QlikView

Joins and Lookups

QlikView Technical Brief

Sept 2012, HIC

www.qlikview.com

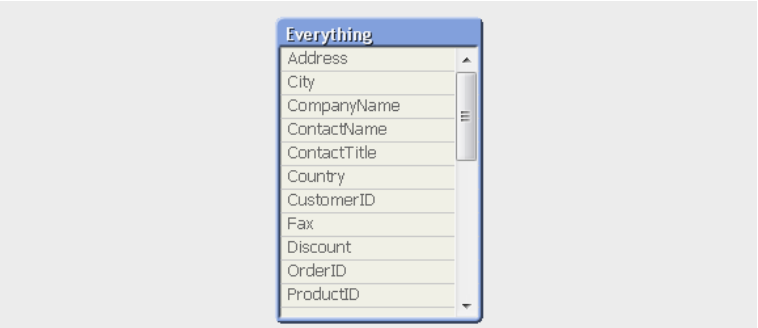
QlikView

Contents

Introduction

The QlikView internal logic allows a multi-table data model. Tables are linked by the naming of the keys so that QlikView “knows” how to evaluate the relations at run-time.

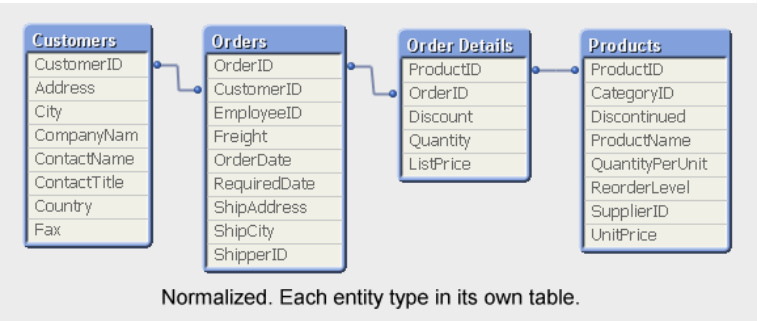
This is very different from many other BI or query tools where, when several tables are used, they are all are joined together into one table and then the output is used for analysis. The most obvious example of this difference is a simple SELECT statement. With it, you can use several tables as input and join them, but the output is always one single, denormalized table.



Completely denormalized. Everything in one table.

With QlikView, in contrast, you can have a multi-table relational data model that is evaluated in real-time. The associations are evaluated as joins at the moment when the user clicks in the application, making a selection.

In most cases it is advantageous to keep many tables in the data model; but sometimes it is better to join them in the script. This document will try to explain when you should do one or the other.



Normalized. Each entity type in its own table.

QlikView table associations

The data model defined in the QlikView script usually contains many tables that are linked with key fields. The links are implicit joins that are not yet made. In other words, they define where joins should be made when the user makes a selection. But since they are not yet evaluated, we prefer not to call them "joins". Instead, we call them "***associations"***.

When the associations are evaluated, they are evaluated as *natural* joins; i.e. QlikView requires the key field to have the same value in the two tables. It is hence not possible to use any other comparison, e.g. the following criterion cannot be used:

TableA.key >= TableB.Key

Further, the QlikView script can contain explicit joins. These are different and these we indeed call "***joins"***. These are executed when the script runs and the resulting table will constitute one table in the QlikView data model, unless it is dropped. See more about these in the next section.

Hence, the main difference between the associations and joins is that the associations are evaluated at demand; as the user makes selections. As opposed to the joins that are evaluated when the script runs.

A second difference is that a join is explicitly an inner join or an outer join (or a left or a right join) whereas the nature of an association depends on the situation. The association can be evaluated to a left join or a right join depending on where the user has made a selection. And with no selection, the association is always evaluated to a full outer join.

For example, if you have a table containing customers and a table containing orders, and you select some customers, then QlikView will make a left join: All the selected customers (left table) will be possible, even though they are not represented in the order table. But *only* the orders (right table) that are represented in the customer table are possible.

So when you make your data model in the QlikView script, you make the necessary data transformations, sometimes using explicit joins, and you make sure to name the keys so that the resulting tables are linked correctly. Also, you also make sure to name non-keys with unique names so that these do not link.

Joins in the script

Sometimes you need to join tables in the script, i.e. you use two or more tables as input, perform the join and get one table as output for the QlikView data model. It could be that you want to denormalize the data model to improve performance or that you, for some other reason, choose to put fields from two tables into one.

When you join two tables, there is always a possibility that the number of records change. In most cases, this is not a problem – rather, it is exactly what you want.

One example could be that you have a table containing order headers and one containing order details. Each header can have several order lines in the order details table. When these two tables are joined, the resulting table should have the same number of records as the order details table: you have a one-to-many relationship so an order can have several order lines, but an order line cannot belong to several order headers. The only problem you encounter is that you can no longer sum a number that resided in the order header table.

If you join over a many-to-many relationship, the new number of records will possibly be larger than the number of records of any of the constituting tables.

In some cases, you get a change of number of records when you don't expect it and this may cause problems. I have often seen this in real life and it usually happens when you join what you think is a one-to-one relationship, when it in fact is a one-to-many relationship. An example is when you have product attributes in another table than the master product table. Then one product can potentially have several attributes, and a join may cause QlikView to calculate some numbers incorrectly. Or rather – QlikView calculates the numbers correctly, but the result is not what you expect.

There are two ways to perform joins in the QlikView script:

The external join – the SQL SELECT join

A join can be defined inside a SELECT statement, which in the script execution is sent as a string to the connector, usually a relational database management system (RDBMS) using the ODBC or OLE DB connection. Then QlikView waits for an answer. In other words: the SQL join is performed on the DB system. This is sometimes more efficient and more robust than if you let QlikView manage it. This is especially true if you have large tables; then you should make the join inside the SELECT statement.

A join inside a SELECT statement is often versatile: you can make any type of join that the database allows, e.g. also joins that are not natural:

```
TableA.key >= TableB.Key
```

One consequence of letting the RDBM system manage the join is that the syntax of the SELECT statement may differ from database to database. QlikView does not interpret the SELECT statement. Instead it is evaluated by the RDBMS.

Examples of SELECT statements joining tables:

```
SELECT Customers.FirstName, Customers.LastName, SUM(Sales.SaleAmount) AS
SalesPerCustomer FROM Customers
LEFT JOIN Sales ON Customers.CustomerID = Sales.CustomerID
GROUP BY Customers.FirstName, Customers.LastName;
```

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers, orders WHERE suppliers.supplier_id = orders.supplier_id;
```

The QlikView join – the Join prefix

The Join prefix in the QlikView script joins the loaded table with one that has been loaded previously in the script run. It is performed by QlikView itself. Just as the associations, QlikView joins are natural joins based on that the two key fields are named the same. If there is no common key, the Cartesian product of the two tables will be generated.

The QlikView join is fast but will need a lot of primary memory. So if the tables are large, the performance may become poor. For smaller tables it is however the best alternative.

The Join prefix can be put in front of a Load statement or in front of a SQL SELECT statement. By default, an outer join is made. But it is also possible to make an inner, left or right join.

Examples of statements using the Join prefix:

```
Join Load … ;
```

```
Left Join (SomePreviousTable) Load … ;
```

```
Inner Join SQL SELECT … ;
```

The Keep prefix

Similar to the Join prefix is the Keep prefix. It works exactly the same way as a join, but it does ***not*** merge the two tables. Instead it keeps them as two separate tables. However, it performs the same type of comparison as a join and then purges the same records as it would have purged, if it had been a join. An inner keep hence keeps only the records where a common key value exists in both tables. It is hence a very good way of removing unnecessary data.

The Keep prefix can be put in front of a Load statement or in front of a SQL SELECT statement. The Keep prefix must be preceded by one of the keywords Inner, Left or Right to make an inner, left or right join.

Examples of statements using the Keep prefix:

```
Left Keep Load … ;

Right Keep (SomePreviousTable) Load … ;

Inner Keep SQL SELECT … ;
```

A common case where you want to use a Keep prefix is when you have loaded a subset of the transactions, e.g. all orders for just one day or one region, and want to load only the corresponding data sets of the master tables:

```
OrderDetails:
Load * From OrderDetails
  Where <some criterion>;

Products:
Left Keep (OrderDetails) Load * From Products;
```

This will reduce the number of records in the product table to fit with what has been loaded in the order details.

Lookup functions

Similar to the joins are the lookup functions. These functions fetch information from another table or look up a specific value in another table and return this.

One case is a mapping or a translation: You could for instance have a product ID and you want to translate that into a specific product attribute. In other words – you never expect more than one value for the specific product ID. If there are several possible values in the lookup table, the lookup function will just take the first one found.

Another case is when you want to load all records from one table, but exclude the records that have an ID listed in another table.

If you want to solve these types of problem using SQL, you would most likely use joins. However, in QlikView, *you should avoid joins* in these situations. Instead you should use one of the lookup functions; Applymap() is usually the best choice. This way you get a faster script execution and you ensure that the number of records do not change when they shouldn't.

If you do expect several values for each of the input values, you should not use a lookup function to solve the problem.

The Lookup() function

The most obvious lookup function is the function with the same name. With it, you can look into another previously loaded table and retrieve a specific field value.

For example, you may while loading a table containing order data want to retrieve the product category for the product found in the order data. You have the product ID in the order data table, but the product category is found in the product table.

Then you could solve this problem by using the Lookup() function:

```
Lookup ('ProductCategory', 'ProductID', ProductID, 'ProductTable')
```

The Lookup() function will then return the value of the field ProductCategory (first parameter) in the table ProductTable (fourth parameter) from the record where the field ProductID (second parameter) has the same value as the field ProductID in the order data table (third parameter).

Note that references to fields in *other* tables must be enclosed by single quotes, whereas references to fields in the same table must *not* be enclosed by single quotes. In this hypothetic example, the Lookup() function is used inside the Load statement of the order data, so fields in the product table must be enclosed by single quotes.

The advantage with the Lookup() function is that it is flexible and can access any previously loaded table. The drawback is that it is slow compared to the Applymap() function.

The Applymap() function

My preferred lookup function is the Applymap() function. With it you can make any translation or mapping – but you need to define the translation in a mapping table before you can use it. Applymap() uses the mapping table as lookup table and returns the appropriate translation.

If we once again look at the case of getting the product category into the order data table, a solution using Applymap() would be the following:

```
Map_ProductID_2_Category:
Mapping Load ProductID, ProductCategory From ProductTable ;

OrderData:
Load *,
  Applymap('Map_ProductID_2_Category',ProductID) as ProductCategory
From OrderData ;
```

The mapping table may only have two columns. How these are named is irrelevant: The first one is always the value from which to translate and the second one is always the value to translate into. The mapping table is discarded at the end of the script execution.

When you use the Applymap() function, you need to enclose the table name (first parameter) with single quotes. Also, I recommend that you use a third parameter to define what value the function should return when the product ID is not found in the mapping table. You may want to use the null() function or the string 'Missing'.

The advantage with the Applymap() function is that it is very fast compared to the Lookup() function. The drawback is that you need to prepare a dedicated mapping table before you can use it.

The Exists() function

The Exists() function is really not a lookup function, but it can be used for similar purposes. It doesn't return a specific value, but it can tell you whether a value has been previously loaded or not.

If you for instance want to load all products from the standard product table, but exclude some that are listed in another table, you can do it with the Exists() function:

```
DiscontinuedProducts:
Load
  ProductID as DiscontinuedProductID
From DiscontinuedProducts;

Products:
Load * From Products
  Where not Exists (DiscontinuedProductID, ProductID);
```

Another case when you may want to use the Exists() function is when you have loaded a subset of the transactions, e.g. all orders for just one day or one region, and want to load only the corresponding data sets of the master tables:

```
OrderDetails:
Load * From OrderDetails
  Where <some criterion>;

Products:
Load * From Products
  Where Exists (Products);
```

This will reduce the number of records in the product table to fit with what has been loaded in the order details. The same reduction can be made using a Keep prefix.

The Peek() function

The Peek() function is similar to the Lookup() function, but instead of addressing the other table using a field value, the peek function needs a row number.

```
Peek ('ProductCategory', -1, 'ProductTable')
```

Although the Peek() function is a lookup function, its use for solving similar problems is limited since you need the row number to get the desired value, and this is rarely the case.

Data model optimization

When creating the QlikView data model, you have a choice of loading several tables as several entities or joining some of them together when the script runs. Joining in the script means that the result of a join is stored in the QlikView data model as one single table.

So what should you do? Is it better to keep the data model normalized (many tables) or is it better to de-normalize (fewer tables)? My view is that it usually is better to keep the data model as normalized as possible. A normalized model has many advantages:

- **It is memory efficient.**
  A normalized solution is, by definition, the data model that uses least memory.
- **It is CPU efficient.**
  In most cases, QlikView calculations in a normalized model are as efficient (or only marginally slower) as in a denormalized model. In some cases a normalized model is faster.
- **It is easier to understand and manage.**
  It should be possible for other developers to read your script: A simple script with as few transformations as possible is a script that is easy for other developers to understand and maintain. A script containing many joins is hard to read and understand.
- **It minimizes the risk for incorrect calculations.**
  Joins potentially change the number of records in the tables, which means that a normal Sum() or Count() function cannot always be used – they would return an incorrect result. You may counter that there is always a way to write a correct formula, but my point is that it should also be easy. Expressions in server objects will be written by users that do not have special knowledge about the data model in the app.

But it is not a clear-cut case.

Often there is a trade-off between memory efficiency and CPU efficiency. In other words, there are cases where you can decrease response time by letting the data model use more memory – where performance will be considerably better if you make the join in the script.

One such case is if you have chart expressions containing fields from different tables. If the aggregation is a simple Sum(X), then QlikView will loop over the records of the table where the field X is found, summing X.

But if the expression is Sum (X*Y) and X and Y are found in different tables, then QlikView has to perform the join in memory generating a virtual table over which the summation will be made. This can be both memory and CPU demanding. In such a case you might get a better performance if you have made the join in the script, especially if the fields are far from each other in the data model. But as long as the dimension tables are relatively small, the performance difference is usually only marginal.

A special situation is if there are several large tables linked to each other, e.g., an Order header table and an Order body table. In such a case you often get a better performance when you join than when you don't.

But think before you join - when you join, you lose the ability to use Sum() and Count() on some fields - in the above case, the fields in the header table. You can still use Count(distinct …), although this aggregation becomes slower on a larger table than on a smaller.

If you can solve the problem using a lookup function instead of a join, I strongly recommend that you do so.

Bottom line is that you'll have to weigh pros and cons. *Don't join if you can avoid it.* If performance is important and you experience a noticeable improvement when you join, then you probably should join. But ask yourself what the implications are. Is the script still manageable? Can a user understand how the formula should be written? Would it be better to put this transformation in the data warehouse or in the metadata layer?

The best join is often the one that never is made. Often – but not always.

HIC