model that could be used to model this scenario:

- Slowly changing dimension of type 2, where we save a new version of a customer every time the rating (or any other attribute) changes. The fact table points to the current version of the customer at the time the fact has been recorded.
- Historical attribute tracking. In this scenario, the rating of the customer is saved in the fact table and the customer is treated as an SCD of type 1, without historical tracking of the attributes.

In Figure 127 you can see these two data models.

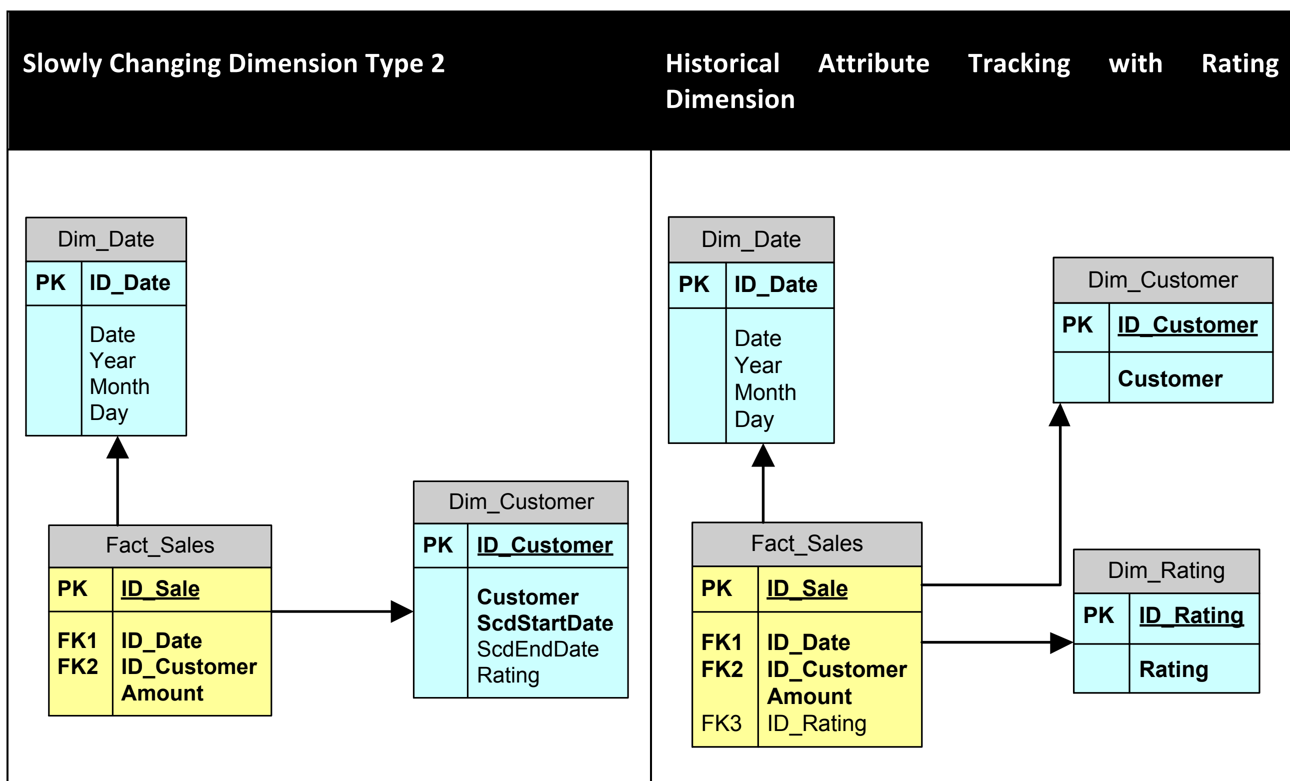


Figure 127 – Transition matrix diagrams

It is worth noting that the second model is not a completely correct one because, if no sales happen when a customer changes the rating, then the rating change itself will not be stored inside the data model. Nevertheless, both data models are widely used and we want to discuss both even if, from a data-modeling point of view, we strongly suggest to avoid the Historical Attribute Tracking in a BISM Tabular model. In Multidimensional, the usage of the HAT data model is sometimes necessary due to performance reasons.

From the analytical point of view, this scenario clearly requires two calendar tables: one will let us select the starting point, the other will be used for the end point. The data model will perform the computation

by selecting the customer that had a specific rating at the starting point, and by slicing them with the rating they had at the ending point.

We will see two of the possible solutions to this scenario:

- **Snapshot table**: to implement this solution we will need to create a snapshot of the rating and use that table to perform the computation.
- **Calculated columns**: this solution does not require the creation of a snapshot table and leverages on the calculated columns.

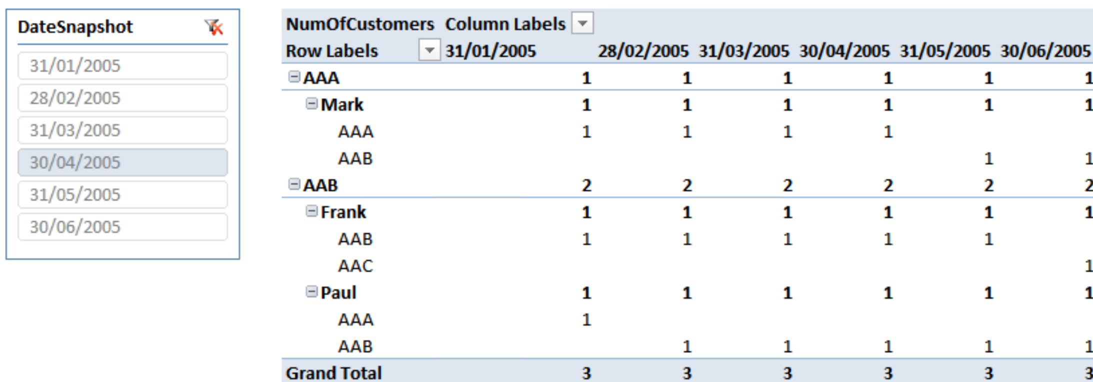
Each of these data models can be implemented over the SCD or the HAT scenarios, leading to four different formulas that we need to analyze.

We start with this set of data:

Customer	Rating	StartDate	EndDate
Mark	AAA	20050131	20050531
Mark	AAB	20050531	
Paul	AAA	20050131	20050228
Paul	AAB	20050228	
Frank	AAB	20050131	20050630
Frank	AAC	20050630	

Table 18 – Customer data for the Transition Matrix Model

We have three customers who changed their rating in different time periods and, at the end, we want to be able to pivot over the data model to produce results like the one shown in Figure 128.



NumOfCustomers Row Labels	Column Labels 31/01/2005	28/02/2005	31/03/2005	30/04/2005	31/05/2005	30/06/2005
AAA	1	1	1	1	1	1
Mark	1	1	1	1	1	1
AAA	1	1	1	1		
AAB					1	1
AAB	2	2	2	2	2	2
Frank	1	1	1	1	1	1
AAB	1	1	1	1	1	
AAC						1
Paul	1	1	1	1	1	1
AAA	1					
AAB		1	1	1	1	1
Grand Total	3	3	3	3	3	3

Figure 128 – Transition matrix Example

What does this report show? We have fixed a date on the DateSnapshot slicers (30/04/2005 in the example) and we want to analyze the customers who had a specific rating at that date, showing how their rating changed over time. For example, Mark had a rating of AAA at April 2005 and the report shows that it had the same rating until April and then switched to AAB on May.

Transition Matrix with Snapshot Table

A snapshot table records the values of the attributes in different points in time. For example, in our scenario we can create a monthly snapshot that would look like this:

DateSnapshot	CustomerSnapshot	RatingSnapshot
20050131	Frank	AAB
20050131	Mark	AAA
20050131	Paul	AAA
20050228	Frank	AAB
20050228	Mark	AAA
20050228	Paul	AAB
20050331	Frank	AAB
20050331	Mark	AAA
20050331	Paul	AAB
20050430	Frank	AAB
20050430	Mark	AAA
20050430	Paul	AAB
20050531	Frank	AAB
20050531	Mark	AAB
20050531	Paul	AAB
20050630	Frank	AAC
20050630	Mark	AAB
20050630	Paul	AAB

Table 19 – Snapshot table

Each customer is repeated for each month, recording the value of the rating at the end of that month. It is clear that snapshot tables have the annoying characteristic of fixing the time window. In this case, having created a monthly snapshot, we will not be able to perform analyses that have a greater granularity than the month. Nevertheless, snapshot tables can be easily created through some ETL code starting from the original data.

Snapshot Table in the Slowly Changing Dimension Scenario

The analytical data model of the snapshot table in the slowly changing dimension pattern looks like the one in Figure 129.

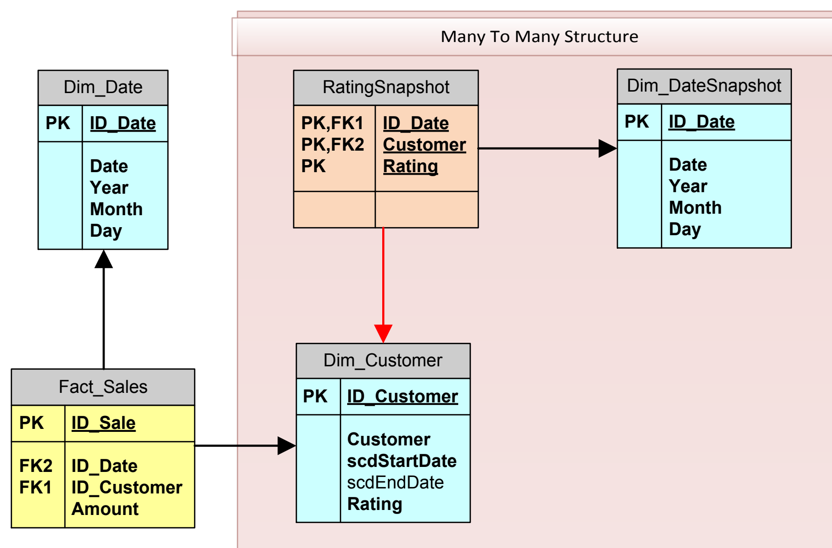


Figure 129 – Transition matrix SCD Diagram

The many-to-many structure is between the date and the customer dimensions, through the snapshot. It is very important to note that the relationship between the snapshot and the customer dimension is not a real relationship. In fact, the customer dimension has a surrogate key that might change during the month and, in consequence of that, we need to track the relationship using the customer code or the customer name. In our example, we used the customer name in order to reduce the number of columns in the tables. In a real world scenario, we would use the customer natural key. This means that a single row in the snapshot table might be related to more than one row in the customer table for the same customer. This situation is definitely something we need to take into account when writing the DAX formulas, because we will not be able to design the relationship inside the data model. Nevertheless, we already know how to use DAX to mimic relationships. Thus, we will leverage advanced DAX filtering instead of following the classical usage of relationships.

Moreover, it is worth noting that the snapshot is not related to the date dimension but to a new date dimension called Dim_DateSnapshot. This is necessary because we are going to use the two dates for different purposes: one is used to filter the snapshot table, the other one is used to filter the sales table.

It is now time to start thinking to the algorithm. Let us start recalling the business scenario: we want to count the number of customers who had a specific rating in a point in time and analyze the changes in their rating over time. Thus, we know that the date of the snapshot will be fixed and we want to be able to filter all the customers who had a rating at that date.

We can start with the classical pattern of many-to-many to select the customers who have a rating at a specific date:

```
NumOfCustomers =

CALCULATE (
    COUNTROWS (Dim_Customers),
    FILTER (
        Dim_Customers,
        CALCULATE (
            COUNTROWS (RatingSnapshot),
            RatingSnapshot[CustomerSnapshot] = EARLIER (Dim_Customers[Customer])
        ) > 0
    )
)
```

This formula computes the number of customers after having filtered the ones that have a specific rating as defined by the snapshot table. It is worth to note the additional condition inside the inner CALCULATE, which is used to force the formula to take into account the "relationship" between the snapshot and the customer dimension through the customer name.

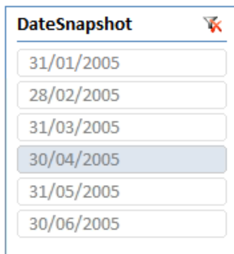
Nevertheless, this formula is not still making what it is supposed to do. In fact, the filter on the customer name always returns all the instances of the customer, regardless of the date. In fact, if we filter the date dimension, that filter will have no effect on the customer dimension, because there are no relationships that tie the customer and the date together. Again, we need to leverage DAX to impose such a filter condition, seeking only the instances of the customer that are active in the date period selected. This consideration will lead us to the final formula:

```
NumOfCustomers =

CALCULATE (
    COUNTROWS (DISTINCT (Dim_Customers[Customer])),
    FILTER (
        Dim_Customers,
        CALCULATE (
            COUNTROWS (RatingSnapshot),
            RatingSnapshot[CustomerSnapshot] = EARLIER (Dim_Customers[Customer])
        ) > 0 &&
        Dim_Customers[scdStartDate] <= MAX (Dim_Date[ID_Date]) &&
        (
            Dim_Customers[scdEndDate] > MIN (Dim_Date[ID_Date]) ||
            ISBLANK (Dim_Customers[scdEndDate])
        )
    )
)
```

By leveraging the scdStartDate and scdEndDate, we further restrict the filter on the customers in order to consider only the instances of the customers that are active in the date selection. Moreover, knowing that during a time period a customer might be returned more than once (for example, it might have changed rating three times in a month), we have changed the COUNTROWS counting the distinct customer names instead of the number of rows. In this way, we always count different instances of the same customer only once.

With this knowledge, we can now analyze the table in Figure 130 and give a correct meaning to the total for 2005, where Mark is counted with rating AAA and AAB as one customer but is counted only once on the grand total since the two instances do not need to be summed up.



NumOfCustomers	Column Labels						
	2005						2005 Total
Row Labels	31/01/2005	28/02/2005	31/03/2005	30/04/2005	31/05/2005	30/06/2005	
AAA	1	1	1	1	1	1	1
Mark	1	1	1	1	1	1	1
AAA	1	1	1	1			1
AAB					1	1	1
AAB	2	2	2	2	2	2	2
Frank	1	1	1	1	1	1	1
AAB	1	1	1	1	1		1
AAC						1	1
Paul	1	1	1	1	1	1	1
AAA	1						1
AAB		1	1	1	1	1	1
Grand Total	3	3	3	3	3	3	3

Figure 130 – Transition matrix SCD Example

It is evident that we can use the same pattern to compute other values, as the amount sold or any other interesting information.

Snapshot Table in the Historical Attribute Tracking Scenario

The bridge table in the scenario with Historical Attribute Tracking is slightly different and can be seen in Figure 131.

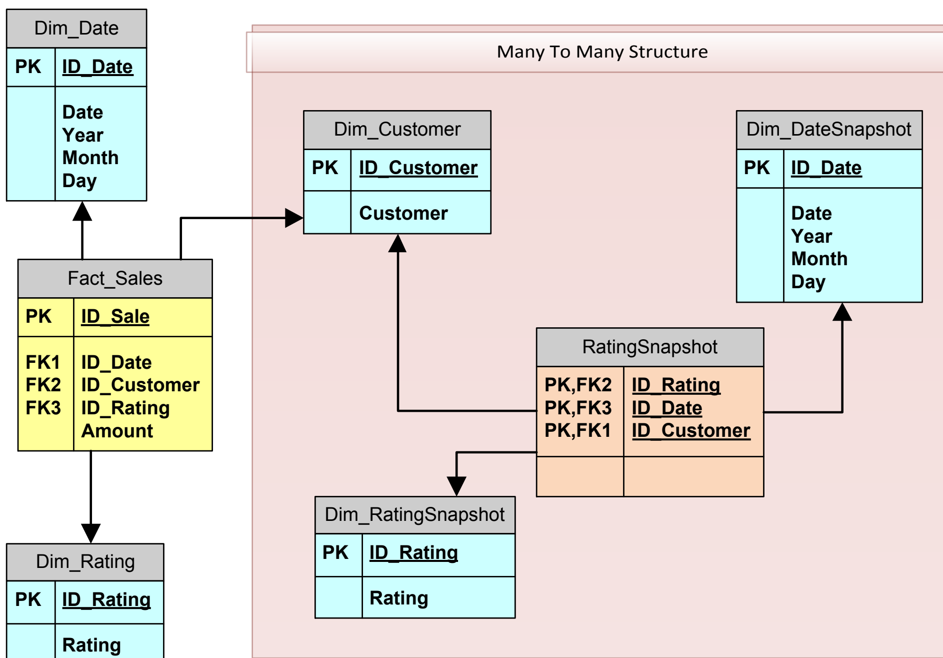


Figure 131 – Transition matrix HAT diagram

The main differences between the previous data model and this one are:

- The presence of a rating dimension. The Rating dimension can be avoided, including the rating directly inside the snapshot. However, in this case we decided to keep it as a separate dimension to maintain coherence with the rating dimension that models the historical rating. From an implementation point of view, the two dimensions can be created as two different views of the same relational table.

- The relationship with the customer is a classical relationship, because each customer is present only once in the dimension: no SCD has been created for the customers in this data model.

In this scenario, the first complex part is the creation of the snapshot table because, having the attribute tracking inside the fact table, some variations might be missing. Nevertheless, this is an issue related to the ETL code and is not something we need to discuss here.

When it comes to writing the DAX formula, we can now leverage the relationships to filter the snapshot table with the customers. Thus, the complex part of the previous scenario is missing now. Nevertheless, since we want to count the number of customers, we need to perform the filtering of the customers in two steps:

- First, we filter the customers who had a specific rating at the date indicated by the DimDateSnapshot date. This will be done by using the classical many-to-many pattern.

- Then, we still need to filter the customers who have a specific rating at the date indicated by the Dim_Date. Remember that, this time, the information about the rating is no longer available in the Dim_Customer. This time, the information is stored inside the Fact_Sales table. Thus, we will use a similar patter to that of the many-to-many to filter only the customers that, in the specified date range, have at least one sale with the specified rating.

With these considerations in mind, the formula is (almost) straightforward:

```
Dim_Customers =

CALCULATE (
    COUNTROWS (
        FILTER (
            Dim_Customers,
            CALCULATE (COUNTROWS (Fact_Sales)) > 0
         )
    ),
    FILTER (
        Dim_Customers,
        CALCULATE (COUNTROWS (RatingSnapshot)) > 0
    )
)
```

You can easily see, in the formula, the presence of the two-steps filtering, one considering the RatingSnapshot table as the bridge table in a classical many-to-many relationship, the other (the innermost) using the Fact_Sales as a bridge table in a second many-to-many relationship between the customers and the ratings.

In this scenario, the formula to count the customer is slightly more complex than any formula that simply needs to aggregate values from the fact table. In fact, in this scenario the formula is more similar to a cascading many-to-many relationship. If we were to aggregate values from the fact table, then the innermost CALCULATE could be avoided because a filter on Dim_Rating directly applies to the fact table, whereas it needs an additional step to be applied to the customers.

The Denali formula, for the same scenario, is identical to the Cascading many-to-many pattern and we leave it as an exercise to the reader.

From a performance point of view, this formula is definitely slower than the previous one, requiring an additional CALCULATE step over the Fact_Sales table. Because we expect the Fact_Sales table to be the biggest among all of our tables, avoiding touching it would be welcome.

If no sales have been recorded when a customer had a specific rating, then the information about the rating change will not be reported by the query in a correct way. Even if we can write formula that simply uses the snapshot table to gather all rating variations, we already know that our system will report wrong data in case more than a single variation happens in a single month. There is no solution to this issue: as we have already discussed, the HAT data model is not the best one to use whenever there is the need to track attribute variations.

Thus, these are the conclusions:

- The SCD data model leads to very fast formulas and always returns correct information about rating variations.

- The HAT data model has a more complex (and slower) formula to compute the count of customers, due to the presence of a cascading many-to-many relationship.

- The HAT data model can return incomplete data if data is missing from the fact table (something that will definitely happen when the customer stops buying from us).

These are the reasons for which we strongly suggest to avoid using the HAT data model and always use a more canonical SCD of type 2 handling in order to record attribute variations. The data model is more accurate and the formulas are easier to write.

Transition Matrix with Calculated Columns

There is an interesting alternative to the creation of a snapshot table that implies the usage of calculated columns. This solution is interesting in some scenarios:

- When the time granularity of the attribute changing can be fixed at a high level (i.e. at the year level) and the data model is stored on a server. In this case, we lose some flexibility but we get a much simpler model both to query and to develop.

- When the data model is to be queried by using PowerPivot for Excel. In this case, we can leverage the interactive nature of PowerPivot and dynamically compute calculated columns thanks to user defined parameters in linked tables.

The basic idea of using calculated columns is to create, at the customer level, a calculated column that contains the value of the rating at a specified point in time. For example, we might want to create one calculated column in the customer table that contains Rating2005, one for Rating2006 and so on. It is clear that, using this solution, we will not be able to look at the customers who had a specific rating at March 2005, because the only column available would be that of the year 2005. Nevertheless, for most of the interesting analysis, a yearly snapshot is a good compromise because we can analyze customers who had a rating at the beginning of 2005 and analyze how their rating changed during the year. The fixed date is only that of the “selection” rating.

Obviously, such a model will need to be updated each year and we will need to add a new column for each year. Because in our small example we have data for a single year, we are going to create a calculated column for each month. The code is not very difficult to write:

```
Rating_2005_01 :=

CALCULATE (
    VALUES (Dim_Customers[Rating]),
    FILTER (
        ALL (Dim_Customers),
        Dim_Customers[Customer] = EARLIER (Dim_Customers[Customer]) &&
        Dim_Customers[ScdStartDate] = CALCULATE (
            MAX (Dim_Customers[ScdStartDate]),
            Dim_Customers[ScdStartDate] <= 20050131,
            ALL (Dim_Customers),
            Dim_Customers[Customer] = EARLIER (Dim_Customers[Customer])
        )
    )
)
```

By means of replacing the constant 20050131 with 20050228, we can easily add new columns for each month of the year. In Figure 132 you can see the Dim_Customers table with three calculated columns:

ID_Custo...	Customer	Rating	scdStartDate	scdEndDate	Rating_2005_01	Rating_2005_02	Rating_2005_03
101	Mark	AAA	20050131	20050531	AAA	AAA	AAA
103	Mark	AAB	20050531		AAA	AAA	AAA
201	Paul	AAA	20050131	20050228	AAA	AAB	AAB
202	Paul	AAB	20050228		AAA	AAB	AAB
301	Frank	AAB	20050131	20050630	AAB	AAB	AAB
303	Frank	AAC	20050630		AAB	AAB	AAB

Figure 132 – Transition matrix with calculated columns

Using these columns in a PivotTable, it is straightforward to perform the selection of the customers who had a specific rating in a point in time (for example, January 2005). Nevertheless, to count the number of such customers we still need to apply a filter over the Dim_Customers to count only the customers who were active in the selected period in time. The DAX code that performs this calculation is somewhat similar to previous formulas:

```
NumOfCustomers =

CALCULATE (
    COUNTROWS (DISTINCT (Dim_Customers[Customer])),
    FILTER (
        Dim_Customers,
        Dim_Customers[scdStartDate] <= MAX (Dim_Date[ID_Date]) &&
        (
            Dim_Customers[scdEndDate] > MIN (Dim_Date[ID_Date]) ||
            ISBLANK (Dim_Customers[scdEndDate])
        )
    )
)
```

The only thing to note is the need of a DISTINCT inside the COUNTROWS, in order to count the customers once even if they appear several times in the selected time period due to many updates in the attribute.

If we want to aggregate values from the fact table, then the formulas will be much easier to write because the filtering of the customers active in the selected time period will be automatically performed by the relationship between the fact table and the calendar table.

Before diving into the details of calculated columns in the HAT model, it is worth to spend some words on the pros and cons of the calculated column approach:

- **Simpler data model**. Apart from any other consideration, this is the greatest advantage of using calculated columns instead of the previous complex data models. Remember that having a simpler data model often means gaining a lot in terms of query speed and user experience.

  The previous models always had two date dimensions: one for the sales fact table and one for the snapshot one. In our experience, having more than one calendar table is quite always a nuisance and the calculated column data model avoids this.

- **Query speed**. This data model uses calculated columns, which are computed at process time, reducing to a minimum the effort needed to answer even complex queries.

- **ETL**: this data model does not require any kind of ETL, because there is no need to create a snapshot table (which is commonly created by an ETL step). This simple consideration makes it the perfect solution for self-service BI scenarios where speed of development is normally the main decision driver.

- **Flexibility**. From a flexibility point of view, the complex data models are obviously the best. Nevertheless, we strongly suggest checking whether this flexibility is really needed or not. We are not saying that flexibility is not one of the major goals of any BI solution but, because this flexibility comes at a cost, we advise to double check your options before taking a choice.

- **Maintenance**. The calculated column approach needs some maintenance. Every time we will need a new column, we will have to spend some time to define it inside the data model. This is a clear disadvantage of the solution, and you should be aware of that.

The last two points are clear disadvantages and cannot be easily addressed in a server-driven BI solution, i.e. when the data model is hosted in a server (SharePoint or SSAS). Nevertheless, in a scenario where we are just using PowerPivot for Excel to analyze data, there is a simple solution that quickly solves both problems.

Let us recall the formula of the calculated column:

```
Rating_2005_01 =

CALCULATE (
    VALUES (Dim_Customers[Rating]),
    FILTER (
        ALL (Dim_Customers),
        Dim_Customers[Customer] = EARLIER (Dim_Customers[Customer]) &&
        Dim_Customers[ScdStartDate] = CALCULATE (
            MAX (Dim_Customers[ScdStartDate]),
            Dim_Customers[ScdStartDate] <= 20050131,
            ALL (Dim_Customers),
            Dim_Customers[Customer] = EARLIER (Dim_Customers[Customer])
        )
    )
)
```

Each calculated column will be identical to this one except for the constant that contains the date of the rating (highlighted in the previous formula). If we were able to transform that constant into a parameter, definable by the user, then we would be able to marry the flexibility of the complex data models with the simplicity of the calculated column approach.

Luckily, this can be easily accomplished by using a parameter table. We can define an Excel table like the one in Figure 133.

	A	B	C	D
1				
2		DateOfHistoricalRating		
3		20050228		
4				
5				
6				

Figure 133 – Transition matrix: parameter table with PowerPivot

This table, named "Parameters", contains only one row and one column and should be loaded inside PowerPivot as a linked table. If, in the future, we will need more parameters, we could always add columns to the table. It is mandatory that only one row exists in the table to make the solution works.

Because the table has only one row, the expression VALUES (Parameters[DateOfHistoricalRating]) will always contain a single value. Thus, DAX will automatically convert it into a scalar value whenever it is needed. We can define the calculated column as:

```
Snapshot_Rating =

CALCULATE (
    VALUES (Dim_Customers[Rating]),
    FILTER (
        ALL (Dim_Customers),
        Dim_Customers[Customer] = EARLIER (Dim_Customers[Customer]) &&
        Dim_Customers[ScdStartDate] = CALCULATE (
            MAX (Dim_Customers[ScdStartDate]),
            Dim_Customers[ScdStartDate] <= VALUES (Parameters[DateOfHistoricalRating]),
            ALL (Dim_Customers),
            Dim_Customers[Customer] = EARLIER (Dim_Customers[Customer])
        )
    )
)
```

This column will be recomputed each time the Parameter table is refreshed and will contain the value of the rating at the time contained in the Excel table. Thus, to perform analyses at any point in time, we will need to refresh the linked table, get the calculated column recomputed and use the Snapshot_Rating column to slice data.

Clearly, because this approach needs a refresh of the data model, this can work with PowerPivot for Excel only and is not affordable in a multi-user solution based on a server, including PowerPivot for SharePoint. Nevertheless, for simple self-service BI, this is by far the best approach to the Transition Matrix scenario.

The last topic is the calculated column approach with the HAT data model. While the logic is not very different, this time the DAX formula is much more complicated, because the ratings, this time, are stored inside the fact table. Thus, to compute the rating at a point in time we need to:

- Search for the last sale of the customer before the desired date
- Compute the ID_Rating of that sale
- Search in the Dim_Rating for the textual representation of that rating The complete formula is the following one:

```
Rating_2005_02 =

CALCULATE (
    VALUES (Dim_Rating[Rating]),
    FILTER (
        Dim_Rating,
        Dim_Rating[ID_Rating] = CALCULATE (
            VALUES (Fact_Sales[ID_Rating]),
            FILTER (
                Fact_Sales,
                Fact_Sales[ID_Date] = CALCULATE (
                    MAX (Fact_Sales[ID_Date]),
                    Fact_Sales[ID_Customer] = EARLIER (Dim_Customers[ID_Customer]),
                    ALL (Fact_Sales),
                    Fact_Sales[ID_Date] <= 20050228
                )
            ),
            Fact_Sales[ID_Customer] = EARLIER (Dim_Customers[ID_Customer])
        )
    )
)
```

Whenever we write a formula with three nested CALCULATE functions, we know that it will be hard to understand. Take your time and study it carefully, it is worth the effort. We cannot expect this calculated column to be computed very quickly because, for each customer, it needs to make a round over the fact table to get the last rating assigned to the customer and we expect the fact table to be a big one.

Nevertheless, as we have already pointed out before, the HAT model is not very convenient in BISM because the formulas are complex and not very fast, but if you need to face such a data model, at least you have learned how to handle it.

Basket Analysis

Basket Analysis is the search for correlations between events stored in the fact table. For this example, we are going to use the AdventureWorks DWH database, because it best fits the scenario.

The question we want to answer is "of all the customers who bought a mountain bike, how many have never bought a mountain tire tube?"

There are two different topics that make this an interesting scenario:

We will need to use the many-to-many pattern using the fact table as the bridge one to solve the problem with Tabular

We are going to search for missing values (customers who have not bought a mountain tire tube) instead of existing ones. This will make the formula somehow harder and shows the tremendous power of BISM Tabular because the final solution will be very fast, while requiring no changes to the existing data model

Let us start taking a look at a simplified version of the AdventureWorks data warehouse. In the diagram shown in Figure 134, we have removed many columns from the original tables, because we are interested in the model and not in the details of each entity.

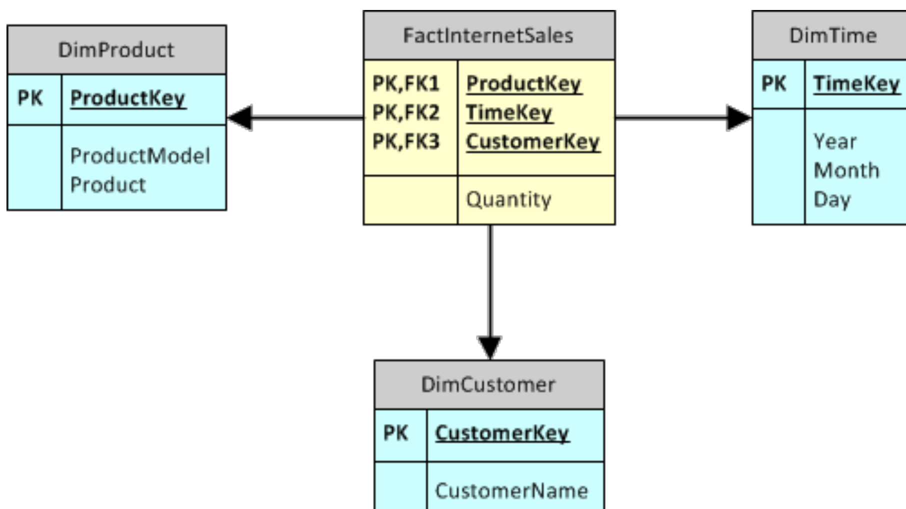


Figure 134 – AdventureWorks Data Warehouse diagram

This data model is not yet enough. Because we want to filter two kinds of products (Mountain Bike and Mountain Tire Tube) and perform different computations with these selections, we need two instances of the DimProduct table in our analytical data model. It is enough to load the DimProduct twice in the data model, giving it another name to make it evident its different meaning, as you can see in Figure 135.

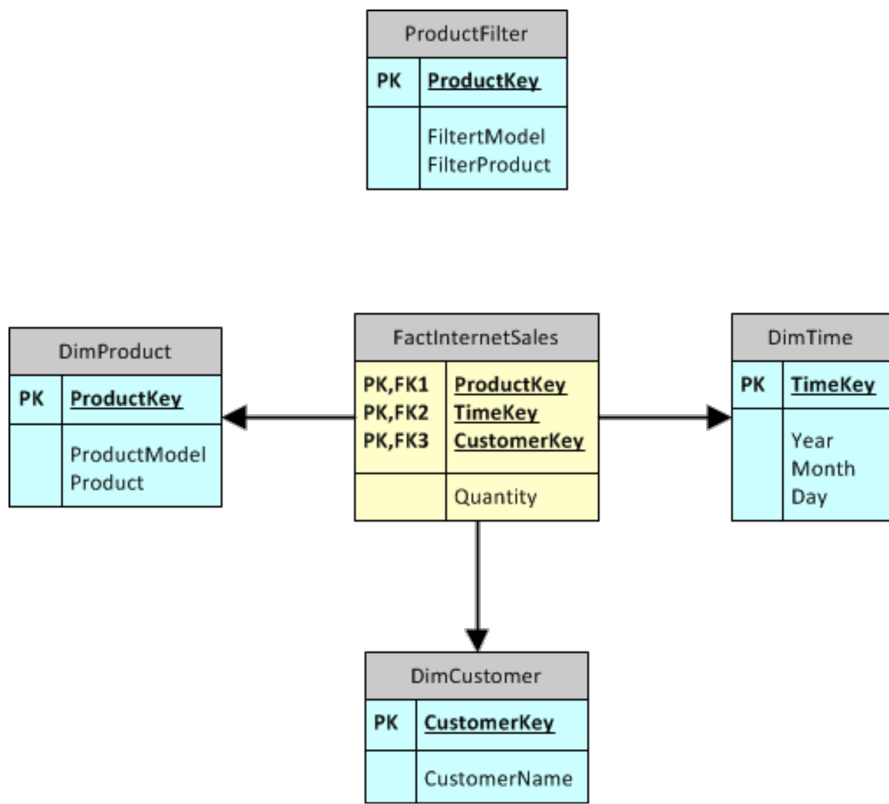
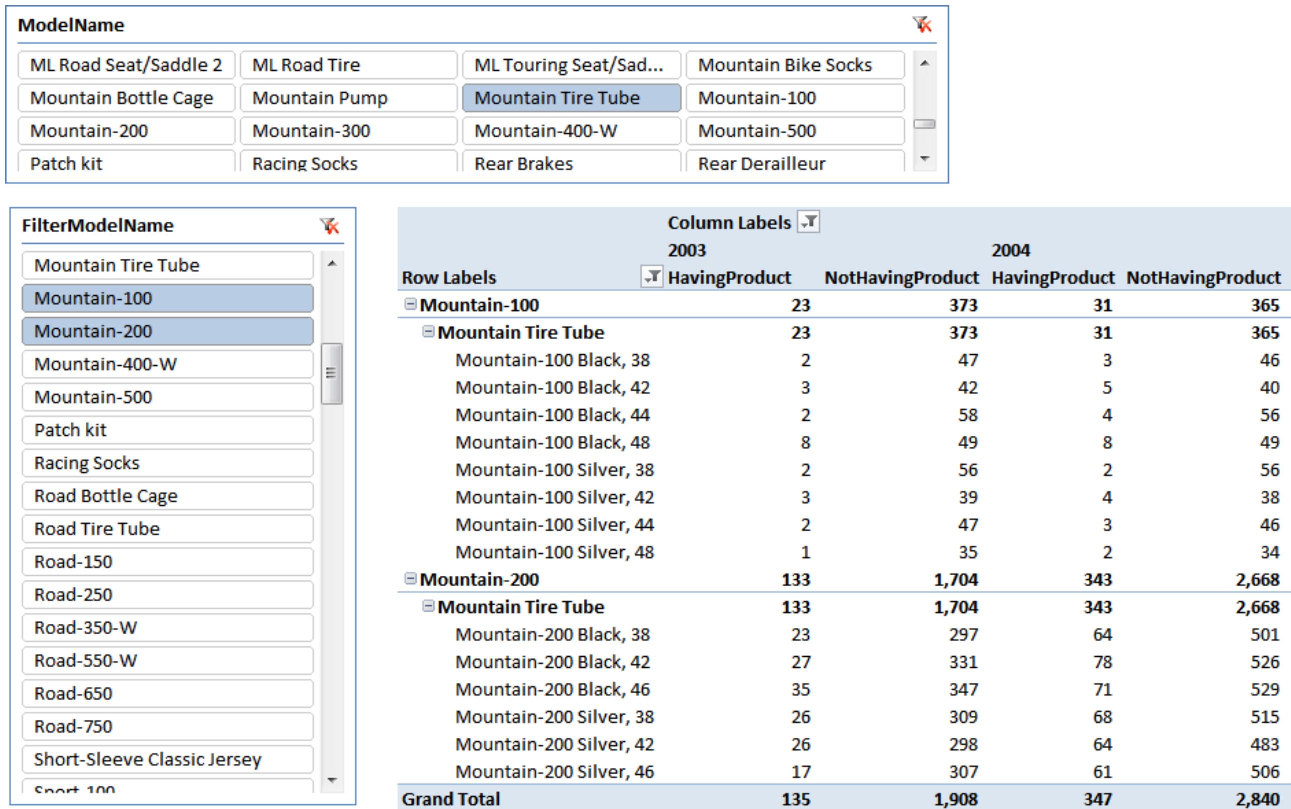


Figure 135 – Basket Analysis Analytical Model

The new table does not have any relationship with the fact table because the only available ProductKey has already been used to relate the fact table with DimProduct. Thus, we will use DAX to mimic the relationship in all of our formulas.

At the end of the chapter, we will show the same scenario solved with the Denali implementation of Vertipaq, where there is the option to create more than one relationship using a single key. We decided to show both solutions (prior to Denali and with Denali) for educational purposes: understanding the formula without the capability to create many relationships is useful.

Let us start from the final result. We want to be able to use a PivotTable to query the model and produce a report like the one in Figure 136.



	Column Labels			
	2003		2004	
Row Labels	HavingProduct	NotHavingProduct	HavingProduct	NotHavingProduct
Mountain-100	**23**	**373**	**31**	**365**
Mountain Tire Tube	**23**	**373**	**31**	**365**
Mountain-100 Black, 38	2	47	3	46
Mountain-100 Black, 42	3	42	5	40
Mountain-100 Black, 44	2	58	4	56
Mountain-100 Black, 48	8	49	8	49
Mountain-100 Silver, 38	2	56	2	56
Mountain-100 Silver, 42	3	39	4	38
Mountain-100 Silver, 44	2	47	3	46
Mountain-100 Silver, 48	1	35	2	34
Mountain-200	**133**	**1,704**	**343**	**2,668**
Mountain Tire Tube	**133**	**1,704**	**343**	**2,668**
Mountain-200 Black, 38	23	297	64	501
Mountain-200 Black, 42	27	331	78	526
Mountain-200 Black, 46	35	347	71	529
Mountain-200 Silver, 38	26	309	68	515
Mountain-200 Silver, 42	26	298	64	483
Mountain-200 Silver, 46	17	307	61	506
Grand Total	**135**	**1,908**	**347**	**2,840**

Figure 136 – Basket Analysis PivotTable

Before diving into the formula, let us spend a few words on the result:

- HavingProduct computes the number of customers who bought a mountain bike and at least one tire tube
- NotHavingProduct computes the number of customers who bought a mountain bike and no tire tubes.

We used the filter table to filter several models, which result in many products and the formula need to take this scenario into account: we do not want to force our user to select a single product

While it is not clearly visible in the figure, the values shown are always TimeToDate. This means that we want to see who bought a mountain bike and a mountain tire tube any time before 2003 or any time before 2004. Thus, the values in 2004 for the measures contain the same values of 2003 plus the values for 2004. This does not mean that the formula is additive, it means that all the time before the selected period need to be taken into account.

Now, let us start to author the formula that will compute the two measures. We start with HavingProduct, which is somehow easier to create. The key points are:

We need to filter the customer who have bought one of the products selected by ProductFilter any time before the end of the selected period

For these customers, we need to check whether they have bought one of the products selected by DimProduct in the same period

Let us start with the first point. The set of customers who bought one of the products in FilterProduct can be computed in this way:

```
FILTER (
    DimCustomer,
    SUMX (
        ProductFilter,
        CALCULATE (
            COUNTROWS (FactInternetSales),
            ALL (FactInternetSales),
            FactInternetSales[CustomerKey] = EARLIER (DimCustomer[CustomerKey]),
            FactInternetSales[ProductKey] = EARLIER (ProductFilter[ProductKey]),
            FILTER (
                ALL (DimTime),
                DimTime[TimeKey] <= MAX (DimTime[TimeKey])
            )
        )
    ) > 0
)
```

There are some interesting techniques used in this formula:

Even if the pattern is very similar to the classical many-to-many, we are now making a SUMX over ProductFilter and check if the final result of SUMX is greater than zero. Each iteration over ProductFilter checks for a single product. In this way, the filter returns all the customers who bought at least one of the products filtered by the slicer.

The highlighted rows are used to mimic the relationship between ProductFilter and the fact table. It is worth to note that we need to use filters to add both the relationship with DimCustomer and ProductFilter because the ALL (FactInternetSales) removes all the filters on the table, which is required in order to clear the filter automatically introduced by the relationship with DimProduct.

The filter on DimTime is required to make the computation consider all the periods of time before the end of the currently selected period. The calculation considers all the individuals who have bought a mountain bike “any time before 2003”.

Now, this filter is not very useful as a filter alone, it needs to be used inside a CALCULATE to compute the values we are interested in. The formula for HavingProduct is, at this point, straightforward:

```
HavingProduct:=
    CALCULATE (
        COUNTROWS (DISTINCT (FactInternetSales[CustomerKey])),
        FILTER (
            ALL (DimTime),
            DimTime[TimeKey] <= MAX (DimTime[TimeKey])
        ),
        FILTER (
            DimCustomer,
            SUMX (
                ProductFilter,
                CALCULATE (
                    COUNTROWS (FactInternetSales),
                    ALL (FactInternetSales),
                    FactInternetSales[CustomerKey] = EARLIER (DimCustomer[CustomerKey]),
                    FactInternetSales[ProductKey] = EARLIER (ProductFilter[ProductKey]),
                    FILTER (
                        ALL (DimTime),
                        DimTime[TimeKey] <= MAX (DimTime[TimeKey])
                    )
                )
            ) > 0
        )
    )
```

The final filter of the first CALCULATE is the filter we have already discussed, the COUNTROWS simply counts the number of customers, the only part of the formula that needs some attention is the duplicate filter over the DimTime, which appears both in the outer and in the inner calculate.

The condition is not duplicated, indeed. If you look carefully at the DAX formula you will discover that the FILTER over DimCustomer and the FILTER over DimTime in the outer CALCULATE are computed in parallel inside the same filter context and they contribute to change the filter context of the COUNTROWS. Nevertheless, these two filters do not interact with each other.

Moreover, the two filters over time represent different meanings:

- The outer one means: “who have bought a mountain tire tube any time before now”
- The inner one means: “who have bought a mountain bike any time before now”

Thus, changing the way the two filters over time are applied you can easily change the semantics of the formula to match your exact specifications.

What makes Basket Analysis a very interesting scenario is that you can use the very same technique to compute the value of NotHavingProduct. The formula of NotHavingProduct is the following:

```
NotHavingProducts :=
    CALCULATE (
        COUNTROWS (
            FILTER (
                DimCustomer,
                CALCULATE (COUNTROWS (FactInternetSales)) = 0
            )
        ),
        FILTER (
            ALL (DimTime),
            DimTime[TimeKey] <= MAX (DimTime[TimeKey])
        ),
        FILTER (
            DimCustomer,
            SUMX (
                ProductFilter,
                CALCULATE (
                    COUNTROWS (FactInternetSales),
                    ALL (FactInternetSales),
                    FactInternetSales[CustomerKey] = EARLIER (DimCustomer[CustomerKey]),
                    FactInternetSales[ProductKey] = EARLIER (ProductFilter[ProductKey]),
                    FILTER (
                        ALL (DimTime),
                        DimTime[TimeKey] <= MAX (DimTime[TimeKey])
                    )
                )
            ) > 0
        )
    )
```

The only change is in the highlighted part of the formula, which now counts the number of customers who did not buy any product, among the filtered ones. All the rest of the formula is identical to the previous one.

It is interesting to note that in DAX you can compute different values using the same pattern. Making the same computation in Multidimensional with MDX is very complex and the best way to perform the computation of NotHavingProduct is to change the data model adding a table and some ETL. This additional step is not necessary in Tabular and clearly shows the power of Tabular.

Denali Implementation

All the code shown up to now works with PowerPivot 1.0 and we decided to show it for educational purposes, because the code in DAX 1.0 clearly shows the technique that need to be used. Nevertheless, we want to extend this chapter with some final considerations writing the very same formula with Denali.

- A note of warning. These formulas, in the CTP3 version of Denali, might make PowerPivot to crash. As it happened with the previous scenario, the usage of USERELATIONSHIP can cause problems to the beta release of the product.

The most interesting feature we can leverage in Denali, regarding this formula, is the capability to create more than one relationship using the same column and mark these relationships as inactive. An inactive

relationship is a relationship that holds true but is not used by default by the Vertipaq engine. If we want to wake the relationship up, we need to use the USERELATIONSHIP keyword.

Let us start with the data model in Denali shown in Figure 137.

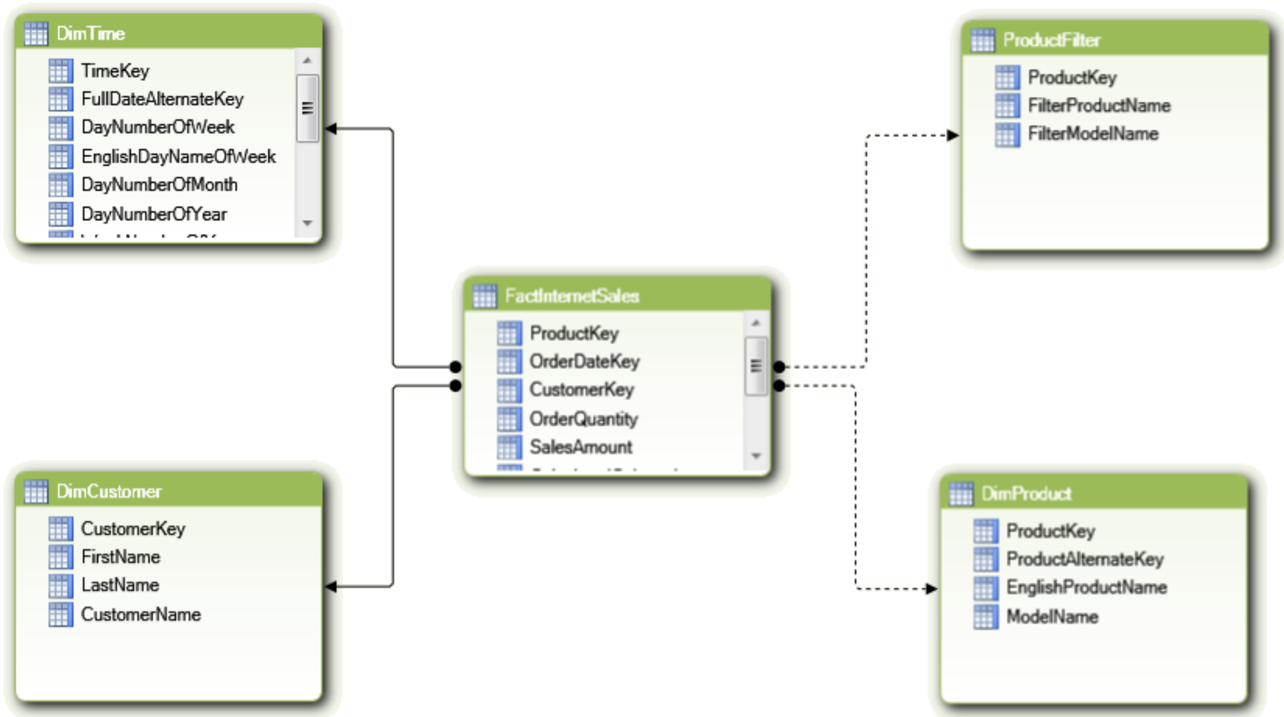


Figure 137 – Basket Analysis Data Diagram in PowerPivot "Denali"

You can clearly see the two dotted lines connecting FactInternetSales with both ProductFilter and DimProduct. These dotted lines represent inactive relationships. Thus, filtering DimProduct or ProductFilter will not result in a cross filtering of FactInternet sales.

With this new data model, the formula of HavingProduct becomes:

```
HavingProducts :=

CALCULATE (
    COUNTROWS (
        FILTER (
            DimCustomer,
            CALCULATE (
                CountRows (FactInternetSales),
                USERELATIONSHIP (FactInternetSales[ProductKey], DimProduct[ProductKey])
            ) > 0
            &&
            CALCULATE (
                CountRows (FactInternetSales),
                USERELATIONSHIP (FactInternetSales[ProductKey], ProductFilter[ProductKey])
            ) > 0
        )
    ),
    FILTER (
        ALL (DimTime),
        DimTime[TimeKey] <= MAX (DimTime[TimeKey])
    )
)
```

There are several aspects of this new formula that make it appealing:

- It is much clearer than the previous ones. Having the chance to model the relationship inside the data model makes it easier both to write and to understand the formula.

- It is faster. In general, in Denali, the usage of USERELATIONSHIP results in faster queries when compared to the usage of FILTER to mimic the relationship, due to internal optimizations introduced by the handling of relationships.

- There is only one iteration over DimCustomer, against the two we needed to develop in the previous formulas

Finally, the formula for NotHavingProducts is straightforward, much easier than before:

```
NotHavingProducts :=

CALCULATE (
    COUNTROWS (
        FILTER (
            DimCustomer,
            CALCULATE (
                CountRows (FactInternetSales),
                USERELATIONSHIP (FactInternetSales[ProductKey], DimProduct[ProductKey])
            ) = 0
            &&
            CALCULATE (
                CountRows (FactInternetSales),
                USERELATIONSHIP (FactInternetSales[ProductKey], ProductFilter[ProductKey])
            ) > 0
        )
    ),
    FILTER (
        ALL (DimTime),
        DimTime[TimeKey] <= MAX (DimTime[TimeKey])
    )
)
```

The only difference from the two formulas now being the test for “equals to zero” instead of “greater than zero” in the check against DimProduct.

The many-to-many pattern is still here, in the two innermost CALCULATE but the introduction of inactive relationships leads to formulas that are easier to write and, as a direct consequence, easier to use and adopt.

You might have noticed that, this time, we did not use the SUMMARIZE pattern to compute the many-to-many filter. There is a reason for that. The HavingProduct can be written using the SUMMARIZE technique in this way:

```
HavingProducts :=

CALCULATE (
    CALCULATE (
        COUNTROWS (DimCustomer),
        CALCULATETABLE (
            SUMMARIZE (FactInternetSales, DimCustomer[CustomerKey]),
            USERELATIONSHIP (FactInternetSales[ProductKey], DimProduct[ProductKey])
        ),
        CALCULATETABLE (
            SUMMARIZE (FactInternetSales, DimCustomer[CustomerKey]),
            USERELATIONSHIP (FactInternetSales[ProductKey], ProductFilter[ProductKey])
        )
    ),
    FILTER (
        ALL (DimTime),
        DimTime[TimeKey] <= MAX (DimTime[TimeKey])
    )
)
```

Again, the Denali syntax is very clear and makes the formula shine.

The problem is with NotHavingProduct. In fact, in NotHavingProduct we are interested in customers who did not buy the product. SUMMARIZE will easily find positive relationships but is not well suited to search for non-existing ones.

In this case, however, it is enough to note that that either a customer has bought a product or he has not. Thus, if we count all customers who have bought a product in ProductFilter and subtract from this value the ones who have bought a product in in DimProduct, we will be able to compute the NotHavingProducts value with a simple subtraction.

Considerations About Multidimensional Models

Many-to-many, although not handled natively in Vertipaq, can be easily managed with the DAX language, as we have demonstrated in the previous chapters and they benefit from the tremendous speed of the Vertipaq engine.

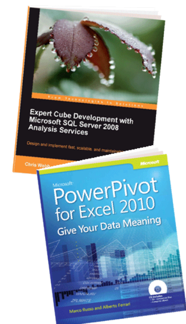
The only thing that is really needed to work with many-to-many in DAX is a good understanding of how the CALCULATE function works which basically means having a good knowledge of the DAX language. Tabular data models are simple, although not very easy to understand at first glance.

One interesting point of many-to-many in Vertipaq is the fact that many different formulas can be computed without changing the data model and without the need to create ETL steps to feed a complex model, created for a specific solution. The Basket Analysis data model is a clear demonstration of the power of the DAX language and the Vertipaq engine: without any modeling step we can compute very complex and interesting measures.

Moreover, it is worth to note that – in general – Tabular models are easier to write and optimize when compared against Multidimensional ones. A good measure of this statement is the length of the Tabular sections compared with the Multidimensional one. We needed far few pages and lines of code to explain the same model in Tabular than in Multidimensional while, hopefully, keeping the same clarity in the explanation.

Links

- http://www.sqlbi.com/articles/many2many/ : the project home page for this paper and related resources
- http://www.sqlbi.com : community dedicated to Business Intelligence with SQL Server
- http://sqlblog.com/blogs/marco_russo : blog of Marco Russo (author)
- http://sqlblog.com/blogs/alberto_ferrari : blog of Alberto Ferrari(author)
- http://www.sqlbi.com/articles/sqlbi-methodology/ : the SQLBI Methodology paper for best practices in BI solution design
- http://www.amazon.com/gp/product/1847197221/?tag=se04-20 : the book "Expert Cube Development with Microsoft SQL Server 2008 Analysis Services" written by Marco Russo, Alberto Ferrari and Chris Webb
- http://www.amazon.com/dp/0735640580/?tag=se04-20 : the book "PowerPivot for Excel 2010: Give your data meaning" written by Marco Russo and Alberto Ferrari.



- http://www.powerpivotworkshop.com : Two-day workshop on PowerPivot and the DAX language, the perfect place where to touch the Karma of DAX with the author of this paper.
- http://tinyurl.com/optimizeM2M : Analysis Services Many-to-Many Dimensions: Query Performance Optimization Techniques