# Bidirectional cross-filtering in SQL Server Analysis Services 2016 and Power BI Desktop

**Microsoft BI and Analytics Technical Article**

**Writer:** Kasper de Jonge, Senior Program Manager, Microsoft Corp.

**Contributor:** Owen Duncan, Senior Content Developer, Microsoft Corp.

**Applies to:** SQL Server 2016 Analysis Services, Power BI Desktop

**Summary:** This paper introduces bidirectional cross-filtering, a new feature in Microsoft SQL Server 2016 Analysis Services and Microsoft Power BI Desktop.

## Copyright

# Contents`

Copyright ..... 1
Introduction ..... 2
How do relationships work in Analysis Services? ..... 3
The traditional BI many-to-many pattern ..... 4
Measures in dimension tables ..... 8
Ambiguous relationships tables, what do end users want to see? ..... 11
Multiple fact tables (a Fact constellation), too much filtering? ..... 14
The date table and bidirectional relationships ..... 18
Use DAX to enable cross-filtering per measure ..... 21
Row level security and bidirectional relationships ..... 24
Summary ..... 29

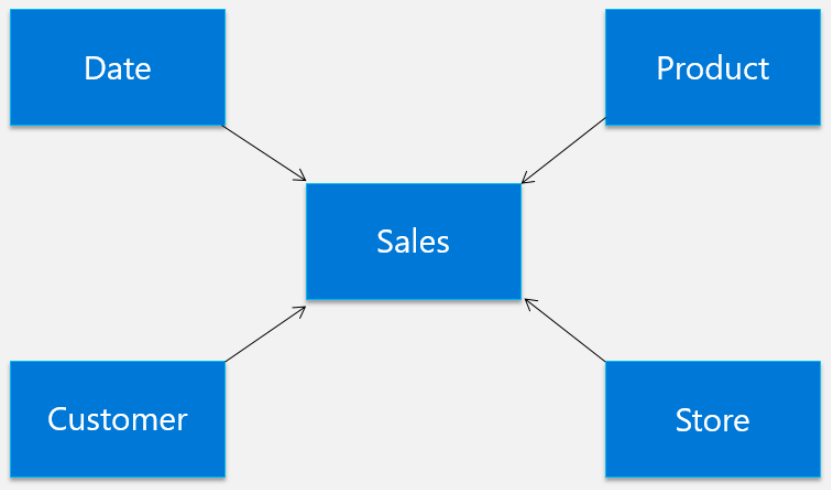
# Introduction

Bidirectional cross-filtering is a new feature for SQL Server 2016 Analysis Services and Power BI Desktop that allows modelers to determine how they want filters to flow for data using relationships between tables. In SQL Server 2014, filter context of a table is based on the values in a related table. With bidirectional cross-filtering the filter context is propagated to a second related table on the other side of a table relationship. This can help you solve the many-to-many problem without writing complicated DAX formula's.

> Note: Even though this whitepaper uses SSDT and SQL Server 2016 as examples, information provided here also applies to Power BI Desktop.

## How do relationships work in Analysis Services?

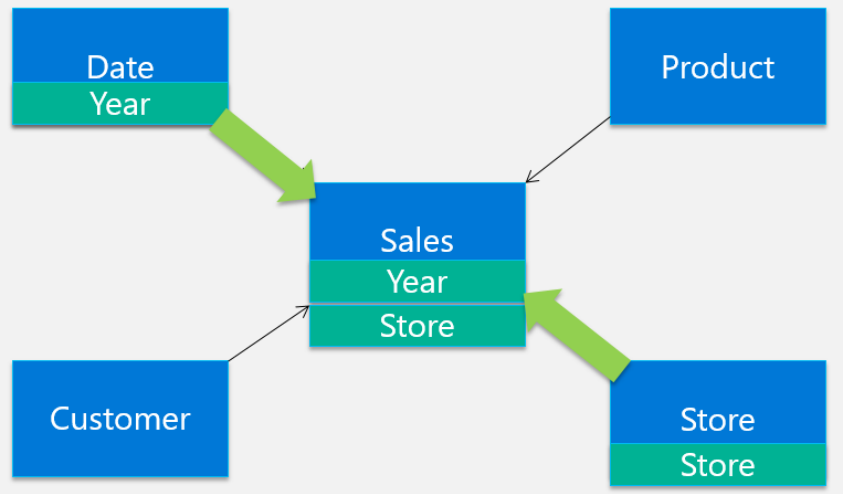
Before we go deep into the details about bidirectional cross-filtering, let's first take a closer look at how relationships work in Analysis Services. Traditionally, business intelligence projects tend to use a star or snowflake schema; a design approach that has become the de facto design standard for data warehouses and cubes over the last few decades. Below is an example of a star schema:




The center of the star is called the fact table. It describes the measurements, facts, or metrics of a business process. In this case, the fact table, Sales, contains sales records; one row for each sales transaction. The center of the star is surrounded by dimensions. Each dimension is a descriptive table that describes attributes of a fact. Here we have, Dates, Customers, Product and Stores, these dimension tables provide more details about the facts.

This arrangement is based on keys inside the tables; for example, the sales table contains ProductKey, and the Product table also contains ProductKey. The ProductKey from the sales table is called a foreign key, and the ProductKey from the Product table a primary key. One single unique product has many different sales for the same product. This is known as a one-to-many relationship.

In a report, the data from the fact table is usually aggregated and sliced by fields from one or more of the dimensions. For example, we might want to look at the sum of Amount by Year and Store. If we visualize how data flows in the schema, we'll see that the Sales table is filtered to show only values for the year and store selected.

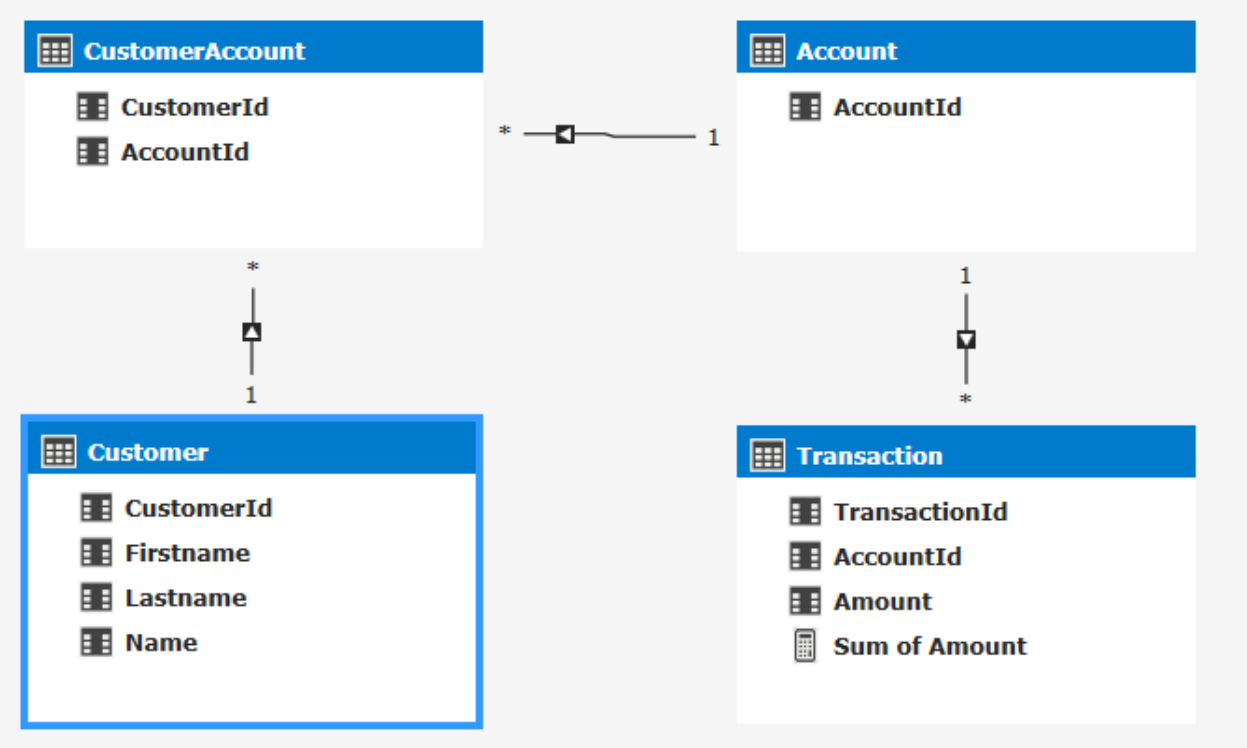



This is the default behavior of filtering for Analysis Services and is the only available filter in SQL Server 2014 and earlier. But business problems are usually never this clear-cut and there are some well-known patterns that are not easily solvable with filters flowing into just one direction. In the following sections, we'll look at some of those scenarios and introduce bidirectional cross-filtering.

## The traditional BI many-to-many pattern

One of the most common patterns that could leave the modeler scratching his head and resorting to complicated DAX expressions is the traditional many-to-many pattern. To illustrate this pattern, let's look at an example from an Analysis Services 2005 help article describing many-to-many for Multidimensional models. This article illustrates the issue very well.

The traditional many-to-many pattern common in BI describes a data structure that doesn't conform to the snowflake or star schema model, where one fact is associated with a single dimension member. For example, in a typical cube analyzing sales data, a single sales transaction is associated with a single customer, a single product, and a single point in time. But, sometimes data structures can be more complex; for example, consider financial transactions in accounts that can have one or more customers. This can be modeled as:




The relationship between transaction and customer is a many-to-many relationship. A single transaction can be associated with many customers and each customer can be associated with many transactions. Say we have three customers, John and Jane Hanover, who share a joint account, and Henry Waxman, who has an individual account. If John and Jane each contribute $100 to their account and Henry contributes $150, the results for all customers looks like this:

| Row Labels | Sum of Amount |
|---|---|
| Henry Waxman | 150 |
| Jane Hanover | 100 |
| John Hanover | 100 |
| **Grand Total** | **250** |

The sum for all amounts isn't the sum of individual amounts for each customer—that would double-count the data that John and Jane share. Instead, the total amount is the sum of all transactions.

When computing a balance for a customer, the amount is the aggregate of all transactions to the accounts that customer is associated with, and not the simple sum of the transactions for each customer.

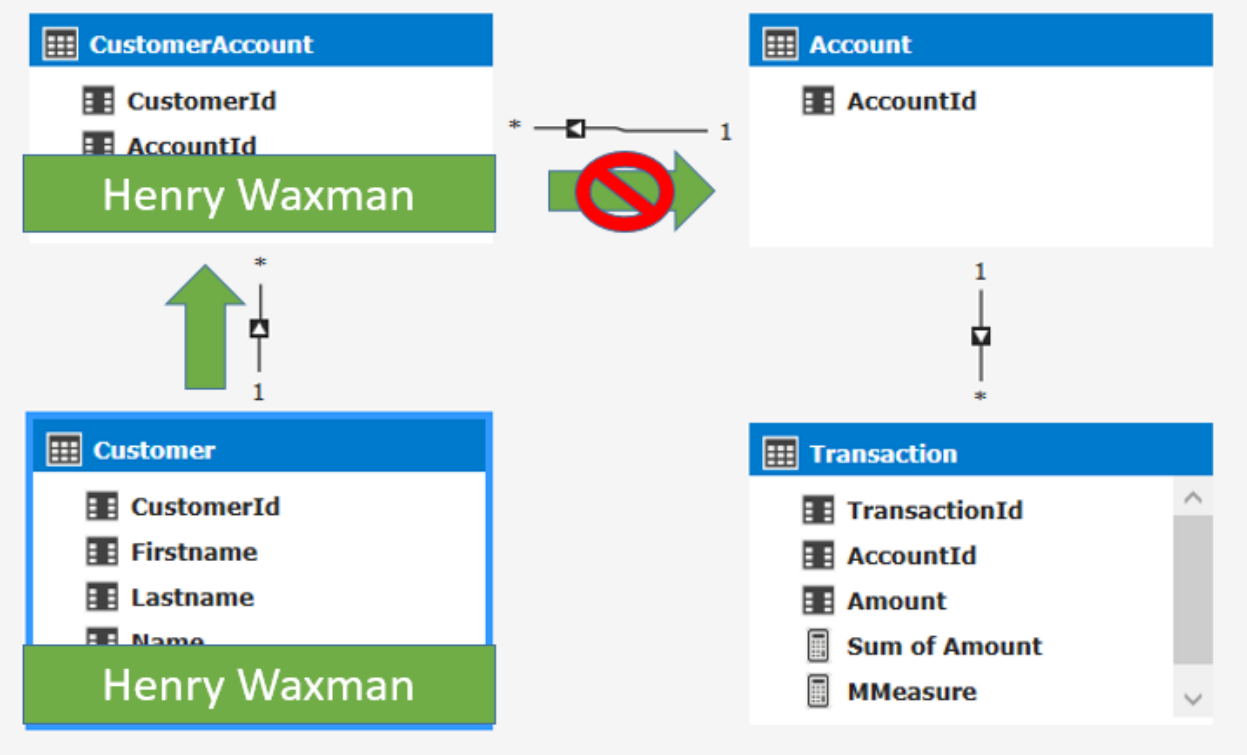
Now, what we just described is the expected behavior, but when trying to recreate this result in SQL Server 2014 tabular models the results in Excel or any other client will be wrong when using just relationships and a simple SUM measure:

| Row Labels | Sum of Amount |
|---|---|
| Henry Waxman | 250 |
| Jane Hanover | 250 |
| John Hanover | 250 |
| **Grand Total** | **250** |

The tabular model before SQL Server 2016 wouldn't have been able to filter the tables in a way that is needed to show the results as expected. Instead it will just repeat the total for every row. So what, exactly, is going on?

Let's start with the basics. We've put Customer on rows and Sum of Amount from the Transaction table on values in an Excel PivotTable. To get the Amount per Customer, we need to filter the rows in the Transaction table for every customer; however, this is where the trouble starts as the tabular model is not able to filter the transaction table by customers.

If you look at the relationships in the model, you'll see arrows on the relationships that indicate how tables will get filtered. Let's highlight how the customer filters are applied to the tables in the model:




You will notice the arrow on the relationship between CustomerAccount and Account is pointing into the wrong direction. The Analysis Services tabular model by default only filters from the 1 to the Many side of the relationship, that is not what we need here as we need to be able to get the transactions for just the accounts by customer. We need the Account table to be filtered.

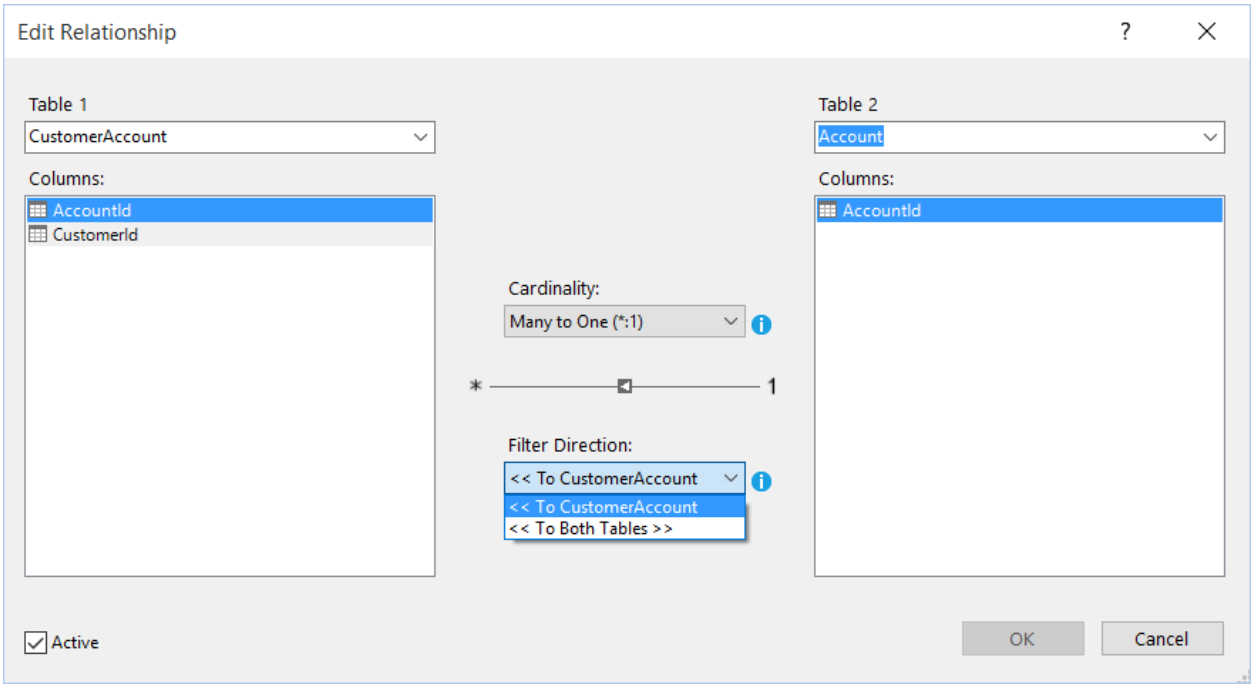
Now we can solve this in two ways:

**1. Using DAX:** In SQL Server 2014 we can write a calculation that pushes the relationship into the Account table by using DAX, the following DAX calculation would give us the expected results:

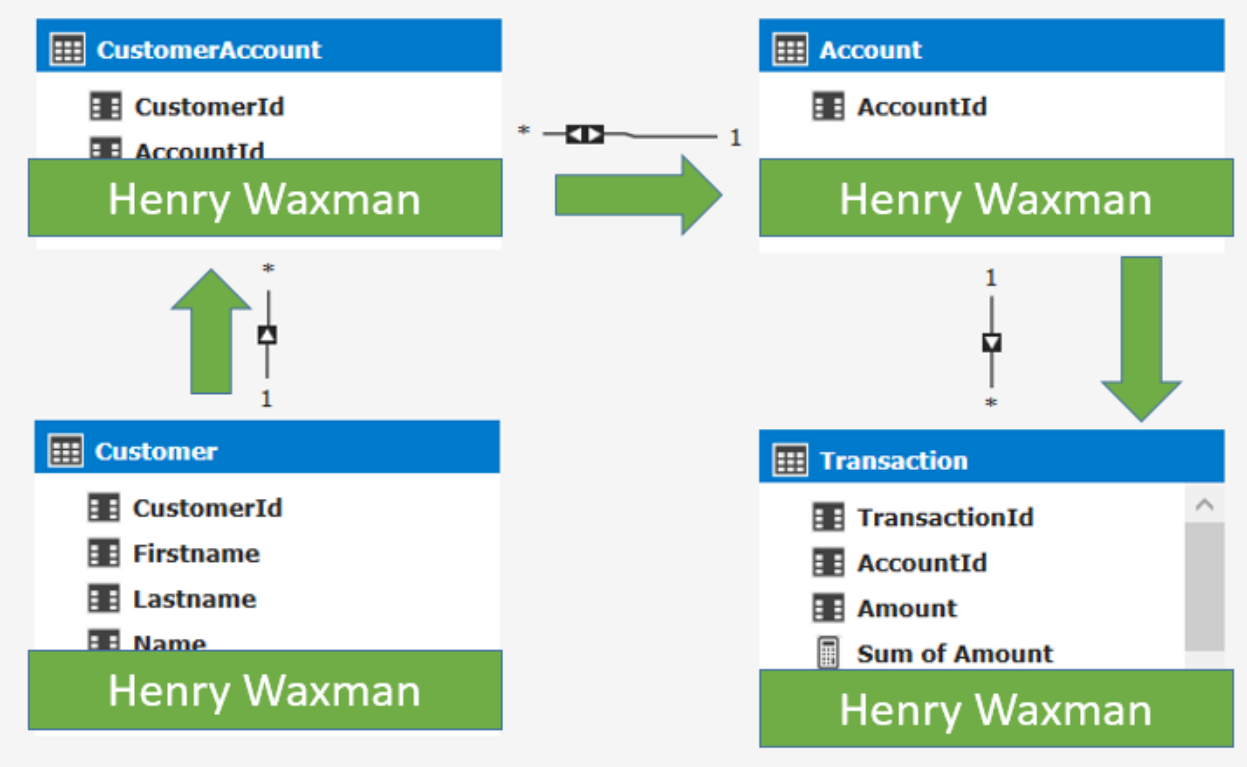
```
=CALCULATE([Sum of Amount]
                    , SUMMARIZE(CustomerAccount
                                ,Account[AccountId])
          )
```

This calculation works in the following way. By using CALCULATE, the Sum of Amount is calculated for rows in the Transaction table where AccountId's exist in the Account table. The AccountId values of the Account table are filtered by the use of the CustomerAccount table in the SUMMARIZE function. This implicitly creates a relationship that wasn't working before. For more details on how this calculation works, see the many-to-many whitepaper written by Marco Russo and Alberto Ferrari. This approach has some downsides; it's complicated and could lead to some performance issues, but mostly it only solves this issue only for this one particular measure. The business user using the report still cannot use any other field in the transaction table to aggregate on.

**2. Use Bidirectional relationships:** In SQL Server 2016 we now allow you to change how filters are flowing when you use the 1200 compatibility level. For each relationship in SSDT you can choose the type of filtering you want to use for between these two tables:




When selecting a single table in the filter direction, you will keep the behavior as it was in SQL Server 2012 and 2014, which only filters data from the 1 to the many side. When you set the filter direction to To Both Tables it means filters are now applied to both sides of the relationship. Changing the Filter Directing from To CustomerAmount to To Both Tables will result in the model pushing the filters to the Account table from the Customer table through the CustomerAccount. When we look at the model with the changed filter direction we'll be able to see the filters flowing as expected:



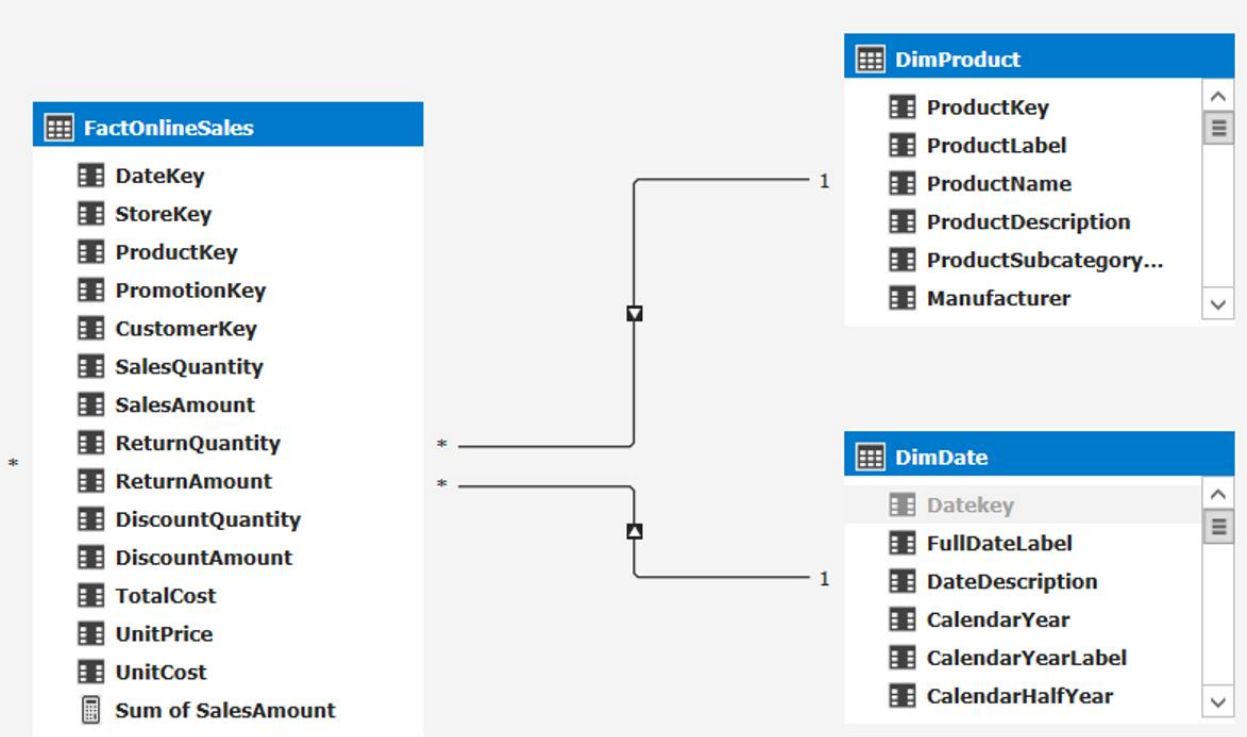

This now gives us the correct results:

| Row Labels | Sum of Amount |
|---|---|
| Henry Waxman | 150 |
| Jane Hanover | 100 |
| John Hanover | 100 |
| **Grand Total** | **250** |

As we've seen using the new filter direction on relationships, we can now change the way data is flowing. This gives us more flexibility to solve more complex modelling issues.

## Measures in dimension tables

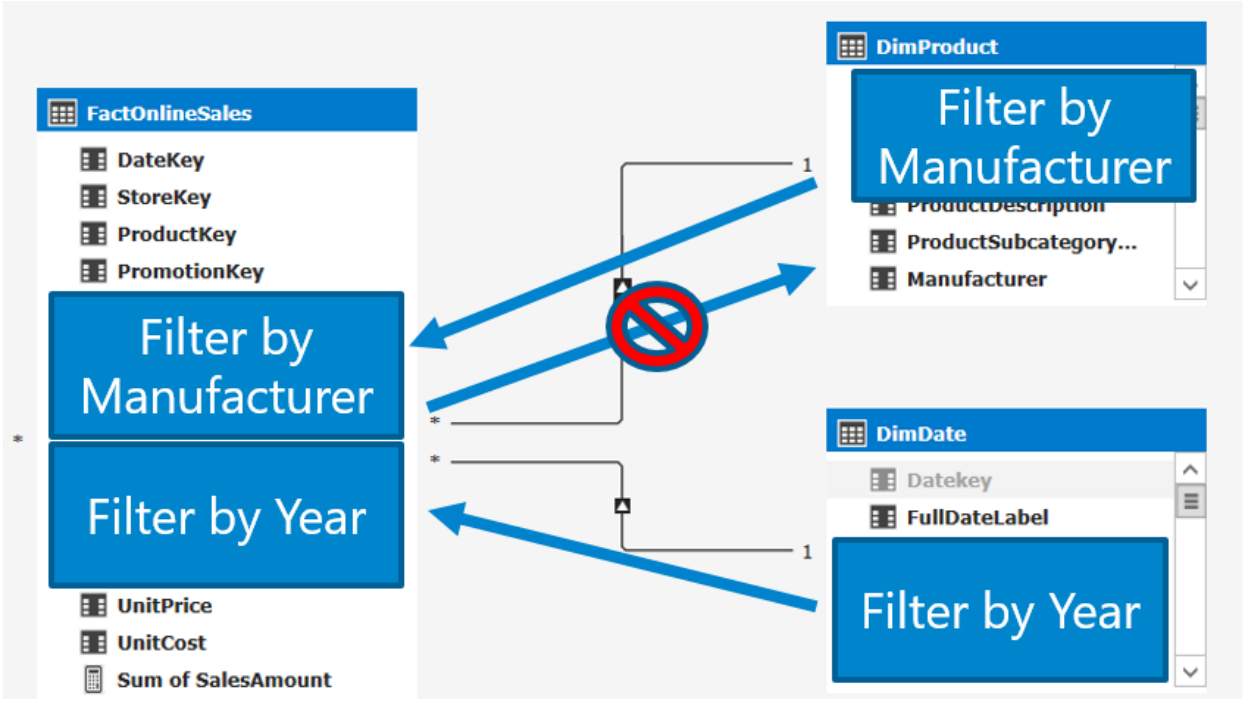
Another scenario where bidirectional cross-filtering will be useful is when we want to count values in a dimension table. Let's take the following schema:




Let's say we want to see the Sum of SalesAmount and Distinct Count of Products by Year and Manufacturer. Like in the previous scenario, by default, we will get the same number of Unique products repeated over and over, indicating there is a problem with the relationships:

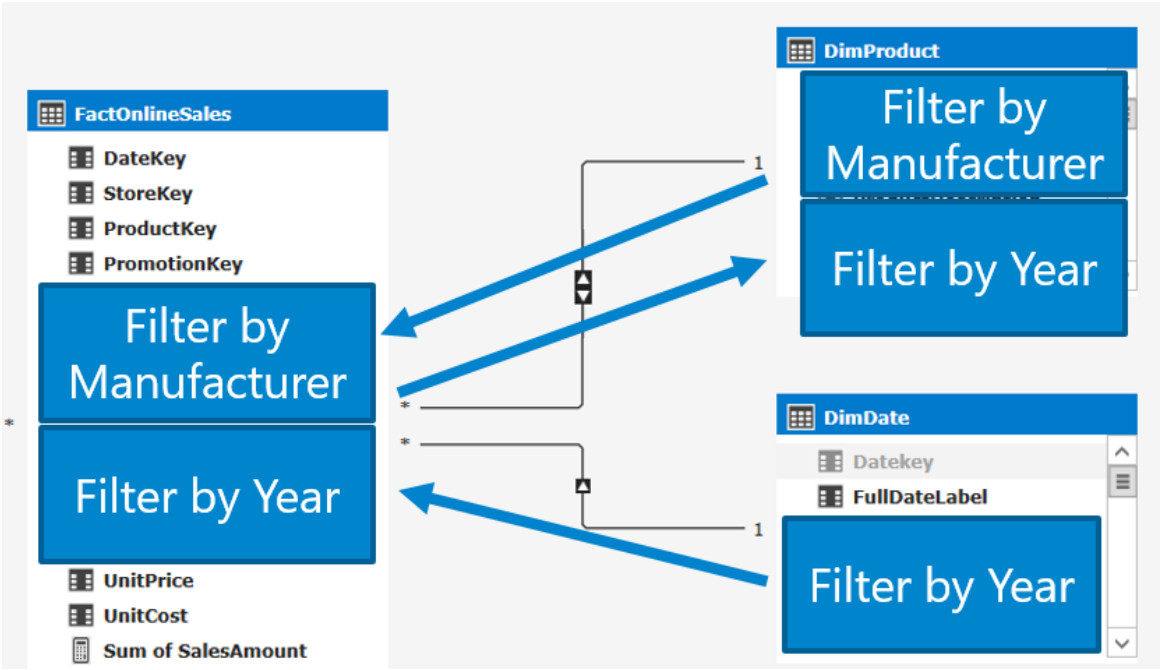
| Row Labels | Sum of SalesAmount | Unique Products |
|---|---|---|
| **A. Datum Corporation** | **$179,451,946.31** | **132** |
| 2005 | | 132 |
| 2006 | | 132 |
| 2007 | $104,361,977.37 | 132 |
| 2008 | $37,053,899.65 | 132 |
| 2009 | $38,036,069.29 | 132 |
| 2010 | | 132 |
| 2011 | | 132 |
| **Adventure Works** | **$370,291,857.54** | **192** |
| 2005 | | 192 |
| 2006 | | 192 |

The problem here is that again the product table isn't filtered by the CalendarYear. Let's visualize the filters in the schema again:




As you can see here the DimProduct table is not filtered by Year. This will make the values repeat over and over again for each year.

Now when we switch the filter direction on the relationship between FactOnlineSale and DimProduct to Both, any filter applied to the FactOnlineSales table is also propagated to the DimProduct table as such:

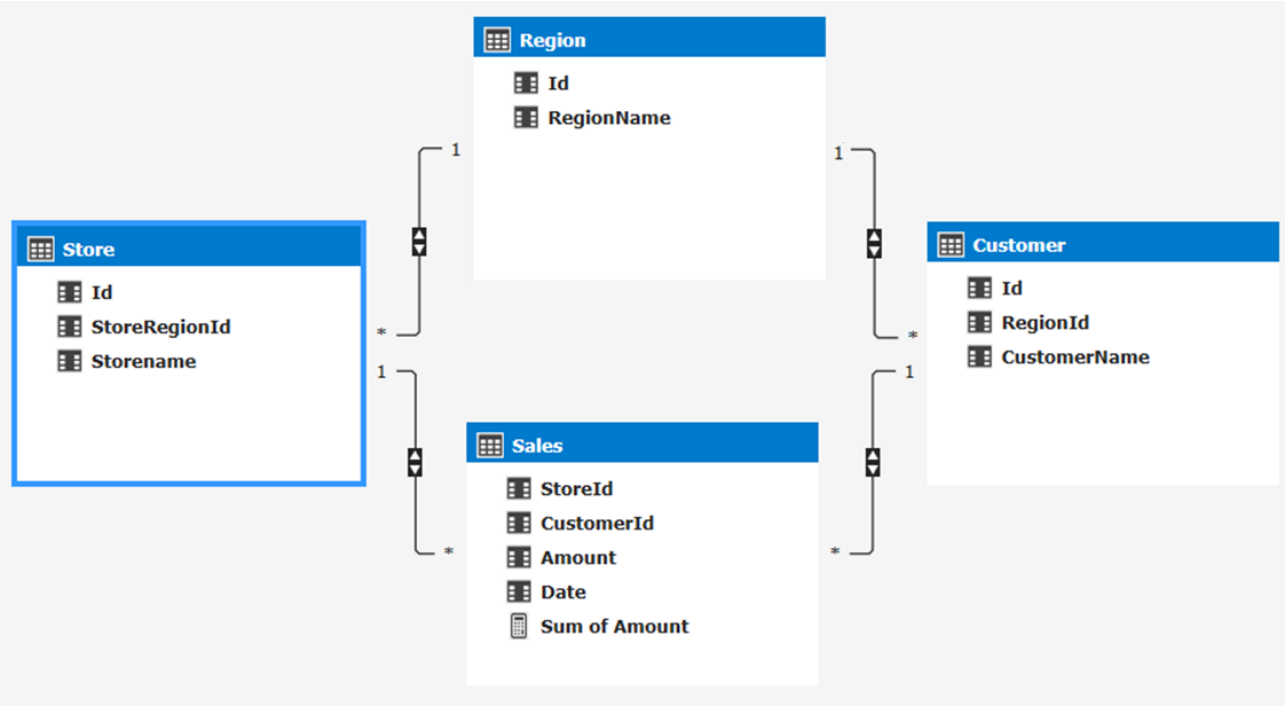



Now we get the result we expect:

| Row Labels | Sum of SalesAmount | Unique Products |
|---|---|---|
| **A. Datum Corporation** | **$179,451,946.31** | **132** |
| 2007 | $104,361,977.37 | 111 |
| 2008 | $37,053,899.65 | 123 |
| 2009 | $38,036,069.29 | 131 |
| **Adventure Works** | **$370,291,857.54** | **192** |
| 2007 | $182,678,563.48 | 131 |
| 2008 | $86,478,475.19 | 175 |
| 2009 | $101,134,818.87 | 189 |
| **Contoso, Ltd** | **$622,430,296.23** | **710** |
| 2007 | $234,490,811.73 | 506 |
| 2008 | $197,559,389.88 | 629 |

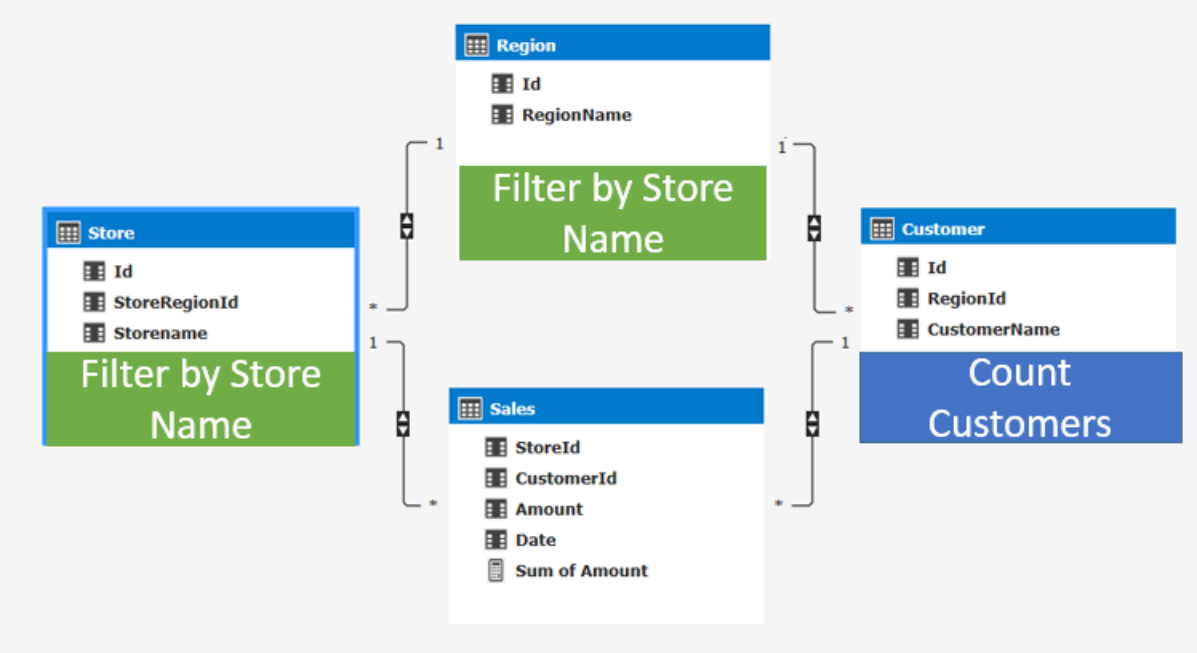
## Ambiguous relationships tables, what do end users want to see?

Now these scenarios are pretty simple with just a few tables. What happens if we create a bit more complicated models that do not follow a script star or snowflake schema? Let's take a look:

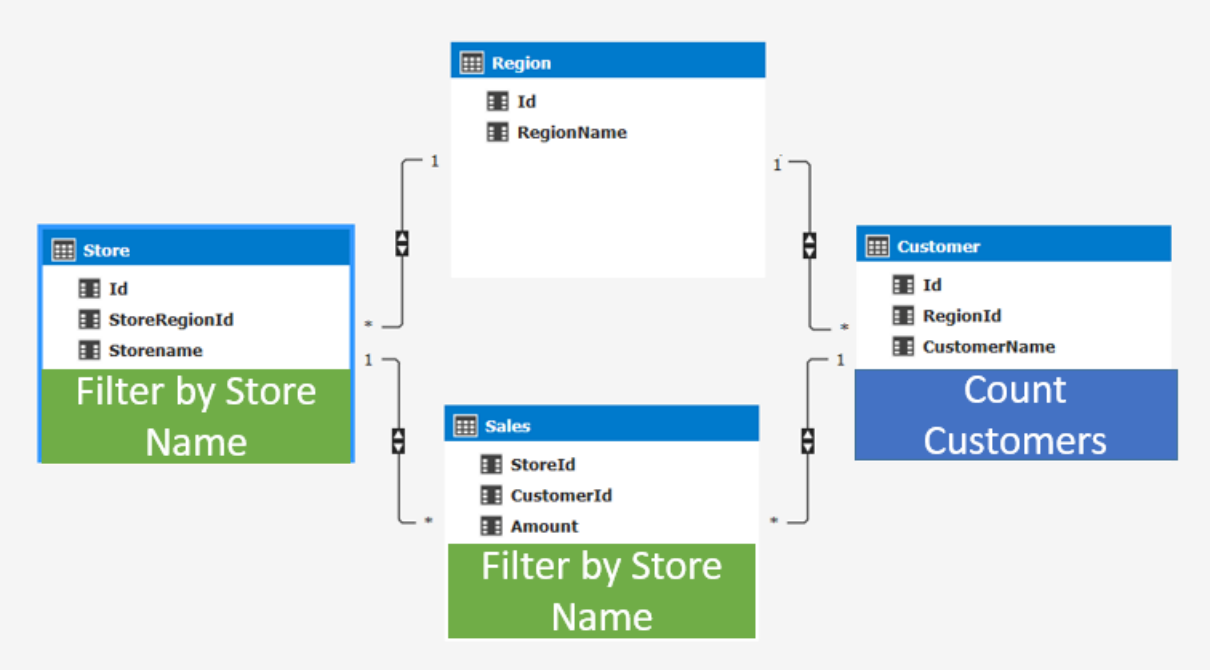



We have stores, where each store has sales and is located in a region. We also have customers who live in a region and also have sales in a store. Now let's try to figure out what happens if we want to see number of customers by store with the schema above.

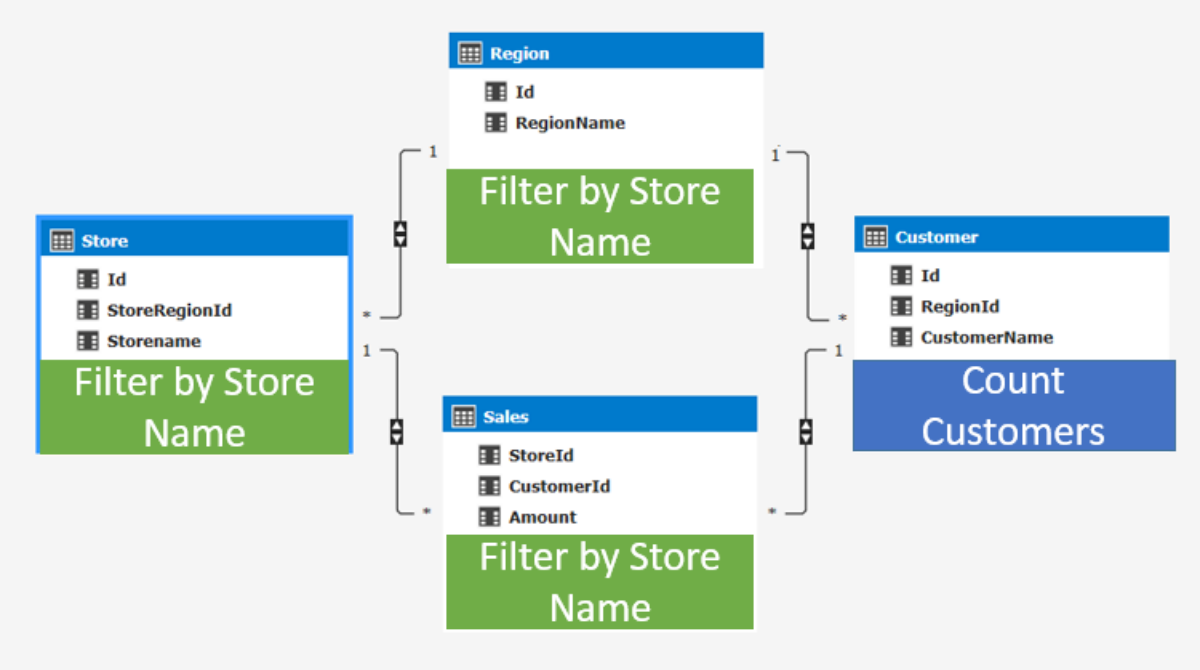
This is where the problem arises, what is it that we want to see? The relationships between the tables in the schema above allows for several options and is ambiguous. Do we want to count the customers who live in same region as the store? This would mean we would filter the customer table to include only the customers which region is the same as the region of the store selected:




Or do we want to count the customers who bought something (had sales) at the store? This would mean we would filter the customer table to include only the customers which had sales in the selected stores:

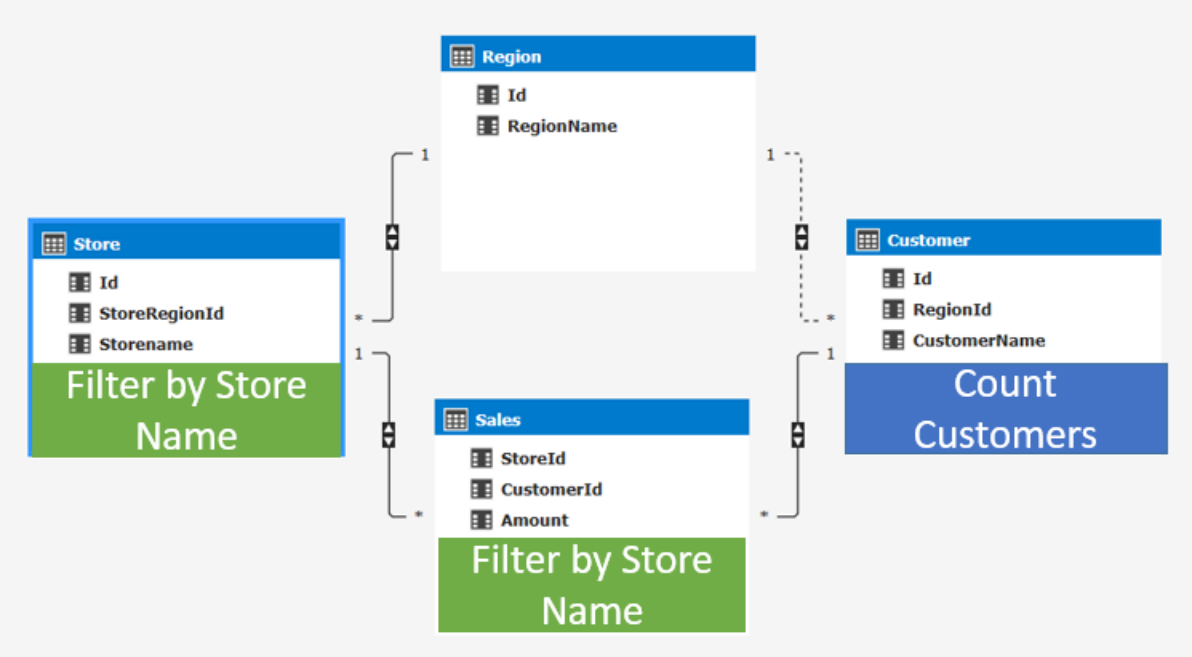



Or do we want customers who live in the same region as the store and bought something in the store This would mean we would filter the customer table to include the customers with sales in the selected stores **AND** to include the customer's which region is the same as the region of the store selected:




The Analysis Services engine cannot know the user's intent and choosing a path at random is also not desired. To make sure the end user doesn't get these random results, Analysis Services will not allow the modeler to create an ambiguous schema. The example above is not allowed in SQL Server 2016 or in Power BI Desktop. The modeler has to make sure there are no ambiguous paths; for example, the scenario above can be solved by disabling a relationship.

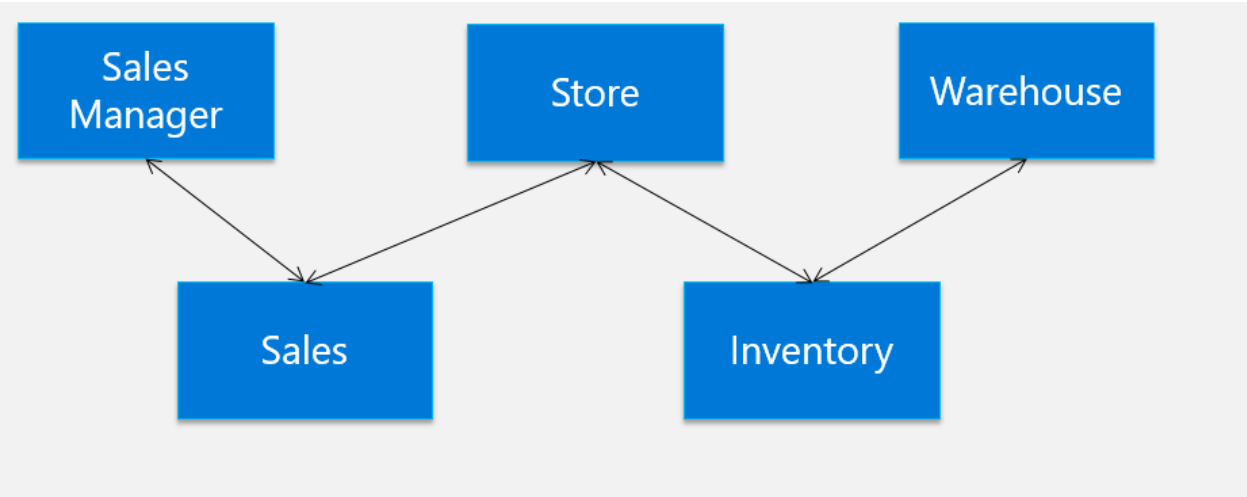
By disabling a relationship, we make the explicit choice on how the filters flow and how end users will see count of customers by stores. For example, by setting the relationship between Region and Customer to inactive we make the choice that end users will always see the count of customers by who bought something at the selected store.



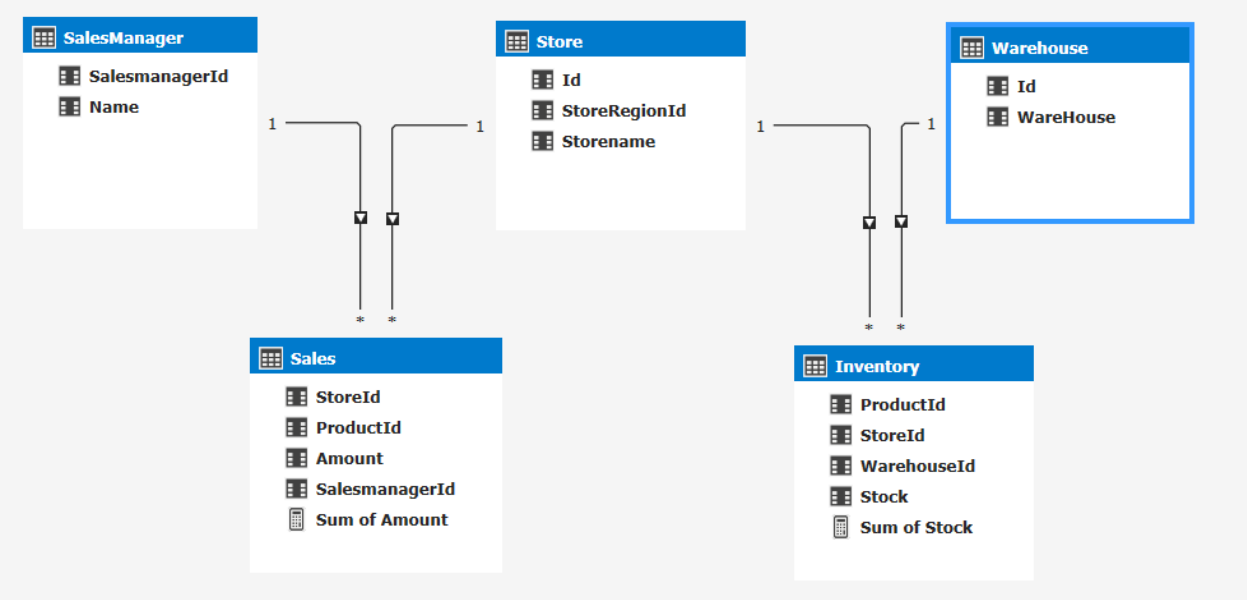

There are many other ways you can achieve non-ambiguous paths, like playing with different variations of filter directions between tables and using DAX to get the results you want. More on this later.

## Multiple fact tables (a Fact constellation), too much filtering?

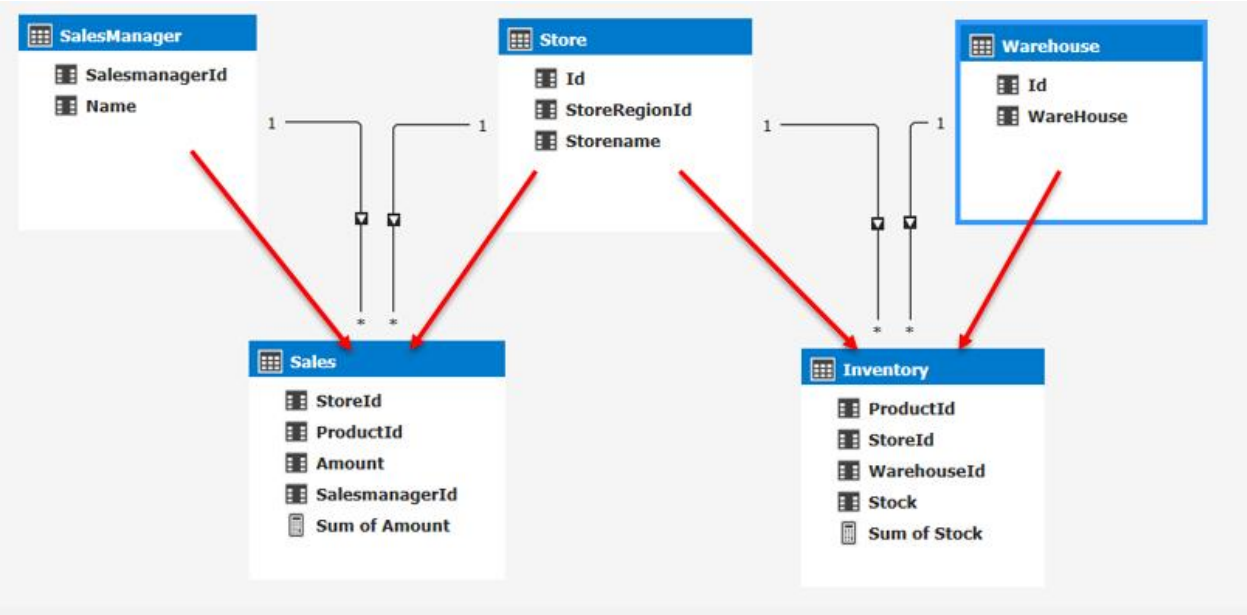
Now let's take it a step further and take a look at a more complicated schema. Imagine we depart completely from a traditional star schema and look at a scenario where we have multiple fact tables (a Fact constellation). The schema below shows us an example of this:




Let's load these tables into SSDT, by default all relationships will be set to single directional:




The tabular model with this schema will allow the fact tables to be filtered only by its related dimensions, in the following manner:

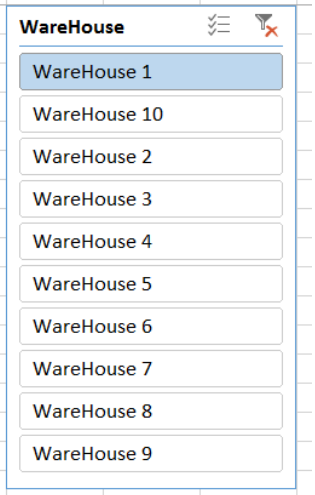



This means the Sales table will only be filtered by columns from the SalesManager and Store tables. The Inventory table will only be filtered by columns from Store and Warehouse tables.

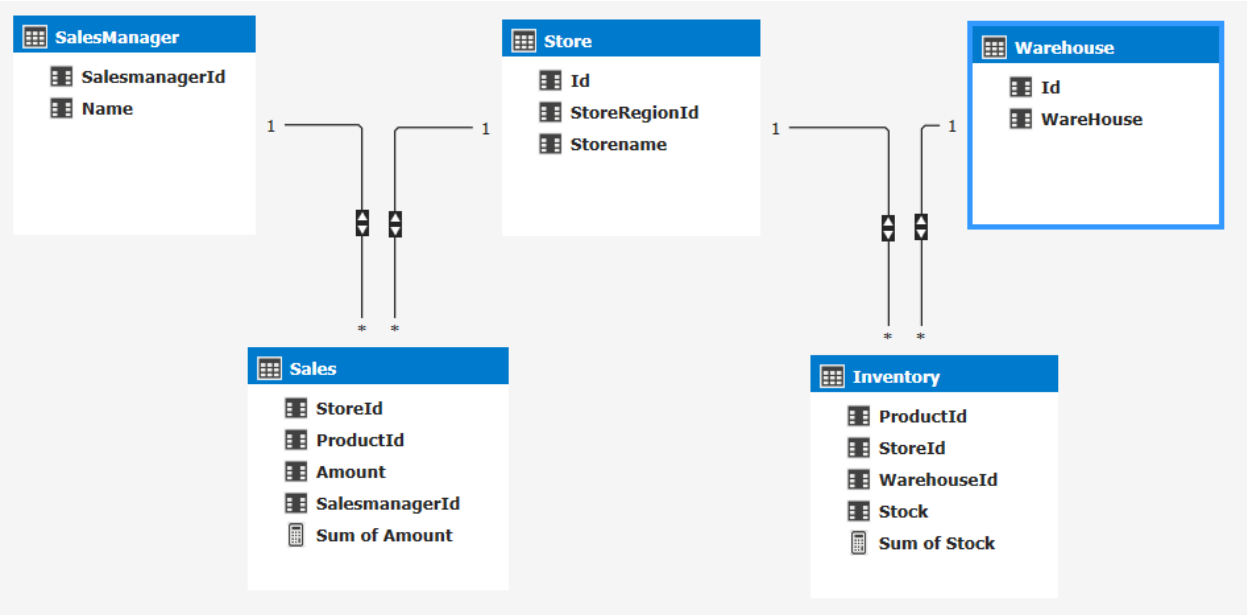
When we look at the data used in Excel, we see what we expect based on the model above: a PivotTable with Sum of Amount by SalesManager and a PivotTable with Sum of Stock available for each store filtered by Warehouse using a slicer. Like this:

| Row Labels | Sum of Amount |
|---|---|
| Sales 1 | 3,428 |
| Sales 2 | 7,104 |
| Sales 3 | 7,024 |
| Sales 4 | 50,664 |
| Sales 5 | 13,352 |
| **Grand Total** | **81,572** |

| Row Labels | Sum of Stock |
|---|---|
| Store 1 | 4,523 |
| Store 3 | 455 |
| **Grand Total** | **4,978** |




The Amount for each SalesManager is not affected by the selection of a Warehouse. Now let's turn on Bidirectional cross-filtering for all relationships in the model:




If we look at the same PivotTable again. we immediately notice several differences. The Total Amount is now much less. We also see that Sales manager Sales 3 has completely disappeared:

| Row Labels | Sum of Amount |
|---|---|
| Sales 1 | 864 |
| Sales 2 | 3,552 |
| Sales 4 | 25,974 |
| Sales 5 | 246 |
| Grand Total | 30,636 |

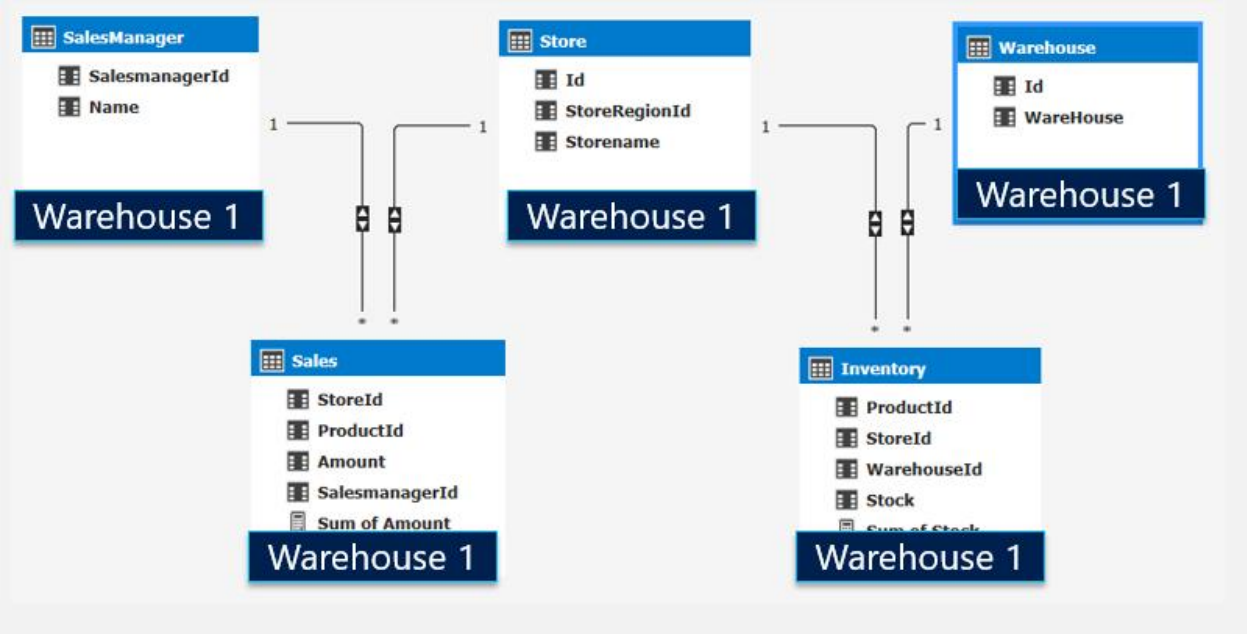
| Row Labels | Sum of Stock |
|---|---|
| Store 1 | 4,523 |
| Store 3 | 455 |
| Grand Total | 4,978 |

| Row Labels | Sum of Amount |
|---|---|
| Sales 1 | 3,428 |
| Sales 2 | 7,104 |
| Sales 3 | 7,024 |
| Sales 4 | 50,664 |
| Sales 5 | 13,352 |
| Grand Total | 81,572 |

WareHouse

- WareHouse 1
- WareHouse 10
- WareHouse 2
- WareHouse 3
- WareHouse 4
- WareHouse 5
- WareHouse 6
- WareHouse 7
- WareHouse 8
- WareHouse 9

What we see happening here is the result of bidirectional cross-filtering. If we project the filter on warehouse to the tables themselves, we'll see that every table is now filtered down to only return data that is related to Warehouse 1.



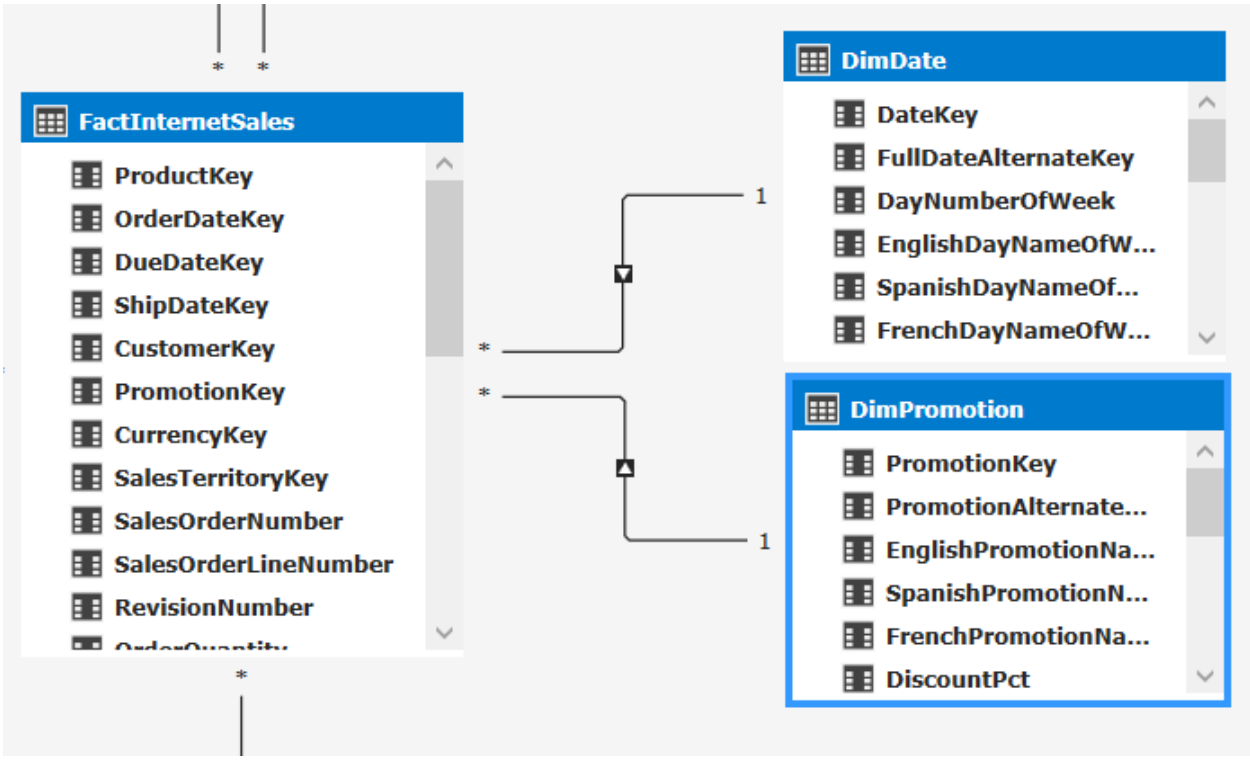

The Inventory table filters out rows that has inventory for Warehouse 1. Consequently, we only see rows for the Store table where we have rows from the already filtered down Inventory table. This now has consequences for the Sales table that will now only show results from those stores where we have inventory from Warehouse1. This will give us the results we see in the PivotTable earlier.

As you can see, turning on bidirectional cross-filtering on all tables in a model can have big effects for the end user, especially when using multiple fact tables. It's important to consider the effects of turning on Bidirectional cross-filtering for a relationship. Later we will see how the bidirectional behavior can be controlled by using DAX.

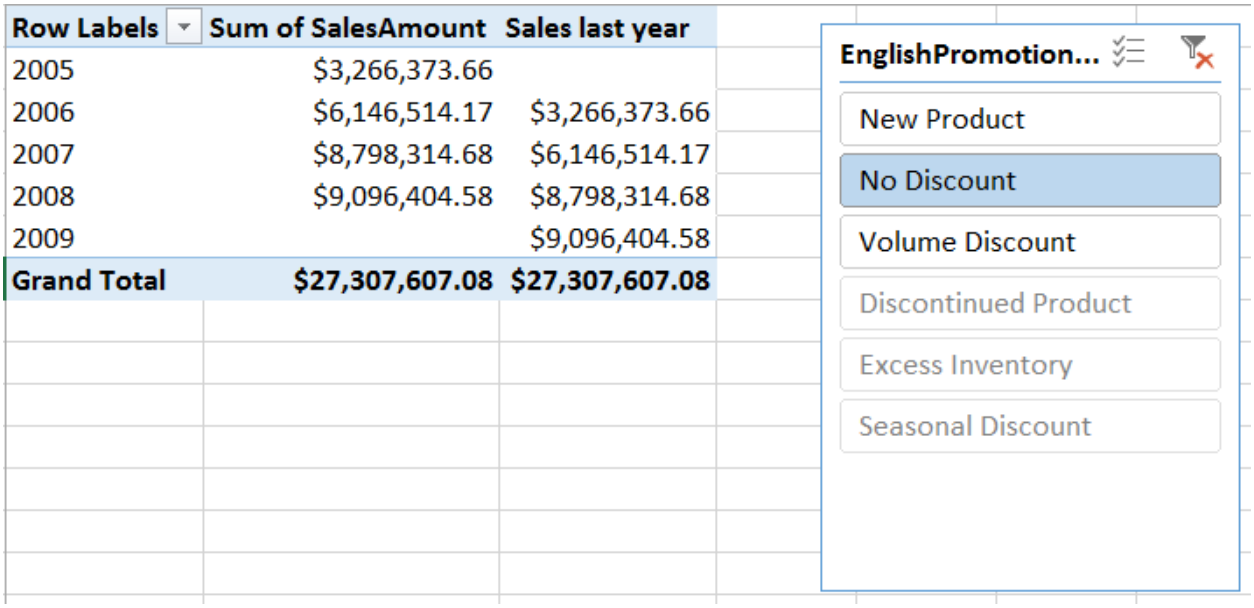
## The date table and bidirectional relationships

One special case to call out is how the date table works together with bidirectional cross-filtering. The general recommendation is to always use a one-way filter for the date table and not use bidirectional relationships at all for date tables.

To illustrate the issue, let's look at a simple case where we have sales, dates, and the promotion with a single directional relationship between the tables:




In a PivotTable we want to see the sales and the sales from the previous year, by year, sliced by promotion:



| Row Labels | Sum of SalesAmount | Sales last year |
|---|---|---|
| 2005 | $3,266,373.66 | |
| 2006 | $6,146,514.17 | $3,266,373.66 |
| 2007 | $8,798,314.68 | $6,146,514.17 |
| 2008 | $9,096,404.58 | $8,798,314.68 |
| 2009 | | $9,096,404.58 |
| **Grand Total** | **$27,307,607.08** | **$27,307,607.08** |



So far so good, all works as expected. The previous year is calculated by using DATEADD time-intelligence function in the Sales last year formula:

```
Sales last year:= CALCULATE([Sum of SalesAmount]
                        , DATEADD(DimDate[FullDateAlternateKey]
                                ,-1
                                ,YEAR
                                )
                        )
```

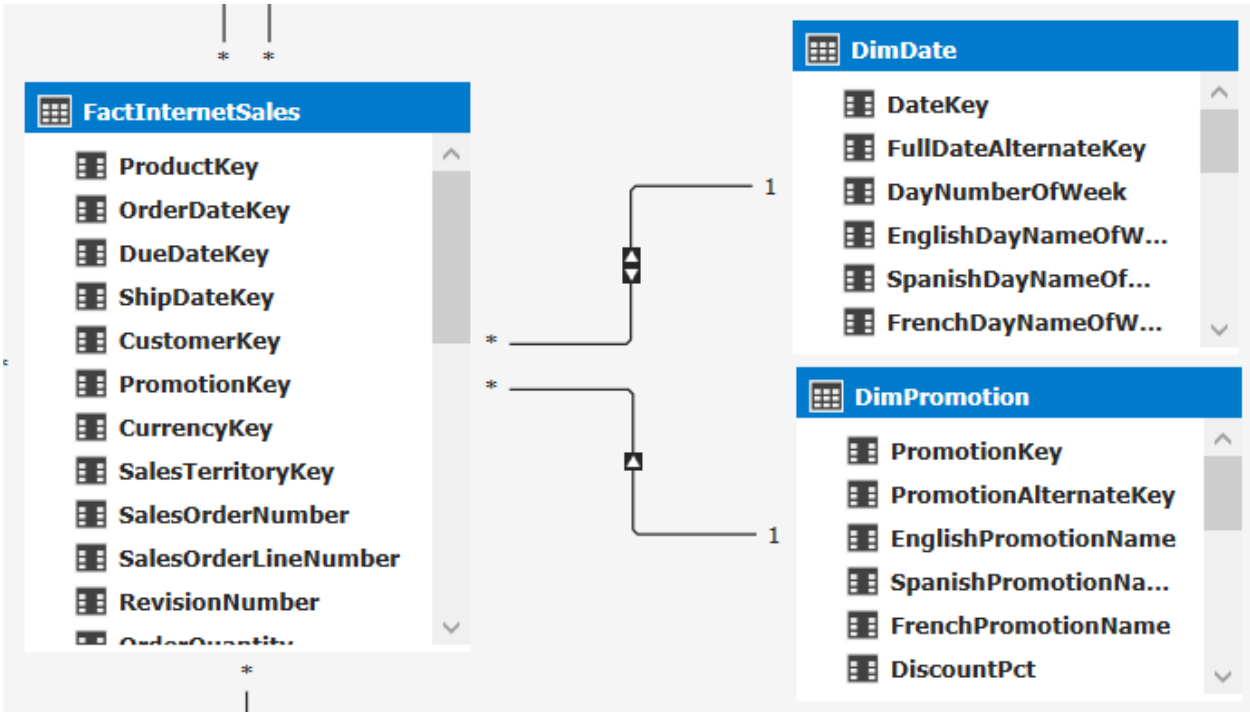
The DATEADD function looks at the dates in the current selection and uses the date column from the DimDate table to get the values from one year ago. You might think, how does this work? In short, the pivot table filters the date table to show only data for the current year. For time-intelligence functions, the DAX engine uses some special functionality to overwrite the filters placed on the date table. [For more details on time-intelligence functions see this blog post](#).

Now, I can also write the same measure without using the time-intelligence function. It looks like this:

```
Sales last year 2:= CALCULATE([Sum of SalesAmount]
                        , DimDate[CalendarYear]=VALUES(DimDate[CalendarYear])-1
                        , ALL(DimDate)
                        )
```

Here we achieve the same result by overwriting the filter on the date table by specifying ALL(DimDate) as an additional parameter of calculate. But this only works for certain scenario's. Time-intelligence functions automatically do some of this work for you.

Now why are these two measures important? Let's turn on bidirectional cross-filtering between the FactInternetSales table and DimDate:



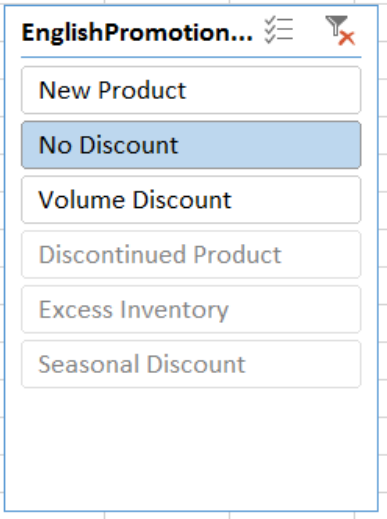

If we refresh the PivotTable, we immediately see an error:

| Row Labels | Sum of SalesAmount | Sales last year |
|---|---|---|
| 2005 | $3,266,373.66 | Error: Calculation error in measure 'FactInternetSales'[Sales last year]: Function 'DATEADD' only works with contiguous date selections. |

This error suggest that we do not have a contiguous data range in the date table. This is our first hint.

Now let's remove the Sales last year measure and add the alternative measure. Here you'll see that it works different from before:

| Row Labels | Sum of SalesAmount | Sales last year 2 |
|---|---|---|
| 2005 | $3,266,373.66 | |
| 2006 | $6,146,514.17 | $3,266,373.66 |
| 2007 | $8,798,314.68 | $6,146,514.17 |
| 2008 | $9,096,404.58 | $8,798,314.68 |



We no longer see any data for 2009, where we don't have any current sales. So what is going on? By enabling bidirectional filtering between FactInternetSales and DimDate, any filters applied to the FactInternetSales automatically flow through to the date table. This means that dates will now be filtered to just those dates where there are sales for the selected promotion. DAX is unable to reach the previous years using its time-intelligence functions. In the custom formula we overwrite the filter context by hand and the previous year is still available, but the dates that do not have any sales for the promotion selected are filtered out, hence we cannot see any previous year sales for 2009.

These are just two examples of what setting bidirectional relationships can do for relationships on a date table. It is recommended to never enable bidirectional cross-filtering between the fact and date table. Analysis Services or Power BI desktop will by default create this relationship single directional. When you do need a bidirectional cross-filter, you should use the DAX formulas described below to turn it on per measure.

## Use DAX to enable cross-filtering per measure

As we've seen in the previous scenarios, it's wise to use the bidirectional filter direction on a relationship with reserve and only in cases where you as the modeler control the scenarios. This doesn't mean you sometimes do want to have this behavior for certain use cases. Luckily we can control the cross-filtering behavior by using DAX as well. This gives us the ability to control cross-filtering behavior per measure instead of for all fields. The DAX CROSSFILTER function is key to this. This function is similar to the USERELATIONSHIP function and allows us to specify the cross-filtering direction to be used in a calculation for a relationship that exists between two columns:

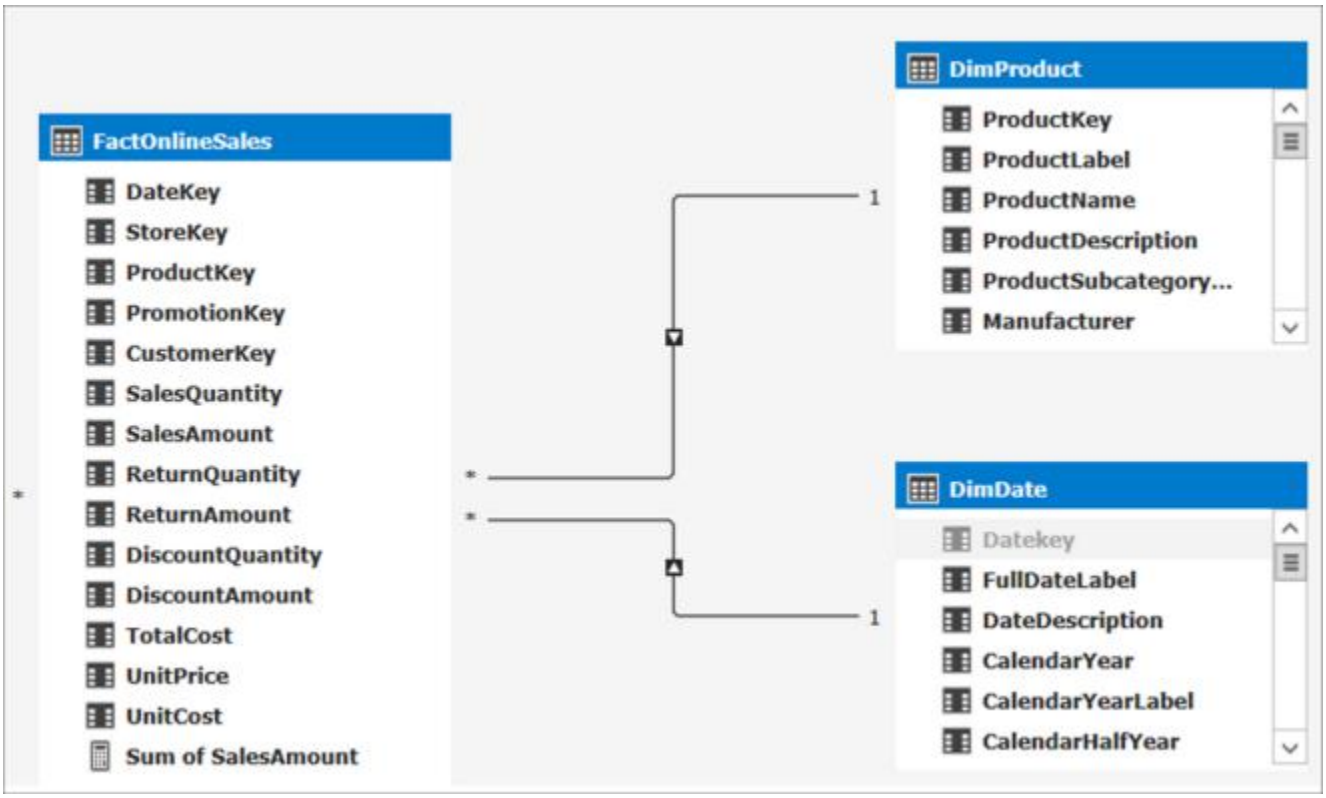
CROSSFILTER(<columnName1>, <columnName2>, <direction>)

It takes 3 parameters:

| Term | Definition |
|---|---|
| columnName1 | The name of an existing column, using standard DAX syntax and fully qualified, that usually represents the many side of the relationship to be used; if the arguments are given in reverse order the function will swap them before using them. This argument cannot be an expression. |
| columnName2 | The name of an existing column, using standard DAX syntax and fully qualified, that usually represents the one side or lookup side of the relationship to be used; if the arguments are given in reverse order the function will swap them before using them. This argument cannot be an expression. |
| Direction | The cross-filter direction to be used. Must be one of the following: none No cross-filtering occurs along this relationship<br>one - Filters on the one or lookup side of the side of the relationship filter the many side.<br>both - Filters on either side filter the other<br>none - No cross-filtering occurs along this relationship |

Let's look at the same example as from section two, where we wanted to see the count of products by year.

Repeating the same schema as in section two, the following model diagram shows both DimProduct and DimDate have a relationship with FactOnlineSales. Both are set to a single direction:




By default, we cannot get the count of Products sold by year:

| Row Labels | Sum of SalesAmount | Distinct Count of ProductKey |
|---|---|---|
| 2005 | $3,266,373.66 | 606 |
| 2006 | $6,530,343.53 | 606 |
| 2007 | $9,791,060.30 | 606 |
| 2008 | $9,770,899.74 | 606 |
| 2009 | | 606 |
| 2010 | | 606 |
| **Grand Total** | **$29,358,677.22** | **606** |

There are two ways to get the count of products by year:

Turn on bidirectional cross-filtering on the relationship. This will change how filters work for all data between these two tables as we have seen in section two. Or, we can use the CROSSFILTER function to change how the relationships work for just this measure.

When using DAX, we can use the CROSSFILTER function to change how the cross-filter direction behaves between two columns defined by a relationship. In this case, the DAX expression looks like this:

BiDi:= CALCULATE([Distinct Count of ProductKey]

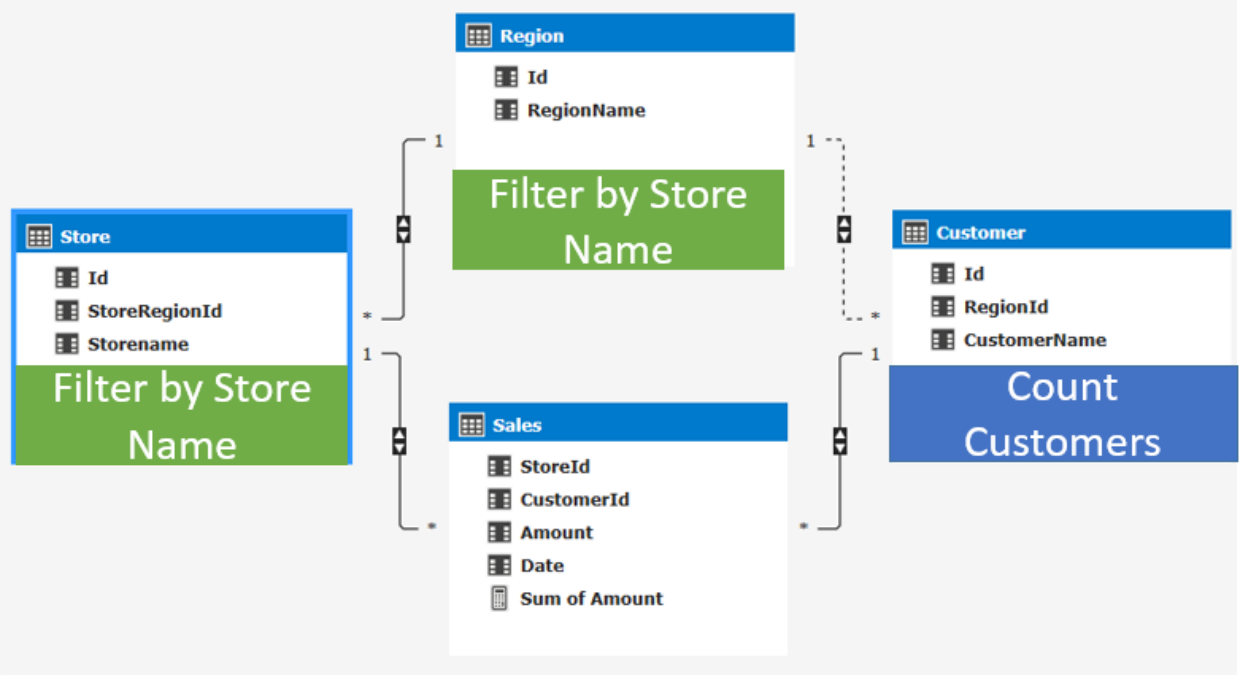
```
, CROSSFILTER(FactInternetSales[ProductKey]
              , DimProduct[ProductKey]
              , Both)
)
```

By using the CROSSFILTER function in our measure expression, we get the expected results for just this measure:

| Row Labels | Sum of SalesAmount | Distinct Count of ProductKey | BiDi |
|---|---|---|---|
| 2005 | $3,266,373.66 | 606 | 25 |
| 2006 | $6,530,343.53 | 606 | 56 |
| 2007 | $9,791,060.30 | 606 | 133 |
| 2008 | $9,770,899.74 | 606 | 102 |
| 2009 | | 606 | |
| 2010 | | 606 | |
| **Grand Total** | **$29,358,677.22** | **606** | **606** |

The use of this function opens many doors and allows you to tune the behavior per measure.

Let's look at one more example. In section four, we made sure the model doesn't have any ambiguity, but it came at a price. By default, based on the active relationships, we count the customers who bought something in the stores region, but maybe we want to count the customers who live in the stores region.



This means we need to use the non-active relationship. We can make use of a combination of CROSSFILTER and USERELATIONSHIP to make this happen:

```
Customers Count Store Region:= CALCULATE([Count of customers]
                                ,CrossFilter(Sales[CustomerId], Customer[Id], none )
                                ,UseRelationship (Customer[RegionId], Region[Id])
                                )
```

What this measure does is calculate the count of customers by disabling the cross-filtering on the relationship between Sales[CustomerId] and Customer[Id], and instead use the inactive relationship between Customer[RegionId] and Region[Id].

This now gives the count of customer for both paths in a single PivotTable:

| Row Labels | Count of customers | Customers Count Store Region |
|---|---|---|
| Store 1 | 4 | |
| Store 2 | 5 | 6 |
| Store 3 | 3 | 1 |
| Store 4 | 2 | 2 |
| Store 5 | 4 | 6 |
| Store 6 | 2 | 1 |

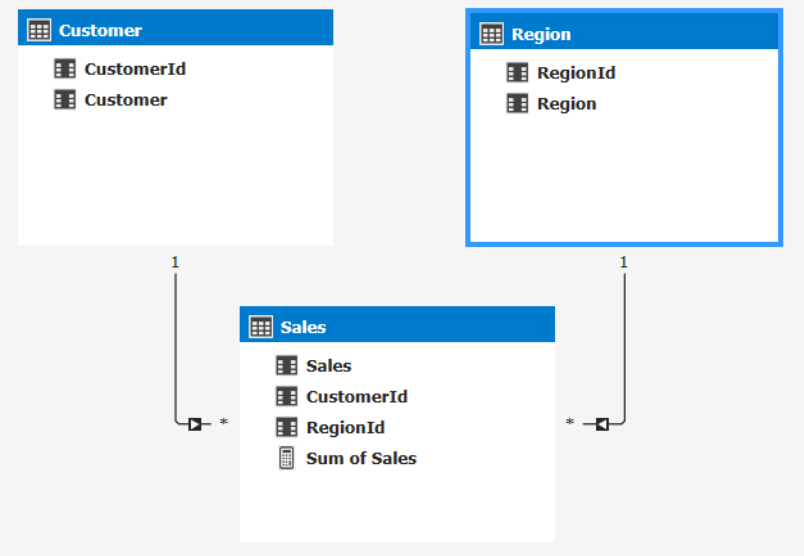
As you've seen, using DAX can give you great flexibility to turn on or off bidirectional cross-filtering per measure.

## Row level security and bidirectional relationships

Another interesting topic is to see how row level security plays (RLS) together with bi directional relationships. More information on row level security is available in this whitepaper.

Both RLS and bi directional relationships work by filtering data. Bi directional relationships explicitly filter data when the columns are actually used on axis, rows, columns or filters in a PivotTable or any other visuals. RLS implicitly filters data based on the expressions defined in the roles.

Let's look at a small example:



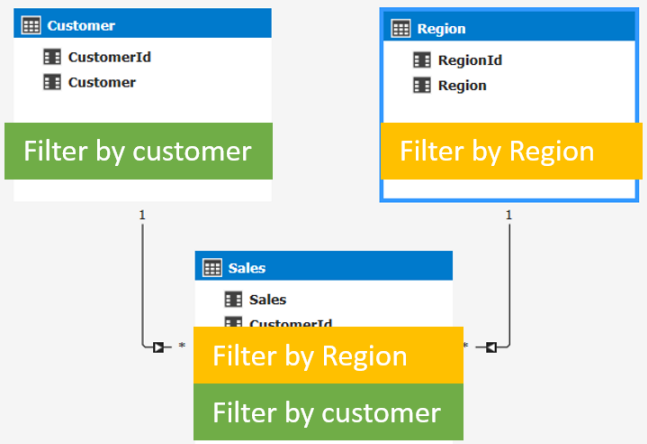

Here I have three tables: Customer, Region and their Sales. I create a role that sets a filter to RegionId = 1. Now when I create a PivotTable and I start by dragging in Customers, I see all customers:

| Row Labels |
|---|
| Customer 1 |
| Customer 2 |
| Customer 3 |
| Customer 4 |
| **Grand Total** |

That's to be expected, there are no RLS filters defined on the Customer table. Now when I drag in Sum of Sales we do see a filtered down list of customers to only those who have sales in Region with Id = 1.

| Row Labels | Sum of Sales |
|---|---|
| Customer 1 | 23 |
| Customer 4 | 43 |
| **Grand Total** | **66** |

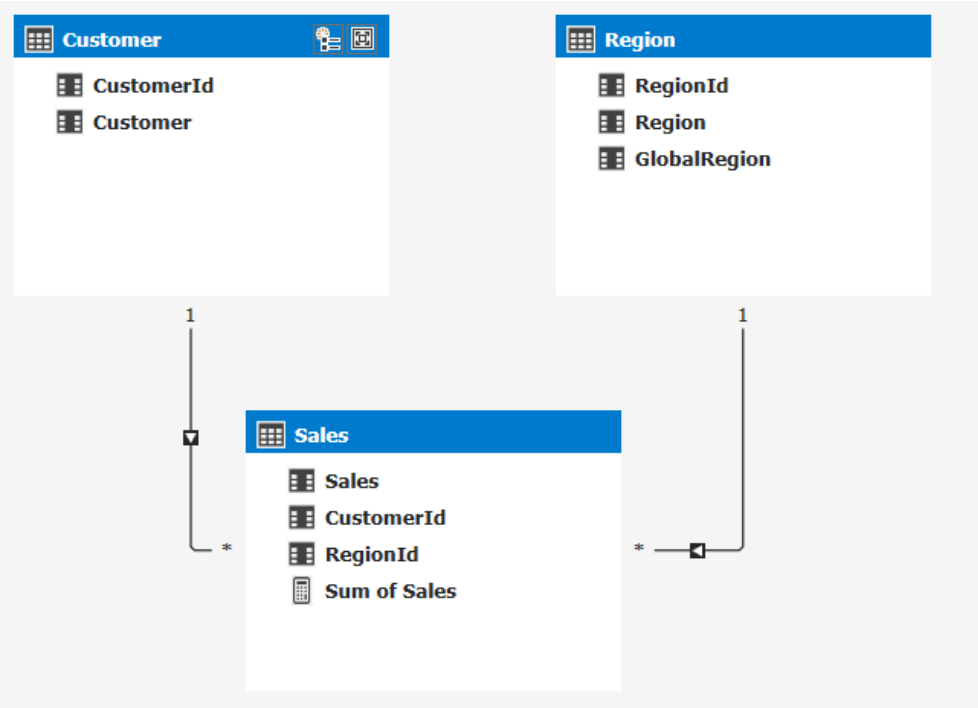
Again, this is to expected as we defined security on the Region table. This will automatically filter down the Sales table to only the values for the region I have access to. I did not have to drag in any field from the Region table:




Now imagine row level security would also honor bidirectional filters, securing the region table would automatically always filter out the Customer table. You would never be able to see Customer 2 and 3, even if you want to look at them in isolation from the Sales table. This is the reason that row level security only uses single directional filters by default.

Now there are some cases you do want to turn on bidirectional cross-filtering when using row level security; for example, let's look at adding dynamic security to our model. Dynamic security provides row-level security based on the user name or login id of the user currently logged on.

I want to secure my model not per region, but a grouping of regions called GlobalRegion:




Next, I add a table that declares which user belongs to a globalregion:


The goal here is to restrict access to the data based on the user name the user uses to connect to the model. The data looks like this:

| | UserId | GlobalRegion |
|---|---|---|
| 1 | Domain\User1 | East |
| 2 | Domain\User2 | East |
| 3 | Domain\User3 | West |
| 4 | Domain\User4 | West |
| 5 | Domain\User4 | East |

Now we can create a row level security expression on the region table that will determine the global region the current connected user has access to. To do that we can create this DAX expression:

=Region[GlobalRegion]=LOOKUPVALUE(Security[GlobalRegion]

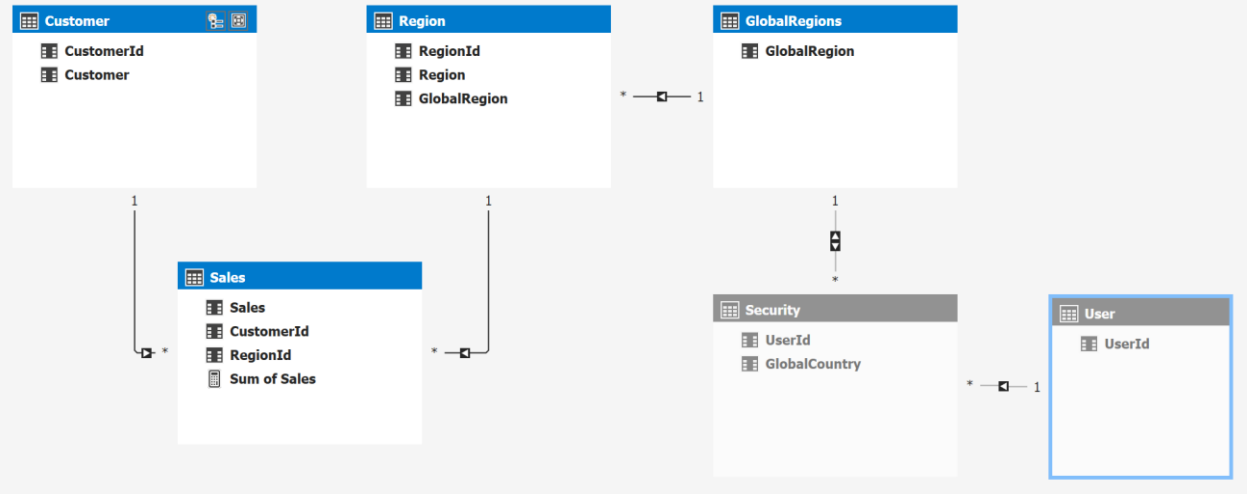
, Security[UserId], USERNAME()

, Region[GlobalRegion], Security[GlobalRegion])

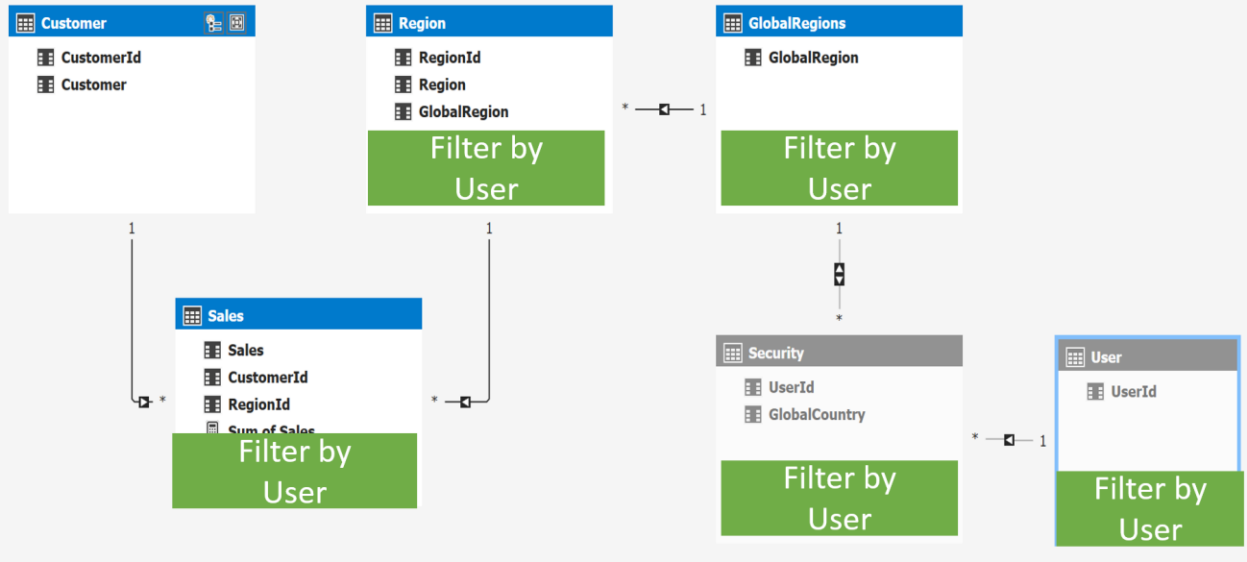
This DAX expression will look up any values for GlobalRegion based on the username of the user connecting to the SSAS model. This is a pretty complicated scenario and also won't work for models in DirectQuery mode as the LOOKUPVALUE function isn't supported. For that reason, we introduced a new property in SQL Server 2016 that will allow you to leverage bi directional cross-filtering to enable the same scenario.

> Note: when using Power BI you should use the USERPRINCIPALNAME() function instead of the USERNAME() function. This will make sure the actual username will get returned, USERNAME will give you an internal representation of the username in Power BI.

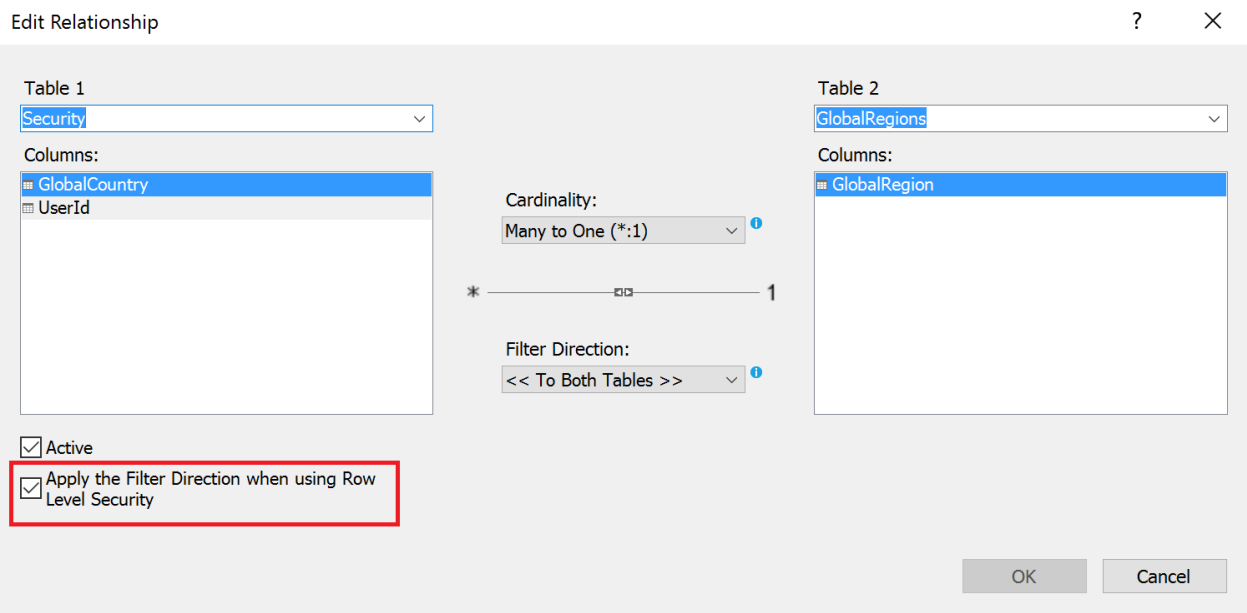
Let's take a look at how this new property works. Instead of leveraging the DAX function to find the username and filter the table, we'll be using the relationships in the model itself. I've extended the model with some additional tables to make this possible. Observe we now have a separate user table and a separate GlobalRegions tables with an intermediate table in the middle that connects the two. The relationship between User and GlobalRegion is a many to many relationship as a user can belong to many global regions and a global region can belong to many users.




The idea here is that we can add a simple row level security filter on the User[UserId] column like =USERNAME() and this filter will now be applied to all of the related tables:




There is just one issue here, as you can see the relationship between Security and GlobalRegions is set to bidirectional cross-filtering, and earlier we determined this will not be honored for RLS scenarios. Luckily there is a way to get this working for RLS scenarios as well. To turn it on I go to the diagram view in SSDT and select the relationship:




Here I can select the property that applies the filter direction when using Row Level Security.

This property only works for certain models and is designed to follow the above pattern with dynamic row level security as shown in the example above. Trying to use this in for other patterns might not work and Analysis Services might throw an error to warn you.

## Summary

As we've seen here, the new bidirectional cross-filtering feature is very powerful and it allows us to solve complex models with more ease. But with this great new functionality also comes potential complexity. It's important to understand what enabling bidirectional cross-filtering will do for your model. You need to check what the behavior and performance characteristics will be for every relationship you want to enable it for.

Note: This feature is sometimes also called "Many-to-Many", based on the feature found in Multidimensional models. But, as discussed here, it allows many other scenarios beside the traditional BI "many-to-many" features. This is why we're referring to it differently for Tabular models.