QlikView

Generic Keys

QlikView Technical Brief

2 October 2012, HIC

www.qlikview.com

QlikView

Contents

Introduction

This document is about **Generic Keys**, which is a way to define keys between tables in a more general way so that their values can represent other things than individual keys; they can represent groups of keys or any key. With them, a more flexible data model can be created and more data modeling problems can be solved.

Generic keys should not be confused with composite keys, which is a different concept. Composite keys are keys that contain information from several individual keys, e.g. by a simple concatenation of the two individual keys, whereas generic keys are keys, often from a *single* field, that contain symbolic key values that represent several or all individual key values.

An example: In many data models, there is a product table where each product has a unique product ID. If you create a generic key for such a dimension, the generic key could contain not only the individual product IDs, but also generic symbols for product categories and a symbol representing ***all*** products.

In most apps, generic keys are not necessary and should not be used. But there are cases where it can solve real problems and then you should not hesitate to use them:

- Authorization table with OR-logic between fields
  If you have an authorization table, either in Section Access or in the Publisher, you sometimes want to have a slightly more complex access restriction than a simple logical AND between fields. It could be e.g., that a user is allowed to see sales for all regions for a specific product and at the same time the European sales for all products. It is straightforward to achieve such an access restriction using generic keys.

- Mixed dimensional granularity in a single fact table
  Often you want to compare actual numbers with budget numbers. The standard method is to concatenate these two tables into one common fact table. However, this new fact table could have mixed granularity in many of the dimensions:

	Dimensions for Actual numbers	Dimensions for Budget numbers
Geography	Customer	Region
Product	Product	Product Group
Organizational	Salesman	Department
Time	Day	Year

  Generic keys can be a help here.

- Multiple fact tables linked using a master link table
  Sometimes you have different keys in the different fact tables, e.g. in the pharmaceutical area you often want to compare actual pharmacy sales to the number of sales calls that

have been made. These two fact tables have different set of keys so they cannot easily be linked. For instance, the physician to which the sales representative has made a call exists in the calls table but not in the sales table. Further, package size exists in the sales table, but not in the calls table. To solve this, you will need to define keys that link to all sales transactions and to all calls. Generic keys can be a help here.

Basics

Let us compare how a dimensional table can be linked to some other table, typically a fact table, an authorization table or a link table, with or without generic keys. In a normal case, the dimension is linked directly to the fact table using the appropriate key; for a product dimension this would be the product ID.

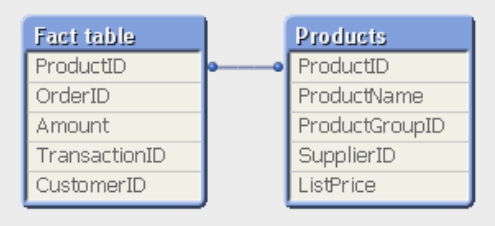

Figure 1. A product table linked to a fact table in a standard solution.

If the same product table would be linked to the fact table, but now using a generic key, the solution would instead look like this:

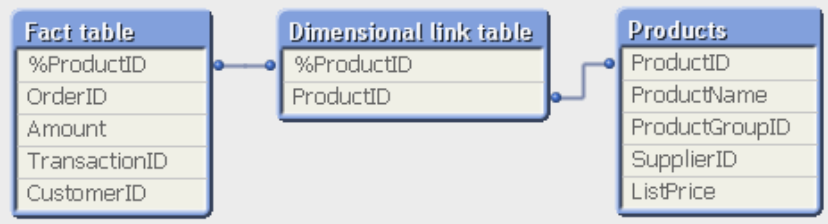

Figure 2. Generic key: A product table linked to a fact table via a generic key.

And the dimensional link table could look like this:

Dimensional link table

%ProductID	ProductID
2	2
1	1
<ANY>	1
<ANY>	2

Figure 3. A simple dimensional link table. All products link to themselves, and in addition, the <ANY> symbol links to all products.

So, basically the dimensional table is linked to the fact table using a bridge; a dimensional link table. The key between the link table and the unmodified dimensional table is the original dimensional key, ProductID. The key between the link table and the fact table is the generic key, %ProductID. The only difference between the generic product key and the original product key is that the symbol '<ANY>' has been introduced as a generic symbol; a key value that links to ***all*** products.

The dimensional link table is created by loading the product ID from the product table twice, using two Load statements:

```
[Dimensional link table]:
Load  ProductID,
      ProductID as %ProductID          From Products;

Load  ProductID,
      '<ANY>' as %ProductID            From Products;
```

With this new data model, we can allow records in the fact table that use '<ANY>' as product id and link to all products. This is useful in some data modeling situations e.g. when you want to load budget and actual in the same fact table or if you have a complex security model involving access restrictions in a matrix of several dimensions.

This is a very basic example, but it shows quite well the basic principles: the dimensional link table as a bridge and the generic key that contains symbols for groups of field values. In real life, however, the model becomes more complex than this. But more about that later.

Groups in the dimension

In most databases, some dimensional values are categorized into groups, e.g. products are categorized into product groups. Also these can be represented in a generic key. Hence, just as the generic key can contain a symbolic value such as '<ANY>', it can also contain generic symbols representing product groups, e.g. '<Group:1>', '<Group:2>', etc.

Data model

In a standard solution, you would link the product group table to the product table in a snowflake scheme. (It is called snowflake since this is what it looks like when you have linked all dimensions to the fact table this way; customers/countries, dates/periods, departments/business units, etc.).

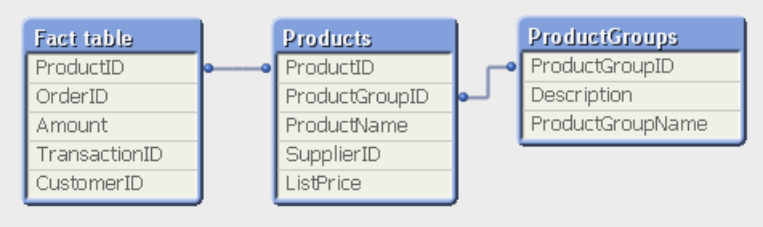

Figure 4. A product group table linked to a fact table via the product table in a standard solution.

But if you use generic keys, it will not work to link product group information to the fact table indirectly via the product table: there may exist records in the fact table that pertain to a product group – and products groups are not found in the product table. So, the attributes in the product group table would not link to these records.

Instead, the product group table should link to the dimensional link table. Only this way can you ensure that selections are propagated correctly through the data model.

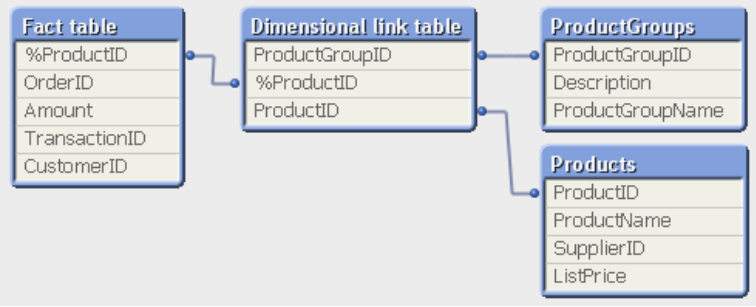

Figure 5. Generic key: A product group table linked to a fact table via the dimensional link table.

Dimensional link table

The dimensional link table would in this situation have three groups of records – one where the original product IDs link to product IDs; a second where the product groups link to the correct product IDs; and a third where the <ANY> symbol links to all product IDs.

The dimensional link table would have three columns: the generic product key, the original product key and the product group key. It could then look like this:

Dimensional link table

%ProductID	ProductGroupID	ProductID
<Product:1>	1	1
<Product:2>	1	2
<Product:3>	2	3
<Product:4>	2	4
<Productgroup:1>	1	1
<Productgroup:1>	1	2
<Productgroup:2>	2	3
<Productgroup:2>	2	4
<ANY>	1	1
<ANY>	1	2
<ANY>	2	3
<ANY>	2	4

Figure 6. A dimensional link table with groups. All products link to themselves, and in addition, the <ANY> symbol as well as the product groups link to the respective products.

Here the generic key has been created by a concatenation of the field label and the field value. But the generic symbols do not have to look exactly like these. However, they ***do*** have to include not only the field value – e.g. the ID of the product or the product group – but also the information about which field it refers to. This way, a product group record cannot by mistake get the same value as another record that refers to a product. In other words, the product group ID which is a number cannot be stored as a number only since it then would collide with the product IDs which also are numbers.

The dimensional link table is created using three consecutive Load or SELECT statements, one for the individual products, one for the product groups, and one for the <ANY> symbol:

```
[Dimensional link table]:
Load  '<Product:' & ProductID & '>'          as %ProductID,
      ProductGroupID,
      ProductID                              From Products;
Load  '<Productgroup:' & ProductGroupID & '>' as %ProductID,
      ProductGroupID,
      ProductID                              From Products;
Load  '<ANY>'                                as %ProductID,
      ProductGroupID,
      ProductID                              From Products;
```

An alternative approach to achieve the same goal is to concatenate the product group ID with the product ID using a known delimiter, e.g.:

Dimensional link table

%ProductID	ProductGroupID	ProductID
1|1	1	1
1|2	1	2
2|3	2	3
2|4	2	4
1|<ANY>	1	1
1|<ANY>	1	2
2|<ANY>	2	3
2|<ANY>	2	4
<ANY>|<ANY>	1	1
<ANY>|<ANY>	1	2
<ANY>|<ANY>	2	3
<ANY>|<ANY>	2	4

Figure 7. An alternative representation of the generic key: concatenation of product group ID and product ID. Whether to choose this or the one above is just a matter of taste. Both work.

Inclusive and exclusive generic symbols

A generic symbol can be inclusive or exclusive. “Inclusive” meaning that the symbol includes all IDs within the group, e.g. <Productgroup:1> does indeed link to all the products within this product group. Hence, the example above uses inclusive generic symbols.

In contrast, an *exclusive* generic symbol would not link to the individual IDs within the group, but it would only link to the group itself. In principle, the <ANY> symbol has been replaced by a <N/A> symbol (Not Applicable). Instead of an <N/A> symbol, you can also use null().

Another way to put it is to say that an inclusive generic symbol links to ***all*** relevant elements in the group, whereas an exclusive generic symbol links to ***no*** elements. It links to the group only.

Dimensional link table

%ProductID	ProductGroupID	ProductID
1|1	1	1
1|2	1	2
2|3	2	3
2|4	2	4
1|<N/A>	1	-
2|<N/A>	2	-
<N/A>|<N/A>	-	-

Figure 8. A dimensional link table with exclusive generic symbols. All products link to themselves, and in addition, the product groups link to the respective product groups but not to the individual products.

Whether to use inclusive or exclusive generic symbols is up to you. In some situations it is better to use inclusive generic symbols; in other situations it is better to use the exclusive generic symbols.

For example, if you have an authorization table where a specific user is allowed to see a specific region in combination with any product, you obviously want the <ANY> symbol to link to ***all*** individual products. Otherwise, the reduction would exclude the relevant records. Hence, you should use inclusive generic symbols.

However, if you have a comparison between actual numbers and budget numbers and you have a mixed granularity, e.g. the actual numbers are per product but the budget is per product group, then the budget is unspecified (N/A) for the individual products. In such a situation, you probably want the budget numbers to disappear when the user selects a specific product, but have them visible when no such selection is made. The budget number would otherwise be misleading. In such a case, you should use exclusive generic symbols.

Just as before, the dimensional link table is created using several consecutive Load or SELECT statements, one for the individual products, one for the product groups, and one for the top <N/A> symbol. The third load statement is really not necessary, since it does not link any generic symbols to real values. But I use it to show the analogy with the inclusive generic symbols.

```
[Dimensional link table]:
Load   ProductGroupID & '|' & ProductID      as %ProductID,
       ProductGroupID,
       ProductID                             From Products;
Load distinct ProductGroupID & '|' & '<N/A>' as %ProductID,
       ProductGroupID,
       Null() as ProductID                   From Products;
Load distinct '<N/A>' & '|' & '<N/A>'        as %ProductID,
       Null() as ProductGroupID,
       Null() as ProductID                   From Products;
```

Composite generic keys

In some cases you want IDs from several dimensions in the same generic key. As always when you create composite keys, it is just a matter of concatenating the different keys with a proper delimiter.

Link table

%MasterKey	ProductID	CustomerID
<ANY>|<ANY>	1	BERGS
<ANY>|<ANY>	1	BLONP
<ANY>|<ANY>	2	BERGS
<ANY>|BERGS	1	BERGS
<ANY>|BERGS	2	BERGS
<ANY>|BLONP	1	BLONP
1|<ANY>	1	BERGS
1|<ANY>	1	BLONP
1|BERGS	1	BERGS
1|BLONP	1	BLONP
2|<ANY>	2	BERGS
2|BERGS	2	BERGS

Figure 9. A master link table. The individual keys are concatenated into a master generic key.

Just as before, the link table is created using several consecutive Load or SELECT statements, one for each combination of the different cases:

```
[Link table]:
Load  ProductID & '|' & CustomerID    as %MasterKey,
      ProductID,
      CustomerID                          From …;
Load  ProductID & '|' & '<ANY>'       as %MasterKey,
      ProductID,
      CustomerID                          From …;
Load  '<ANY>' & '|' & CustomerID      as %MasterKey,
      ProductID,
      CustomerID                          From …;
Load  '<ANY>' & '|' & '<ANY>'         as %MasterKey,
      ProductID,
      CustomerID                          From …;
```

One minor complication when creating composite keys is that you can no longer create the keys by loading records from the dimensional tables: There is rarely one single dimensional table that contains keys from several dimensions. Instead you must create the link table loading from the fact table(s).

This sometimes leads to the next complication: You may not have all the necessary keys in your fact table. For example, you probably do not have the key for product group. Should you need a key that does not exist there, then the best way to get that information is to use the applymap function. With this, you can make a lookup and get the information you need.

Example 1: Complex security - Authorization bridge table

The first example is on authorization, i.e. access restriction where a user is allowed to see some, but not all data. I assume that the process of authentication (user identification) has been made so that QlikView "knows" which user it is that holds the session. The step of authorization is then a matter of determining which data the user is allowed to see.

QlikView can reduce the data so that the user only can see that which has been approved for that specific user. Such a reduction can be made either with a distribution within the QlikView Publisher, or using the Section Access within the QlikView script. In both cases, you need one or several reducing fields that are connected to the user IDs; you need an authorization table listing field values approved for each user. QlikView or QlikView Publisher will then make the selection as defined by the listed field values and purge all excluded data from the session or from the file.

If you want to make a reduction in a single field, you do not need generic keys.

If you want to limit data to the intersection of filters set in two or more fields, e.g. a user is allowed to see records that pertain to product X ***and*** customer A, but nothing else; then you can just link the two reducing fields to your data model. You do not need generic keys.

ACCESS	NTNAME	PRODUCT_ID	CUSTOMER_ID
ADMIN	ADMIN	ANY	ANY
USER	ALAN	ANY	1;2;3
USER	BETTY	2;3;4	ANY
USER	CHRISTA	3;4	5;6
USER	DAVID	ANY	5;7
USER	DAVID	6;7	ANY

Figure 10. Source table with authorization information. The user "DAVID" is allowed to see all products, but just for customer 5 and 7. In addition, he is also allowed to see all customers, but just for product 6 or 7.

However, if you want to limit data to the union of filters set in two or more fields, e.g. either to product X ***or*** to customer A, then you will need generic keys. Such an authorization table could look like the one above.

This authorization table already contains symbols to for groups of products or groups of customers. To load this in QlikView, you will need to interpret the generic keys in an authorization bridge table. The authorization bridge table (second table from the right in the picture below) will link your authorization table with your transaction table using AuthID and %AuthID, which both are composite keys with information on both product ID and customer ID. In addition, %AuthID is a generic key.

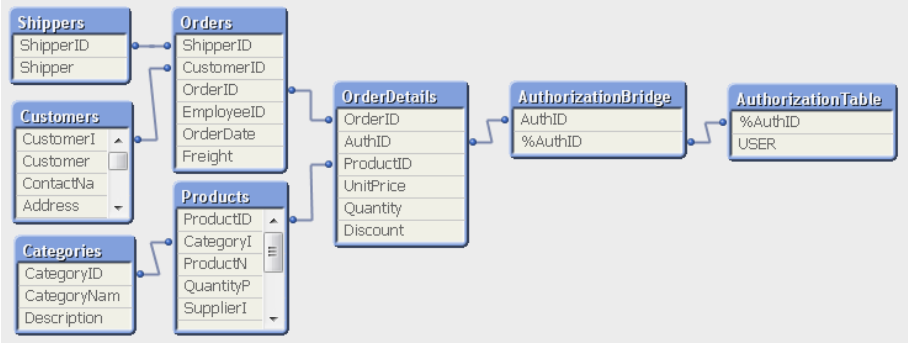

Figure 11. Example of data model when using an authorization bridge table.

Authorization table in the QlikView data model

The first step is to load the authorization table (rightmost table in the picture above). To do this, I expand the individual rows into their components, using the subfield function. The subfield function will make the Load statement loop over the individual records so that each subfield ends up in its own record. But I do not want to map the <ANY> symbols to individual products and individual customers yet. Instead I will load this as a generic key.

Further, I will in this example use a copy of the user ID (NTNAME) as reducing field. Hence,

```
AuthorizationTable:
Load Upper(NTNAME) as USER,
    Subfield(PRODUCT_ID, ';') & '|' & Subfield(CUSTOMER_ID, ';') as %AUTH_ID
    From AuthorizationTable ;
```

Authorization table

USER	%AuthID
ADMIN	<ANY>|<ANY>
ALAN	<ANY>|1
ALAN	<ANY>|2
ALAN	<ANY>|3
BETTY	2|<ANY>
BETTY	3|<ANY>
BETTY	4|<ANY>
CHRISTA	3|5
CHRISTA	3|6
CHRISTA	4|5
CHRISTA	4|6
DAVID	<ANY>|5
DAVID	<ANY>|7
DAVID	6|<ANY>
DAVID	7|<ANY>

Figure 12. The expanded authorization table when loaded into QlikView.

Authorization ID in the transaction table

I also need to define the authorization ID in the transaction table. In my example, this means the order details table. However, there is no customer ID in this table, so I need to fetch this from the order header table. But rather than joining the two tables, I use the applymap function.

```
OrderID_to_CustID:
Mapping Load OrderID, CustomerID                From Orders;

OrderDetails:
Load  *,
    ProductID & '|' & Applymap('OrderID_to_CustID', OrderID, 'NONE') as AuthID
                                                From OrderDetails ;
```

Authorization bridge table

The next step is to create the authorization bridge table, which is the table that will link the generic symbols to their real values. This is a table that potentially can become very large: If I would generate all combinations of product ID and customer ID, with the possibility of generic keys, I would face a very large number of records.

Authorization bridge table

AuthID	%AuthID
1|1	<ANY>|1
2|3	<ANY>|3
4|24	4|<ANY>
11|2	<ANY>|2
11|3	<ANY>|3

Figure 13. Example of content of an authorization bridge table.

Instead of generating all possible combinations, I load only the combinations that exist in the transaction table and at the same time in the authorization table. But to do this I need four Load statements, each with a preceding load. Hence:

```
AuthorizationBridge:
Load distinct * Where Exists(%AuthID);
Load  AuthID as %AuthID,
      AuthID                                    resident OrderDetails ;
Load distinct * Where Exists(%AuthID);
Load  '<ANY>' & '|' & Applymap('OrderID_to_CustID', OrderID, 'NONE') as %AuthID,
      AuthID                                    resident OrderDetails;
Load distinct * Where Exists(%AuthID);
Load  ProductID & '|' & '<ANY>' as %AuthID,
      AuthID                                    resident OrderDetails;
Load distinct * Where Exists(%AuthID);
Load  '<ANY>' & '|' & '<ANY>' as %AuthID,
      AuthID                                    resident OrderDetails;
```

In other words: I load from the order details, so I only get the combinations that really exist in the transactional data. In addition, I pipe the result into a preceding load to filter it further; I only save the records where the corresponding generic key exists in the authorization table.

Finally, I need to create the Section Access table with the user names. The reduction will then be made on the USER field.

```
Section Access;
Load ACCESS, NTNAME, Upper(NTNAME)      as USER
    From AuthorizationTable;            Section Application;
```

An alternative to the section access table is to distribute the app using the QlikView Publisher. All you need to do then is to reduce and distribute on the field USER.

And with this the problem is solved.

Example 2: Concatenation of fact tables - Actual and Budget

Another common case is when you want to compare actual numbers from a sales or a cost data base with the budgeted numbers. The best way to do this is in my mind to concatenate the two tables into a common fact table.

When doing so you often encounter the problem of different granularity. The actual numbers are e.g. per day where the budget numbers are per month; the actual numbers are per customer where the budget numbers are per region or country; the actual numbers are per salesman where the budget numbers are per sales unit; etc. The pictures below show a typical situation.

Actual numbers

OrderID	CustomerID	SalesRepID	ProductID	OrderDate	ShipperID	Amount
10248	4	2	11	2004-06-29	2	34.344 $
10253	34	3	31	2004-07-05	2	16.560 $
10254	19	2	24	2003-01-06	2	6.780 $
10258	17	7	2	2004-07-12	2	100.100 $
10258	17	7	5	2004-07-12	2	39.845 $

Figure 14. Fact table for actual numbers with five foreign keys: CustomerID, SalesRepID, ProductID, OrderDate and ShipperID.

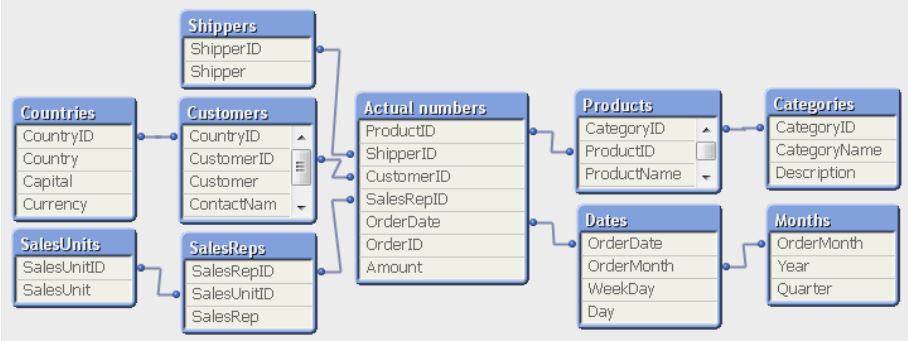

Figure 15. Snowflake data model for actual numbers with its corresponding five dimensions: shippers; customers and countries; sales reps and sales units; products and product groups; and finally the master calendar with dates and months.

Budget numbers

CountryID	SalesUnitID	CategoryID	Month	Amount
France	2	1	2012-01	2.523.000 $
France	3	1	2012-01	3.928.000 $
Brazil	6	1	2012-01	3.615.000 $
Germany	6	1	2012-01	455.000 $
Ireland	6	1	2012-01	4.567.000 $

Figure 16. Fact table for budget numbers with four foreign keys: CountryID, SalesUnitID, product CategoryID and Month.

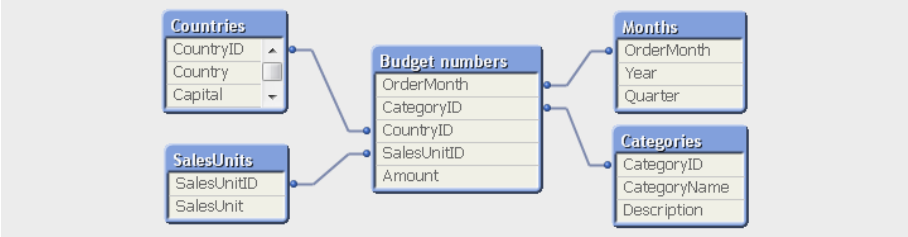

Figure 17. Data model for budget numbers with its corresponding four dimensions: customer countries, sales units, product groups and months. Note that the shippers' dimension is missing as well as the detailed level of the four existing dimensions.

Although the two data models are different, they share a lot of information. The four dimensional tables in the budget data model are identical to the corresponding tables in the data model for the actual numbers. The goal is to merge the two data models without losing any information, which can be done using generic keys.

Loading the actual numbers

First, I load the actual numbers. However, instead of the normal keys for the four common dimensions, I use generic keys. But these cannot be created without knowledge about the dimensional groups, e.g. which product group a specific product belongs to. This is information that can be fetched from the dimensional tables using the applymap function. Hence:

```
CustomerID_to_CountryID:
Mapping Load CustomerID, CountryID          From Customers;
SalesRepID_to_SalesUnitID:
Mapping Load SalesRepID, SalesUnitID        From SalesReps;
ProductID_to_CategoryID:
Mapping Load ProductID, CategoryID          From Products;

Facts:
Load Amount, OrderID, ShipperID,
    Applymap('CustomerID_to_CountryID',   CustomerID,null()) & '|' & CustomerID   as %CustomerID,
    Applymap('SalesRepID_to_SalesUnitID',SalesRepID,null()) & '|' & SalesRepID   as %SalesRepID,
    Applymap('ProductID_to_CategoryID',    ProductID,null())  & '|' & ProductID    as %ProductID,
    Num(MonthStart(OrderDate))  & '|' & Num(Floor(OrderDate))                      as %OrderDate,
    'Actual' as Type                                From ActualNumbers ;
```

In other words: The CountryID is fetched from the customer table; the SalesUnitID is fetched from the salesman table; and the CategoryID is fetched from the product table. All three are stored in the corresponding generic keys. For the fourth generic key I use the date serial number as integers; first for the month, then for the date.

Loading the budget numbers

Next step is to append the budget numbers onto this table using the concatenate prefix. Also here I use generic keys, but this time I use the N/A symbol for the detailed level:

```
Concatenate (Facts) Load Amount,
    CountryID    & '|' & 'N/A'                    as %CustomerID,
    SalesUnitID & '|' & 'N/A'                    as %SalesRepID,
    CategoryID  & '|' & 'N/A'                    as %ProductID,
    Num(MakeDate(Year, Month)) & '|' &  'N/A'    as %OrderDate,
    'Budget' as Type                             From Budget ;
```

The fields OrderID and ShipperID are missing so they will get NULL values in the budget part of the fact table, which is OK since these fields are irrelevant for the budget numbers. The field Type finally, is a created field with just two values: ‘Actual’ and ‘Budget’. It can be used as dimension in charts and as flag in expressions.

Dimensions and dimensional link tables

Further, I create the dimensional link tables, just as described above in the section “Groups in the dimensions”. The group symbols should be exclusive, since I do not want the individual dimensional values to link to budget numbers.

```
[%Products]:
Load  CategoryID & '|' & ProductID      as %ProductID,
      CategoryID,
      ProductID                                 From Products;

Load  CategoryID & '|' & '<N/A>'        as %ProductID,
      CategoryID,
      Null() as ProductID                       From Products;
```

The other dimensional link tables are created in a similar way.

Finally, when loading the dimensions like the product table, these must not contain the key to the dimensional group like the product group. The obtained data model now looks like in the picture below. All information is there and all links work correctly.

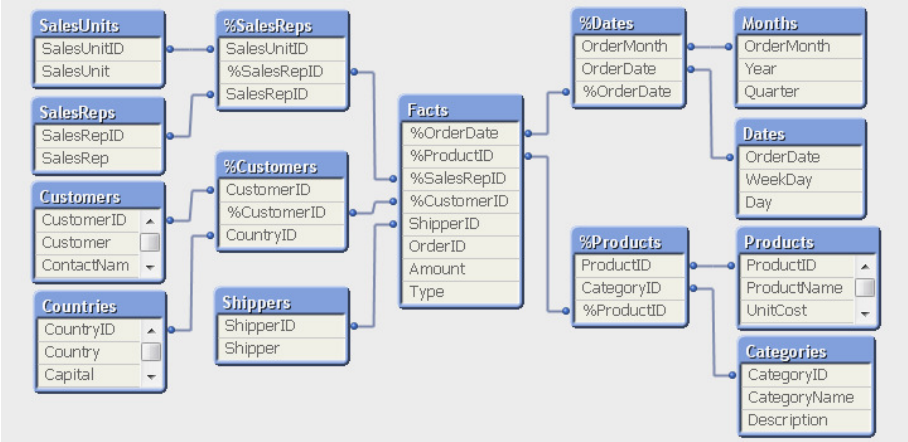

Figure 18. The data model with the actual numbers merged with the budget numbers.

Example 3: Link table and multiple fact tables

In some cases, the fact tables are so different that you do not want to concatenate them. Then you can create a solution using generic keys in combination with a master link table.

My example comes from the pharmaceutical industry where the situation often is that you want to compare sales numbers from pharmacies with the sales calls that your sales representatives have made to physicians. The problem is similar to the previous example, however much more complex.

The sales numbers are per product package, i.e. a sub-group in the product dimension. Geographically the sales numbers are reported per physician, per pharmacy or per territory (depending on country), which only indirectly is connected to the sales representative.

The sales calls are found in two tables: Visits and VisitDetails. These describe how a sales representative visits a physician and shows products. Several products can be shown in a sales call.

Finally, the sales representatives have targets on how many calls they should make and how many products they should show. These are stored in the Targets table.

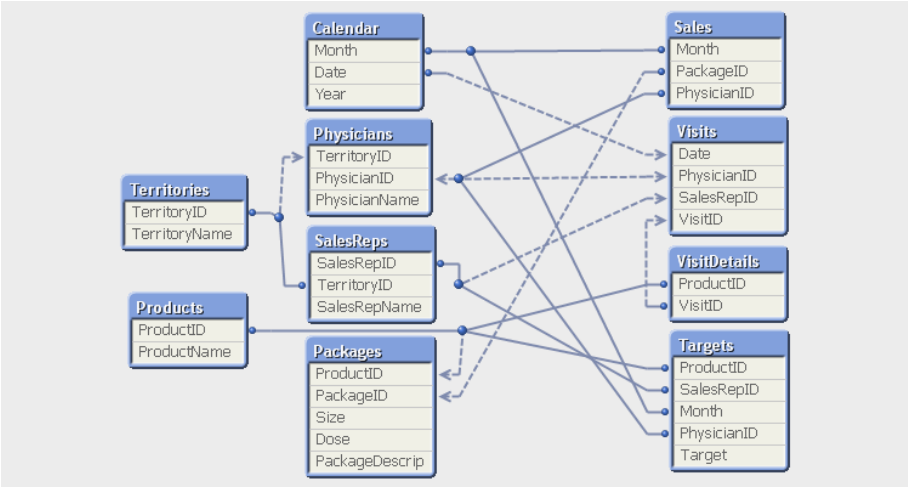

Figure 19. Sales data model from the pharmaceutical industry – if tables were loaded as-is. Dimensions to the left; fact tables to the right.

An additional complication is the forked territorial dimension: Both physicians and sales representatives belong to territories, but there is no direct connection between a specific sales representative and a specific physician.

The details of the challenge differ from country to country and from company to company, but the example still describes the general problem well.

Loading these tables into QlikView as they are will obviously not work. There are far too many links that create circular dependencies. Instead the data must be remodelled into a snowflake scheme. For this, generic keys in combination with a master link table can be a great help.

There are four table types in this solution: Dimensions, Dimensional link tables, Fact tables and the Master link table. For each type there are some things to be aware of:

Dimensions and dimensional link tables

First there are the dimensional tables. These should be loaded in a standard way, however without the keys to the groups. For instance, the product table should not contain a key to product groups. The dimensional groups should still be loaded as individual dimensional tables. In this case it means that products, months and territories should be loaded as separate tables.

All fields that the user will use to make selections should exist in the dimensional tables.

Then there are the dimensional link tables. These should be loaded as described above in the section “Groups in the dimension”, i.e. with several consecutive Load statements so that also the <ANY> symbol links to all individual dimensional elements.

The question is which generic keys you should have. In my example I choose to have territories as a separate independent dimension, but I combine date and month into one generic key and package and product into another. All in all, I get five generic keys: %TerritoryID, %ProductID, %SalesRepID, %PhysicianID and %Date.

Fact tables

The third group of tables is the fact tables. These should be loaded in such a way that they only contain a master key and numbers that should be used in aggregations. They may not contain any fields used for selections. The master key should contain information from all keys used as generic keys. If the key is not relevant for the specific table, this part of the master key should have the <ANY> symbol. The master key should be defined in the same way in all fact tables.

In this case it means that the master key should consist of the information that constitutes the five generic keys:

%TerritoryID + %ProductID + %SalesRepID + %PhysicianID + %Date

But this is just half the truth: %Product and %Date have groups within the generic key, so in practice the master key will be a concatenation of seven keys:

TerritoryID + ProductID + PackageID + SalesRepID + PhysicianID + Month + Date

Some of the keys are not directly available and need to be fetched from other tables using applymap.

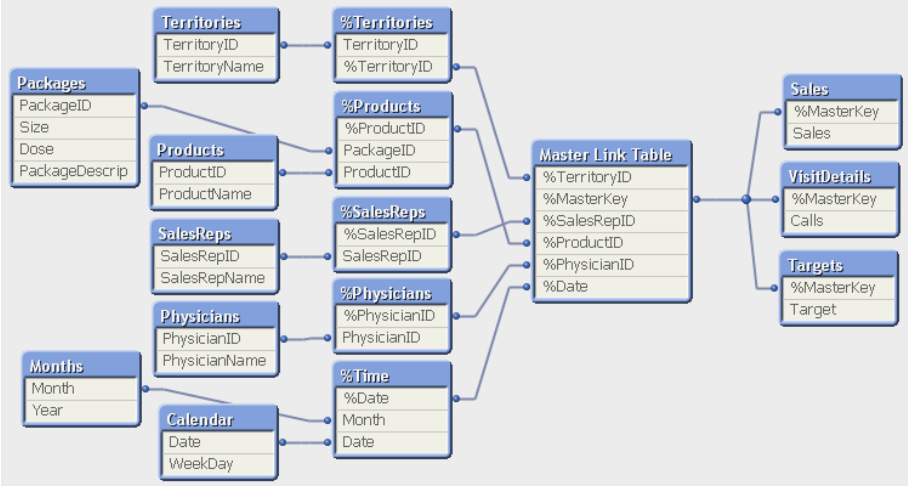

Figure 20. Example of a correct sales data model from the pharmaceutical industry. Tables are loaded using generic keys and a link table. All dimensions are to the left and all fact tables to the far right.

First, I look at the sales table: It has keys for month, package and physician. The territory and the product can be deduced from these keys using applymap. However, the sales representative and date is not applicable, so these should be replaced with the <ANY> symbol. Hence, the corresponding Load statement will be

```
Sales:
Load Sales,
    Applymap('PhysicianID_to_TerritoryID', PhysicianID, null())  & '|' &    // -- TerritoryID
    Applymap('PackageID_to_ProductID', PackageID,null())         & '|' &    // -- ProductID
    PackageID                                                    & '|' &    // -- PackageID
    '<ANY>'                                                      & '|' &    // -- SalesrepID
    PhysicianID                                                  & '|' &    // -- PhysicianID
    Num(Month)                                                   & '|' &    // -- Month
    '<ANY>'                                                                // -- Date
                    as %MasterKey                                From Sales ;
```

Visits and VisitDetails are joined and this new table has date, physician, sales representative and product as keys. Hence, the corresponding Load statement will be

```
Visits:
Load 1 as Calls,
    Applymap('PhysicianID_to_TerritoryID', PhysicianID, null())  & '|' &    // -- TerritoryID
    ProductID                                                    & '|' &    // -- ProductID
    '<ANY>'                                                      & '|' &    // -- PackageID
    SalesRepID                                                   & '|' &    // -- SalesrepID
    PhysicianID                                                  & '|' &    // -- PhysicianID
    Num(MonthStart(Date))                                        & '|' &    // -- Month
    Num(Floor(Date))                                                       // -- Date
                        as %MasterKey                            From Visits ;
```

The third fact table is the table with the target numbers. This table has month, physician, sales representative and product as keys. Hence, the corresponding Load statement will be

```
Targets:
Load Target,
    Applymap('PhysicianID_to_TerritoryID', PhysicianID, null())  & '|' &    // -- TerritoryID
    ProductID                                                    & '|' &    // -- ProductID
    '<ANY>'                                                      & '|' &    // -- PackageID
    SalesRepID                                                   & '|' &    // -- SalesrepID
    PhysicianID                                                  & '|' &    // -- PhysicianID
    Num(Month)                                                   & '|' &    // -- Month
    '<ANY>'                                                                // -- Date
                        as %MasterKey                            From Targets ;
```

Master link table

Finally there is the master link table. It should contain only the keys that link it to the dimensional link tables and the master key that links it to the fact tables. No other fields should be loaded in this table.

Potentially this table can become very large, so it is important that is loaded in a way so that the number of records is minimized. Therefore I load only the master keys that exist in the fact tables. But as a consequence, three consecutive Load statements are needed to create the link table, one for each fact table.

Further, the link table should be loaded with the distinct clause.

The Load statement loading the records from the sales table hence becomes:

```
[Master Link Table]:
Load distinct
    Applymap('PhysicianID_to_TerritoryID', PhysicianID, null())  as      %TerritoryID,
    Applymap('PackageID_to_ProductID', PackageID,null())         & '|' &
    PackageID                                                    as      %ProductID,
    '<ANY>'                                                      as      %SalesRepID,
    PhysicianID                                                  as      %PhysicianID,
    Num(Month)                                                   & '|' &
    '<ANY>'                                                      as      %Date,

    Applymap('PhysicianID_to_TerritoryID', PhysicianID, null())  & '|' &  // -- TerritoryID
    Applymap('PackageID_to_ProductID', PackageID,null())         & '|' &  // -- ProductID
    PackageID                                                    & '|' &  // -- PackageID
    '<ANY>'                                                      & '|' &  // -- SalesrepID
    PhysicianID                                                  & '|' &  // -- PhysicianID
    Num(Month)                                                   & '|' &  // -- Month
    '<ANY>'                                                               // -- Date
                    as %MasterKey                                From Sales ;
```

There are three things to note here:

1) The definition of the master key is identical to the definition used when loading the sales fact table.
2) The Load statement has two parts; one where the dimensional generic keys are defined and one where the master key is defined.
3) The definitions of the dimensional generic keys (first part of the load) are identical to the sub-parts of master key (second part of the load).

Similar load statements are created for the other two fact tables.

This concludes the description of how to load multiple fact tables using generic keys and a fact table. The resulting data model can be seen in Figure 20.

Some practical tips

Hide the generic keys

If you name all your generic keys with e.g. a percent sign as first character, and then set this character as the “HidePrefix”, the generic keys will be hidden for the users.

```
Set HidePrefix    = % ;
```

Use variables for the generic symbols

To ensure that you only store the actual text representing the <ANY> symbol in one place, you could store it in a variable that you can access in your load statements. You can do the same with the <N/A> symbol and your concatenation symbol:

```
Set  ANY          = <ANY> ;
Set  NA           = <N/A> ;
Set  &            = & '|' & ;
```

The load statement then becomes much more compact, e.g.:

```
Load  ProductID  $(&)  '$(ANY)'                 as %MasterKey,
      ProductID,
      CustomerID                                From …;
```

Use Autonumber

The generic keys may become long since they are concatenated strings. One way to make them shorter and less memory consuming is to use autonumber, a function which assigns an integer instead of the long string. No information is lost.

```
Load  Autonumber( ProductID  $(&)  '$(ANY)'  )   as %MasterKey,
      ProductID,
      CustomerID                                From …;
```

Use Applymap

If you do not have a key that you need, you can use applymap as a lookup function. The below load statement uses the order ID as key to make a lookup for the correct customer ID.

```
OrderID_to_CustID:
Mapping Load OrderID, CustomerID                From Orders;

OrderDetails:
Load  *,
    ProductID  $(&) Applymap('OrderID_to_CustID', OrderID, 'NONE') as AuthID
                                                From OrderDetails ;
```

HIC