



IntervalMatch and Slowly Changing Dimensions

QlikView Technical Brief

June 03, 2013, HIC

www.qlikview.com

Contents

Contents	2
Introduction	3
Matching numbers against intervals	3
Method 1: Using the IntervalMatch prefix	3
Method 2: Using a While loop creating enumerable values	5
Method 3: Using a join made with the Join prefix	6
Method 4: Using a join made with a While loop	6
Simplifying the data model.....	7
Removing the bridge table.....	7
Removing the synthetic key.....	7
Open and closed intervals	9
Creating a Date Interval from a Single Date	10
Slowly changing dimensions.....	12
Creating the bridge table	13
Joining the bridge table onto the transaction table.....	15
Using a While loop and Applymap.....	16
Models with multiple interval tables	17
Interval Partitioning	17
Two interval tables mapped against a common dimension ID and a common time line	19
Two-level slowly changing dimension	20

Introduction

A common problem in business intelligence is when you want to link a number to an interval. It could be that you have a date in one table and an interval – a “From” date and a “To” date – in another table, and you want to link the two tables.

Sometimes you have one or several keys in addition to the number. One example is if you have a salesperson that belongs to a department for a limited period of time. In such a case you will have a salesperson table containing not only salesperson data, but also the department key and a date interval that defines the department affiliation. This table should now be linked to the transaction table, where you have the date and the salesperson key. This is called a *Slowly Changing Dimension* and is described below in a separate section.

In SQL, you would probably join the tables using a BETWEEN clause when comparing the number with the intervals. But how do you solve this in QlikView, where you want to keep the data in a multi-table data model and avoid joins?

This technical brief is about how to solve such problems in QlikView.

Matching numbers against intervals

The most basic interval match is when you have a list of numbers or dates in one table – below called events – and a list of intervals in a second table. The goal is to link the two tables.



In the general case, this is a many-to-many relationship, i.e. an interval can have many dates belonging to it and a date can belong to many intervals. To solve this, you need to create a bridge table between the two original tables. There are several ways to do this.

Method 1: Using the IntervalMatch prefix

The by far simplest way to solve this problem in QlikView is to use the IntervalMatch. It is a prefix that can be put in front of either a Load or a SELECT statement. The Load/SELECT statement needs to contain two fields only: the “From” and the “To” fields defining the intervals. The IntervalMatch prefix will then generate all combinations between the loaded intervals and a previously loaded numeric field, specified as parameter to the prefix.

Typically, you would first load the table with the individual numbers (The Events), then the table with the Intervals, and finally an IntervalMatch that creates a third table that bridges the two first tables.

Events:

```
Load EventDate, ... From Events;
```

Intervals:

```
Load IntervalBegin, IntervalEnd, ... From Intervals;
```

BridgeTable:

```
IntervalMatch (EventDate)
```

```
Load distinct IntervalBegin, IntervalEnd Resident Intervals;
```

The resulting data model contains three tables:

- The Events table that contains exactly one record per event.
- The Intervals table that contains exactly one record per interval.
- The bridge table that contains exactly one record per combination of event and interval, and that links the two previous tables.



Note again that an event may belong to *several intervals* if the intervals are overlapping. And an interval can of course have several events belonging to it.

This data model is optimal, in the sense that it is normalized and compact. The Events table and the Intervals table are both unchanged and contain exactly the right numbers of records. All QlikView calculations operating on these tables e.g. Count(EventID) will work and will be evaluated correctly. This means that you should **not** – if you have a many-to-many relationship – join the bridge table onto one of the original tables. Joining it onto another table may even cause QlikView to calculate aggregations incorrectly, since the join will change the number of records in a table.

Further, the data model contains a composite key (the IntervalBegin and IntervalEnd fields) which will manifest itself as a QlikView synthetic key. But have no fear. This synthetic key **should** be there. It is **not** dangerous. It does **not** use an excessive amount of memory. You do **not** need to remove it.

The advantage of this method is that it is easy and fast to implement and it executes very fast. However, it has a couple of drawbacks: First, it is not possible to have additional fields in the bridge table. Secondly, it considers all intervals as closed and cannot handle open and half-open intervals.

Method 2: Using a While loop creating enumerable values

You can often achieve almost the same bridge table using a while loop that creates enumerable values between the lower and upper bounds of the interval.

If you have a primary key for the intervals – the IntervalID – the only difference in the script will be how the bridge table is created:

```
BridgeTable:
Load distinct * Where Exists(EventDate);
Load IntervalBegin + IterNo() - 1 as EventDate, IntervalID
Resident Intervals
While IntervalBegin + IterNo() - 1 <= IntervalEnd ;
```



If you don't have a primary key in the interval table, you can create one using the following definition:

```
RecNo() as IntervalID,
```

This solution has the advantage that other fields can be included in the bridge table, e.g. the primary key from the interval table, so that you can use this instead of the composite key in the above example.

It however has the drawback that it can only handle enumerable values, e.g. integers and dates.

Method 3: Using a join made with the Join prefix

A third way to solve this problem is to use the join prefix to create all combinations of the number and the intervals and then, in a second pass, filter out the relevant ones:

```

Tmp_BridgeTable:
Load distinct EventDate Resident Events ;
Join
Load IntervalID, IntervalBegin, IntervalEnd Resident Intervals ;

BridgeTable:
Load distinct EventDate, IntervalID
  Resident Tmp_BridgeTable
  Where IntervalBegin <= EventDate and EventDate < IntervalEnd ;

Drop Table Tmp_BridgeTable ;

```

This solution has the advantage that other fields can be included in the bridge table, e.g. the primary keys from the both the event table and the interval table, so that you can use these for the links to the initial tables. Further, the method can handle both open and closed intervals (see the relational operators in the where clause).

However, it does have one major drawback: The join is very memory consuming, so this method should not be used for large data amounts. In fact, method 4 below is often a better solution. It has all the mentioned advantages but uses less memory.

Bottom line: Don't use this method unless you have very small tables or you have an additional key between the tables (i.e. a slowly changing dimension. See further below.)

Method 4: Using a join made with a While loop

A fourth way to solve this problem is to use a while loop with a Peek() function to create the bridge table. The idea is to make a join "manually" by making a lookup in each loop of the while clause.

The first (bottom) load statement loads data from the interval table and has a while loop that loads all records from the Events table. Effectively it creates the Cartesian product between the two tables. The preceding load then filters out the relevant records:

```

BridgeTable:
Load distinct IntervalID, EventDate
  Where IntervalBegin <= EventDate and EventDate < IntervalEnd;
Load IntervalID, IntervalBegin, IntervalEnd,
  Peek('EventDate', IterNo() - 1, 'Events') as EventDate
  Resident Intervals
  While IterNo() <= NoOfRows('Events') ;

```

Just as the method using the Join prefix, this solution has the advantage that other fields can be included in the bridge table. Also, the method can handle both open and closed intervals.

It however has the drawback that it is slower than the two initial methods.

Simplifying the data model

The above three-table data model is a general one: Normalized with each entity in its own table. And this is the way you should think of the different entities in your data modeling work – at least initially.

In real life there could however be good reasons to make the data model simpler, and here are two common changes that you may consider:

Removing the bridge table

In the general case, the solution with three tables is the best one, because it allows for a many-to-many relationship between intervals and events. But a very common situation is that you know that an event can only belong to one single interval. In such a case, the bridge table is really not necessary: The interval ID can be stored directly in the event table. There are several ways to achieve this, but the most common is to join the bridge table with the event table.

A word of caution: If you join the bridge table with the transaction table in a situation where an event belongs to several intervals, you will change the number of records in the event table and QlikView may make incorrect calculations: Some events will be counted twice.

Bottom line: Double check that you really have a many-to-one relationship and not a many-to-many relationship.

Removing the synthetic key

The synthetic key may look like a wart on an otherwise beautiful data model. But I can only re-iterate that a synthetic key like the one you get from using IntervalMatch is harmless, and that you do **not** need to remove it. Synthetic keys need to be removed only if one of the constituent fields shouldn't be a key, or if you have a synthetic key created from another synthetic key.

On the other hand, it is fairly straightforward to remove the synthetic key: Just create a composite key manually from the constituent fields and use this instead:

```
IntervalBegin & '|' & IntervalEnd as IntervalID,
```

or slightly more advanced:

```
Autonumber(Num(IntervalBegin) & '|' & Num(IntervalEnd)) as IntervalID,
```

The Num() functions format the dates to integers, which means that the created strings are independent of the date format. If this is of no concern, the Num() function calls are not needed.

The Autonumber() function converts the string to an integer that takes a lot less memory space. If this is of no concern, the Autonumber() function call is not needed.

You may need to run through the bridge table a second pass to achieve this solution, but since it normally is a fairly small table, this should be no problem.

The script using method one above then becomes

```

Events:
Load EventDate, ... From Events;

Intervals:
Load IntervalBegin, IntervalEnd, ...,
    Autonumber(Num(IntervalBegin) & '|' & Num(IntervalEnd)) as IntervalID
From Intervals;

BridgeTable:
IntervalMatch (EventDate)
Load distinct IntervalBegin, IntervalEnd Resident Intervals;

Join (Events)
Load EventDate, Autonumber(Num(IntervalBegin) & '|' & Num(IntervalEnd)) as IntervalID
Resident BridgeTable;

Drop Table BridgeTable;
    
```



Finally, I have seen cases where the developer joins the bridge table onto the interval table, instead of the events table, to get rid of the synthetic key. This is not a good idea: Since there usually are many more events than intervals, the new interval table grows dramatically in size and the whole point of having the data in two tables is lost.

Bottom line: In a simple interval match where you don't have additional keys, you do not need to remove the synthetic key. However, if you know that an event only belongs to one interval, you can remove both the bridge table and the synthetic key by moving the interval ID into the events table.

Open and closed intervals

Whether an interval is open or closed is determined by the endpoints – whether these are included in the interval or not.

- 1) If the endpoints are included, it is a closed interval; $[a,b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$
- 2) If the endpoints are not included, it is an open interval; $]a,b[= \{x \in \mathbb{R} \mid a < x < b\}$
- 3) If one endpoint is included, it is a half-open interval; $[a,b[= \{x \in \mathbb{R} \mid a \leq x < b\}$

If you have a case where the intervals are overlapping and a number can belong to more than one interval, you usually want to use closed intervals.

However, in some cases you do not want overlapping intervals – you want a number to belong to *one interval only*. Hence, you will get a problem if one and the same point is the end of one interval and at the same time the beginning of next. A number with exactly this value will be attributed to both intervals. Hence, you want half-open intervals.

A practical solution to this problem is to subtract a very small amount from the end value of all intervals, thus creating closed, but non-overlapping intervals. If your numbers are dates, the simplest way to do this is to use the function DayEnd() which returns the last millisecond of the day:

Intervals:

```
Load ..., DayEnd(IntervalEnd - 1) as IntervalEnd From Intervals ;
```

But you can also subtract a small amount manually. If you do, make sure the subtracted amount isn't too small since the operation will be rounded to 52 significant binary digits (14 decimal digits). If you use a too small amount, the difference will not be significant and you will be back using the original number.

If your numbers are dates, you can subtract an amount as small as 2^{-37} ($=\text{Pow}(2,-37) \approx 0.000000000007$) which is slightly less than a microsecond.

```
Let vEpsilon = Pow(2,-37);
```

Intervals:

```
Load ..., Date(IntervalEnd - $(#vEpsilon)) as IntervalEnd From Intervals ;
```

If the subtracted amount is smaller than 2^{-27} , the time difference will not be visible in the date and timestamp formats. The reason is that the date and time functions display the formatted date or time for the nearest millisecond while keeping a numeric value that has higher precision. I.e. the Date() function will display the original date. If you want the time difference to be visible, you can at most subtract 2^{-27} ($=\text{Pow}(2,-27) \approx 0.000000007$) which is slightly less than a millisecond.

It is also possible to use the Dual() function to create the desired display value:

```
Dual(IntervalEnd, IntervalEnd - $(#vEpsilon)) as IntervalEnd
```

Creating a Date Interval from a Single Date

Sometimes the intervals are not stored explicitly with a beginning and an end. Instead they are implied by only one field – the change timestamp.

It could be like in the table to the right where you have currency rates for multiple currencies: Each currency rate change is on its own row; each with a new conversion rate. Also, the table could contain rows with empty dates corresponding to the initial conversion rate, before the first change was made.

Currency	Change Date	Rate
EUR		8.59
EUR	1/28/2013	8.69
EUR	2/15/2013	8.45
USD		6.50
USD	1/10/2013	6.56
USD	2/3/2013	6.30

This table defines a set of non-overlapping intervals, where the begin data is called “Change Date” and the end date is defined by the beginning of the following interval. But since the end date isn’t explicitly stored in a column of its own, we need to create such a column, so that the new table will become a list of intervals. Here’s how you do it:

- 1) Determine which time range you want to work with. The beginning of the range must be before the first date in data and the end of the range must be after the last.
- 2) Load the source data, but change empty dates to the beginning of the range defined in the previous bullet. The change date should be loaded as “From Date”.
- 3) Sort the table first according to Currency, then according to the “From Date” descending so that you have the latest dates on top.
- 4) Run a second pass through data where you calculate the “To Date”. If the current record has a different currency from the previous record, then it is the first record of a new currency (but its last interval), so you should use the end of the range defined in bullet 1. If it is the same Currency, you should take the “From Date” from the previous record, subtract a small amount of time, and use this value as “To Date” in the current record.

Currency	From Date	To Date	Rate
EUR	2/15/2013	vEndTime	8.45
EUR	1/28/2013	2/14/2013 23:59:59	8.69
EUR	vBeginTime	1/27/2013 23:59:59	8.59
USD	2/3/2013	vEndTime	6.30
USD	1/10/2013	2/2/2013 23:59:59	6.56
USD	vBeginTime	1/9/2013 23:59:59	6.50

In the QlikView script, it could look like this:

```

Let vBeginTime = Num('1/1/2013');
Let vEndTime   = Num(Now());
Let vEpsilon   = Pow(2,-27);

Tmp_Rates:
Load Currency, Rate,
    Date(If(IsNum([Change Date]), [Change Date], $(#vBeginTime))) as FromDate
From Rates ;

```

Rates:

```

Load Currency, Rate, FromDate,
  Date(If( Currency=Peek(Currency),
    Peek(FromDate) - $({#vEpsilon}),
    ${#vEndTime}
  )) as ToDate
Resident Tmp_Rates
Order By Currency, FromDate Desc;

Drop Table Tmp_Rates;

```

When this is done, you will have a table listing the intervals correctly. This table can subsequently be used in a comparison with an existing date using any of the above mentioned intervalmatch methods.

Currency	FromDate	ToDate	Rate
USD	1/1/2013	1/9/2013	6.50
USD	1/10/2013	2/2/2013	6.56
USD	2/3/2013	2/22/2013	6.30
EUR	1/1/2013	1/27/2013	8.59
EUR	1/28/2013	2/14/2013	8.69
EUR	2/15/2013	2/22/2013	8.45

Slowly changing dimensions

As one creates QlikView applications one sometimes encounters a data modeling problem where a dimensional attribute for a record varies over time. It could be that a salesperson changes department or a product is reclassified to belong to another class of products.

This problem is called **Slowly Changing Dimensions** and is a challenge for any Business Intelligence tool. Creating a query or a report with static dimensions is simple enough, but when a salesperson is transferred from one department to another, you will have to ask yourself how you want this change to be reflected in your sales report.

First of all, a changed attribute must be recorded in a way that the historical information is preserved. If the old value is lost when it is overwritten by the new attribute value, there is of course nothing QlikView can do to save the situation:

Before			After: Original Department is lost		
SPID	Name	Department	SPID	Name	Department
1	Bob	Dept B	1	Bob	Dept B
2	Cynthia	Dept C	2	Cynthia	Dept D

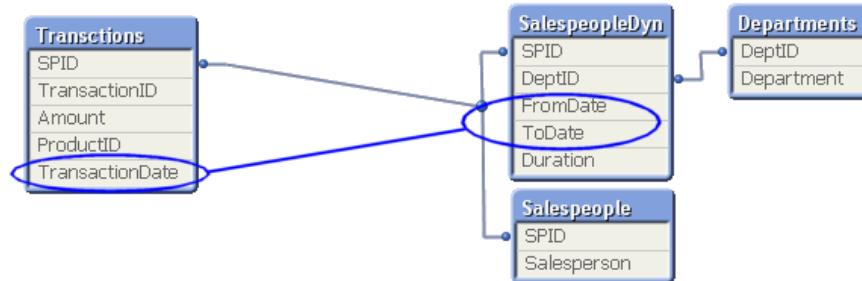
In such a case, the new attribute value will be used also for the old transactions and sales numbers will in some cases be attributed to the wrong department.

However, if the change has been recorded in a way so that historical data persists, then QlikView can show the changes very well. Normally, historical data are stored by adding a new record in the database for each new situation, with a change date that defines the beginning of the validity period.

However, it is not uncommon that there are two tables for each dimension, one with the history containing fields that may change and one with static data, similar to the following:

Dynamic dimension data			Static dimension data	
SPID	ChangeDate	Department	SPID	Name
1		Dept B	1	Bob
2		Dept C	2	Cynthia
2	12/31/2011	Dept D		

In such a case, we will have four tables that need to be linked correctly: A transaction table, a dynamic salesperson dimension, a static salesperson dimension and a department dimension. Also this is an intervalmatch - the transaction date needs to be matched against the intervals defined in the dynamic salesperson dimension.

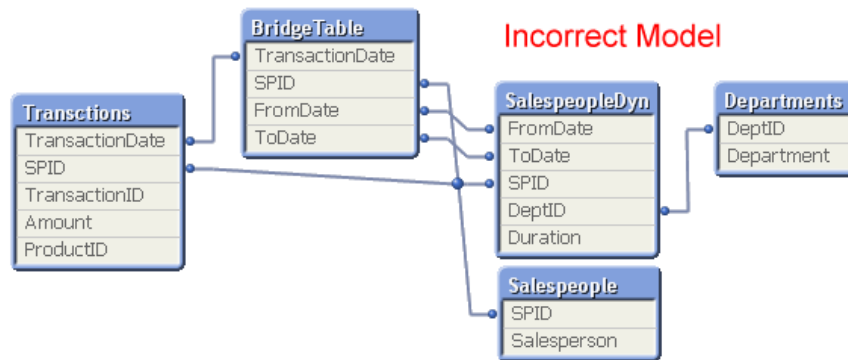


Creating the bridge table

Hence, we need to link the intervals with the transaction dates **and** at the same time link the salesperson ID (SPID). For this, we can use the method one above – the IntervalMatch prefix – in its extended syntax where it is possible to have additional keys:

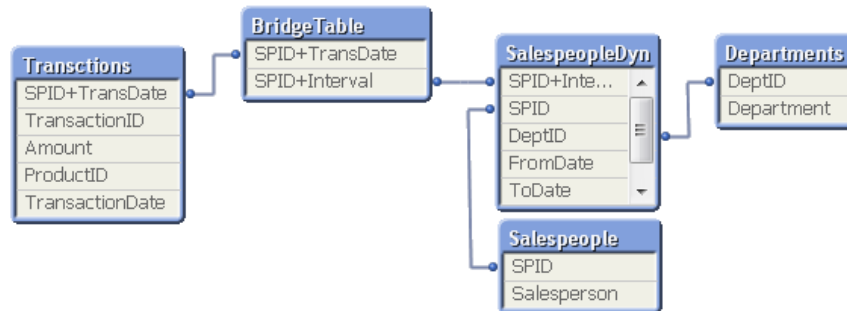
```
IntervalMatch (TransactionDate, SPID)
Load distinct FromDate, ToDate, SPID Resident SalespeopleDyn;
```

But just creating the bridge table and joining it together with the three other tables will not work since too many links will be created:



Instead, we must **create** the structure we want: We need to create a bridge table between the transaction table and the dimension tables. And it should be **the only link** between them. This means that the link from the transaction table to the bridge table should be a composite key consisting of *SPID* and *TransactionDate*.

It also means that the next link, the one from the bridge table to the dimension tables, should also be a composite key, but now consisting of *SPID*, *FromDate* and *ToDate*. Finally, the field *SPID* may only exist in the dimension tables and must hence be removed from the transaction table.



To achieve this, the bridge table is created in the following steps:

- 1) The dimension tables are loaded. In the dynamic salesperson dimension, a composite key is created from the salesperson ID and the interval dates.
- 2) The transaction table is loaded. The salesperson ID is loaded as a temporary field *TmpSPID*. In addition, a composite key is created from the salesperson ID and the date.
- 3) The intervalmatch is made, assigning an interval to each combination of sales person and transaction date. The resulting table is a temporary table: the first version of the bridge table and it will have four fields: *TransactionDate*, *FromDate*, *ToDate* and *TmpSPID*.
- 4) In a second pass over the temporary bridge table, these four fields are used to create two new fields that are composite keys: First the key that links to the transaction table and then the key that links to the dimension tables. The definitions of the composite keys must be the same in the bridge table as in the two tables in bullet 1 and 2.
- 5) Finally, the temporary bridge table and the temporary salesperson ID are dropped.

In the QlikView script, the main part of the script could be implemented like this:

SalespeopleDyn:

```
Load *,
    SPID & '|' & FromDate & '|' & ToDate as [SPID+Interval]
From SalespeopleDyn ;
```

Transactions:

```
Load TransactionDate, ProductID, Amount, SPID as TmpSPID,
    SPID & '|' & TransactionDate as [SPID+TransDate]
From Transactions ;
```

TmpBridgeTable:

```
IntervalMatch (TransactionDate,TmpSPID)
Load distinct FromDate, ToDate, SPID as TmpSPID
Resident SalespeopleDyn ;
```

```

BridgeTable:
Load
    TmpSPID & '|' & TransactionDate as [SPID+TransDate],
    TmpSPID & '|' & FromDate & '|' & ToDate as [SPID+Interval]
Resident Tmp_BridgeTable;

Drop Field TmpSPID;
Drop table TmpBridgeTable;

```

Using the above method to link a dynamic dimension to the transaction table, the problem with a Slowly Changing Dimension can be solved. All transactions will be connected to the appropriate department and QlikView will always show the correct numbers.

Joining the bridge table onto the transaction table

In most cases of slowly changing dimensions, a salesperson (or product, customer, etc.) can only belong to one department (or product group, region, etc.) at a time. In other words, the salesperson - interval relationship is a many-to-one relationship. If so, you should join the bridge table onto the transaction table. This join will not change the number of records in the transaction table.

A word of caution: I have more than once been in a situation where the application developer with great confidence says that it is a many-to-one relationship; e.g. that there can only be **one** product group per product. So we make the join - only to find out that the number of records in the transaction table changes. I.e. the application developer was wrong: in the data a product could in fact belong to more than one product group. Such a join may cause QlikView to make incorrect calculations. Bottom line: Double-check before you join.

With a join, the script changes only marginally: Bullets 1-3 are almost identical, with the small difference that the composite key with salesperson ID and date is no longer necessary.

Further, bridge table should not be stored as a separate table, but the Load is instead preceded by a join prefix. The salesman ID and the date are loaded instead of the composite key, so the join with the transaction table has the correct keys.

In the QlikView script, bullet 3-5 could be implemented like this:

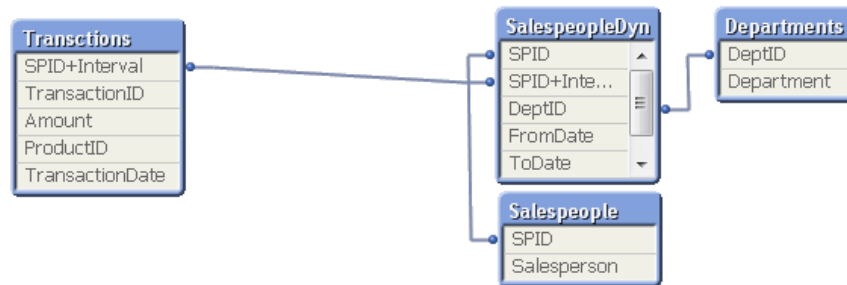
```

TmpBridgeTable:
IntervalMatch (TransactionDate,TmpSPID)
Load distinct FromDate, ToDate, SPID as TmpSPID
Resident SalespeopleDyn ;

Left Join (Transactions)
Load TmpSPID, TransactionDate,
    TmpSPID & '|' & FromDate & '|' & ToDate as [SPID+Interval]
Resident Tmp_BridgeTable;

```

Drop Field TmpSPID;
Drop table TmpBridgeTable;



Using a While loop and Applymap

It is also possible to solve the problem without the IntervalMatch prefix along the ideas of method two above described in the first section: Using a While loop. The idea is to generate all possible discrete values in the interval and then match these against the transaction table.

This can be combined with the Applymap function, so that the correct keys are loaded directly into the transaction table when this table is loaded. The dimension tables are loaded as described above, with the composite key made from the salesman ID and the intervals. Loading the transaction table could then look like the following:

// Create mapping table that maps key+date to key+interval.

BridgeTable:

Mapping Load distinct

SPID & '|' & Num(FromDate + IterNo() - 1) as [SPID+TransDate],

SPID & '|' & FromDate & '|' & ToDate as [SPID+Interval]

Resident SalespeopleDyn

While FromDate + IterNo() - 1 <= ToDate;

// Load transaction table using the above mapping table to create the correct key.

Transactions:

Load TransactionID, TransactionDate, ProductID, Amount,

Applymap('BridgeTable', SPID & '|' & Num(TransactionDate)) as [SPID+Interval]

From Transactions ;

Models with multiple interval tables

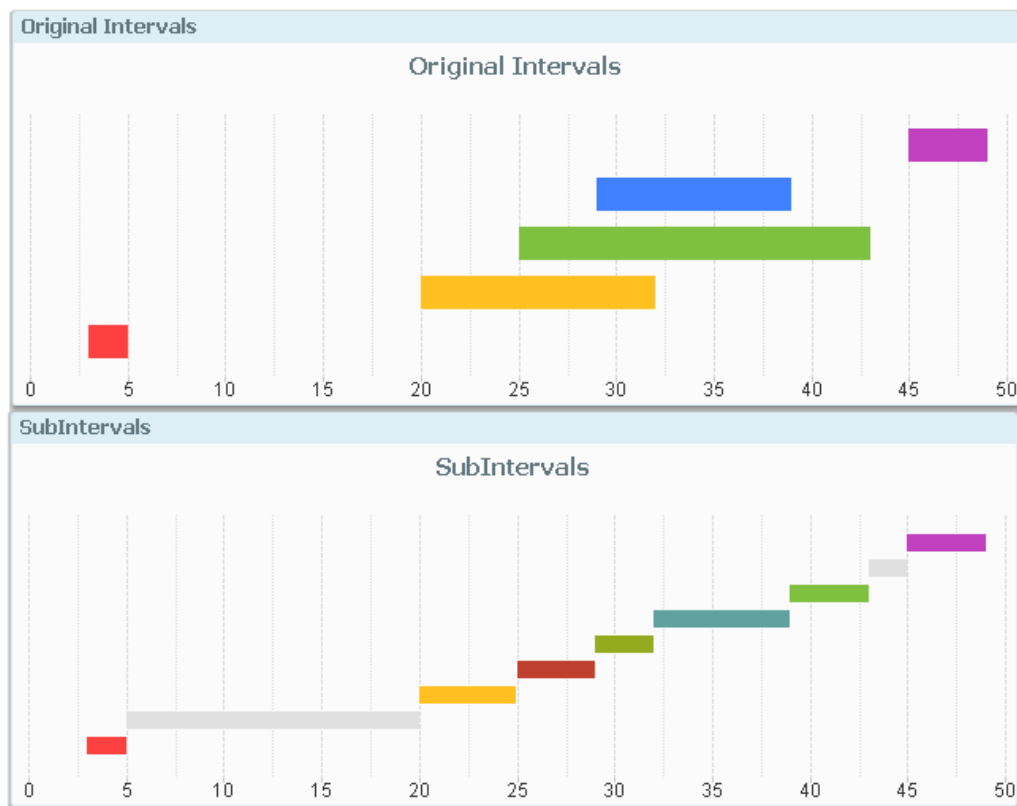
In more advanced data models you will encounter situations where you have several interval tables that need to be matched against each other or against a common date. In this section I will elaborate a little around these cases.

Interval Partitioning

In some cases you will need to transform a set of overlapping intervals into its most basic components: the unique sub-intervals. This is called partitioning.

The pictures below show an example. The upper graph shows five intervals that are partly overlapping. If you would compare these intervals to a set of events, you will get a data model where each event can belong to *several* intervals.

The lower graph shows the sub-intervals for the same data. If you instead would compare these sub-intervals to the same set of events, you will have a situation where you know that each event can only belong to *one* interval.



By finding the sub-intervals, you can in some situations simplify your data model.

Note that in the picture above, you don't have any additional dimension. But in real life, you usually have an additional dimensional key, e.g. salesperson ID, which makes it is a slowly changing dimension. In such a case, the intervals should only be compared *within* each salesperson and not between sales people.

If you have one table with overlapping intervals, you can do it this way:

- 1) Create a table containing all beginnings and ends of the original intervals, stored in one field. In addition, the table needs the dimensional key.
- 2) Convert this table to intervals as described in the section "Creating a Date Interval from a Single Date".

The script will look similar to the following, assuming that the dimensional key is SPID (salesperson ID).

```

Let vEpsilon = Pow(2,-27);
// Find all breakpoints of the intervals.
TempSubIntervals:
Load distinct SPID, Begin as SubFromDate Resident OriginalIntervals;
Load distinct SPID, End as SubFromDate Resident OriginalIntervals;

// Create intervals from single dates.
SubIntervals:
Load SPID, SubFromDate, RecNo() as SubIntervalID,
Dual(SubToDate, SubToDate - $(#vEpsilon)) as SubToDate
Where not IsNull(SubToDate) ;
Load SPID, SubFromDate,
If(SPID = Previous(SPID), Previous(SubFromDate)) as SubToDate
Resident TempSubIntervals
Order By SPID, SubFromDate Desc;

Drop Table TempSubIntervals;

```

With this script, you have created a new set of intervals that you can use as a replacement for the original intervals. Each subinterval can belong to several original intervals and each original interval can have several subintervals, so you may need to use an intervalmatch to connect the new and old intervals.

Two interval tables mapped against a common dimension ID and a common time line

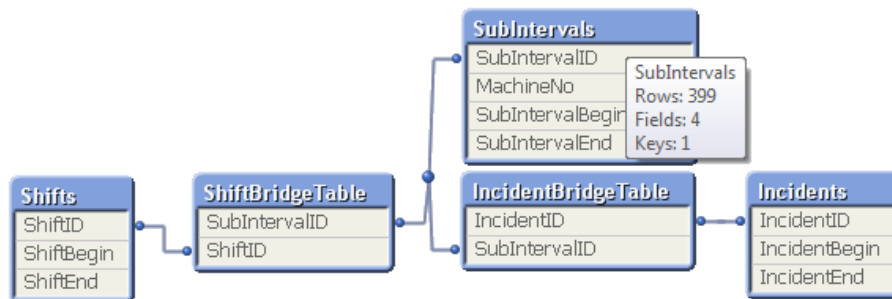
A case where you could use partitioning is when you have two tables with different entities, and both with intervals. An example could be that you have one table with incidents – events that have both a begin time and an end time – and another table with working the shift schedule of the working staff. Both tables should be mapped against the same time line and both tables also contain the ID of the production machine where the incident took place.

And the question is: which work shift managed which incident?

To solve this, you need to partition the intervals and map both original tables against a common sub-interval table.

One simple partition is to use discrete minutes as the common time line – it solves the problem. However, you will probably get too many combinations of minutes and machine IDs, so that the data amount will be too large. Further, it is only roughly right: The beginnings and the ends of the events and shifts need to be rounded to discrete minutes, so you lose the information about seconds.

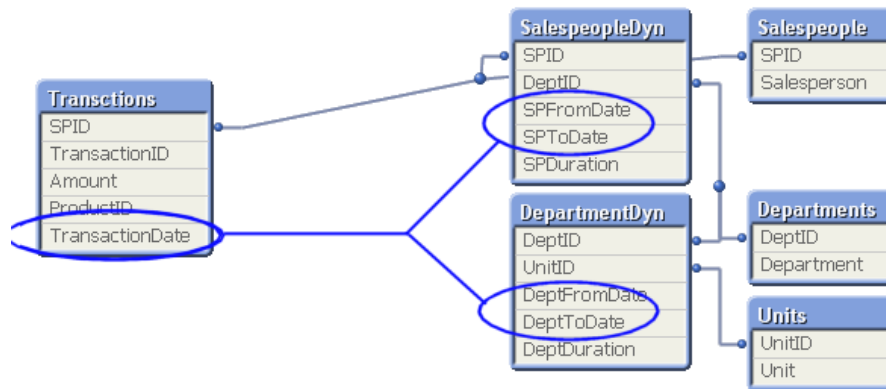
Then it might be better to make the partitioning properly and reduce the number of records in the sub-interval table. See how this can be done in the attached example.



Two-level slowly changing dimension

A second case where you could use partitioning is when you have a slowly changing dimension in two levels. Example:

- The salesperson can change department. Which department he/she belongs to is described in the dynamic salesperson table, which contains the department ID and the intervals for when the department was relevant.
- The department can change unit. Which unit it belongs to is described in the dynamic department table, which contains the unit ID and the intervals for when the unit was relevant.



In such a situation, you will need the date from the transaction table to determine not only the relevant department, but also the relevant unit. Partitioning the intervals the way it is described previously could be one way to solve this problem. But since the salesperson ID is missing in one of the two interval tables, we have to use another algorithm for the partitioning: Joining, using an algorithm which is very similar to method three described in the first section of this tech brief.

By joining the two interval tables on department ID into a temporary table, and in a second step check where the two interval tables have overlaps, it is possible to create the relevant set of subintervals. The script for creating the subintervals will look similar to the following:

```
// Load the dynamic department table.
DepartmentDyn:
Load ..., DeptID as TmpDeptID, DeptIntervalID
From DepartmentDyn;
```

```

// Load the dynamic salesperson table.
SalespeopleDyn:
Load ..., DeptID as TmpDeptID, SPID as TmpSPID, SPIntervalID
  From SalespeopleDyn;

// Join the two above tables into a temporary table using TmpDeptID as key.
TempSubIntervals:
Load distinct DeptIntervalID, TmpDeptID, UnitID, DeptFromDate, DeptToDate
  Resident DepartmentDyn;
Inner Join
Load distinct SPIntervalID, TmpSPID, TmpDeptID, SPFromDate, SPToDate
  Resident SalespeopleDyn;

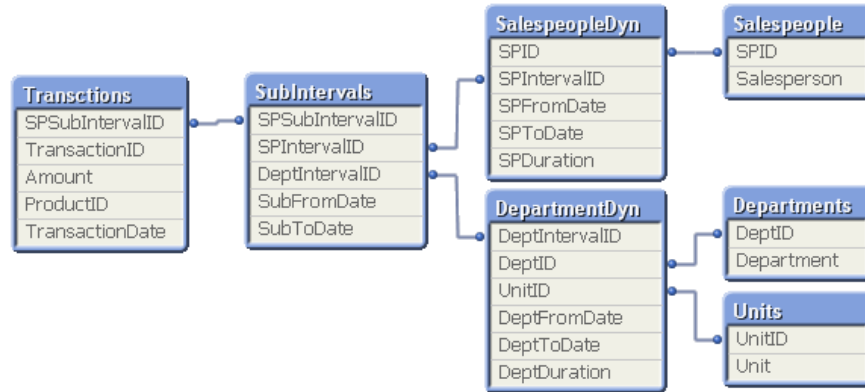
// Select only the records where the intervals overlap. Calculate new begin and end.
SubIntervals:
Load *,
  Autonumber(TmpSPID & '|' & SubFromDate & '|' & SubToDate) as SPSubIntervalID;
Load
  DeptIntervalID, SPIntervalID, TmpSPID,
  Date(RangeMax(SPFromDate,DeptFromDate)) as SubFromDate,
  Date(RangeMin(SPToDate,DeptToDate)) as SubToDate
  resident TempSubIntervals
  Where SPFromDate <= DeptToDate and DeptFromDate <= SPToDate ;

// Clean up.
Drop Field TmpSPID, TmpDeptID;
Drop Table TempSubIntervals;

```

The Where clause picks out the combinations where the two interval types have overlaps. The RangeMax and RangeMin functions determine the beginning and the end of the overlap.

Once the subintervals table has been created, it is straightforward to connect this new table to the transaction table using the method for slowly changing dimensions described previously. You will then get a data model similar to the following:



See more on how this can be done in the attached example.

HIC /Lund, Jun 03, 2013