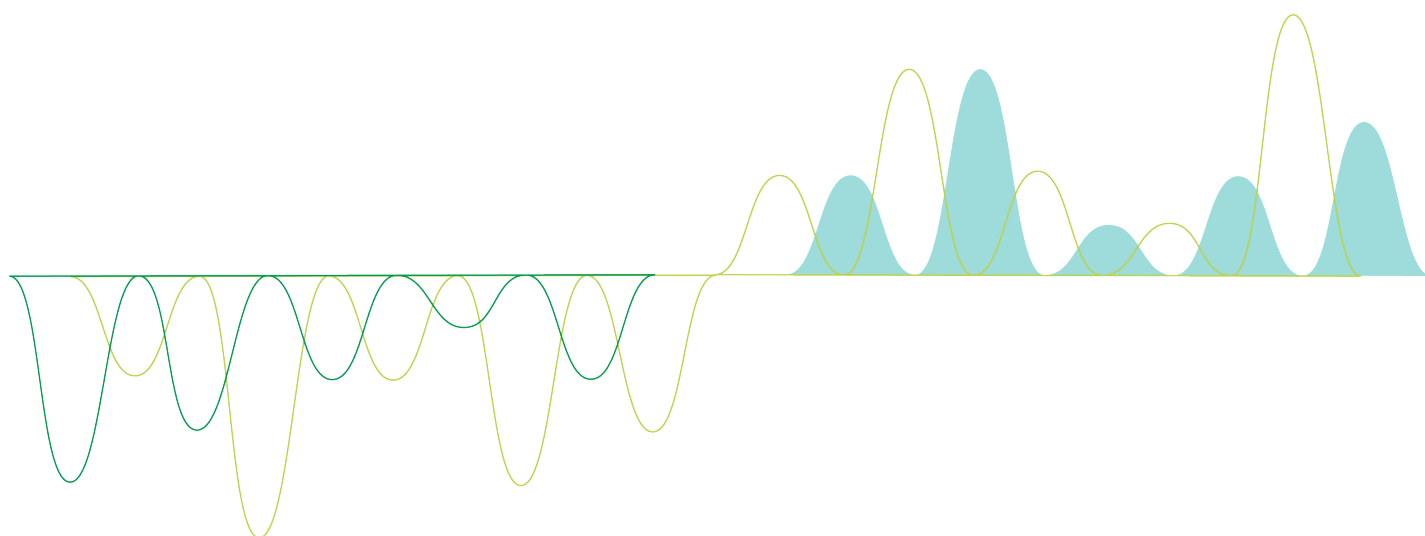


Tutorial - Next Steps in Scripting

Qlik Sense®

February 2019

Copyright © 1993-2019 QlikTech International AB. All rights reserved.



1 Welcome to this tutorial!	5
1.1 About this tutorial	5
1.2 Prerequisites	5
Creating a data connection in Qlik Sense Enterprise	6
Placing tutorial source files in Qlik Sense Desktop	6
1.3 Further reading and resources	6
2 Data loading in Qlik Sense	8
2.1 Creating a new app	8
2.2 Adding connections	8
File data connection	9
2.3 Editing the data load script	9
Accessing syntax help for commands and functions	9
Inserting a prepared test script	10
Searching and replacing text	10
Commenting in the script	11
Selecting all code	12
2.4 Debugging the load script	12
Debug toolbar	13
Output	13
3 Transforming data	14
3.1 Using the Crosstable prefix	14
Crosstable prefix	14
Clearing the memory cache	17
3.2 Combining tables with Join and Keep	17
Join	18
Using Join	18
Keep	21
Inner	21
Left	22
Right	23
3.3 Using inter-record functions	24
Peek	24
Previous	24
Exists	25
Using Peek() and Previous()	25
Using Exists()	29
3.4 Matching intervals and iterative loading	31
Using the IntervalMatch prefix	32
4 Data cleansing	39
4.1 Mapping tables	39
Rules:	39
4.2 Using Mapping	39
Mapping prefix	39
ApplyMap() function	40

MapSubstring() function	42
Map ... Using	44
Unmap	44
5 Handling hierarchical data	45
5.1 Hierarchy prefix	45
5.2 HierarchyBelongsTo prefix	46
Authorization	47
6 QVD files	49
6.1 Working with QVD files	49
6.2 Creating QVD files	50
Store	50
Buffer	51
6.3 Reading data from QVD files	52
6.4 Thank you!	52

1 Welcome to this tutorial!

Welcome to this tutorial, which will introduce you to more advanced scripting in Qlik Sense. You find the complete syntax reference in the online help and in the *Script syntax and chart functions*, available at help.qlik.com.

In Qlik Sense, scripting is mainly used to specify what data to load from your data sources. In this tutorial you will learn how to transform and manipulate data from databases and files using the data load editor.

1.1 About this tutorial

This tutorial guides you through some advanced steps of scripting, using the example files provided.

These are some of the subjects in this tutorial:

- Editing scripts
- Transforming data
- Data cleansing
- Hierarchical data
- Dollar-sign expansions

When you have completed the tutorial, you should have a fair understanding of some of the more advanced steps involved in scripting in Qlik Sense. A deeper understanding of scripting can be gained by taking a training course that is available from the Qlik website.

The screenshots in this tutorial are taken in Qlik Sense Enterprise. Some differences may occur if you are using Qlik Sense Desktop.


1.2 Prerequisites

To get the most out of this tutorial, we recommend that you have fulfilled the following prerequisites before you begin:

- Qlik Sense Desktop is installed or you have access to Qlik Sense Enterprise or access to Qlik Sense Cloud.
To install Qlik Sense Desktop, follow the instructions available at help.qlik.com.
- You are familiar with the basics of Qlik Sense.
That is, you know how to make selections in an app and how to interpret the results of your selection. You have also successfully loaded data into Qlik Sense. If you are not familiar with how to do these things, you can learn all about them in the tutorial *Tutorial - Building an App*, available at help.qlik.com.
- You have completed the *Tutorial - Scripting for Beginners* that is available at help.qlik.com.

Creating a data connection in Qlik Sense Enterprise

If you are using Qlik Sense Enterprise you need some help from your system administrator before you can start the tutorial. You might need help to access the hub and the *Tutorial source files* folder must be saved by the system administrator on the server machine. Also, before you can start loading the data files, your system administrator must prepare the data connection for you, by doing the following:

1. Save the *Tutorial source files* folder on the server machine.
2. Open the hub and open an unpublished app. If there are no unpublished apps, click **Create new app**. Give the app a name, click **Create** and then click **Open app**.
3. Click  and select **Data load editor**.
4. Click **Create new connection**.
5. Select **All files**.
6. Browse to the *Tutorial source files* folder stored on the server machine.
7. Type *Tutorial source files* in the **Name** field.
8. Click **Create**.

Make sure you have access rights to use the connection. The access rights for the data connection are managed from the Qlik Management Console (QMC):

- a. Select **Data connections** on the QMC start page.
- b. Select the *Tutorial source files* connection and click **Edit**.
- c. Select **Security rules** under **Associated items**.
- d. Either create a new rule or edit an existing rule to give you access to the connection.
- e. Edit the security rule for administrative access of the data connection and click **Apply**.

The connection is now prepared for you to use.

Placing tutorial source files in Qlik Sense Desktop

If you are using Qlik Sense Desktop, before you begin this tutorial, you need to place the *Tutorial source files* folder in the *Sense* folder. The folder *Tutorial source files* is included in the zip file and contains the data files.

Do the following:

1. Open the folder *Documents*. (It is sometimes called *My Documents*.) From there, the path is *Qlik\Sense*.
2. Place the *Tutorial source files* folder in the *Sense* folder.

1.3 Further reading and resources

Qlik offers a wide variety of resources when you want to learn more.

At help.qlik.com, you find the Qlik Sense online help and a number of downloadable guides.

If you visit www.qlik.com, you will find the following:

- Training, including free online courses
- Demo apps
- Qlik Community, where you will find discussion forums, blogs, and more

These are all valuable sources of information and are highly recommended.

2 Data loading in Qlik Sense

Qlik Sense uses a data load script, which is managed in the data load editor, to connect to and retrieve data from various data sources. In the script, the fields and tables to load are specified. It is also possible to manipulate the data structure by using script statements and expressions. It is also possible to load data into Qlik Sense using the data manager, but when you want to create, edit and run a data load script you use the data load editor.

During the data load, Qlik Sense identifies common fields from different tables (key fields) to associate the data. The resulting data structure of the data in the app can be monitored in the data model viewer. Changes to the data structure can be achieved by renaming fields to obtain different associations between tables.

After the data has been loaded into Qlik Sense, it is stored in the app. The app is the heart of the program's functionality and it is characterized by the unrestricted manner in which data is associated, its large number of possible dimensions, its speed of analysis and its compact size. The app is held in RAM when it is open.

Before we can start looking more closely at the script in the data load editor, you need to create an empty app and a couple of data connections to be able to load data into Qlik Sense.

The example files that you need for this tutorial are:

- *Product.xlsx*
- *Transactions.csv*
- *Employees.xlsx*
- *Salesman.xlsx*
- *Data.xlsx*
- *Events.txt*
- *Intervals.txt*
- *Winedistricts.txt*

2.1 Creating a new app

Throughout the tutorial you will be asked to create new apps to enable certain scripts to be run in isolation.

Do the following:

1. Open Qlik Sense.
2. Click **Create new app**.
3. Give the app a name, for example, *Advanced Scripting tutorial* and click **Create**.

2.2 Adding connections

In the **Data load editor** in Qlik Sense, under the section **Data connections**, you can save shortcuts to the data sources you use: databases, local files or remote files. **Data connections** lists the connections you have saved in alphabetical order. You can use the search box to narrow the list down to connections with a certain name or

type.

You can find more detailed descriptions of how to create data connections, and the supported file types, in the *Scripting for beginners* tutorial available on the Qlik Sense help site. The following sections are simply reminders of the procedures used in the Tutorial - Scripting for Beginners.



*Before you load data into your app for the first time, there is an option to use **Add data** to easily load data from files. However, in this tutorial we want to see the script, so we will use the data load editor.*

File data connection

Loading data from files, such as Microsoft Excel or any other supported file formats, is easily done by using the data selection dialog in the **data load editor**.

Do the following:

1. In the app *Advanced Scripting tutorial* you created in the previous section, open the **Data load editor**.
2. Click **Create new connection** and select **All files**.
3. Locate the tutorial folder you want to connect to and give it a name.
4. Click **Create**.

The folder connection is now complete and you are now ready to connect to the files and start selecting which data to load.

2.3 Editing the data load script

You write the script in the text editor of the **Data load editor**. Here you can make manual changes to the **LOAD** or **SELECT** statements you have generated when selecting data, and type new script.

The script, which must be written using the Qlik Sense script syntax, is color coded to make it easy to distinguish the different elements. Comments are highlighted in green, whereas Qlik Sense syntax keywords are highlighted in blue. Each script line is numbered.


There are a number of functions available in the editor to assist you in developing the load script, and they are described in this section.

Accessing syntax help for commands and functions

There are several ways to access syntax help for a Qlik Sense syntax keyword.

Accessing the help portal

You can access detailed help in the Qlik Sense help portal in two different ways.

- Click  in the toolbar to enter syntax help mode. In syntax help mode you can click on a syntax keyword (marked in blue and underlined) to access syntax help.

- Place the cursor inside or at the end of the keyword and press Ctrl+H.



You cannot edit the script in syntax help mode.

Using the auto-complete function

If you start to type a Qlik Sense script keyword, you get an auto-complete list of matching keywords to select from. The list is narrowed down as you continue to type, and you can select from templates with suggested syntax and parameters. A tooltip displays the syntax of the function, including parameters and additional statements, as well as a link to the help portal description of the statement or function.



You can also use the keyboard shortcut Ctrl+Space to show the keyword list, and Ctrl+Shift+Space to show a tooltip.

Inserting a prepared test script

You can insert a prepared test script that will load a set of inline data fields. You can use this to quickly create a data set for test purposes.

Do the following:



- Press Ctrl + 00.

The test script code is inserted into the script.

Indenting code

You can indent the code to increase readability.

Do the following:

1. Select one or several lines to change indentation.
2. Click  to indent the text (increase indentation) or, click  to outdent the text (decrease indentation).



You can also use keyboard shortcuts:

Tab (indent)





Shift+Tab (outdent)

Searching and replacing text

You can search and replace text throughout script sections.

Searching for text

Open the data load editor. Do the following:





1. Click  in the toolbar.
The search drop-down dialog is displayed.
2. In the search box, type the text you want to find.
The search results are highlighted in the current section of the script code. Also, the number of text instances found is indicated next to the section label.
3. You can browse through the results by clicking  and .
4. Click  in the toolbar to close the search dialog.



Also, you can select **Search in all sections** to search in all script sections. The number of text instances found is indicated next to each section label. You can select **Match case** to make case sensitive searches.

Replacing text

Do the following:

1. Click  in the toolbar.
The search drop-down dialog is displayed.
2. Type the text you want to find in the search box.
3. Type the replacement text in the replace box and click **Replace**.
4. Click  to find next instance of search text and do one of the following:
 - Click **Replace** to replace text.
 - Click  to find next.
5. Click  in the toolbar to close the search dialog.



You can also click **Replace all in section** to replace all instances of the search text in the current script section. The replace function is case sensitive, and replaced text will have the case given in the replace field. A message is shown with information about how many instances that were replaced.


Commenting in the script

You can insert comments in the script code, or deactivate parts of the script code by using comment marks. All text on a line that follows to the right of `//` (two forward slashes) will be considered a comment and will not be executed when the script is run.

The data load editor toolbar contains a shortcut for commenting or uncommenting code. The function works as a toggle. That is, if the selected code is commented out it will be commented, and vice versa.

Commenting


Do the following:

1. Select one or more lines of code that are not commented out, or place the cursor at the beginning of a line.
2. Click , or press Ctrl + K.

The selected code is now commented out.

Uncommenting

Do the following:

1. Select one or more lines of code that are commented out, or place the cursor at the beginning of a commented line.
2. Click , or press Ctrl + K.

The selected code will now be executed with the rest of the script.



There are more ways to insert comments in the script code:

- Using the **Rem** statement.
- Enclosing a section of code with `/*` and `*/`.

Example:

```
Rem This is a comment ;
```

```
/* This is a comment  
   that spans two lines */
```

```
// This is a comment as well
```

Selecting all code

You can select all code in the current script section.


Do the following:

- Press Ctrl + A.

All script code in the current section is selected.

2.4 Debugging the load script

To show the debug panel, do the following:

- Click  in the data load editor toolbar.
The debug panel is opened at the bottom of the data load editor.



You cannot create connections, edit connections, select data, save the script or load new data while you are running in debug mode, that is, from when you have started debug execution until the script is executed or execution has been ended.

Debug toolbar

The debug panel for the data load editor has a toolbar with the following options to control the debug execution:

Limited load	<p>Select the check box to limit how many rows of data to load from each data source. This is useful for large data sources, as it reduces the execution time.</p> <p>Enter the number of rows you want to limit the load to.</p> <div> <i>This only applies to physical data sources. Automatically-generated and inline loads will not be limited, for example.</i> </div>
	Start or continue execution in debug mode until the next breakpoint is reached.
	Step to the next line of code.
	End execution here.

Output

Output displays all messages that are generated during debug execution. You can select to lock the output from scrolling when new messages are displayed by clicking .

Additionally, the output menu () contains the following options:

Clear	Click this to delete all output messages.
Select all text	Click this to select all output messages.
Scroll to bottom	Click this to scroll to the last output message.



***Variables** and **Breakpoints** will not be handled in this tutorial.*

3 Transforming data

This section introduces you to the transformation and manipulation of data that you can perform through the data load editor before using the data in your app.

One of the advantages to data manipulation is that you can choose to load only a subset of the data from a file, such as a few chosen columns from a table, to make the data handling more efficient. You can also load the data more than once to split up the raw data into several new logical tables. It is also possible to load data from more than one source and merge it into one table in Qlik Sense.

In this section you will learn how to load data using **Crosstable**. You will also learn how to join tables, use inter-record functions such as **Peek** and **Previous**, and load the same row several times using **While Load**.

3.1 Using the Crosstable prefix

Cross tables are a common type of table featuring a matrix of values between two orthogonal lists of header data. Whenever you have a cross table of data, the Crosstable prefix can be used to transform the data and create the desired fields.

Crosstable prefix

In the input table below you have one column per month and one row per product.

Input						
Product	Jan 2014	Feb 2014	Mar 2014	Apr 2014	May 2014	Jun 2014
A	100	98	108	83	103	82
B	284	279	297	305	294	292
C	50	53	50	54	49	51



Output		
Product	Month	Sales
A	Jan 2014	100
A	Feb 2014	98
A	Mar 2014	108
A	Apr 2014	83
A	May 2014	103
A	Jun 2014	82
B	Jan 2014	284
B	Feb 2014	279
B	Mar 2014	297
B	Apr 2014	305



If this table is simply loaded into Qlik Sense the result is a table with one field for *Product* and one field for each of the months. But if you want to analyze this data, it is much easier to have all numbers in one field and all months in another, that is, in a three-column table, one for each category (*Product*, *Month*, *Sales*).

The Crosstable prefix converts the data to a table with one column for *Month* and another for *Sales*. Another way to express it is to say that it takes field names and converts these to field values.

Example:

```
Crosstable (Month, Sales) LOAD Product, [Jan 2014], [Feb 2014], [Mar 2014], ... From ... ;
```

Do the following:

1. In the app, *Advanced Scripting tutorial*, open the **Data load editor**.
2. Click **Create new connection** and select **All files**.
3. Locate the folder where the tutorial file *Product.xlsx* is stored, and give it the name *Tutorial Files*.
4. Click **Save**.
5. Add a new script section by clicking on  in the upper left corner.
6. Give the section the name *Product* and press Enter.
7. Click  on the **Tutorial Files** data connection you created earlier to select a file to load data from.
8. Select *Product.xlsx* and click **Select**.
9. Select *Product*.




Under **Field names**, make sure that **Embedded field names** is selected to include the names of the table fields when you load the data.

10. Click **Insert script**.

The script looks like this:


```
LOAD
    Product,
    "Jan 2014",
    "Feb 2014",
    "Mar 2014",
    "Apr 2014",
    "May 2014",
    "Jun 2014"
FROM 'lib://Tutorial Files/Product.xlsx'
(ooxml, embedded labels, table is Product);
```

11. In the upper right corner, click **Load data**.
When you click **Load data**, the script is saved.
12. When the script execution is finished, click **Close**.
13. To view the table, select **Data model viewer**, click  to expand the table and verify your data.

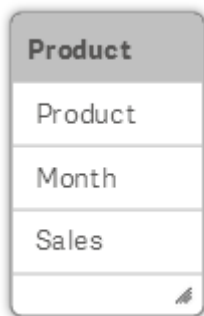
This script produces a table with one field for product and one for each of the months.



Product
Product
Jan 2014
Feb 2014
Mar 2014
Apr 2014
May 2014
Jun 2014

14. Select **Data load editor** and open the *Product* script.
15. Add a new row at the top of the script and add the following to that row:
`CrossTable(Month, Sales)`
16. In the upper right corner, click **Load data**.
17. When the script execution is finished, click **Close**.
18. Click  and select **Data model viewer** to verify that your data is loaded.

The loaded table is a three-column table, one column for each category (Product, Month, Sales):



Product
Product
Month
Sales

Usually the input data has only one column as a qualifier field; as an internal key (Product in the above example). But you can have several. If so, all qualifying fields must be listed before the attribute fields in the **LOAD** statement, and the third parameter to the **Crosstable** prefix must be used to define the number of qualifying fields.

It is not possible to have a preceding **LOAD** or a prefix in front of the **Crosstable** keyword. Auto-concatenate can, however, be used.

The numeric interpretation will not work for the attribute fields. This means that if you have months as column headers, these will not be automatically interpreted. The work-around is to use the crosstable prefix to create a temporary table, and to run a second pass through it to make the interpretations as in the following example:

tmpData:


```
Crosstable (MonthText, Sales)
LOAD Product, [Jan 2014], [Feb 2014], ... From 'lib://Tutorial Files/Product.xlsx'
(ooxml, embedded labels, table is Product);

Final:
LOAD Product,
Date(Date#(MonthText, 'MMM YYYY'), 'MMM YYYY') as Month,
Sales
Resident tmpData;
Drop Table tmpData;
```

Clearing the memory cache

We need sometimes to delete tables we have created during the script to clear the memory cache. When you load into a temporary table as in the previous section then you should drop it when it is not needed anymore.

The following examples show the various options available with **Drop Table**:

```
DROP TABLE Name1[, Name2 [,Name3 ...]];
DROP TABLES Name1[, Name2 [,Name3 ...]];
```

The command **Drop** also lets you delete one or several fields:

```
DROP FIELD Name1[, Name2 [, Name3 ...]] ;
DROP FIELDS Name1[, Name2 [, Name3 ...]] ;
```

As you may notice, the keyword FIELD or TABLE can set in plural (FIELDS, TABLES) even if you have to delete one single table or field.

3.2 Combining tables with Join and Keep

A join is an operation that takes two tables and combines them into one. The records of the resulting table are combinations of records in the original tables, usually in such a way that the two records contributing to any given combination in the resulting table have a common value for one or several common fields, a so-called natural join. In Qlik Sense, joins can be made in the script, producing logical tables.

The Qlik Sense logic will then not see the separate tables, but rather the result of the join, which is a single internal table. In some situations this is needed, but there are disadvantages:

- The loaded tables often become larger, and Qlik Sense works slower.
- Some information may be lost: the frequency (number of records) within the original table may no longer be available.

The **Keep** functionality, which has the effect of reducing one or both of the two tables to the intersection of table data before the tables are stored in Qlik Sense, has been designed to reduce the number of cases where explicit joins needs to be used.



*In this documentation, the term **join** is usually used for joins made before the internal tables are created. The association, made after the internal tables are created, is however essentially also a join.*

Join

The simplest way to make a join is with the **Join** prefix in the script, which joins the internal table with another named table or with the last previously created table. The join will be an outer join, creating all possible combinations of values from the two tables.

Example:

```
LOAD a, b, c from table1.csv;  
join LOAD a, d from table2.csv;
```

The resulting internal table has the fields a, b, c and d. The number of records differs depending on the field values of the two tables.





*The names of the fields to join over must be exactly the same. The number of fields to join over is arbitrary. Usually the tables should have one or a few fields in common. No field in common will render the cartesian product of the tables. All fields in common is also possible, but usually makes no sense. Unless a table name of a previously loaded table is specified in the **Join** statement the **Join** prefix uses the last previously created table. The order of the two statements is thus not arbitrary.*

Using Join

The explicit **Join** prefix in the Qlik Sense script language performs a full join of the two tables. The result is one table. In many cases such joins will result in very large tables.


Do the following:

1. In the app, *Advanced Scripting tutorial*, open the **Data load editor**.
2. Add a new script section by clicking on  in the upper left corner.
3. Give the section the name *Transactions* and press Enter.
4. Click  on the **Tutorial files** data connection to select a file to load data from.
5. Select *Transactions.csv* and click **Select**.



*Under **Field names**, make sure that **Embedded field names** is selected to include the names of the table fields when you load the data.*

6. Click **Insert script**.

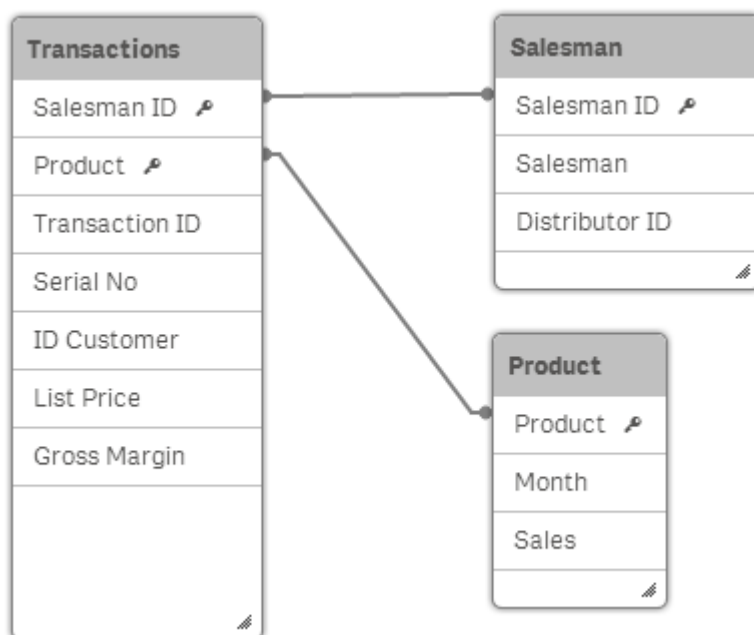
7. Click  on the **Tutorial files** data connection to select a file to load data from.
8. Select *Salesman.xlsx* and click **Select**.
9. Select *Salesman* and click **Insert script**.

The script looks like this:

```
LOAD
    "Transaction ID",
    "Salesman ID",
    Product,
    "Serial No",
    "ID Customer",
    "List Price",
    "Gross Margin"
FROM 'lib://Tutorial Files/Transactions.csv'
(txt, codepage is 1252, embedded labels, delimiter is ',', msq);

LOAD
    "Salesman ID",
    Salesman,
    "Distributor ID"
FROM 'lib://Tutorial Files/Salesman.xlsx'
(ooxml, embedded labels, table is Salesman);
```

Clicking on **Load data** at this point would produce the following data model:



However, having the Transactions and Salesman tables separated may not be the required result. It may be better to join the two tables.

Do the following:


1. To set a name for the joined table, add a new row at the top of the script and enter the following:
Transactions:
2. To join the Transactions and Salesman tables, on the empty line above the second **LOAD** statement, add the following:

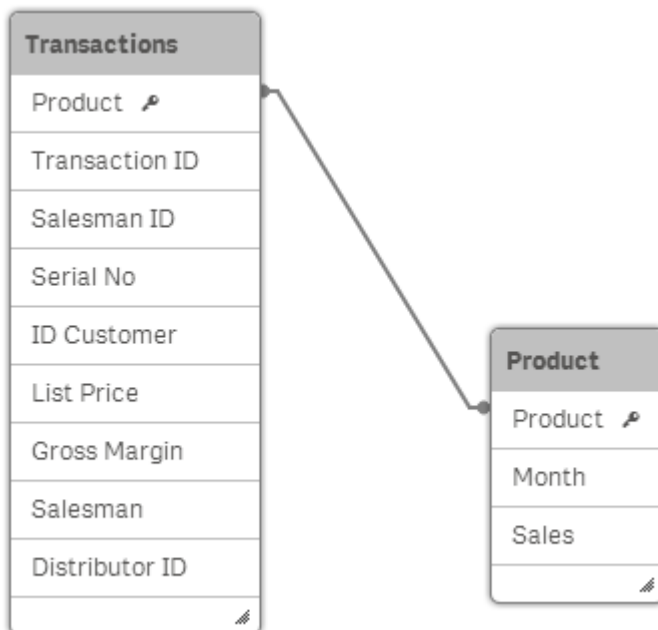
Join (Transactions)

Your script should now look like this:

Transactions:

```
LOAD
    "Transaction ID",
    "Salesman ID",
    Product,
    "Serial No",
    "ID Customer",
    "List Price",
    "Gross Margin"
FROM 'lib://Tutorial Files/Transactions.csv'
(txt, codepage is 1252, embedded labels, delimiter is ',', msq);
Join (Transactions)
LOAD
    "Salesman ID",
    Salesman,
    "Distributor ID"
FROM 'lib://Tutorial Files/Salesman.xlsx'
(ooxml, embedded labels, table is Salesman);
```

3. Click **Load data**.
When you click **Load data**, the script is saved.
4. When the script execution is finished, click **Close** in the **Progress** pop-up.
5. Click  and select **Data model viewer** to verify that your data is loaded.



All the fields of the Transactions and Salesman tables are now combined into a single Transactions table.

Keep

One of the main features of Qlik Sense is its ability to make associations between tables instead of joining them, which reduces space in memory, increases speed and gives enormous flexibility. The keep functionality has been designed to reduce the number of cases where explicit joins need to be used.

The **Keep** prefix between two **LOAD** or **SELECT** statements has the effect of reducing one or both of the two tables to the intersection of table data before they are stored in Qlik Sense. The **Keep** prefix must always be preceded by one of the keywords **Inner**, **Left** or **Right**. The selection of records from the tables is made in the same way as in a corresponding join. However, the two tables are not joined and will be stored in Qlik Sense as two separately named tables.

Inner

The **Join** and **Keep** prefixes in the Qlik Sense script language can be preceded by the prefix **Inner**.

If used before **Join**, it specifies that the join between the two tables should be an inner join. The resulting table contains only combinations between the two tables with a full data set from both sides.

If used before **Keep**, it specifies that the two tables should be reduced to their common intersection before being stored in Qlik Sense.

Example:

In these examples we use the source tables Table1 and Table2:

Table1		Table2	
A	B	A	C
1	aa	1	xx
2	cc	4	yy
3	ee		

Inner examples source tables

First, we perform an **Inner Join** on the tables, resulting in VTable, containing only one row, the only record existing in both tables, with data combined from both tables.

VTable:
SELECT * from Table1;
inner join SELECT * from Table2;

VTable		
A	B	C
1	aa	xx

Inner Join example

If we perform an **Inner Keep** instead, you will still have two tables. The two tables are of course associated via the common field A.

```
VTab1:  
SELECT * from Table1;  
VTab2:  
inner keep SELECT * from Table2;
```

VTab1		VTab2	
A	B	A	C
1	aa	1	xx

Inner Keep example

Left

The **Join** and **Keep** prefixes in the Qlik Sense script language can be preceded by the prefix **Left**.

If used before **Join**, it specifies that the join between the two tables should be a Left Join. The resulting table only contains combinations between the two tables with a full data set from the first table.

If used before **Keep**, it specifies that the second table should be reduced to its common intersection with the first table before being stored in Qlik Sense.

Example:

In these examples we use the source tables Table1 and Table2:

Table1		Table2	
A	B	A	C
1	aa	1	xx
2	cc	4	yy
3	ee		

Left examples source tables

First, we perform a **Left Join** on the tables, resulting in VTable, containing all rows from Table1, combined with fields from matching rows in Table2.

```
VTable:  
SELECT * from Table1;  
left join SELECT * from Table2;
```

VTable		
A	B	C
1	aa	xx
2	cc	—
3	ee	—

Left Join example

If we perform an **Left Keep** instead, you will still have two tables. The two tables are of course associated via the common field A.

VTab1:

```
SELECT * from Table1;
```

VTab2:

```
left keep SELECT * from Table2;
```

VTab1		VTab2	
A	B	A	C
1	aa	1	xx
2	cc		
3	ee		

Left Keep example

Right

The **Join** and **Keep** prefixes in the Qlik Sense script language can be preceded by the prefix **Right**.

If used before **Join**, it specifies that the join between the two tables should be a Right Join. The resulting table only contains combinations between the two tables with a full data set from the second table.

If used before **Keep**, it specifies that the first table should be reduced to its common intersection with the second table before being stored in Qlik Sense.

Example:

In these examples we use the source tables Table1 and Table2:

Table1		Table2	
A	B	A	C
1	aa	1	xx
2	cc	4	yy
3	ee		

Right examples source tables

First, we perform a **Right Join** on the tables, resulting in VTable, containing all rows from Table2, combined with fields from matching rows in Table1.

VTable:

```
SELECT * from Table1;
```

```
right join SELECT * from Table2;
```

VTable		
A	B	C
1	aa	xx
4	—	yy

Right Join example

If we perform an **Right Keep** instead, you will still have two tables. The two tables are of course associated via the common field A.

```
VTab1:  
SELECT * from Table1;  
VTab2:  
right keep SELECT * from Table2;
```

VTab1		VTab2	
A	B	A	C
1	aa	1	xx
		4	yy

Right Keep example

3.3 Using inter-record functions

These functions are used when a value from previously loaded records of data is needed for the evaluation of the current record.

In this part of the tutorial we will be examining the **Peek**, **Previous** and **Exists** functions. More detailed information on these functions can be found on the Qlik Sense help site.

Peek

Peek() finds the value of a field in a table for a row that has already been loaded or that exists in internal memory. The row number can be specified, as can the table.

Syntax:

```
Peek(fieldname [ , row [ , tablename ] ] )
```

Row must be an integer. 0 denotes the first record, 1 the second and so on. Negative numbers indicate order from the end of the table. -1 denotes the last record read.

If no row is stated, -1 is assumed.

tablename is a table label without the ending colon. If no **tablename** is stated, the current table is assumed. If used outside the **LOAD** statement or referring to another table, the **tablename** must be included.

Previous

Previous() finds the value of the **expr** expression using data from the previous input record that has not been discarded because of a **where** clause. In the first record of an internal table, the function will return NULL.

Syntax:

```
Previous(expression)
```


The **Previous** function may be nested in order to access records further back. Data are fetched directly from the input source, making it possible to refer also to fields which have not been loaded into Qlik Sense, that is, even if they have not been stored in the associated database.

Exists

Exists() determines whether a specific field value has already been loaded into the field in the data load script. The function returns TRUE or FALSE, so can be used in the **where** clause of a **LOAD** statement or an **IF** statement.

Syntax:

```
Exists(field [ , expression ] )
```

The field must exist in the data loaded so far by the script. **Expression** is an expression evaluating to the field value to look for in the specified field. If omitted, the current record's value in the specified field will be assumed.

Using Peek() and Previous()


In their simplest form, **Peek()** and **Previous()** are used to identify specific values within a table. The following example is based on the simple table *Employees*.

	A	B	C	D	E	F
1	Date	Hired	Terminated			
2	1/1/2011		6	0		
3	2/1/2011		4	2		
4	3/1/2011		6	1		
5	4/1/2011		5	2		
6	5/1/2011		3	2		
7	6/1/2011		4	1		
8	7/1/2011		6	2		
9	8/1/2011		4	1		
10	9/1/2011		4	0		
11	10/1/2011		1	0		
12	11/1/2011		1	0		
13	12/1/2011		3	2		
14	1/1/2012		1	1		
15	2/1/2012		1	0		
16	3/1/2012		3	1		
17	4/1/2012		3	4		
18	5/1/2012		2	1		

Currently this only collects data for month, hires and terminations, so we are going to add fields for Employee Count and Employee Var, using the **Peek** and **Previous** functions, to see the monthly difference in total employees.

Do the following:

1. In the app, *Advanced Scripting tutorial*, open the **Data load editor**.
2. Add a new script section by clicking on **+** in the upper left corner.
3. Give the section the name *Employees* and press Enter.

- Click  on the *Tutorial files* data connection to select a file to load data from.
- Select *Employees.xlsx* and click **Select**.



Under **Field names**, make sure that **Embedded field names** is selected to include the names of the table fields when you load the data.

- Click **Insert script**.

The script looks like this:

```
LOAD
    "Date",
    Hired,
    Terminated
FROM 'lib://Tutorial Files/Employees.xlsx'
(ooxml, embedded labels, table is Sheet2);
```

- Modify the script so that it now looks like this:

```
[Employees Init]:
LOAD
    rowno() as Row,
    Date(Date) as Date,
    Hired,
    Terminated,
    If(rowno()=1, Hired-Terminated, peek([Employee Count], -1)+(Hired-Terminated)) as
[Employee Count]
FROM 'lib://Tutorial Files/Employees.xlsx'
(ooxml, embedded labels, table is Sheet2);
```

The dates in the Date field in the Excel sheet are in the format MM/DD/YYYY. To ensure dates are interpreted correctly using the format from the system variables, the Date function is applied to the Date field.

The **Peek()** function lets you identify any value loaded for a defined field.

```
If(rowno()=1, Hired-Terminated, peek([Employee Count], -1)+(Hired-Terminated)) as [Employee Count]
```

In this expression we first look to see if the rowno() is equal to 1. If it is equal to 1, no Employee Count will exist, so we will just populate the field with the difference of Hired minus Terminated. If the rowno() is greater than 1, we look at last month's Employee Count and use that number to add to the difference of that month's Hired minus Terminated employees.

Notice too that in the **Peek()** function we are using a (-1). This tells Qlik Sense to look at the record above the current record. If the (-1) is not specified, Qlik Sense will assume that you want to look at the previous record.

- Add the following below the script you have just modified

```
[Employee Count]:
LOAD
    Row,
    Date,
    Hired,
    Terminated,
    [Employee Count],
    If(rowno()=1,0,[Employee Count]-Previous([Employee Count])) as [Employee var]
Resident [Employees Init] Order By Row asc;
```

```
Drop Table [Employees Init];
```

The **Previous()** function lets you identify the last value loaded for a defined field.

```
If(rowno()=1,0,[Employee Count]-Previous([Employee Count])) as [Employee Var]
```

In this expression we first look to see if the rowno() is equal to 1. If it is equal to 1, we know that there will be no Employee Var because there is no record for the previous month's Employee Count so we simply enter 0 for the value. If the rowno() is greater than 1, we know that there will be an Employee Var so we look at last month's Employee Count and subtract that number from the current month's Employee Count to create the value in the Employee Var field.

Your completed script should look like this:

```
[Employees Init]:
LOAD
    rowno() as Row,
    Date(Date) as Date,
    Hired,
    Terminated,
    If(rowno()=1, Hired-Terminated, peek([Employee Count], -1)+(Hired-Terminated)) as
[Employee Count]

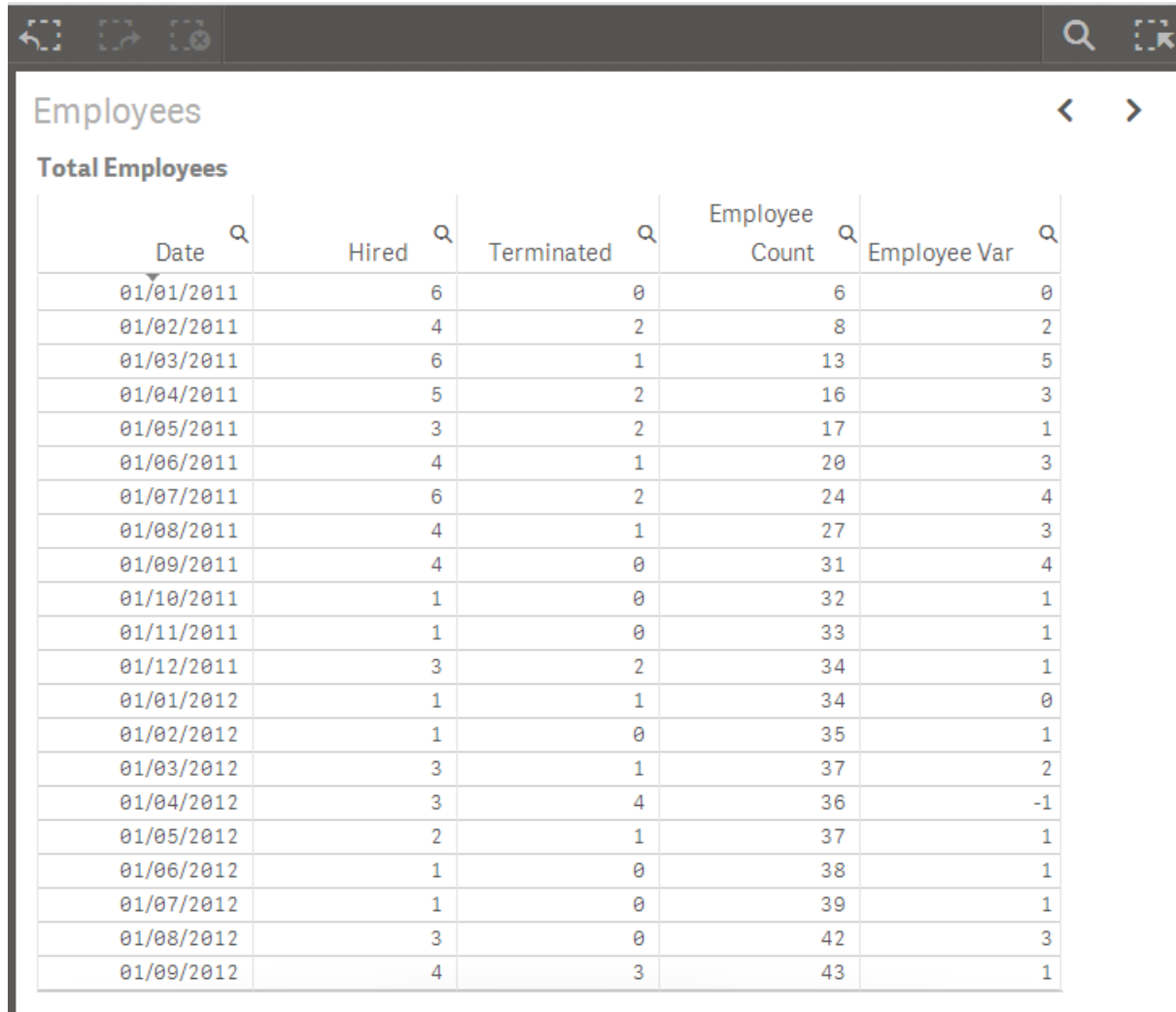
FROM 'lib://Tutorial Files/Employees.xlsx'
(ooxml, embedded labels, table is Sheet2);

[Employee Count]:
LOAD
    Row,
    Date,
    Hired,
    Terminated,
    [Employee Count],
    If(rowno()=1,0,[Employee Count]-Previous([Employee Count])) as [Employee Var]
Resident [Employees Init] Order By Row asc;

Drop Table [Employees Init];
```

9. In the upper right corner, click **Load data**.

If, in a new sheet in the app overview, you now create a standard table using Date, Hired, Terminated, Employee Count and Employee Var as the columns of the table, you should get a result similar to this:



The screenshot shows a Qlik Sense interface with a table titled "Employees". The table has five columns: "Date", "Hired", "Terminated", "Employee Count", and "Employee Var". The data spans from January 2011 to September 2012. The "Employee Count" column is calculated as the sum of "Hired" and "Terminated" from the previous month. The "Employee Var" column is calculated as the difference between the current month's "Hired" and "Terminated" counts.

Date	Hired	Terminated	Employee Count	Employee Var
01/01/2011	6	0	6	0
01/02/2011	4	2	8	2
01/03/2011	6	1	13	5
01/04/2011	5	2	16	3
01/05/2011	3	2	17	1
01/06/2011	4	1	20	3
01/07/2011	6	2	24	4
01/08/2011	4	1	27	3
01/09/2011	4	0	31	4
01/10/2011	1	0	32	1
01/11/2011	1	0	33	1
01/12/2011	3	2	34	1
01/01/2012	1	1	34	0
01/02/2012	1	0	35	1
01/03/2012	3	1	37	2
01/04/2012	3	4	36	-1
01/05/2012	2	1	37	1
01/06/2012	1	0	38	1
01/07/2012	1	0	39	1
01/08/2012	3	0	42	3
01/09/2012	4	3	43	1

Peek() and **Previous()** allow users to target defined rows within a table. The biggest difference between the two functions is that the **Peek()** function allows the user to look into a field that was not previously loaded into the script whereas the **Previous()** function can only look into a previously loaded field. **Previous()** operates on the Input to the **LOAD** statement, whereas **Peek()** operates on the output of the **LOAD** statement. (Same as the difference between **RecNo()** and **RowNo()**.) This means that the two functions will behave differently if you have a **Where**-clause.

So the **Previous()** function would be better suited for when a user needs to show the current value versus the previous value. In the example we calculated the employee variance from month to month.

The **Peek()** function would be better suited when the user is targeting either a field that has not been previously loaded into the table or if the user needs to target a specific row. This was shown in the example where we calculated the Employee Count by peeking into the previous month's Employee Count and adding the difference between the hired and terminated employees for the current month. Remember that Employee Count was not a field in the original file.

Using Exists()

The **Exists()** function is often used with the **Where** clause in the script in order to load data if related data has already been loaded in the data model.

In the following example we are also using the **Dual()** function to assign numeric values to strings.

Do the following:

1. Create a new app and give it a name.
2. Open the **Data load editor** and enter the following script:

```
//Add dummy people data
PeopleTemp:
LOAD * INLINE [
    PersonID, Person
    1, Jane
    2, Joe
    3, Shawn
    4, Sue
    5, Frank
    6, Mike
    7, Gloria
    8, Mary
    9, Steven,
    10, Bill
];

//Add dummy age data
AgeTemp:
LOAD * INLINE [
    PersonID, Age
    1, 23
    2, 45
    3, 43
    4, 30
    5, 40
    6, 32
    7, 45
    8, 54
    9,
    10, 61
    11, 21
    12, 39
];

//LOAD new table with people
People:
NoConcatenate LOAD
    PersonID,
    Person
Resident PeopleTemp;

Drop Table PeopleTemp;

//Add age and age bucket fields to the People table
```

```

Left Join (People)
LOAD
    PersonID,
    Age,
    If(IsNull(Age) or Age='', Dual('No age', 5),
        If(Age<25, Dual('Under 25', 1),
            If(Age>=25 and Age <35, Dual('25-34', 2),
                If(Age>=35 and Age<50, Dual('35-49' , 3),
                    If(Age>=50, Dual('50 or over', 4)
                        )))) as AgeBucket

Resident AgeTemp
Where Exists(PersonID);

DROP Table AgeTemp;

```

3. Click **Load data**.

When you click **Load data**, the script is saved.

In the script, the Age and AgeBucket fields are loaded only if the PersonID has already been loaded in the data model.

Notice in the AgeTemp table that there are ages listed for PersonID 11 and 12 but since those IDs were not loaded in the data model (in the People table), they are excluded by the **Where Exists(PersonID)** clause. This clause can also be written like this: **Where Exists(PersonID, PersonID)**.

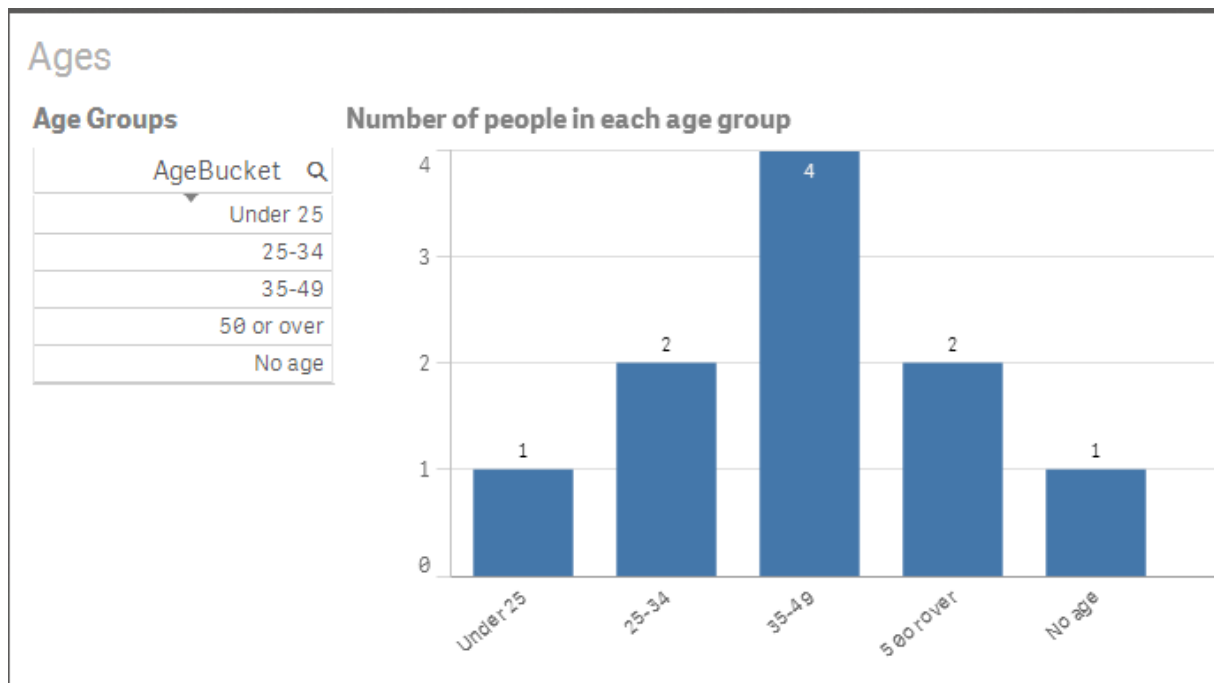
The final results of the script look like this:

Person Details			
PersonID	Person	Age	AgeBucket
9	Steven		No age
1	Jane	23	Under 25
4	Sue	30	25-34
6	Mike	32	25-34
5	Frank	40	35-49
3	Shawn	43	35-49
7	Gloria	45	35-49
2	Joe	45	35-49
8	Mary	54	50 or over
10	Bill	61	50 or over

If none of the PersonIDs in the AgeTemp table had been loaded into the data model, then the Age and AgeBucket fields would not have been joined to the People table. Using the **Exists** function can help to prevent orphan records/data in the data model, that is, Age and AgeBucket fields that do not have any associated people.

4. In the **App overview**, create a new sheet and give it a name.
5. Open the new sheet and click **Edit**.
6. Add a standard table to the sheet with the dimension AgeBucket and name the visualization *Age Groups*.
7. Add a bar chart to the sheet with the dimension AgeBucket, and the measure Count([AgeBucket]) and name the visualization *Number of people in each age group*.
8. Adjust the properties of the table and bar chart to your preference and click **Done**.

You should now have a sheet similar to this:



The **Dual()** function is very useful in the script, or in a chart expression, when there is the need to assign a numeric value to a string.

```
LOAD
    PersonID,
    Age,
    If(IsNull(Age) or Age='', Dual('No age', 5),
        If(Age<25, Dual('Under 25', 1),
            If(Age >=25 and Age <35, Dual('25-34', 2),
                If(Age>=35 and Age<50, Dual('35-49', 3),
                    If(Age>=50, Dual('50 or over', 4)
                ))) as AgeBucket
Resident AgeTemp
Where Exists(PersonID);
```

In the script you have an application that loads ages, and you have decided to put those ages in buckets so that you can create visualizations based on the age buckets versus the actual ages. There is a bucket for people under 25, between 25 and 35, and so on. By using the **Dual()** function, the age buckets can be assigned a numeric value that can later be used to sort the age buckets in a list box or in a chart. So, as in the app sheet, the sort puts "No age" at the end of the list.

3.4 Matching intervals and iterative loading

The **Intervalmatch** prefix to a **LOAD** or **SELECT** statement is used to link discrete numeric values to one or more numeric intervals. This is a very powerful feature which can be used, for example, in production environments.

Using the IntervalMatch prefix

The most basic interval match is when you have a list of numbers or dates (events) in one table, and a list of intervals in a second table. The goal is to link the two tables. In general, this is a many to many relationship, that is, 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.

By far the simplest way to solve this problem in Qlik Sense is to use the **IntervalMatch** prefix 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.

Do the following:

1. Create a new app and give it a name.
2. Open the **Data load editor** and create a new connection to the folder containing the tutorial sample files and give it the name *Tutorial Files*.
3. Load the Events table, then the Intervals table.
Use the *Events.txt* and *Intervals.txt* sample files provided with the tutorial.
4. At the end of the script add an IntervalMatch to create a third table that bridges the two first tables using the following script:

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

5. The complete script should look similar to this:

```
Events:
LOAD
    EventID,
    EventDate,
    EventAttribute
FROM 'lib://Tutorial Files/Events.txt'
(txt, utf8, embedded labels, delimiter is '\t', msq);
```

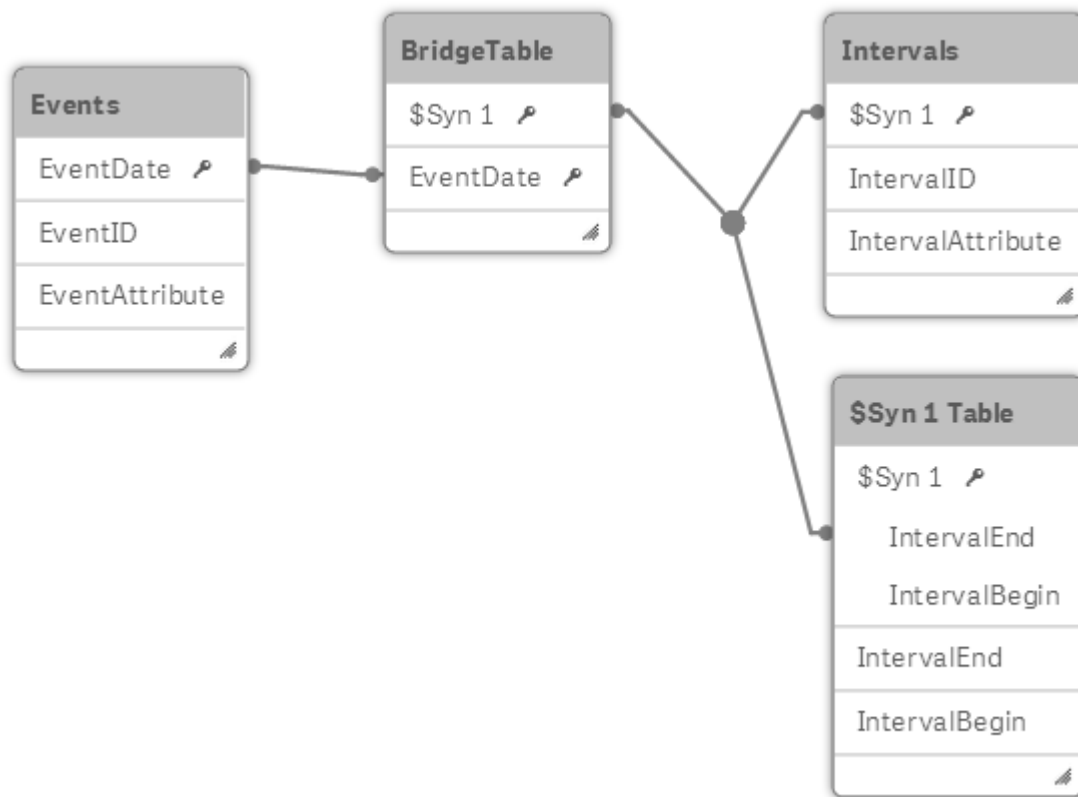
```
Intervals:
LOAD
    IntervalID,
    IntervalAttribute,
    IntervalBegin,
    IntervalEnd
FROM 'lib://Tutorial Files/Intervals.txt'
(txt, utf8, embedded labels, delimiter is '\t', msq);
```

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

6. Click **Load data**.

When you click **Load data**, the script is saved.

The data model contains a composite key (the IntervalBegin and IntervalEnd fields) which will manifest itself as a Qlik Sense synthetic key:



The basic tables are:

- 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 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 the original number of records. All Qlik Sense calculations operating on these tables, for example, Count(EventID), will work and will be evaluated correctly.

Using a While loop and iterative loading IterNo()

You can achieve almost the same bridge table using a **While** loop and **IterNo()** that creates enumerable values between the lower and upper bounds of the interval.

A loop inside the **LOAD** statement can be created using the **While** clause:

```
LOAD Date, IterNo() as Iteration From ... while IterNo() <= 4 ;
```

Such a **LOAD** statement will loop over each input record and load this over and over as long as the expression in the **While** clause is true. The **IterNo()** function returns “1” in the first iteration, “2” in the second, and so on.

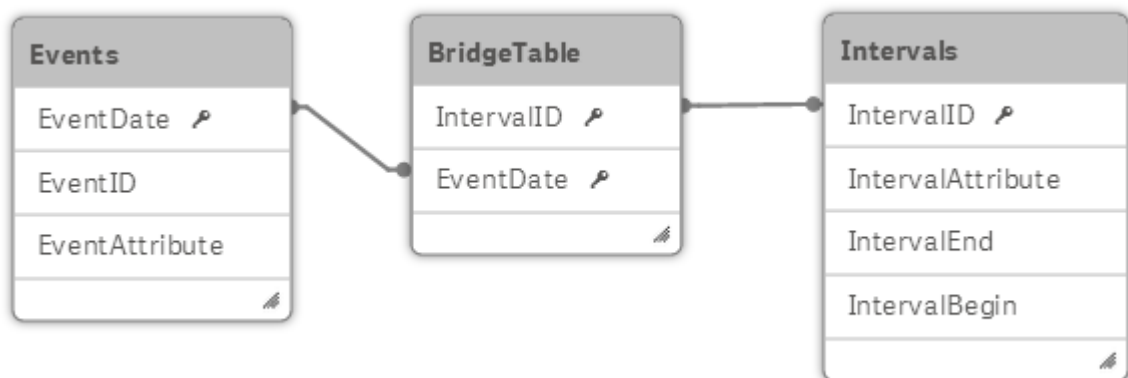
You have a primary key for the intervals, the IntervalID, so the only difference in the script will be how the bridge table is created:

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

Do the following:

1. Replace the existing Bridgetable statements with the above script.
2. Click **Save** in the toolbar.
3. In the upper right corner, click **Load data**.

The data model should look like this:



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 IntervalID can be stored directly in the event table. There are several ways to achieve this, but the most useful is to join Bridgetable with the Events table.

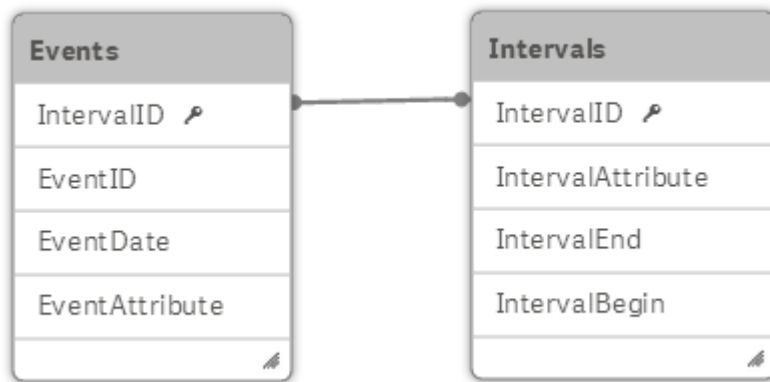
4. Add the following script to the bottom of existing script:

```
Join (Events)
LOAD EventDate, IntervalID
    Resident BridgeTable;
```

```
Drop Table BridgeTable;
```

5. Click **Save** in the toolbar.
6. In the upper right corner, click **Load data**.

The data model now looks like this:



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.

- If the endpoints are included, it is a closed interval:
 $[a,b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$
- If the endpoints are not included, it is an open interval:
 $]a,b[= \{x \in \mathbb{R} \mid a < x < b\}$
- 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 need 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 point is the end of one interval and, at the same time, the beginning of next. A number with 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.

Creating a date interval from a single date

Sometimes time 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 as in the table below 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 contains rows with empty dates corresponding to the initial conversion rate, before the first change was made.

Currency rates

Currency	Change Date	Rate
EUR	-	8.59
EUR	28/01/2013	8.69
EUR	15/02/2013	8.45
USD	-	6.50
USD	10/01/2013	6.56
USD	03/02/2013	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.

In this script example the table `In_Rates` is created by an inline load. Make sure that the dates in the `Change Date` column are in the same format as the local date format.

```
In_Rates:
LOAD * Inline [
Currency,Change Date,Rate
EUR,,8.59
EUR,28/01/2013,8.69
EUR,15/02/2013,8.45
USD,,6.50
USD,10/01/2013,6.56
USD,03/02/2013,6.30
];
```

Do the following:

1. Determine which time range you want to work with. The beginning of the range must be before the first date in the data and the end of the range must be after the last.

```
Let vBeginTime = Num('1/1/2013');
Let vEndTime = Num('1/3/2013');
Let vEpsilon = Pow(2,-27);
```

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”. Sort the table first according to `Currency`, then according to the “From Date” descending so that you have the latest dates on top.

```
Tmp_Rates:
LOAD Currency, Rate,
      Date(If(IsNum([Change Date]), [Change Date], $(#vBeginTime))) as FromDate
Resident In_Rates;
```

3. 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 step 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.

Rates:

```
LOAD Currency, Rate, FromDate,
    Date(If( Currency=Peek('Currency'),
        Peek('FromDate') - $(#vEpsilon),
        $(#vEndTime)
    )) as ToDate
Resident Tmp_Rates
Order By Currency, FromDate Desc;
```

4. Drop the input table and the temporary table.

```
Drop Table Tmp_Rates;
```

The script listed below will update the source table in the following manner:

Updated source table

Currency	Rate	FromDate	ToDate
EUR	8.45	15/02/2013	vEndTime
EUR	8.69	28/01/2013	14/02/2013 23:59:59
EUR	8.59	vBeginTime	28/01/2013 23:59:99
USD	6.30	03/02/2013	vEndTime
USD	6.56	10/01/2013	2/02/2013 23:59:59
USD	6.50	vBeginTime	9/01/2013 23:59:59

When the script is run, you will have a table listing the intervals correctly. Use the **Preview** section of the data model viewer to view the resulting table.

Preview of data

Currency	Rate	FromDate	ToDate
EUR	8.45	15/02/2013	01/03/2013
EUR	8.69	28/01/2013	14/02/2013
EUR	8.59	01/01/2013	28/01/2013
USD	6.30	03/02/2013	01/03/2013
USD	6.56	10/01/2013	2/02/2013
USD	6.50	01/01/2013	9/01/2013

This table can subsequently be used in a comparison with an existing date using the **Intervalmatch** methods.

Example:

The entire Qlik Sense script looks like this:

```
Let vBeginTime = Num('1/1/2013');
Let vEndTime = Num('1/3/2013');
Let vEpsilon = Pow(2,-27);

In_Rates:
LOAD * Inline [
Currency,Change Date,Rate
EUR,,8.59
EUR,28/01/2013,8.69
EUR,15/02/2013,8.45
USD,,6.50
USD,10/01/2013,6.56
USD,03/02/2013,6.30
];

Tmp_Rates:
LOAD Currency, Rate,
      Date(If(IsNum([Change Date]), [Change Date], $(#vBeginTime))) as FromDate
Resident In_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;
```

4 Data cleansing

There are times when the source data that you load into Qlik Sense is not necessarily how you want it in the Qlik Sense app. Qlik Sense provides a host of functions and statements that allow us to transform our data into a format that works for us.

Mapping can be used in a Qlik Sense script to replace or modify field values or names when the script is run, so mapping can be used to clean up data and make it more consistent or to replace parts or all of a field value.

When you load data from different tables, field values denoting the same thing are not always consistently named. Since this lack of consistency hinders associations, the problem needs to be solved. This can be done in an elegant way by creating a mapping table for the comparison of field values.

4.1 Mapping tables

Tables loaded via **Mapping load** or **Mapping select** are treated differently from other tables. They are stored in a separate area of the memory and used only as mapping tables when the script is run. After the script is run these tables are automatically dropped.

Rules:

- A mapping table must have two columns, the first one containing the comparison values and the second the desired mapping values.
- The two columns must be named, but the names have no relevance in themselves. The column names have no connection to field names in regular internal tables.

4.2 Using Mapping

The following mapping functions/statements will be addressed in this tutorial:

- **Mapping** prefix
- **ApplyMap()**
- **MapSubstring()**
- **Map ... Using** statement
- **Unmap** statement

Mapping prefix

The **Mapping** prefix is used in a script to create a mapping table. The mapping table can then be used with the **ApplyMap()** function, the **MapSubstring()** function or the **Map ... Using** statement.

Do the following:

1. Create a new app and give it a name.
2. Open the **Data load editor** and create a new connection to the folder containing the tutorial sample files and give it the name *Tutorial Files*.
3. Enter the following script:

```
CountryMap:
MAPPING LOAD * INLINE [
Country, NewCountry
U.S.A., US
U.S., US
United States, US
United States of America, US
];
```

The CountryMap table stores two columns: Country and NewCountry. The Country column stores the various ways country has been entered in the Country field. The NewCountry column stores how the values will be mapped. This mapping table will be used to store consistent US country values in the Country field. For instance, if U.S.A. is stored in the Country field, map it to be US.


ApplyMap() function

ApplyMap() allows the user to replace data in a field based on a previously created mapping table. The mapping table need to be loaded before the **ApplyMap()** function can be used. In the sample file, *Data.xlsx*, data that includes people and the country they reside in is loaded. The raw data looks like this:

ID	Name	Country	Code
1	John Black	U.S.A.	SDFGBS1DI
2	Steve Johnson	U.S.	2ABC
3	Mary White	United States	DJY3DFE34
4	Susan McDaniels	u	DEF5556
5	Dean Smith	US	KSD111DKFJ1

In the table above, notice that the country is entered in various ways. In order to make the country field consistent, the mapping table is loaded and then the **ApplyMap()** function is used.

Do the following:

1. In the **Data load editor**, click  on the data connection you created in the previous section to select a file to load data from.
2. Select *Data.xlsx* and click **Select**.
3. Click **Insert script**.
4. Insert a line above the newly created **LOAD** statement and enter the following:

Data:

The script should now look like this:

```
CountryMap:
MAPPING LOAD * INLINE [
Country, NewCountry
```



```

U.S.A., US
U.S., US
United States, US
United States of America, US
];
Data:
LOAD
    ID,
    Name,
    Country,
    Code
FROM [lib://Tutorial Files/Data.xlsx]
(ooxml, embedded labels, table is Sheet1);

```

5. Modify the line containing Country, as follows:

```
ApplyMap('CountryMap', Country) as Country,
```

The script should now look like this:

```

CountryMap:
MAPPING LOAD * INLINE [
Country, NewCountry
U.S.A., US
U.S., US
United States, US
United States of America, US
];
Data:
LOAD
    ID,
    Name,
    ApplyMap('CountryMap', Country) as Country,
    Code
FROM 'lib://Tutorial Files/Data.xlsx'
(ooxml, embedded labels, table is Sheet1);

```

The first parameter of the **ApplyMap()** function has the map name enclosed in single quotes. The second parameter is the field that has the data that is to be replaced.

6. In the upper right corner, click **Load data**.

When you click **Load data**, the script is saved.

The resulting table looks like this:

ID	Name	Country	Code
1	John Black	US	SDFGBS1DI
2	Steve Johnson	US	2ABC
3	Mary White	US	DJY3DFE34
4	Susan McDaniels	u	DEF5556
5	Dean Smith	US	KSD111DKFJ1

Use the Preview section of the data model viewer to view the resulting table.

The various spellings of the United States have all been changed to US. There is one record that was not spelled correctly so the **ApplyMap()** function did not change that field value. Using the **ApplyMap()** function, you can use the third parameter to add a default expression if the mapping table does not have a matching value.

7. Add 'us' as the third parameter of the **ApplyMap()** function, to handle such cases when the country may have been entered incorrectly:

```
ApplyMap('CountryMap', Country, 'US') as Country,
```

The script should now look like this:

```
CountryMap:
MAPPING LOAD * INLINE [
Country, NewCountry
U.S.A., US
U.S., US
United States, US
United States of America, US
];
Data:
LOAD
    ID,
    Name,
    ApplyMap('CountryMap', Country, 'US') as Country,
    Code
FROM [lib://Tutorial Files/Data.xlsx]
(ooxml, embedded labels, table is Sheet1);
```

8. Click **Load data**.

Now the loaded data will look like this:

ID	Name	Country	Code
1	John Black	US	SDFGBS1DI
2	Steve Johnson	US	2ABC
3	Mary White	US	DJY3DFE34
4	Susan McDaniels	US	DEF5556
5	Dean Smith	US	KSD111DKFJ1

MapSubstring() function

The **MapSubstring()** function allows you to map parts of a field.

In the table created by **ApplyMap()** we now want the numbers to be written as text, so the **MapSubstring()** function will be used to replace the numeric data with text.

In order to do this a mapping table first needs to be created.

Do the following:

1. In the **Data load editor**, add the following script lines at the end of the CountryMap section, but before the Data section.

The script should look like this:

```
CountryMap:
MAPPING LOAD * INLINE [
Country, NewCountry
U.S.A., US
U.S., US
United States, US
United States of America, US
];
CodeMap:
MAPPING LOAD * INLINE [
F1, F2
1, one
2, two
3, three
4, four
5, five
11, eleven
];
Data:
LOAD
    ID,
    Name,
    ApplyMap('CountryMap', Country, 'US') as Country,
    Code
FROM [lib://Tutorial Files/Data.xlsx]
(ooxml, embedded labels, table is Sheet1);
```

In the CodeMap table, the numbers 1 through 5, and 11 are mapped.

2. In the Data section of the script modify the Code statement as follows:

```
MapSubString('CodeMap', Code) as Code
```

The Data section of the script should now look like this:

```
Data:
LOAD
    ID,
    Name,
    ApplyMap('CountryMap', Country, 'US') as Country,
    MapSubString('CodeMap', Code) as Code
FROM 'lib://Tutorial Files/Data.xlsx'
(ooxml, embedded labels, table is Sheet1);
```

3. Click **Load data**.

Now let us take a look at the results of the **MapSubString()** function. The loaded data now looks like this:

ID	Name	Country	Code
1	John Black	US	SDFGBSoneDI
2	Steve Johnson	US	twoABC
3	Mary White	US	DJYthreeDFEthreefour
4	Susan McDaniels	US	DEffivefivefive6
5	Dean Smith	US	KSDelevenoneDKFJone

Use the **Preview** section of the data model viewer to view the resulting table.

The numeric characters were replaced with text in the Code field. If a number appears more than once as it does for ID=3, and ID=4, the text is also repeated. ID=4, Susan McDaniels had a 6 in her code. Since 6 was not mapped in the CodeMap table, it remains unchanged. ID=5, Dean Smith, had 111 in his code. This has been mapped as 'elevenone'.

Map ... Using

The **Map ... Using** statement can also be used to apply a map to a field but it works a little differently than **ApplyMap()**. While **ApplyMap()** handles the mapping every time the field name is encountered, **Map ... using** handles the mapping when the value is stored under the field name in the internal table.

Let's take a look at an example. Assume we were loading the Country field multiple times in the script and wanted to apply a map every time the field was loaded. The **ApplyMap()** function could be used as illustrated earlier in this tutorial or **Map ... Using** can be used.

If **Map ... Using** is used then the map is applied to the field when the field is stored in the internal table. So in the example below, the map is applied to the Country field in the Data1 table but it would not be applied to the Country2 field in the Data2 table. This is because the **Map ... using** statement is only applied to fields named Country. When the Country2 field is stored to the internal table it is no longer named Country. If you want the map to be applied to the Country2 table then you would need to use the **ApplyMap()** function.

```
Map Country Using CountryMap;
Data1:
LOAD
ID,
Name,
Country
FROM 'lib://Tutorial Files/Data.xlsx'
(ooxml, embedded labels, table is Sheet1);
Data2:
LOAD
ID,
Country as Country2
FROM 'lib://Tutorial Files/Data.xlsx'
(ooxml, embedded labels, table is Sheet1);
UNMAP;
```

Unmap

The **Unmap** statement ends the **Map ... Using** statement so if Country were to be loaded after the **Unmap** statement, the CountryMap would not be applied.

5 Handling hierarchical data

Hierarchies are an important part of all business intelligence solutions, used to describe dimensions that naturally contain different levels of granularity. Some are simple and intuitive whereas others are complex and demand a lot of thinking to be modeled correctly.

From the top of a hierarchy to the bottom, the members are progressively more detailed. For example, in a dimension that has the levels Market, Country, State and City, the member Americas appears in the top level of the hierarchy, the member U.S.A. appears in the second level, the member California appears in the third level and San Francisco in the bottom level. California is more specific than U.S.A., and San Francisco is more specific than California.

Storing hierarchies in a relational model is a common challenge with multiple solutions. There are several approaches:

- The Horizontal hierarchy
- The Adjacency list model
- The Path enumeration method
- The Nested sets model
- The Ancestor list

For the purposes of this tutorial we will be creating an Ancestor list since it presents the hierarchy in a form that is directly usable in a query. Further information on the other approaches can be found in the Qlik Community.


5.1 Hierarchy prefix

The **Hierarchy** prefix is a script command that you put in front of a **LOAD** or **SELECT** statement that loads an adjacent nodes table. The **LOAD** statement needs to have at least three fields: An ID that is a unique key for the node, a reference to the parent and a name.

```
Hierarchy (NodeID, ParentID, NodeName)
LOAD      NodeID,
          ParentID,
          NodeName
```

The prefix will transform a loaded table into an expanded nodes table; a table that has a number of additional columns; one for each level of the hierarchy.

Do the following:

1. Create a new app and give it a name.
2. Open the data load editor and create a new connection to the folder containing the tutorial sample files and give it the name *Tutorial Files*.
3. In the **Data load editor**, click  on the data connection to select a file to load data from.
4. Select *Winedistricts.txt* and click **Select**.

5. Uncheck the Lbound and RBound fields so they are not loaded.

6. Click **Insert script**.

The loaded script should look like this:

```
LOAD
    NodeID,
    ParentID,
    NodeName
FROM [lib://Tutorial Files/winedistricts.txt]
(txt, utf8, embedded labels, delimiter is '\t', msq);
```

7. Insert a new line above the **LOAD** statement, and enter the following:

```
Hierarchy (NodeID, ParentID, NodeName)
```

8. Click **Load data**.

The loaded table should look like this:

NodeID	ParentID	NodeName	NodeName1	NodeName2	NodeName3	NodeName4
588	178	Ukraine	The World	Europe	Ukraine	-
591	178	United Kingdom	The World	Europe	United Kingdom	-
600	599	Australia	The World	Oceania	Australia	-
671	599	New Zealand	The World	Oceania	New Zealand	-
5	4	Atlas Mountains	The World	Africa	Morocco	Atlas Mountains
7	6	Breede River Valley	The World	Africa	South Africa	Breede River Valley

Use the **Preview** section of the data model viewer to view the resulting table.

The resulting expanded nodes table has exactly the same number of records as its source table: One per node. The expanded nodes table is very practical since it fulfills a number of requirements for analyzing a hierarchy in a relational model:

- All the node names exist in one and the same column, so that this can be used for searches.
- In addition, the different node levels have been expanded into one field each; fields that can be used in drill-down groups.
- It can be made to contain a path unique for the node, listing all ancestors in the right order.
- It can be made to contain the depth of the node, i.e. the distance from the root.

5.2 HierarchyBelongsTo prefix

Just as the **Hierarchy** prefix, the **HierarchyBelongsTo** is a script command that you put in front of a **LOAD** or **SELECT** statement that loads an adjacent nodes table:

```
HierarchyBelongsTo (NodeID, ParentID, NodeName, BelongsToID, BelongsTo)
LOAD
    NodeID,
    ParentID,
    NodeName
```

Also here, the **LOAD** statement needs to have at least three fields: An ID that is a unique key for the node, a reference to the parent and a name. The prefix will transform the loaded table into an ancestor table, a table that has every combination of an ancestor and a descendant listed as a separate record. Hence, it is very easy to find all ancestors or all descendants of a specific node.

Do the following:

1. In the **Data load editor** modify the **Hierarchy** statement so that it reads as follows:
`HierarchyBelongsTo (NodeID, ParentID, NodeName, BelongsToID, BelongsTo)`
2. In the upper right corner, click **Load data**.

The loaded table should look like this:

NodeID	NodeName	BelongsToID	BelongsTo
1	The World	1	The World
2	Africa	2	Africa
2	Africa	1	The World
20	Americas	20	Americas
20	Americas	1	The World
158	Asia	158	Asia

Use the **Preview** section of the data model viewer to view the resulting table.

The ancestor table is very practical since it fulfills a number of requirements for analyzing a hierarchy in a relational model:

- If the node ID represents the single nodes, the ancestor ID represents the entire trees and sub-trees of the hierarchy.
- All the node names exist both in the role as nodes and in the role as trees, and both can be used for searches.
- It can be made to contain the depth difference between the node depth, and the ancestor depth, that is, the distance from the root of the sub-tree.

Authorization

It is not uncommon that a hierarchy is used for authorization. One example is an organizational hierarchy. Each manager should obviously have the right to see everything pertaining to their own department, including all its sub-departments. But they should not necessarily have the right to see other departments.



This means that different people will be allowed to see different sub-trees of the organization. The authorization table may look like the following:

ACCESS	NTNAME	Person	Position	Permissions
USER	ACME\JRL	John	CPO	HR
USER	ACME\CAH	Carol	CEO	CEO
USER	ACME\JER	James	Director Engineering	Engineering
USER	ACME\DBK	Diana	CFO	Finance
USER	ACME\RNL	Bob	COO	Sale & Marketing
USER	ACME\LFD	Larry	CTO	Product

In this case, Carol is allowed to see everything pertaining to the CEO and below; Larry is allowed to see the Product organization; and James is allowed to see the Engineering organization only.

Further information on how to work with hierarchical data is available at community.qlik.com

6 QVD files

QVD files are one of the recognized types of files that can be used as data connections.

6.1 Working with QVD files

A QVD (QlikView Data) file is a file containing a table of data exported from Qlik Sense or QlikView. QVD is a native Qlik format and can only be written to and read by Qlik Sense or QlikView. The file format is optimized for speed when reading data from a Qlik Sense script but it is still very compact. Reading data from a QVD file is typically 10-100 times faster than reading from other data sources.

QVD files can be read in two modes: standard (fast) and optimized (faster). The selected mode is determined automatically by the Qlik Sense script engine. Optimized mode can be utilized only when all loaded fields are read without any transformations (formulas acting upon the fields), although renaming of fields is allowed. A **Where** clause causing Qlik Sense to unpack the records will also disable the optimized load.

A QVD file holds exactly one data table and consists of three parts:

- An XML header (in UTF-8 char set) describing the fields in the table, the layout of the subsequent information and some other metadata.
- Symbol tables in a byte-stuffed format.
- Actual table data in a bit-stuffed format.

QVD files can be used for many purposes. Four major uses can be easily identified. More than one may apply in any given situation:

- Increasing data load speed
By buffering non-changing or slowly-changing blocks of input data in QVD files, script execution becomes considerably faster for large data sets.
- Decreasing load on database servers
The amount of data fetched from external data sources can also be greatly reduced. This reduces the workload on external databases and network traffic. Furthermore, when several Qlik Sense scripts share the same data, it is only necessary to load it once from the source database into a QVD file. The other applications can make use of the same data through this QVD file.
- Consolidating data from multiple Qlik Sense applications.
With the **Binary** script statement it is possible to load data from only one single Qlik Sense application into another one, but with QVD files a Qlik Sense script can combine data from any number of Qlik Sense applications. This opens up possibilities for applications consolidating similar data from different business units etc.
- Incremental load
In many common cases the QVD functionality can be used for facilitating incremental load by exclusively loading new records from a growing database.

6.2 Creating QVD files

A QVD file can be created in two ways:

- Explicit creation and naming using the **Store** command in the Qlik Sense script.
State in the script that a previously-read table, or part thereof, is to be exported to an explicitly-named file at a location of your choice.
- Automatic creation and maintenance from script.
By preceding a load or select statement with the **Buffer** prefix, Qlik Sense will automatically create a QVD file, which under certain conditions, can be used instead of the original data source when reloading data.

There is no difference between the resulting QVD files, with regard to reading speed.

Store

This script function creates a QVD or a CSV file.

Syntax:

```
Store[ *fieldlist from] table into filename [ format-spec ];
```

The statement will create an explicitly named QVD or CSV file. The statement can only export fields from one data table. If fields from several tables are to be exported, an explicit join must be made previously in the script to create the data table that should be exported.

The text values are exported to the CSV file in UTF-8 format. A delimiter can be specified, see **LOAD**. The store statement to a CSV file does not support BIFF export.

Examples:

```
Store mytable into xyz.qvd (qvd);
Store * from mytable into xyz.qvd;
Store Name, RegNo from mytable into xyz.qvd;
Store Name as a, RegNo as b from mytable into xyz.qvd;
store mytable into myfile.txt (txt);
store * from mytable into 'lib://FolderConnection/myfile.qvd';
```

Do the following:


1. In the app, *Advanced Scripting tutorial*, open the **Data load editor**.
2. In the script section select the section *Product*.

The script should look like this:

```
CrossTable(Month, Sales)
LOAD
    Product,
    "Jan 2014",
    "Feb 2014",
```

```
"Mar 2014",  
"Apr 2014",  
"May 2014",  
"Jun 2014"  
FROM 'lib://Tutorial Files/Product.xlsx'  
(ooxml, embedded labels, table is Product);
```

3. Add a new line at the end of the script. For this tutorial we will take the last example above, modified for the *Product* script:

```
store * from Product into 'lib://Tutorial Files/Product.qvd';
```
4. In the upper right corner, click **Load data**.
When you click **Load data**, the script is saved.
5. Click  on the *Tutorial files* data connection to view the available files. The *Product.qvd* file should now be in the list of files.

This data file is the result of the **Crosstable** script and is a three-column table, one column for each category (Product, Month, Sales). This data file could now be used to replace the entire *Product* script section.

Buffer

QVD files can be created and maintained automatically via the **Buffer** prefix. This prefix can be used on most **LOAD** and **SELECT** statements in script. It indicates that QVD files are used to cache/buffer the result of the statement.

Syntax:

```
Buffer [ (option [ , option])] ( loadstatement | selectstatement )  
option::= incremental | stale [after] amount [(days | hours)]
```

If no option is used, the QVD buffer created by the first execution of the script will be used indefinitely.

Example 1:

```
Buffer select * from MyTable;
```

incremental

The **incremental** option enables the ability to read only part of an underlying file. The previous size of the file is stored in the XML header in the QVD file. This is particularly useful with log files. All records loaded at a previous occasion are read from the QVD file whereas the following new records are read from the original source and finally an updated QVD file is created.

Example 2:

```
Buffer (stale after 7 days) select * from MyTable;
```

Note that the **incremental** option can only be used with **LOAD** statements and text files and that incremental load cannot be used where old data is changed or deleted!

stale [after] amount [(days | hours)]

Amount is a number specifying the time period. Decimals may be used. The unit is assumed to be days if omitted.

The **stale after** option is typically used with database sources where there is no simple timestamp on the original data. A **stale after** clause simply states a time period from the creation time of the QVD buffer after which it will no longer be considered valid. Before that time the QVD buffer will be used as source for data and after that the original data source will be used. The QVD buffer file will then automatically be updated and a new period starts.

Example 3:

```
Buffer (incremental) load * from MyLog.log;
```

QVD buffers will normally be removed when no longer referenced anywhere throughout a complete script execution in the app that created it or when the app that created it no longer exists. The **Store** statement should be used if you wish to retain the contents of the buffer as a QVD or CSVfile.

6.3 Reading data from QVD files

A QVD file can be read into or accessed by Qlik Sense by the following methods:

- Loading a QVD file as an explicit data source. QVD files can be referenced by a load statement in the Qlik Sense script just like any other type of text files (csv, fix, dif, biff, and so on).

Example:

```
LOAD * from xyz.qvd (qvd);  
LOAD Name, RegNo from xyz.qvd (qvd);  
LOAD Name as a, RegNo as b from xyz.qvd (qvd);
```

- Automatic loading of buffered QVD files. When using the buffer prefix on load or select statements, no explicit statements for reading are necessary. Qlik Sense will determine the extent to which it will use data from the QVD file as opposed to acquiring data using the original **LOAD** or **SELECT** statement.
- Accessing QVD files from the script. A number of script functions (all beginning with QVD) can be used for retrieving various information on the data found in the XML header of a QVD file.

6.4 Thank you!

Now you have finished this tutorial, and hopefully you have gained some more knowledge about scripting in Qlik Sense. Please visit our website for more information on the further training available.