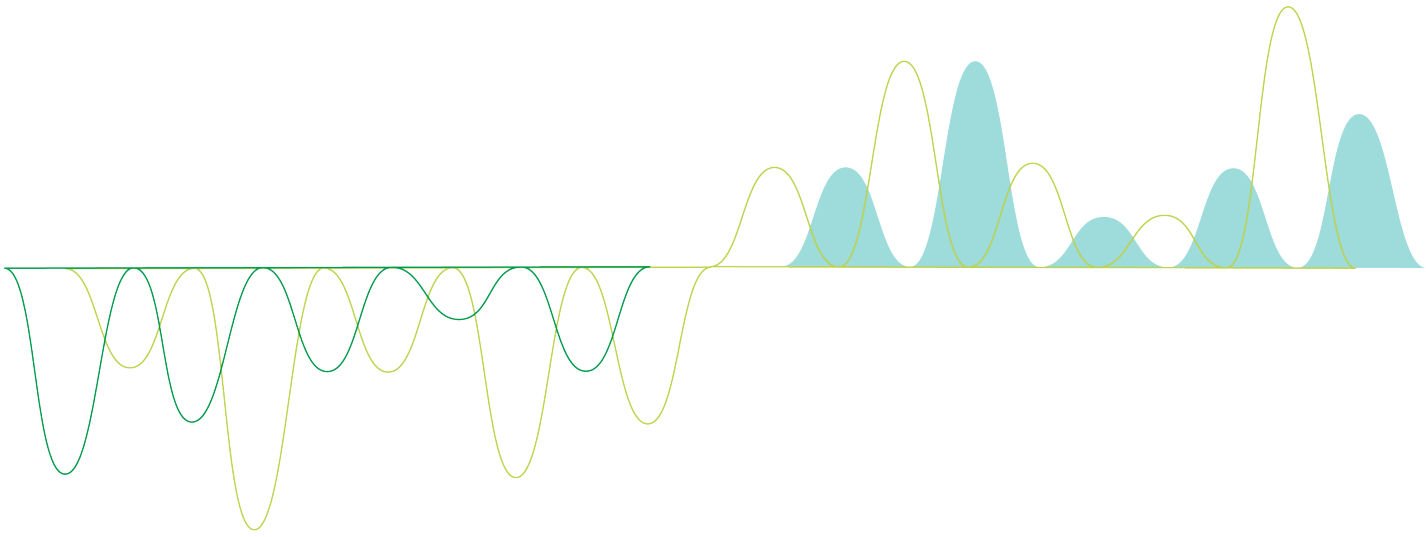
Tutorial - Next Steps in Scripting

Qlik Sense®
February 2023
Copyright © 1993-2023 QlikTech International AB. All rights reserved.



© 2023 QlikTech International AB. All rights reserved. All company and/or product names may be trade names, trademarks and/or registered trademarks of the respective owners with which they are associated.

Contents

1 Welcome to this tutorial! 5
1.1 What you will learn 5
1.2 Who should take this course 5
1.3 Package contents 5
1.4 Lessons in this tutorial 6
1.5 Further reading and resources 6
2 LOAD and SELECT statements 7
3 Transforming data 8
3.1 Using the Crosstable prefix 8
Crosstable prefix 8
Clearing the memory cache 12
3.2 Combining tables with Join and Keep 12
Join 13
Using Join 13
Keep 16
Inner 16
Left 18
Right 19
3.3 Using inter-record functions: Peek, Previous, and Exists 20
Peek() 20
Previous() 21
Exists() 21
Using Peek() and Previous() 21
Using Exists() 24
3.4 Matching intervals and iterative loading 27
Using the IntervalMatch() prefix 28
Using a While loop and iterative loading IterNo() 30
Open and closed intervals 31
4 Data cleansing 33
4.1 Mapping tables 33
Rules: 33
4.2 Mapping functions and statements 33
4.3 Mapping prefix 33
4.4 ApplyMap() function 34
4.5 MapSubstring() function 36
4.6 Map … Using 38
5 Handling hierarchical data 40
5.1 Hierarchy prefix 40
5.2 HierarchyBelongsTo prefix 41
Authorization 42
6 QVD files 45
6.1 Creating QVD files 45
Store 46
6.2 Reading data from QVD files 47
Buffer 48

6.3 Thank you! 51

1 Welcome to this tutorial!

Welcome to this tutorial, which will introduce you to advanced scripting in Qlik Sense.

Once you are familiar with the basics of scripting, you can start to perform more sophisticated operations on your data as you load it into Qlik Sense. This can include, for example, transforming data using cross-tables, cleansing data, and creating and loading data from Qlik data files known as QVD files.

1.1 What you will learn

After completing this tutorial, you should be comfortable with loading data using some of the more advanced scripting functions in Qlik Sense.

1.2 Who should take this course

You should be familiar with the basics of scripting in Qlik Sense. That is, you have loaded data and manipulated data using scripts.

If you have not already done so, we recommend completing the Scripting for beginners tutorial.

You require access to the data load editor and should be allowed to load data in Qlik Sense Enterprise on Windows.

The instructions also apply generally for Qlik Sense Cloud Business.

1.3 Package contents

The zip package that you downloaded contains the following data files that you need to complete the tutorial:

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

The package also contains a copy of the *Advanced Scripting Tutorial* app. Additional script sections in the app contain the scripts for the other apps that you create in this tutorial. You can upload the app to your hub.

We recommend building the app yourself as described in the tutorial to maximize your learning. Additionally, you would have to upload and connect to your data files as described in the tutorial for the data loads to work.

However, if you run into problems, the app may help you troubleshoot. We have indicated which script segments are associated with each lesson.

1.4 Lessons in this tutorial

Depending on your experience with Qlik Sense, this tutorial should take 3-4 hours to complete. The topics are designed to be completed in sequence. However, you can step away and return at any time. There are, mercifully, no tests.

Transforming data

Using the Crosstable prefix

Combining tables with Join and Keep

Using inter-record functions: Peek, Previous, and Exists

Matching intervals and iterative loading

Data cleansing

Handling hierarchical data

QVD files

1.5 Further reading and resources

- Qlik offers a wide variety of resources when you want to learn more.
- Qlik online help is available.
- Training, including free online courses, is available in the Qlik Continuous Classroom.
- Discussion forums, blogs, and more can be found in Qlik Community.

2 LOAD and SELECT statements

You can load data into Qlik Sense using the LOAD and SELECT statements. Each of these statements generates an internal table. LOAD is used to load data from files, while SELECT is used to load data from databases.

In this tutorial, you will be using data from files, so you will be using LOAD statements.

You can also use a preceding LOAD to be able to manipulate the content of the data loaded. For example, renaming fields has to be done in a LOAD statement, whereas the SELECT statement does not permit any changes to field names.

The following rules apply when loading data into Qlik Sense:

- Qlik Sense does not differentiate between tables generated by a LOAD or a SELECT statement. This means that if several tables are loaded, it does not matter whether the tables are loaded by LOAD or SELECT statements or by a mix of the two.
- The order of the fields in the statement or in the original table in the database is unimportant to the Qlik Sense logic.
- Field names are case sensitive and are used to establish associations among data tables. Due to this, at times it is necessary to rename fields in the load script to achieve a desired data model.

3 Transforming data

You can transform and manipulate data in the data load editor before using the data in your app.

One of the advantages of 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.

The following exercises will show you how to load data using the Crosstable prefix. 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, you can use the Crosstable prefix to transform the data and create the desired fields.

Crosstable prefix

In the following *Product* table you have one column per month and one row per product.

Product table

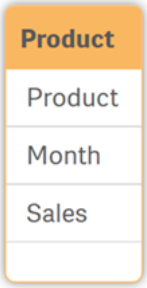
Product	Jan 2014	Feb 2014	Mar 2014	Apr 2014	May 2014	Jun 2014
A	100	98	100	83	103	82
B	284	279	297	305	294	292
C	50	53	50	54	49	51

When you load the table, the output is a table with one field for *Product* and one field for each of the months.

Product table with Product field, and one field each for the months

If you want to analyze this data, it is much easier to have all numbers in one field and all months in another. In this case, that is a three-column table with one column for each category (*Product, Month, Sales*).

Product table with Product, Month, and Sales fields



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.

Do the following:

1. Create a new app and call it *Advanced Scripting Tutorial*.
2. Add a new script section in the **Data load editor**.
3. Name the section *Product*.
4. Under **AttachedFiles** in the right menu, click **Select data**.
5. Upload and then select *Product.xlsx*.
6. Select the *Product* table in the **Select data from** window.

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

7. Click **Insert script**.

   Your script should look like this:

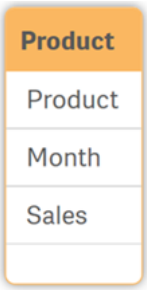
   ```
   LOAD
       Product,
       "Jan 2014",
       "Feb 2014",
       "Mar 2014",
       "Apr 2014",
       "May 2014",
       "Jun 2014"
   FROM [lib://AttachedFiles/Product.xlsx]
   (ooxml, embedded labels, table is Product);
   ```

8. Click **Load data**.
9. Open the **Data model viewer**. The data model looks like this:

Product table with Product field, and one field each for the months

10. Click the *Product* tab in the **Data load editor**.
11. Enter the following above the LOAD statement:
    ```
    CrossTable(Month, Sales)
    ```
12. Click **Load data**.
13. Open the **Data model viewer**. The data model looks like this:

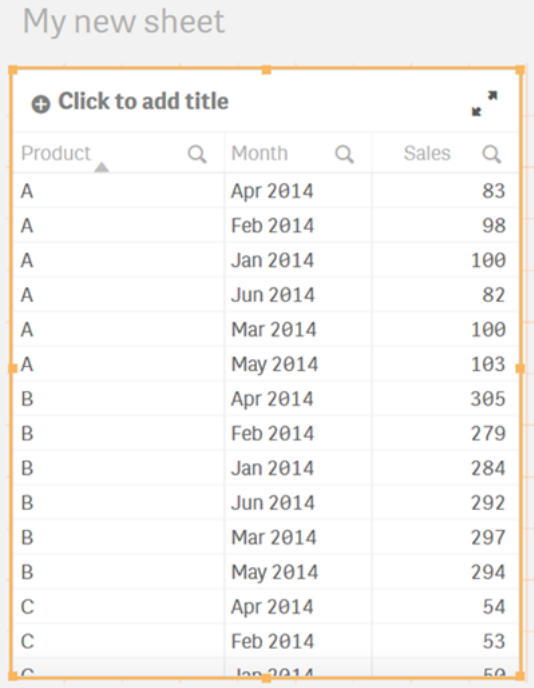
    Product table with Product, Month, and Sales fields

    Note that the input data typically 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. You cannot have a preceding LOAD or a prefix in front of the Crosstable keyword. However, you can use auto-concatenate.

    In a table in Qlik Sense, your data looks like this:

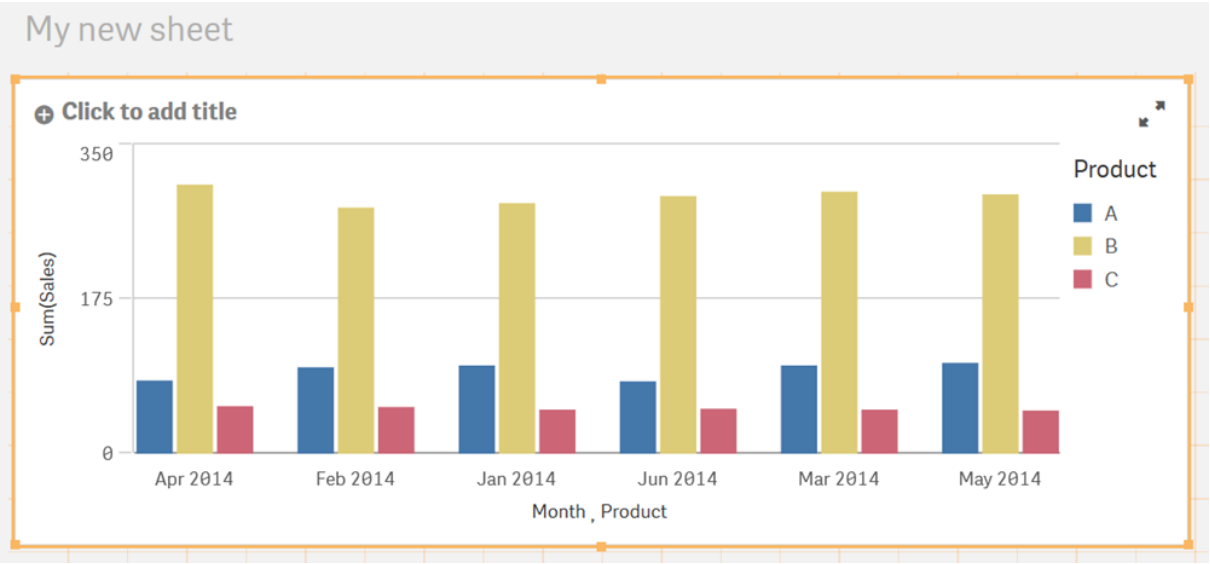
Table showing data loaded using Crosstable prefix



Product	Month	Sales
A	Apr 2014	83
A	Feb 2014	98
A	Jan 2014	100
A	Jun 2014	82
A	Mar 2014	100
A	May 2014	103
B	Apr 2014	305
B	Feb 2014	279
B	Jan 2014	284
B	Jun 2014	292
B	Mar 2014	297
B	May 2014	294
C	Apr 2014	54
C	Feb 2014	53

You can now, for example, create a bar chart using the data:

Bar chart showing data loaded using Crosstable prefix



To learn more about Crosstable, see this blog post in Qlik Community: The Crosstable Load. The behaviors are discussed in the context of QlikView. However, the logic applies equally to Qlik Sense.

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 shown in the following example.

Note that this is an example only. There are no accompanying exercises to be completed in Qlik Sense.

```
tmpData:
Crosstable (MonthText, Sales)
LOAD Product, [Jan 2014], [Feb 2014], [Mar 2014], [Apr 2014], [May 2014], [Jun 2014]
FROM ...

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

Clearing the memory cache

You can delete tables that you create to clear the memory cache. When you load into a temporary, as in the previous section, you should drop it when it is not needed anymore. For example:

```
DROP TABLE Table1, Table2, Table3, Table4;
DROP TABLES Table1, Table2, Table3, Table4;
```

You can also drop fields. For example:

```
DROP FIELD Field1, Field2, Field3, Field4;
DROP FIELDS Field1, Field2, Field3, Field4;
DROP FIELD Field1 from Table1;
DROP FIELDS Field1 from Table1;
```

As you can see, the keywords TABLE and FIELD and can be singular or plural.

3.2 Combining tables with Join and Keep

A join is an operation that uses 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.

It is possible to join tables already in the script. 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 need 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. Such joins can often result in very large tables.

Do the following:

1. Open the *Advanced Scripting Tutorial* app.
2. Add a new script section in the **Data load editor**.
3. Call the section *Transactions*.
4. Under **AttachedFiles** in the right menu, click **Select data**.
5. Upload and then select *Transactions.csv*.

   *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. In the **Select data from** window, click **Insert script**.

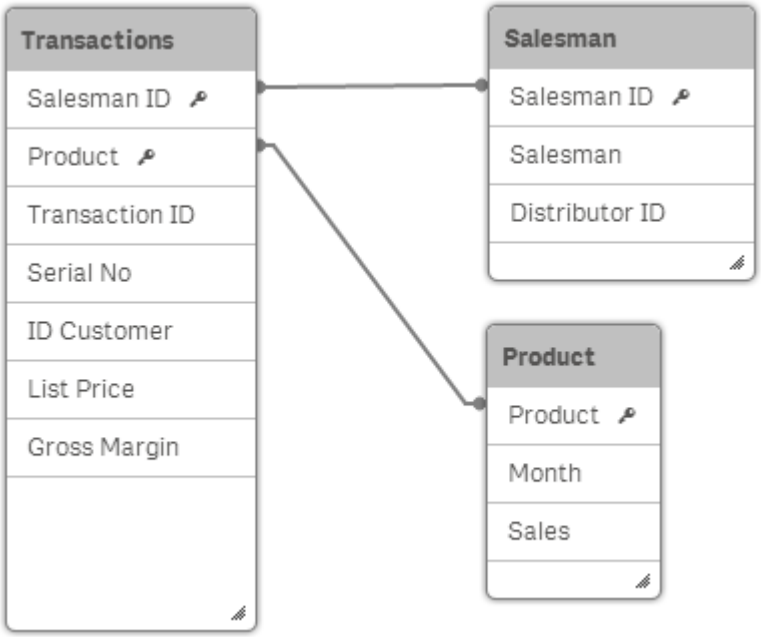
7. Upload and then select *Salesman.xlsx*.
8. In the **Select data from** window, click **Insert script**.

   Your script should look like this:

```
LOAD
    "Transaction ID",
    "Salesman ID",
    Product,
    "Serial No",
    "ID Customer",
    "List Price",
    "Gross Margin"
FROM [lib://AttachedFiles/Transactions.csv]
(txt, codepage is 28591, embedded labels, delimiter is ',', msq);

LOAD
    "Salesman ID",
    Salesman,
    "Distributor ID"
FROM [lib://AttachedFiles/Salesman.xlsx]
(ooxml, embedded labels, table is Salesman);
```

9. Click **Load data**.
10. Open the **Data model viewer**. The data model looks like this:

    Data model: Transactions, Salesman, and Product tables



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

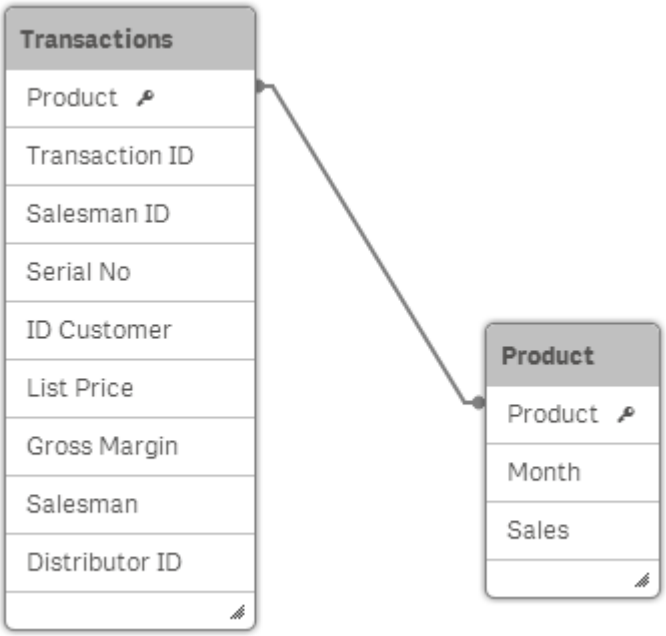
Do the following:

1. To set a name for the joined table, add the following line above the first LOAD statement:
   ```
   Transactions:
   ```
2. To join the *Transactions* and *Salesman* tables, add the following line above the second LOAD statement:
   ```
   Join(Transactions)
   ```

   Your script should look like this:

   ```
   Transactions:
   LOAD
       "Transaction ID",
       "Salesman ID",
       Product,
       "Serial No",
       "ID Customer",
       "List Price",
       "Gross Margin"
   FROM [lib://AttachedFiles/Transactions.csv]
   (txt, codepage is 28591, embedded labels, delimiter is ',', msq);

   Join(Transactions)
   LOAD
       "Salesman ID",
       Salesman,
       "Distributor ID"
   FROM [lib://AttachedFiles/Salesman.xlsx]
   (ooxml, embedded labels, table is Salesman);
   ```
3. Click **Load data**.
4. Open the **Data model viewer**. The data model looks like this:

Data model: Transactions and Product tables



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

To learn more about when to use Join, see these blog posts in Qlik Community: To Join or not to Join, Mapping as an Alternative to Joining. *The behaviors are discussed in the context of QlikView. However, the logic applies equally to Qlik Sense.*

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 reduces 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 data load script 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*.

Note that these are examples only. There are no accompanying exercises to be completed in Qlik Sense.

Table 1

A	B
1	aa
2	cc
3	ee

Table2

A	C
1	xx
4	yy

Inner Join

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 Keep

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

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

VTab1

A	B
1	aa

VTab2

A	C
1	xx

Left

The Join and Keep prefixes in the data load script 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

A	B
1	aa
2	cc
3	ee

Table2

A	C
1	xx
4	yy

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	-

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

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

VTab1

A	B
1	aa
2	cc
3	ee

VTab2

A	C
1	xx

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

A	B
1	aa
2	cc
3	ee

Table2

A	C
1	xx
4	yy

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

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

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

VTab1

A	B
1	aa

VTab2

A	C
1	xx
4	yy

3.3 Using inter-record functions: Peek, Previous, and Exists

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.

Peek()

Peek() returns the value of a field in a table for a row that has already been loaded. The row number can be specified, as can the table. If no row number is specified, the last previously loaded record will be used.

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 is fetched directly from the input source, making it possible to also refer 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. Here is a sample of the data in the *Employees* table that you will load in this exercise.

Sample of data from Employees table

Date	Hired	Terminated
1/1/2011	6	0
2/1/2011	4	2
3/1/2011	6	1
4/1/2011	5	2

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. Open the *Advanced Scripting Tutorial* app.
2. Add a new script section in the **Data load editor**.

3. Call the section *Employees*.
4. Under **AttachedFiles** in the right menu, click **Select data**.
5. Upload and then select *Employees.xlsx*.

   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. In the **Select data from** window, click **Insert script**.

   Your script should look like this:

   ```
   LOAD
       "Date",
       Hired,
       Terminated
   FROM [lib://AttachedFiles/Employees.xlsx]
   (ooxml, embedded labels, table is Sheet1);
   ```

7. 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://AttachedFiles/Employees.xlsx]
   (ooxml, embedded labels, table is Sheet1);
   ```

   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. In the 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 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 use 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.

8. Add the following to the end of your script:

   ```
   [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. In the 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.

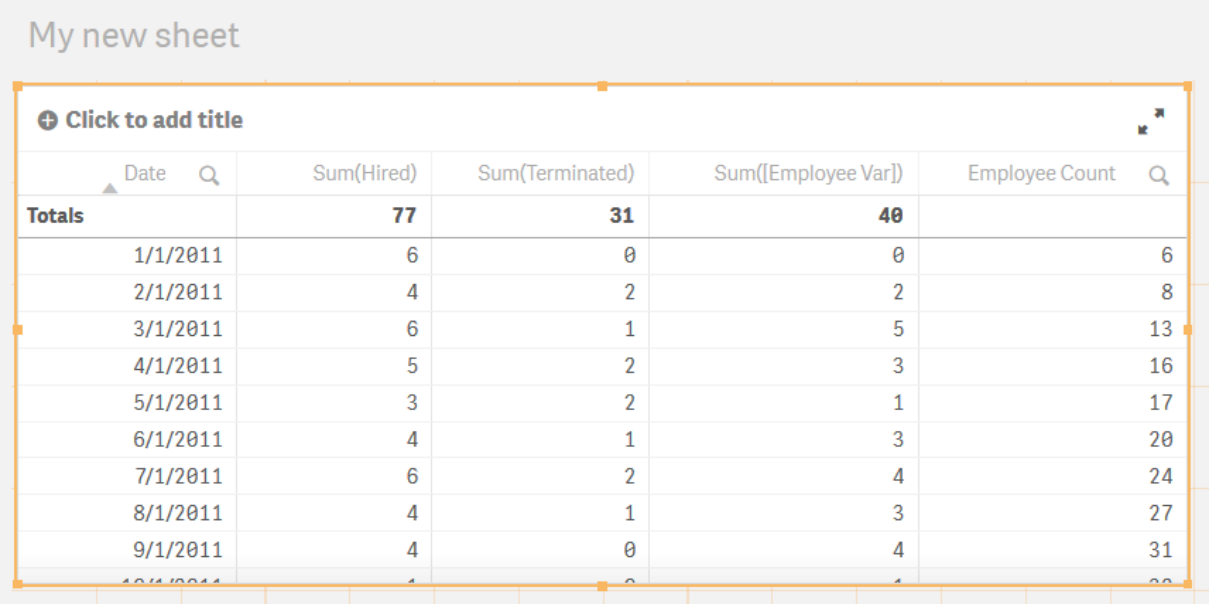
Your script should look like this:

```
[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://AttachedFiles/Employees.xlsx]
(ooxml, embedded labels, table is Sheet1);

[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. Click **Load data**.

   In a new sheet in the app overview, create a table using *Date, Hired, Terminated, Employee Count* and *Employee Var* as the columns of the table. The resulting table should look like this:

Table following use of Peek and Previous in script



Date	Sum(Hired)	Sum(Terminated)	Sum([Employee Var])	Employee Count
Totals	**77**	**31**	**40**	
1/1/2011	6	0	0	6
2/1/2011	4	2	2	8
3/1/2011	6	1	5	13
4/1/2011	5	2	3	16
5/1/2011	3	2	1	17
6/1/2011	4	1	3	20
7/1/2011	6	2	4	24
8/1/2011	4	1	3	27
9/1/2011	4	0	4	31

Peek() and Previous() allow you 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 when you need 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 when you are targeting a field that has not been previously loaded into the table, or when youo need 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 then added the difference between the hired and terminated employees for the current month. Remember that *Employee Count* was not a field in the original file

To learn more about when to use Peek() and Previous(), see this blog post in Qlik Community: Peek() vs Previous() – When to Use Each. The behaviors are discussed in the context of QlikView. However, the logic applies equally to Qlik Sense.

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 also use the Dual() function to assign numeric values to strings.

Do the following:

1. Create a new app and give it a name.
2. Add a new script section in the **Data load editor**.
3. Call the section *People*.
4. 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;
```

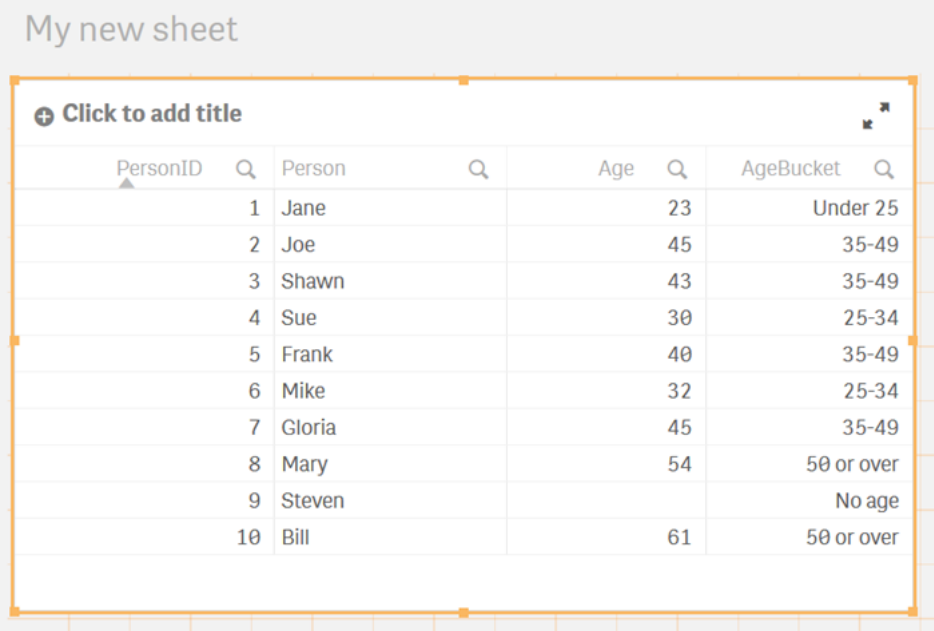
5. Click **Load data**.

   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 output of the script look like this:

   Table following use of Exists in script

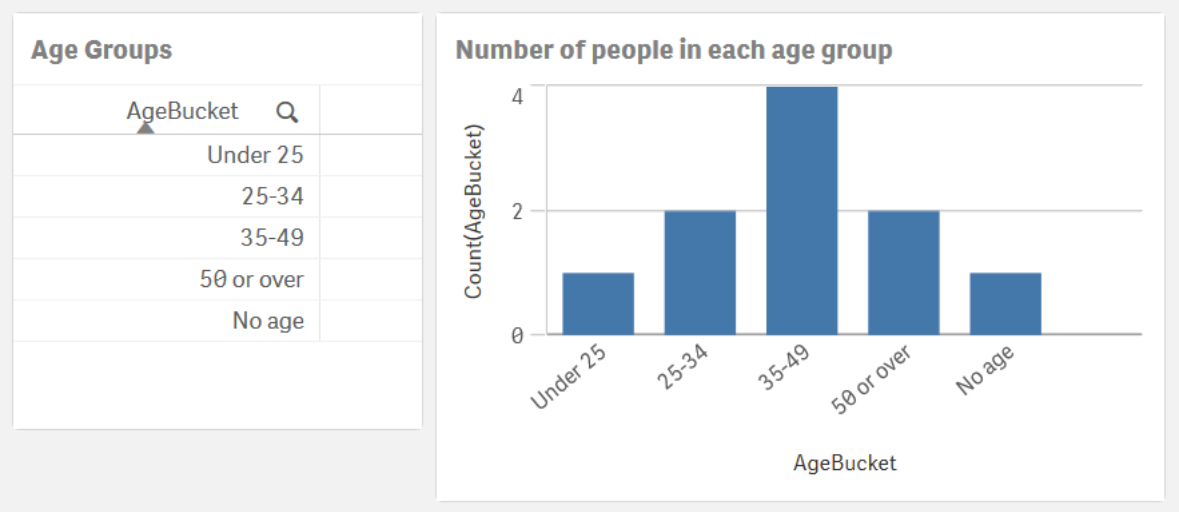


PersonID	Person	Age	AgeBucket
1	Jane	23	Under 25
2	Joe	45	35-49
3	Shawn	43	35-49
4	Sue	30	25-34
5	Frank	40	35-49
6	Mike	32	25-34
7	Gloria	45	35-49
8	Mary	54	50 or over
9	Steven		No age
10	Bill	61	50 or over

   If none of the *PersonID*s 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.

6. Create a new sheet and give it a name.

7. Open the new sheet and click **Edit sheet**.
8. Add a standard table to the sheet with the dimension *AgeBucket* and name the visualization *Age Groups*.
9. Add a bar chart to the sheet with the dimension *AgeBucket*, and the measure *Count([AgeBucket])*. Name the visualization *Number of people in each age group*.
10. Adjust the properties of the table and bar chart to your preference and then click **Done**.
    Your sheet should look similar to this:
    Sheet with groupings by age



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

In the script above 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.

To learn more about Exists() and Dual(), see this blog post in Qlik Community: Dual & Exists – Useful Functions

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.

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 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.

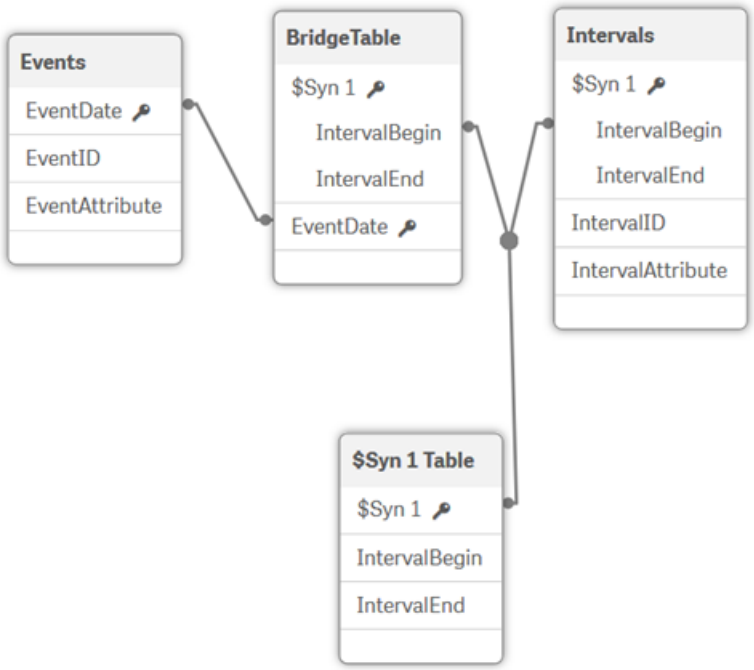
Do the following:

1. Create a new app and give it a name.
2. Add a new script section in the **Data load editor**.
3. Call the sections *Events*.
4. Under **AttachedFiles** in the right menu, click **Select data**.
5. Upload and then select *Events.txt*.
6. In the **Select data from** window, click **Insert script**.
7. Upload and then select *Intervals.txt*.
8. In the **Select data from** window, click **Insert script**.
9. In the script, name the first table *Events*, and name the second table *Intervals*.
10. At the end of the script add an IntervalMatch to create a third table that bridges the two first tables:
    ```
    BridgeTable:
    IntervalMatch (EventDate)
    LOAD distinct IntervalBegin, IntervalEnd
    Resident Intervals;
    ```
11. Your script should look like this:
    ```
    Events:
    LOAD
        EventID,
        EventDate,
        EventAttribute
    FROM [lib://AttachedFiles/Events.txt]
    (txt, utf8, embedded labels, delimiter is '\t', msq);

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

    BridgeTable:
    ```

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

12. Click **Load data**.
13. Open the **Data model viewer**. The data model looks like this:

    Data model: Events, BridgeTable, Intervals, and $Syn1 tables



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.

To learn more about IntervalMatch(), see this blog post in Qlik Community: Using IntervalMatch()

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. For example:

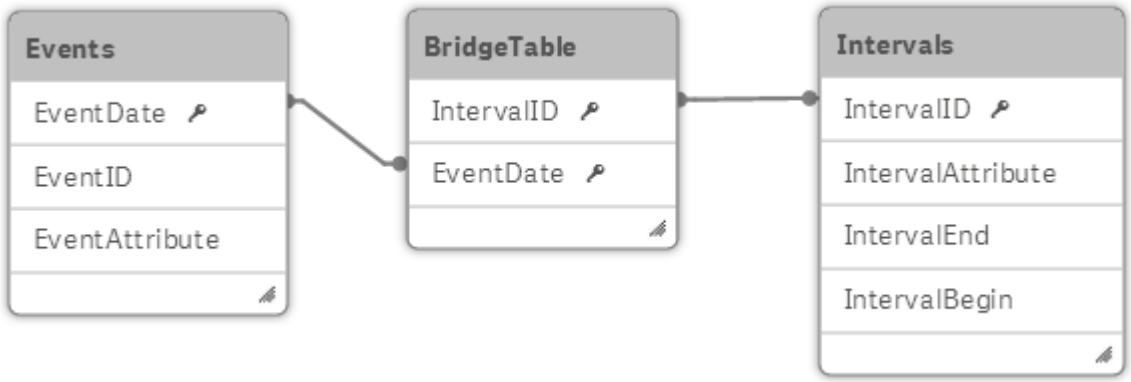
```
LOAD Date, IterNo() as Iteration From … While IterNo() <= 4;
```

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:

Do the following:

1. Replace the existing Bridgetable statements with the following script:
   ```
   BridgeTable:
   LOAD distinct * Where Exists(EventDate);
   LOAD IntervalBegin + IterNo() - 1 as EventDate, IntervalID
       Resident Intervals
       While IntervalBegin + IterNo() - 1 <= IntervalEnd;
   ```
2. Click **Load data**.
3. Open the **Data model viewer**. The data model looks like this:
   Data model: Events, BridgeTable, and Intervals tables

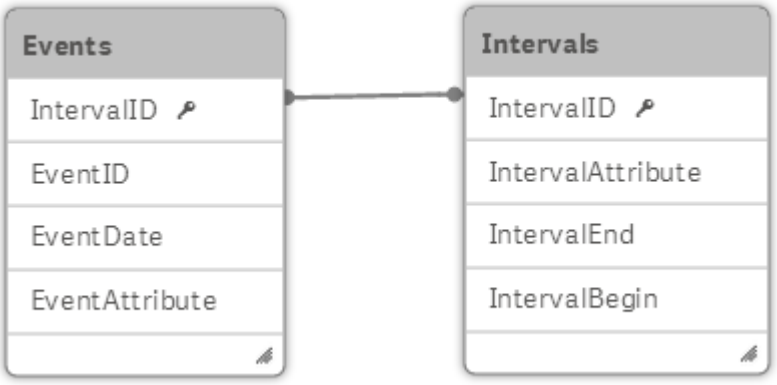
   

   Generally, the solution with three tables is the best one, because it allows for a many to many relationship between intervals and events. But a common situation is that you know that an event can only belong to one single interval. In this case, the bridge table is really not necessary. The *IntervalID* 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 end of your script:

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

Drop Table BridgeTable;
```

5. Click **Load data**.
6. Open the **Data model viewer**. The data model looks like this:

   Data model: Events and Intervals tables



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;
```

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 too small of an amount, the difference will not be significant and you will be back using the original number.

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 you 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 Mapping functions and statements

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

• Mapping prefix

• ApplyMap()

• MapSubstring()

• Map … Using statement

• Unmap statement

4.3 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. Add a new script section in the **Data load editor**.
3. Call the section *Countries*.
4. 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*.

4.4 ApplyMap() function

Use ApplyMap() 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. The data in the *Data.xlsx* table that you will load looks like this:

Data table

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

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. Beneath the script you entered above, select and load *Data.xlsx*, and then insert the script.
2. Enter the following above the newly created LOAD statement:

```
Data:
```

   Your script should look like this:

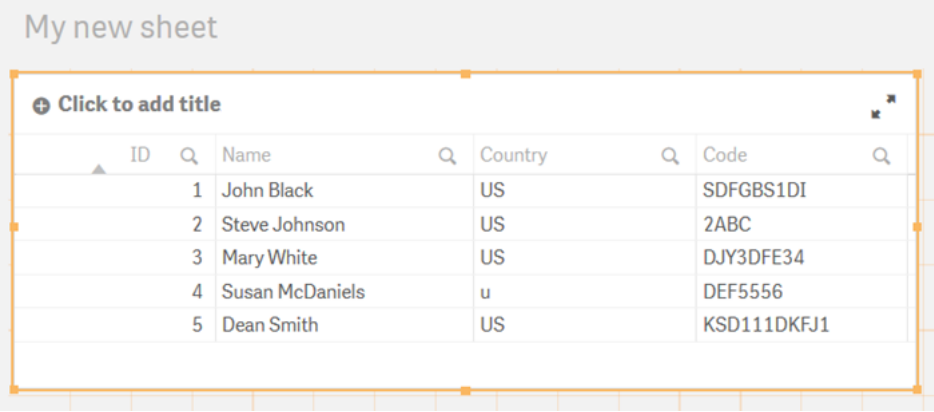
```
CountryMap:
MAPPING LOAD * INLINE [
    Country, NewCountry
    U.S.A., US
    U.S., US
    United States, US
    United States of America, US
];

Data:
LOAD
    ID,
    Name,
    Country,
    Code
FROM [lib://AttachedFiles/Data.xlsx]
(ooxml, embedded labels, table is Sheet1);
```

3. Modify the line containing `Country`, as follows:
   `ApplyMap('CountryMap', Country) as Country,`

   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.

4. Click **Load data**.
   The resulting table looks like this:

   Table showing data loaded using ApplyMap() function



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

   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.

5. 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,`

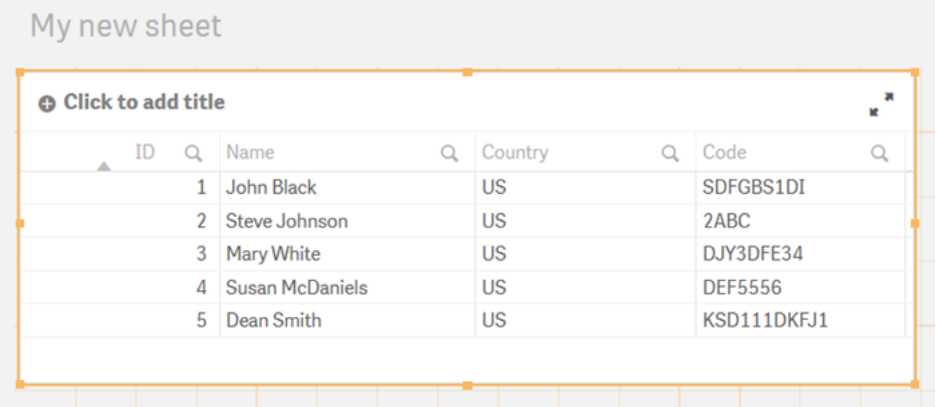
   Your script should look like this:

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

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

6. Click **Load data**.

   The resulting table looks like this:

   Table showing data loaded using ApplyMap function



To learn more about ApplyMap(), see this blog post in Qlik Community: Don't join - use Applymap instead

4.5 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. Add the following script lines after the *CountryMap* section, but before the *Data* section.

   ```
   CodeMap:
   MAPPING LOAD * INLINE [
   F1, F2
   1, one
   2, two
   3, three
   4, four
   5, five
   11, eleven
   ];
   ```

   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
   ```

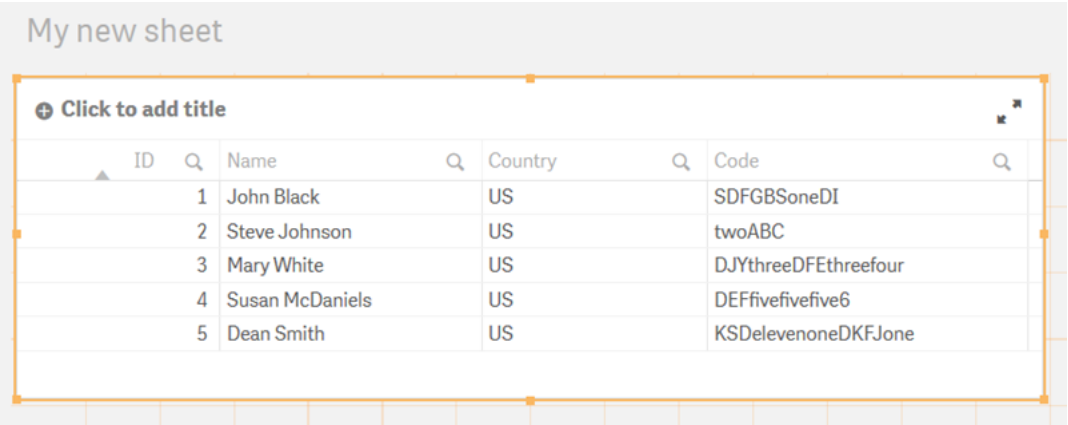
   Your script should look like this:

   ```
   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,
       MapSubString('CodeMap', Code) as Code
   FROM [lib://AttachedFiles/Data.xlsx]
   (ooxml, embedded labels, table is Sheet1);
   ```

3. Click **Load data**.

   The resulting table looks like this:

Table showing data loaded using MapSubString function



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

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'.

To learn more about MapSubstring(), see this blog post in Qlik Community: Mapping … and not the geographical kind

4.6 Map … Using

The Map … Using statement can also be used to apply a map to a field. However, 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.

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.

Do the following:

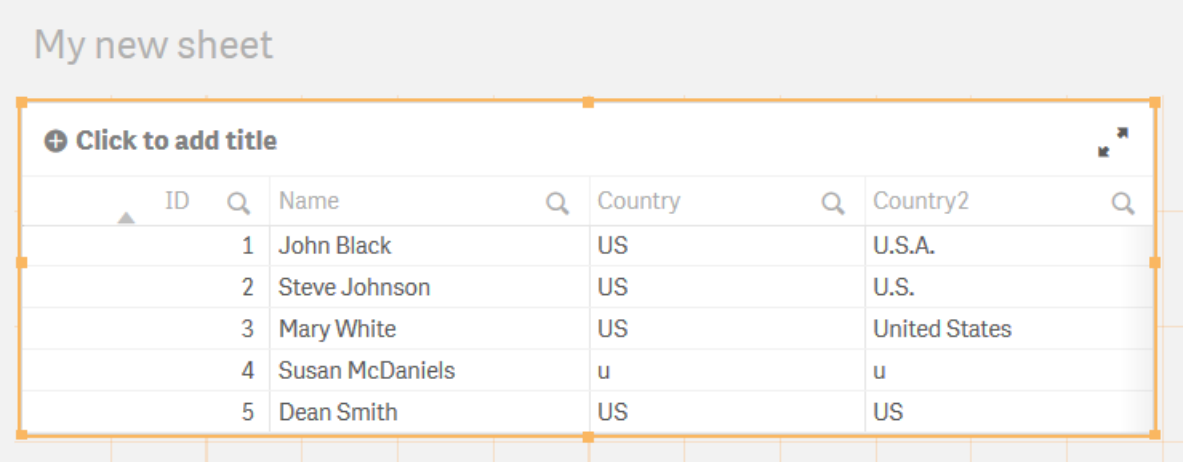
1. Replace the script for the *Data* table with the following:

```
Map Country Using CountryMap;
Data1:
    LOAD
    ID,
    Name,
    Country
FROM [lib://AttachedFiles/Data.xlsx]
(ooxml, embedded labels, table is Sheet1);

Data2:
LOAD
    ID,
    Country as Country2
FROM [lib://AttachedFiles/Data.xlsx]
(ooxml, embedded labels, table is Sheet1);
UNMAP;
```

2. Click **Load data**.

   The resulting table looks like this:

   Table showing data loaded using Map … Using function



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 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.

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. Add a new script section in the **Data load editor**.
3. Call the section *Wine*.
4. Under **AttachedFiles** in the right menu, click **Select data**.
5. Upload and then select *Winedistricts.txt*.
6. In the **Select data from** window, uncheck the *Lbound* and *RBound* fields so that they are not loaded.
7. Click **Insert script**.
8. Enter the following above the LOAD statement:

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

Your script should look like this:

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

9. Click **Load data**.
10. 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 or as dimensions in pivot tables.
    - 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.

    The resulting table looks like this:

    Table showing sample of data loaded using Hierarchy prefix

My new sheet

NodeID	ParentID	NodeName	NodeName1	NodeName2	NodeName3	NodeName4	NodeName5	NodeName6
289	288	Bas-Médoc	The World	Europe	France	Bordeaux	Médoc	Bas-Médoc
290	289	Listrac	The World	Europe	France	Bordeaux	Médoc	Bas-Médoc
291	289	Pauillac	The World	Europe	France	Bordeaux	Médoc	Bas-Médoc
292	289	Saint-Estèphe	The World	Europe	France	Bordeaux	Médoc	Bas-Médoc
293	289	Saint-Julien	The World	Europe	France	Bordeaux	Médoc	Bas-Médoc
294	288	Haut-Médoc	The World	Europe	France	Bordeaux	Médoc	Haut-Médoc
295	294	Margaux	The World	Europe	France	Bordeaux	Médoc	Haut-Médoc

To learn more about hierarchies, see this blog post in Qlik Community: Hierarchies

5.2 HierarchyBelongsTo prefix

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

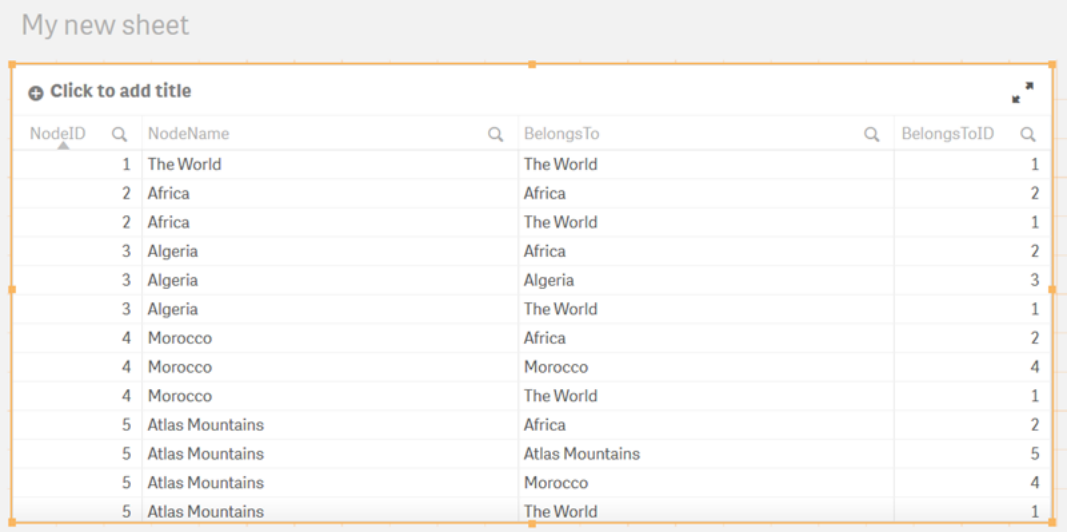
Also here, the LOAD statement needs to have at least three fields: An ID that is a unique key for the node, a reference to the parent and a name. The prefix will transform the loaded table into an ancestor table, a table that has every combination of an ancestor and a descendant listed as a separate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

Do the following:

1. Modify the Hierarchy statement in the **Data load editor** so that it reads as follows:
   `HierarchyBelongsTo (NodeID, ParentID, NodeName, BelongsToID, BelongsTo)`
2. Click **Load data**.
3. Use the **Preview** section of the **Data model viewer** to view the resulting table.
   The ancestor table 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.

   The resulting table looks like this:

   Table showing data loaded using HierarchyBelongsTo prefix



NodeID	NodeName	BelongsTo	BelongsToID
1	The World	The World	1
2	Africa	Africa	2
2	Africa	The World	1
3	Algeria	Africa	2
3	Algeria	Algeria	3
3	Algeria	The World	1
4	Morocco	Africa	2
4	Morocco	Morocco	4
4	Morocco	The World	1
5	Atlas Mountains	Africa	2
5	Atlas Mountains	Atlas Mountains	5
5	Atlas Mountains	Morocco	4
5	Atlas Mountains	The World	1

Authorization

It is not uncommon that a hierarchy is used for authorization. One example is an organizational hierarchy. Each manager should 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.

Organizational hierarchy example

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

Authorization table

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	Sales
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.

Example:

Often the hierarchy is stored in an adjacent nodes table. In this example, to solve this, you can load the adjacent nodes table using a HierarchyBelongsTo and name the ancestor field *Tree*.

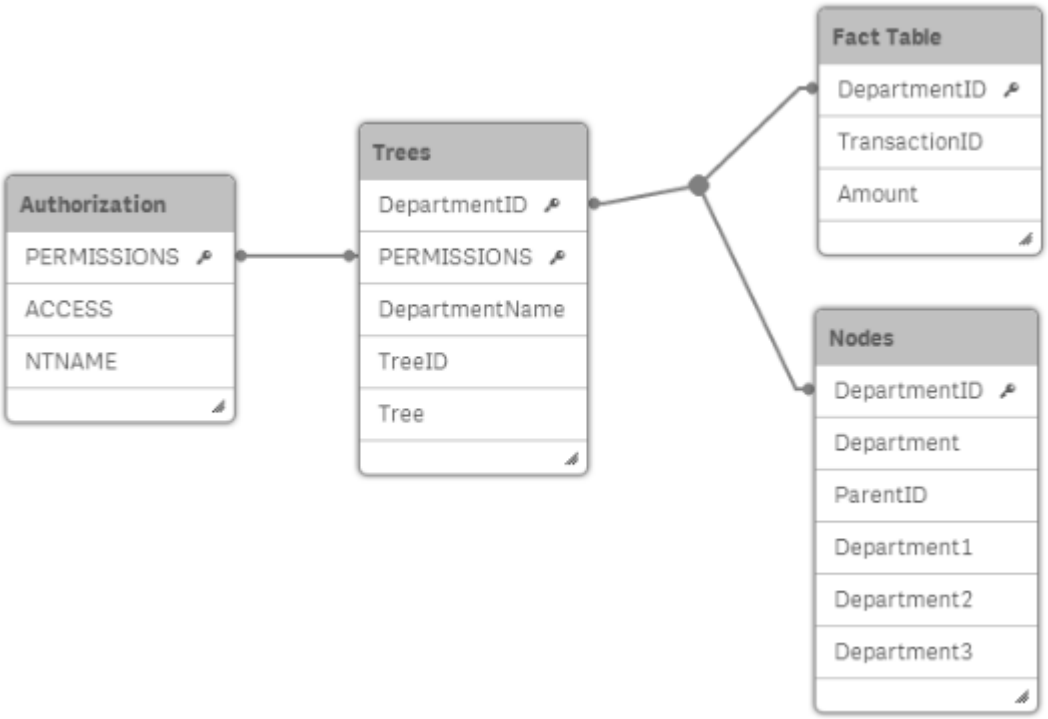
If you want to use Section Access, load an upper case copy of *Tree* and call this new field *PERMISSIONS*. Finally, you need to load the authorization table. These two last steps can be done using the following script lines. Note that the TempTrees table is the table created by the HierarchyBelongsTo statement.

Note that this is an example only. There is no accompanying exercise to be completed in Qlik Sense.

```
Trees:
LOAD *,
    Upper(Tree) as PERMISSIONS
    Resident TempTrees;
Drop Table TempTrees;

Section Access;
Authorization:
LOAD   ACCESS,
       NTNAME,
       UPPER(Permissions) as PERMISSIONS
From Organization;
Section Application;
```

This example would produce the following data model:

Data model: Authorization, Trees, Fact, and Nodes tables



6 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.

6.1 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 an an explicitly named QVD, CSV, or txt file.

Syntax:

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

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 [lib://AttachedFiles/xyz.qvd];
Store * from mytable into [lib://FolderConnection/xyz.qvd];
Store myfield from mytable into 'lib://FolderConnection/xyz.qvd';
Store myfield as renamedfield, myfield2 as renamedfield2 from mytable into
[lib://AttachedFiles/xyz.qvd];
Store mytable into 'lib://FolderConnection/myfile.txt';
Store * from mytable into 'lib://FolderConnection/myfile.csv';
```

Do the following:

1. Open the *Advanced Scripting Tutorial* app.
2. Click the *Product* script section.
3. Add the following to the end of the script:

   ```
   Store * from Product into [lib://AttachedFiles/ProductData.qvd](qvd);
   ```

   Your script should look like this:

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

```
FROM [lib://AttachedFiles/Product.xlsx]
(ooxml, embedded labels, table is Product);

Store * from Product into [lib://AttachedFiles/ProductData.qvd](qvd);
```

4. Click **Load data**.
   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.

6.2 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).

  Examples:

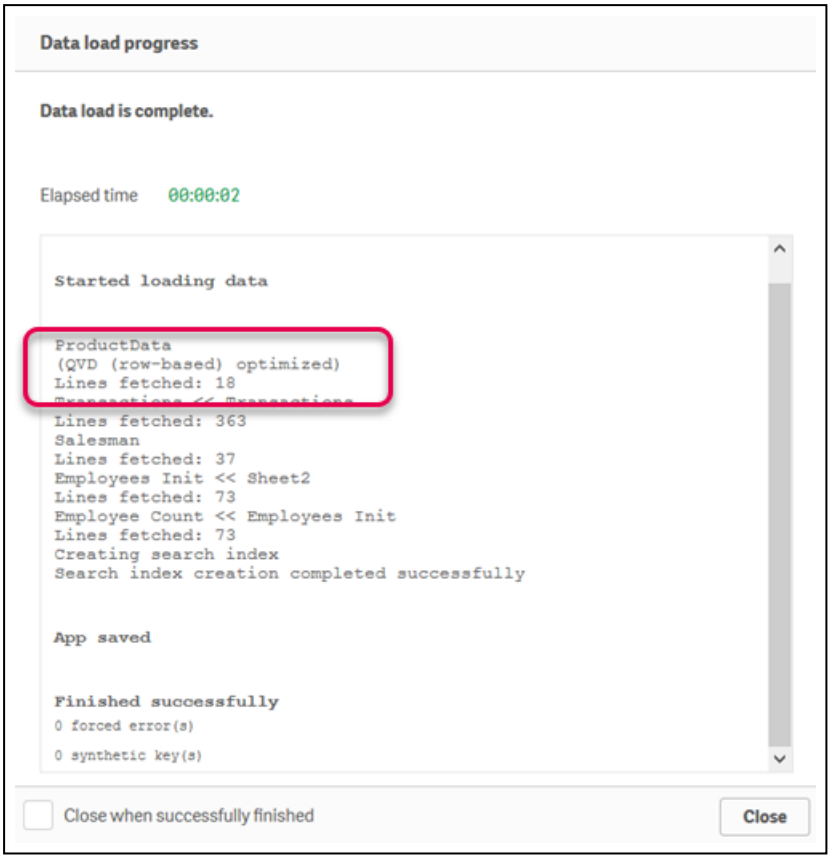
  ```
  LOAD * from 'lib://FolderConnection/xyz.qvd' (qvd);
  LOAD fieldname1, fieldname2 from [lib://FolderConnection/xyz.qvd] (qvd);
  LOAD fieldname1 as newfieldname1, fieldname2 as newfieldname2 from
  [lib://AttachedFiles/xyz.qvd](qvd);
  ```

- 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.

Do the following:

1. Comment out the entire script in the *Product* script section.
2. Enter the following script:
   ```
   Load * from [lib://AttachedFiles/ProductData.qvd](qvd);
   ```
3. Click **Load data**.
   The data is loaded from the QVD file.

Data load progress window



To learn about using QVD files for incremental loads, see this blog post in Qlik Community: Overview of Qlik Incremental Loading

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:

```
Buffer load * from MyTable;
```

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:

```
Buffer (stale after 7 days) load * 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.

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.

Example:

```
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.

Do the following:

1. Create a new app and give it a name.
2. Add a new script section in the **Data load editor**.
3. Under **AttachedFiles** in the right menu, click **Select data**.
4. Upload and then select *Cutlery.xlsx*.
5. In the **Select data from** window, click **Insert script**.
6. Comment out the fields in the load statement, and change the load statement to the following:
   ```
   Buffer LOAD *
   ```

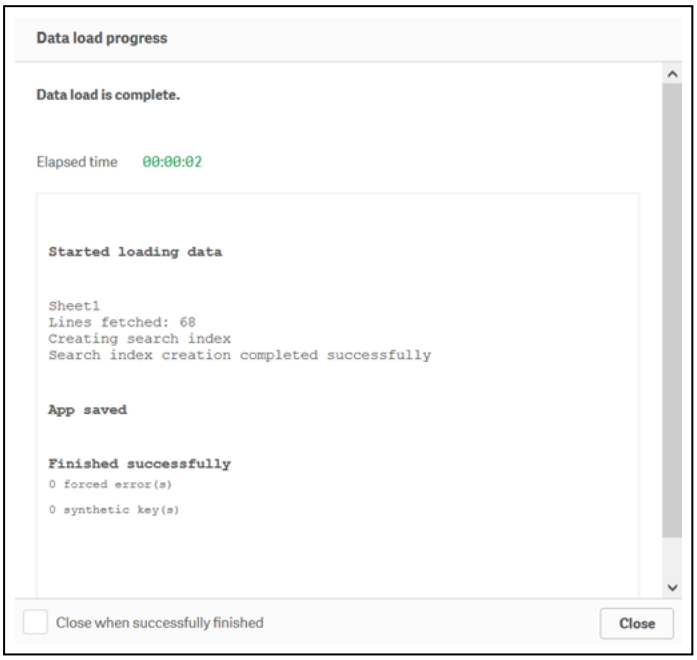
   Your script should look like this:

   ```
   Buffer LOAD *
   //     "date",
   //     item,
   //     quantity
   FROM [lib://AttachedFiles/Cutlery.xlsx]
   (ooxml, embedded labels, table is Sheet1);
   ```

7. Click **Load data**.

   The first time that you load data, it is loaded from *Cutlery.xlsx*.

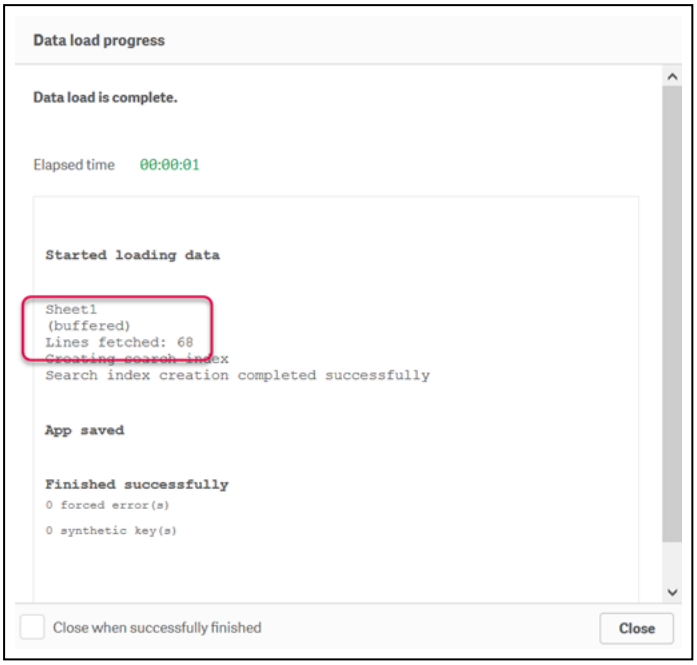
   Data load progress window



   The Buffer statement also creates a QVD file and stores it in Qlik Sense. In a Qlik Sense Enterprise on Windows deployment, it is stored in a directory on the Qlik Sense server.

8. Click **Load data** again.
9. This time the data is loaded from the QVD file created by the Buffer statement when you loaded the data for the first time.

Data load progress window



6.3 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.