QlikView Expressor Product Training

Contents

Introductory Concepts

1 Product Introduction

QlikView Expressor consists of three components.

1. Desktop: The graphical user interface used to develop and test data integration applications.
2. Date Integration Engine: The command line executable that runs QlikView Expressor data integration applications.
3. Repository: The version control system that integrates with Desktop and the Engine, supporting team development and controlled deployment.

While all three components may be installed onto computers running Windows XP, Windows 7/8, or Windows Server 2003/2008/2012, the standard approach is to install Desktop onto developer computers running Windows XP or Windows 7/8 and the Engine and Repository onto computers running Windows Server.[1]

A customer generally deploys Desktop onto many developer computers, configuring them to use a common Repository. This arrangement allows developers to work privately, in what is termed a Standalone Workspace, or within a team, in a Repository Workspace. Once an application's development and testing is complete, a compiled version of the application may be checked out of Repository onto the computer hosting the Data Integration Engine. While many deployments install the Engine and Repository onto separate computers, it is perfectly acceptable to install these two components onto the same computer.

The Engine and Repository components of QlikView Expressor must be licensed and will not run until the license is installed. Desktop is functional without an externally applied license, but does not support the Write Teradata DT operator until an appropriate purchased license key has been installed.

2 Installation

When installing the QlikView Expressor components there are two approaches.

1. If the installation is limited to Desktop, you may use the installer that installs only this component.
2. If the installation is to include multiple components, either on the same or different computers, you use the full platform installer.

You will need a RDBMS to develop the solution to the exercises dealing with managing slowly changing dimension tables/incremental loads and semantic types. Unless you have access to another RDBMS, install Oracle XE version 10g or Microsoft SQL Server onto the computer on which you installed QlikView Expressor.

[1] To support the QlikView extension and Read QVX Connection, QlikView Expressor must be installed on a computer running a 64-bit version of Windows.

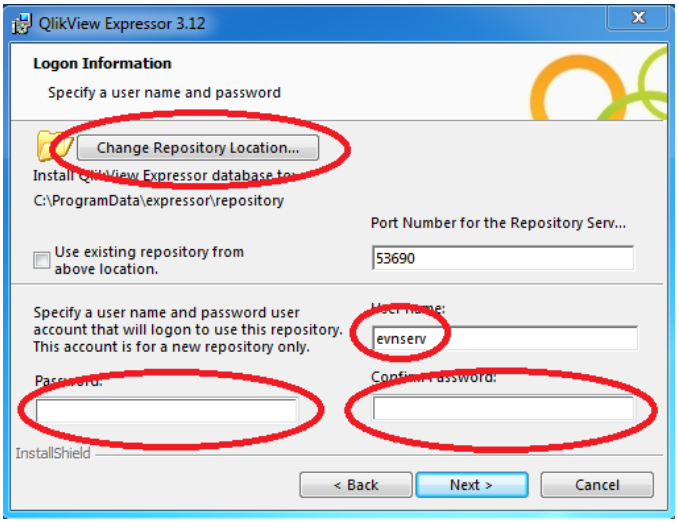
2.1 Installing Desktop

The executable `QlikViewExpressorDesktopInstaller.exe`, which is approximately 200 MB, is downloaded from the QlikView Web site. When you run this application, it confirms that the required prerequisites (for example, the correct .NET framework) are installed and if necessary downloads and installs any prerequisites that are missing. The process then extracts and runs the actual Desktop installer, `QlikViewExpressorDesktopInstaller.msi`.

During this installation, the only decision you will need to make is to select an installation directory if you do not choose to install under the Program Files, 32 bit operating system, or Program Files(x86), 64 bit operating system, directory.

2.2 Installing Repository and/or Engine

The executable `QlikViewExpressorFullInstaller.exe`, which is approximately 200 MB, is downloaded from the QlikView Web site. When you run this application, it confirms that the required prerequisites (for example, the correct .NET framework) are installed and if necessary downloads and installs any prerequisites that are missing. The process then extracts and runs the actual installer, `QlikViewExpressorFullInstaller.msi`.



During this installation, you must make several decisions in addition to selection of an installation directory. If installing the Repository, you must specify the file system location in which this version control system will store its content and you must pick the TCP/IP port number over which communication with the Repository will occur. You also need to provide username and password credentials for the initial user account. When entering these credentials, DO NOT USE your Windows credentials. These credentials, which are only required to communicate with Repository, are stored in open text with the Repository installation. As an administrative task, you will later add credentials for other users so that the Repository version control system can track individual users' changes.

2.3 Licensing

This training is based on the Expressor Desktop Edition for which licensing is not necessary. However, after completing the Repository and/or Engine installation, you must install a license. Desktop includes a menu item that will allow you to install a license by making several entries into a form, although

installing a license into Desktop is generally not required. On computers where you have installed only Repository and/or Engine, you will need to run a utility in a command window to affect the licensing.

Select the **Start – All Programs – QlikView – expressor3 – expressor command prompt** menu item to open a command window. Since installing QlikView Expressor does not alter any environmental variables, you must use this menu item to open a properly configured command window whenever you want to run a command line utility that is part of QlikView Expressor. You will then use the `elicense` command line utility to license the Repository and Engine. If both the Repository and Engine are installed onto the same computer, you only need to run this utility once. If these components are installed onto different computers, you must run this utility on each computer.

NOTE: In order to perform the exercises, you must also have QlikView Personal Edition or QlikView Desktop Edition installed on your computer.

Exercise: QlikView Expressor Desktop Installation

1. Obtain **QlikViewExpressorDesktopInstaller.exe**.
    a. On the Qlik Web site, register and download the free installer.
    b. QlikTech employees, partners and current customers may also download from the Qlik download site.
        i. http://download.qlikview.com
        ii. Log in with your trigram and single sign-on password. Not your domain password.
        iii. Click the **Bookmarked Downloads** link.
        iv. Click the **QlikView Expressor** link.
        v. Select the **QlikViewExpressorDesktopInstaller.exe** link (be certain to select the most recent release) and save to a convenient file system location.
2. Run the installer by double-clicking on its entry in Windows Explorer.
    a. Accept the default installation suggestions.

Exercise: Data Files

Extract the data.zip file directly to your C:\ drive, creating the directory C:\data, which contains the data files used in this tutorial.

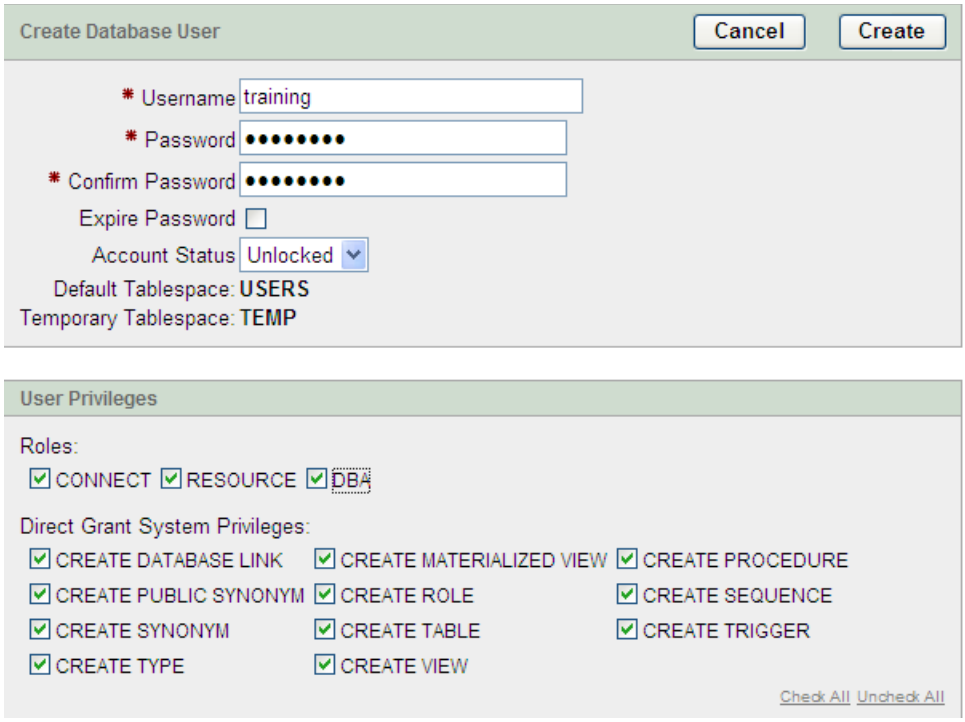
Exercise: Decoda IDE Installation

The Decoda Lua development and debugging tool can be downloaded from Unknown Worlds Entertainment. Simply run the executable to install.

Exercise: Oracle XE RDBMS Installation

Expressor has been developed and tested with Oracle versions 10 and 11. This course has been developed using Oracle XE 10g. QlikView Expressor ships with the necessary database driver, which is transparently used when creating a database connection artifact.

1. Download the Oracle XE 10g installer.
2. Double-click on the installer icon to begin the installation process.
3. When prompted, enter a password for the SYS and SYSTEM database accounts. _____
4. On the last screen, before clicking **Finish**, be certain to select the **Launch the Database homepage** checkbox.
5. Use the SYSTEM account, and password from step 3, to log into the database homepage.
6. Click on the **Administration** then **Database Users** icons.
7. Then click **Create>** to open the Create Database User page.
8. Create the user **training** with the password **training**.
    a. Grant all user privileges.

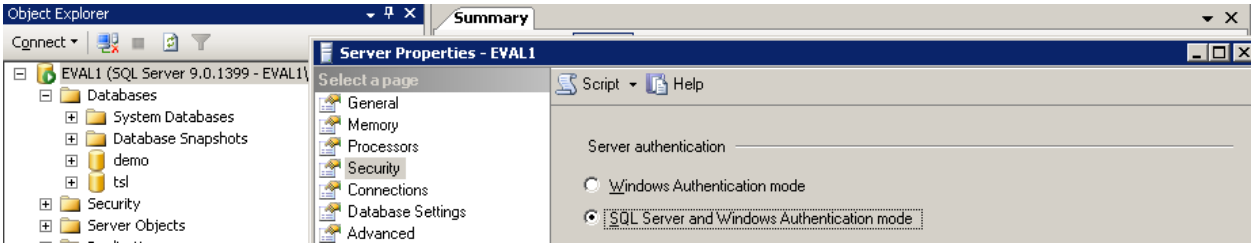


    b. Then click **Create**.
9. Log out of the database homepage.
10. You should also unlock the HR account.
    a. Be certain to supply a password.
11. To view the database tables, etc., log in through the database home page using the HR or training accounts or open a command window and log in using the account's credentials. For example,

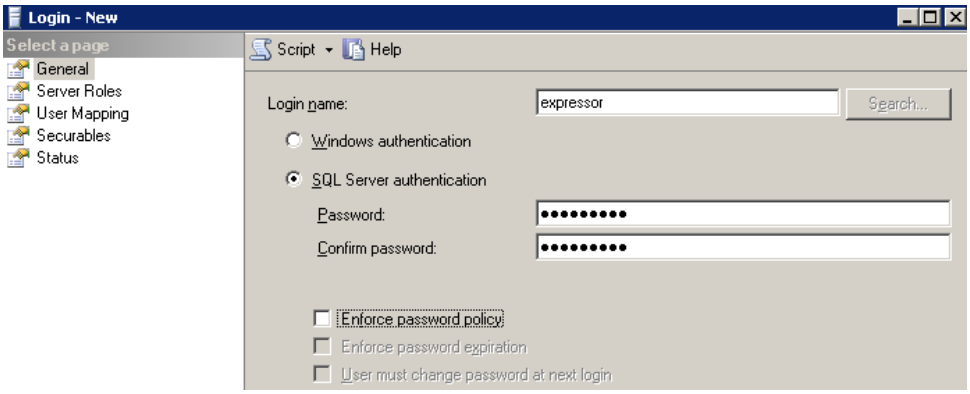
```
sqlplus training/training
```

Exercise: Microsoft SQL Server Installation

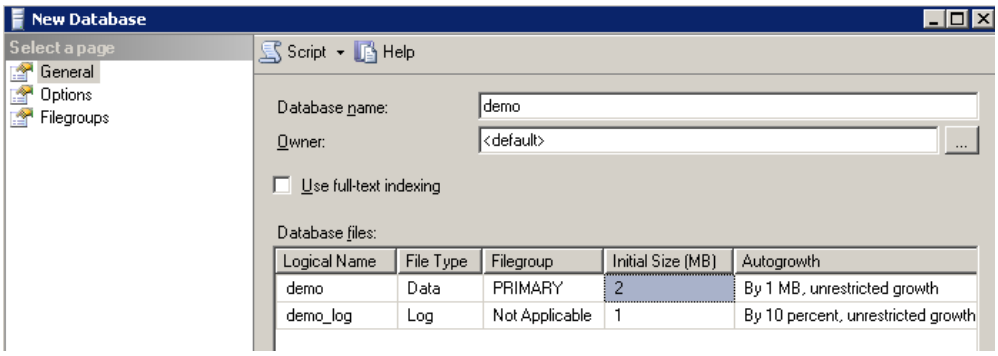
1. Double-click on the installer icon to start the installation process.
    a. Perform a Typical install accepting all of the default suggestions.
2. Use the SQL Server Management Studio to set up and configure the database and user account.
    a. Highlight the top level entry in the Object Explorer, right-click, and select Properties form the popup menu.
    b. In the Server Properties window, select the Security page and in the Server authentication grouping, select the SQL Server and Windows Authentication mode radio button.



    c. Highlight the Security > Logins entry in the Object Explorer, right-click, and select New Login… from the popup menu.
        i. In the Login Properties window, General page, enter the Login name (expressor), select the SQL Server authentication radio button, and enter the Password (expressor).
        ii. Uncheck the Enforce password policy checkbox control, which disables the two other checkbox controls.
            1. These will be the credentials used in the database connection artifacts.
        iii. Click OK to save the login information.

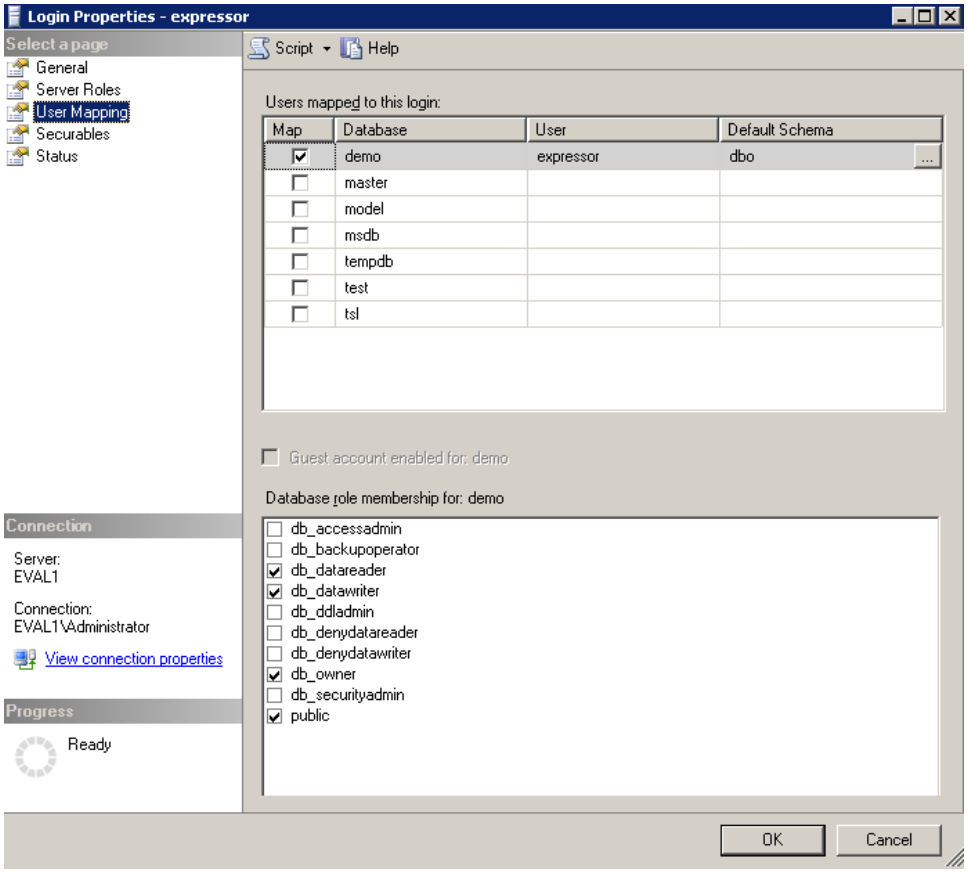


    d. Highlight the Databases entry in the Object Explorer, right-click, and select New Database… from the popup menu.
        i. On the General page, create a database named demo.
        ii. Click OK to create the database.

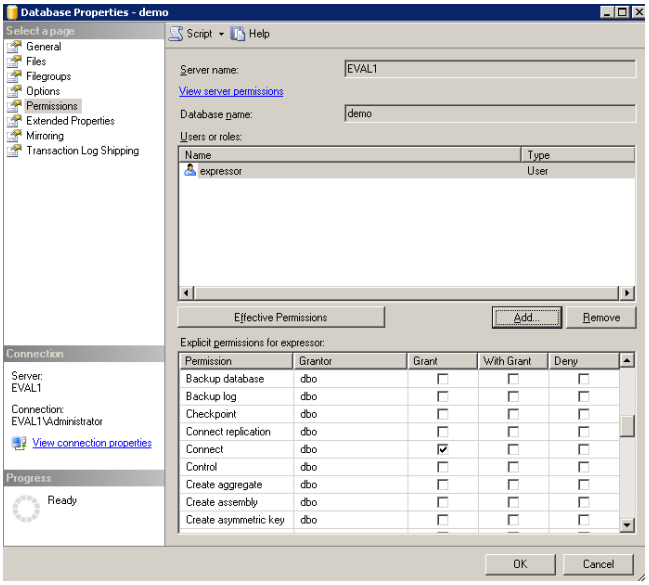


e. In order for the Expressor schema wizard to display the dbo database schema when creating an Expressor schema, you must create at least one table.
    i. Expand the demo entry under Databases, select the Tables entry, right-click, and select New Table… from the popup menu.
        1. A simple table with a single varchar column is sufficient.

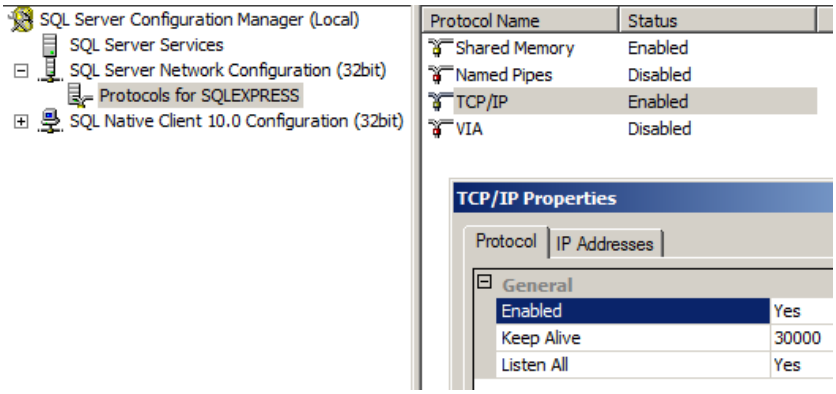
f. Once the demo database and sample table exist, reopen the Security > Logins > expressor Properties window and select the User Mapping page.
    i. Map the user expressor to the demo database and assign the datareader, datawriter, owner, and public roles.



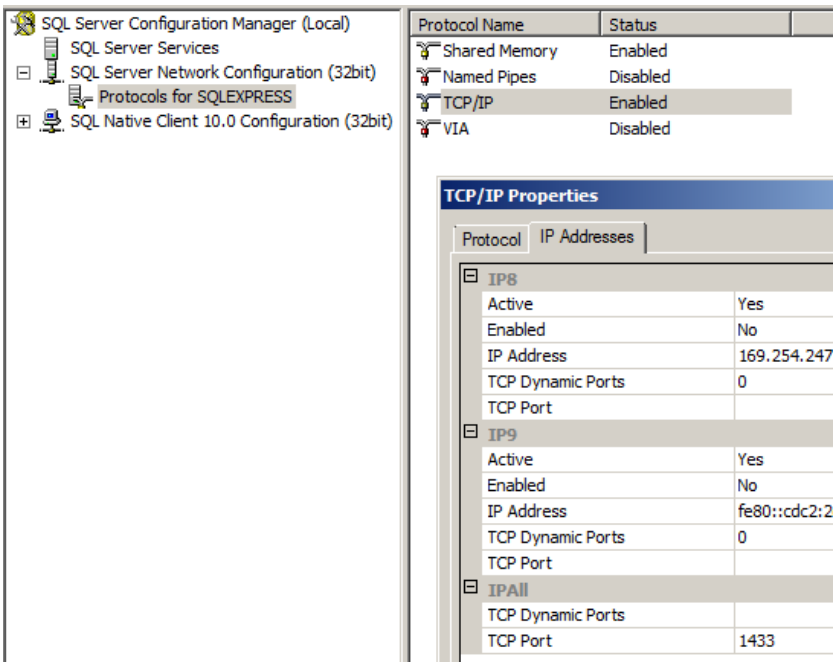
g. Next, in the Object Explorer, highlight the demo entry under the Databases entry, right-click, and select Properties from the popup menu.
    i. In the Database Properties window, select the Permissions page.
    ii. Then in the Users or roles grouping, click on the user expressor entry.
    iii. In the Explicit permissions for expressor list, confirm that the Connect > Grant checkbox is selected.



3. Use the SQL Server Configuration Manager to set the TCP/IP port used by SQL Server.
    a. Expand the SQL Server Network Configuration entry and highlight the Protocols for SQL Express entry.
    b. In the right-hand panel, double-click on the TCP/IP protocol entry.
        i. On the Protocol tab, confirm that the protocol is enabled.

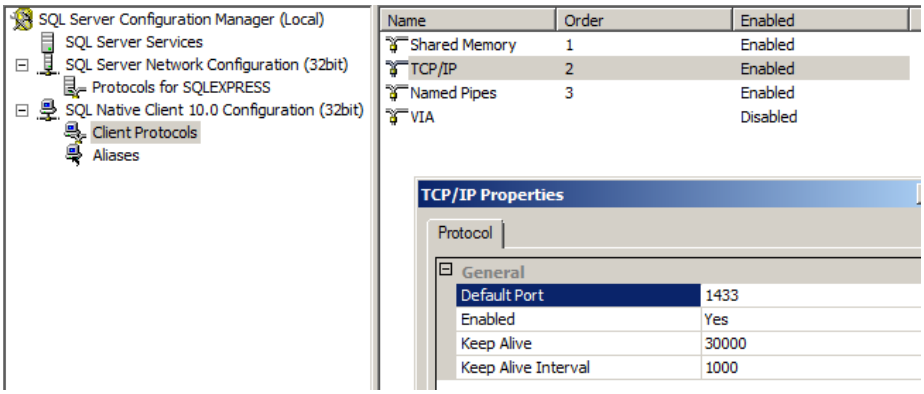


        ii. On the IP Addresses tab, confirm that IP1 and IP2 are active and enabled and that the TCP Port is set to 1433.
            1. Be certain that the IPAll TCP Dynamic Ports entry is blank.



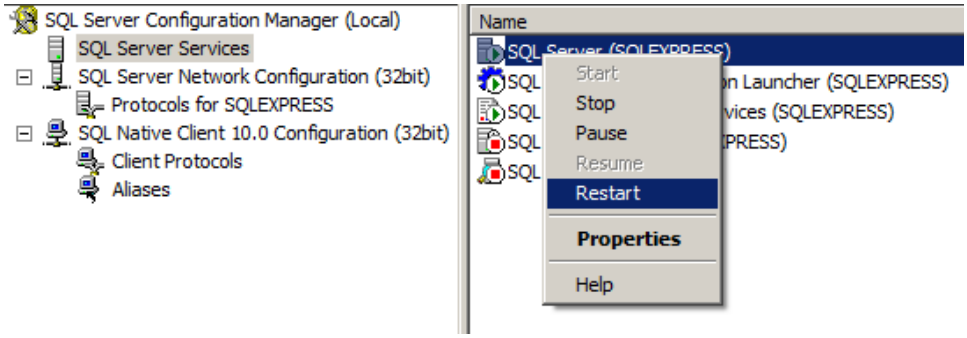
iii. Expand the SQL Native Client 10.0 Configuration entry highlight the Client Protocols entry.

1. In the right-hand panel, double-click on the TCP/IP protocol.
2. On the Protocol tab, confirm that the port is set to 1433 and that the protocol is enabled.



c. Finally, highlight the SQL Server Services entry.

i. In the right-hand panel, select the SQL Server (SQLEXPRESS) entry, right-click and select Restart from the popup menu.

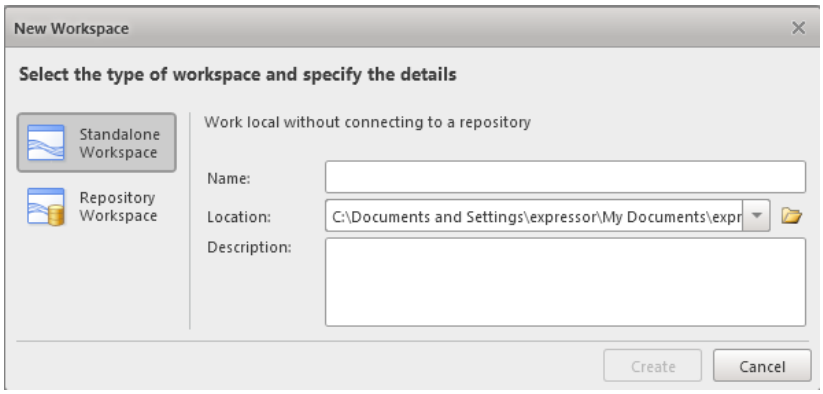


ii. Once the server restarts, close the configuration manager.

3 Workspaces, Projects and Libraries

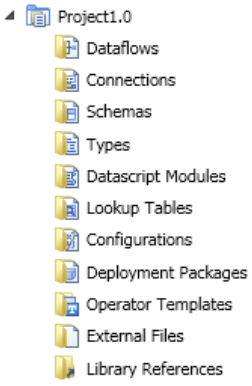
A workspace is a collection of QlikView Expressor data integration applications that are related to one another. For example, a group of applications that are used to process customer orders. These applications most certainly share resources such as database tables or file system locations, and processing logic developed for one of the applications might also be used in other applications. Within QlikView Expressor Desktop, you may create as many separate workspaces as desired. When you are developing a project or library, you work within the scope of a single workspace.

When you first start QlikView Expressor Desktop, there are no current workspaces although QlikView Expressor Desktop does ship with a sample workspace that includes a few simple data integration applications. Your first responsibility is to create a new workspace.



Note that in the New Workspace window there are two types of workspaces: Standalone and Repository. With the QlikView Expressor Desktop Edition product, you may only work in a Standalone Workspace; all work is stored on your computer and you do not have access to the enterprise and team development features. If you upgrade to the Standard or Enterprise Editions, you will have the option of working within a Repository Workspace, which allows you to check your work into, and retrieve work from, the centralized QlikView Expressor Repository version control system.

A workspace contains one or more projects and/or libraries.



Projects and Libraries are containers for QlikView Expressor artifacts: Dataflows, Connections, Schemas, Semantic Types, Operator Templates, Datascript Modules, Deployment Packages (projects only), Configurations and Lookup Tables.

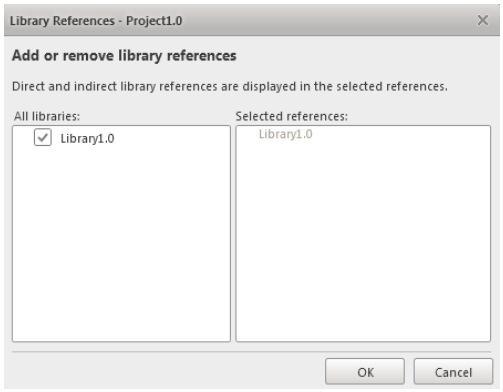
Additionally, either a project or a library can reference another library, which means that any artifacts contained in the referenced library may be used as if they were contained within the referencing project or library.

Only projects may be deployed onto the QlikView Expressor Data Integration

Engine, the production server. When you deploy an application, all of the required artifacts, from both the project and cross referenced libraries are compiled into a standalone package that may be easily moved onto the computer running the data integration engine.

Exercise: Workspace and Project Setup

1. From the Start menu, open QlikView Expressor Desktop.
2. Click on the **New Workspace** link and create a new standalone workspace named QVE_Labs.
    a. Click **Desktop > Manage Extensions…**. The **expressor_QlikViewLibrary.0** extension should already be enabled.
        i. If not, select the Current Workspace Settings tab, select the extension and click **Enable**.
    b. If desired, enable the other extensions.
3. The Create tab shows the most commonly chosen tasks.
    a. Click **Project**.
    b. Accept the default project name (Project1) and click **Create**.
    c. In Desktop's Explorer panel, check to see that Project1.0 has been created.
4. Click on the Create tab of the ribbon bar.
    a. Click **Library**.
    b. Create a new library with the default library name: Library1.0.
5. Under Project1.0, right click on the **Library References** folder, select **Library References…** and check the **Library1.0** entry within the **All libraries:** listing.



    a. This associates the library with the project.
    b. Any artifacts you create within the library will be available to artifacts you create within the project.
    c. Click **OK** to close the window.
6. Click on the question mark in the top right hand corner of Desktop. Choose the **Desktop Help (F1)** option to open the embedded Help document.
    a. Navigate to and read the "Workspaces, Projects and Libraries" topic.
7. Get version information.
    a. Click on the question mark in the top right hand corner of Desktop.

i. Choose the **About QlikView Expressor Desktop** option to open an informational window that shows the Build Revision and product version.
ii. If you need to contact QlikView support, be certain to include this information in your enquiry.

4 Connection Artifacts

Connection artifacts specify the location of the resource to be accessed. They are used by Input and Output operators. A connection artifact may also be required when configuring an operator, for example, the Aggregate operator or Lookup Table, which may write data to disk.

In QlikView Expressor Desktop Edition there are several types of connection artifacts: File Connection and Database Connection. If you install the QlikView Expressor Sales Force Dot Com library extension, you will also be able to define a Sales Force Connection.

4.1 File Connection

A File Connection is simply the path to a file system location. You may specify either a relative or absolute path, however using a relative path requires a little forethought as the path is relative to the file system location where the application runs, which may be different when you are developing within Desktop or executing on the data integration engine.

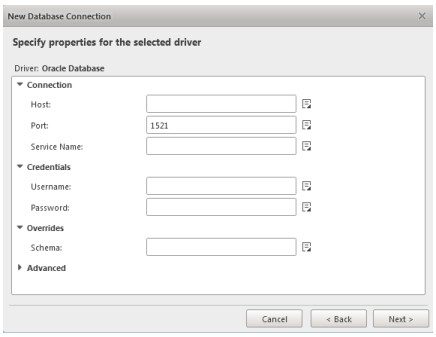
4.2 Database Connection

A Database Connection contains the connection details needed to access a relational database management system. The details that you must provide differ slightly depending on the brand of database you are connecting to but generally include the host computer, TCP/IP port, user credentials, and identity of the target database.

When you want to configure a database connection, QlikView Expressor offers three approaches.

1. Create a connection using a supported driver supplied with QlikView Expressor.
2. Create a connection using a supported driver not supplied with QlikView Expressor.
3. Create a connection using an existing ODBC DSN that uses a driver that adheres to the ODBC 3.5 specification, e.g., the Microsoft ODBC Driver to Access.

QlikView Expressor Desktop includes database drivers for many of the most popular relational database management systems, so using the first approach is preferred. Popular database whose drivers not included with QlikView Expressor include Informix, Netezza and MySQL Community Edition. In order to use these databases, you will need to install the drivers. The QlikView Expressor Desktop Database Connection wizard will detect when these drivers are installed and enable its entry in the appropriate control.



Once you select a driver, you are presented with a window in which you enter the required connection parameters. The information entered within the Connection and Credentials groupings represent the fewest connection properties that must be set to enable connectivity. For some of these connection properties a default value will be automatically filled in. Enter optional connection parameters into the Advanced grouping.

Exercise: Create a File Connection

1. In QlikView Expressor Desktop, in the Explorer panel expand Library1.0.
    a. Right click on the **Connections** category and choose **New – File Connection**.
2. Enter **C:\data** into the **Path** text box.
    a. Click **Next**.
    b. Accept the default suggestion for the artifact's name.
        i. Be certain that Library1.0 is the selection in the Project drop-down control.
    c. Click **Finish**.
3. Confirm that the connection artifact appears in the Explorer panel under Library1.0.

A file connection artifact tells QlikView Expressor where your input files are and/or where your output files will be created. This artifact will be used by the input and output operators that work with files (delimited files, Excel files, QlikView .qvx or .qvd files).

Some operators, such as the Aggregate and Join, may need to temporarily write to a file system location. A File Connection will be used to specify this location as well.

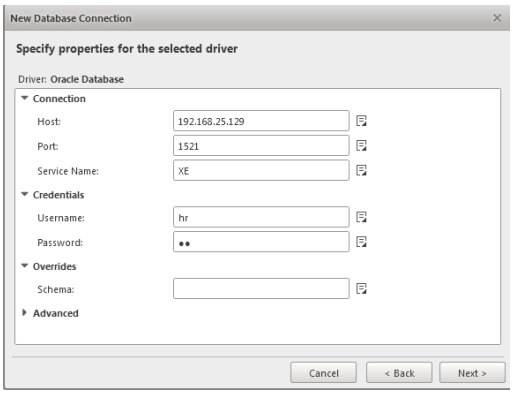
Finally, the data stored in a Lookup Table is retained as a file. You will also use a File Connection to specify the file system location where these files should be stored.

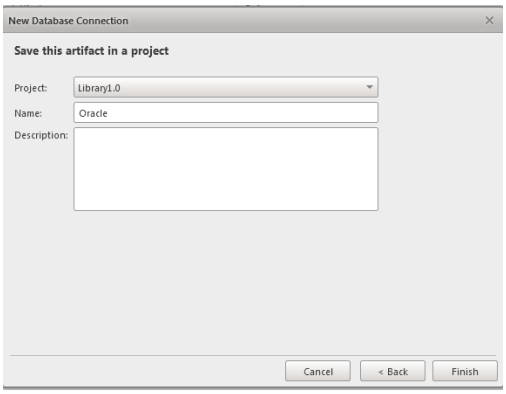
Exercise: Create a Database Connection

If you have any of the supported RDBMS's installed on your computer, or can access a remote RDBMS, create a database connection.

1. In QlikView Expressor Desktop, in the Explorer panel expand Library1.0.
    a. Right click on the **Connections** category and choose **New – Database Connection**.
2. From the **Supplied database drivers** drop down list, select the entry for the RDBMS.
3. Enter the appropriate connection information.
    a. For Oracle, the instance name is **XE**.
    b. For SQL Server, if you followed the installation procedure described in this document, the database name is **demo**.
    c. Enter the credentials for the training user account.

4. Test your connection by clicking **Next**.
    a. If you have made valid entries, the 'Save this artifact in a project' window will appear.
    b. Give the connection a meaningful name and save the artifact in either the project or library.



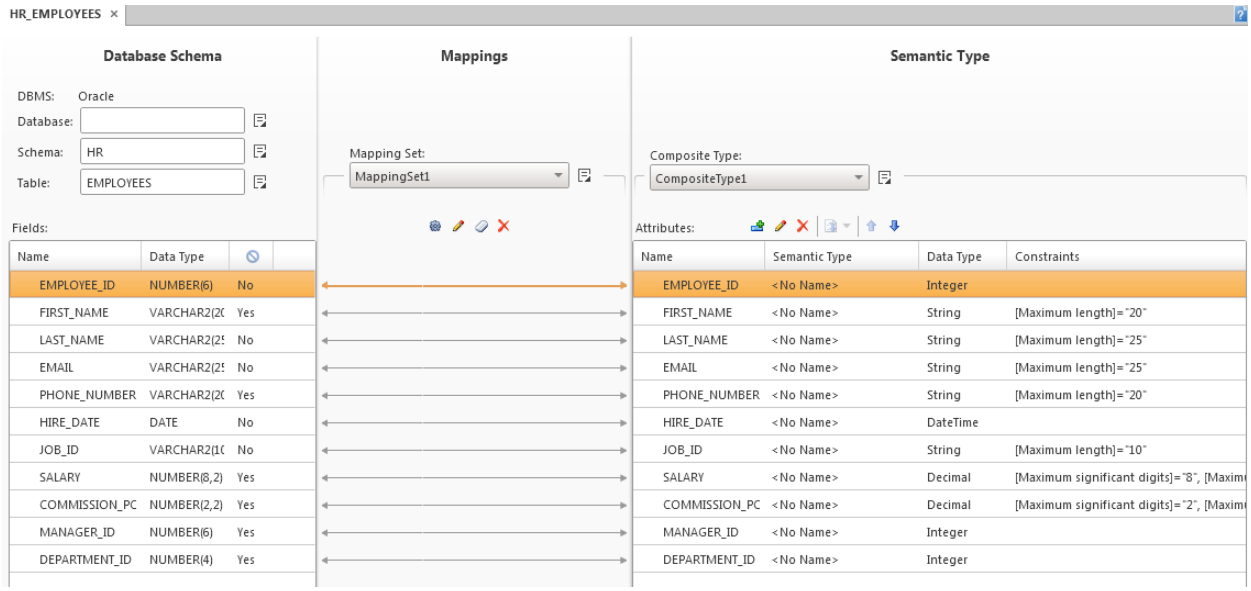


5 Schema Artifacts and Composite Types

A schema artifact describes the structure of the data that is read from or written to a data source. Additionally, it describes how each data type in the data external to the QlikView Expressor application will be converted into a QlikView Expressor data type. For example, while the external data resource may type a specific piece of data as a string, you may want to convert the value into a datetime or numeric value. The schema not only describes this transformation but includes the directives the QlikView Expressor engine will use to affect this transformation.

There are multiple ways to create a schema artifact. A schema artifact may be created from a delimited file by copying a few lines of the file into the delimited schema wizard, or a schema artifact may be created from metadata extracted from a relational database table or QlikView .qvx, .qvd or .qvw file, or a schema may be created from a composite semantic type. Which approach you take depends on whether the external data source already exists, for example, a file with input data or a populated database table, or whether the data source will be created by the QlikView Expressor application, for example, an output file.

The schema artifact contains three types of information: a description of the record structure in the external data source, a description of the record structure within the QlikView Expressor application, and a mapping between these two descriptions. The following figure shows the structure of a schema created from the EMPLOYEES table in an Oracle database.



The left-hand panel documents the structure of the database table providing the column name, type and size, and whether null values are accepted. The right-hand panel describes how the data will be represented within the QlikView Expressor application. By default, the name of an attribute is the same as its corresponding table column, but that is not a requirement and you are free to change the attribute names to something more descriptive or meaningful. The number of data types within QlikView Expressor is much smaller than the number of data types within a database, so many database types are mapped to the same QlikView Expressor data type. For example, all integer table types are mapped to the same QlikView Expressor integer type and both the datetime and timestamp table types are mapped to the QlikView Expressor datetime type.

An attribute may be associated with one or more constraints, which are like business rules applied to a specific piece of data. Note how the maximum size of the FIRST_NAME, LAST_NAME, and EMAIL attributes is constrained to the width of the corresponding table columns. During execution, QlikView Expressor will constantly check the size of the value contained by these attributes and if the stored value exceeds the maximum size limit, an error will be raised. Your application then handles the error in a manner consistent with your business and processing objectives.

Note that in the right-hand panel, the collection of attributes is referred to as a semantic type. When you first generate a schema artifact, this semantic type is given a default name (CompositeType1) and its scope is limited to the schema artifact. However, you may create additional composite types within the schema artifact, renaming the attributes or applying different constraints as necessary. For example, a schema artifact created against a database table could be used to read each row in the table by selecting a composite type that includes an attribute for each table column, but it could also be used to update table records by selecting a composite type that includes only the columns to be updated and the key column(s).

As your applications become more involved, you will find it extremely useful to convert a local composite type into a shared type. This creates a type artifact that may be used as a description of a record at many places within the data integration application. You can also use the composite type as a template for creating other schema artifacts. For example, a composite type that describes a database table could be used to create a schema artifact that describes a delimited file, allowing you to easily read the contents of the database table and write to a file. When used in this way, the QlikView Expressor engine takes care of all the type conversions so that all table data types can be written as strings to the file.

The center panel describes the mapping between the external data fields and their corresponding attributes. The mappings have two functions:

1. They associate each field in the external data resource with its corresponding attribute in the composite type that represents this data within a QlikView Expressor application.
2. They describe any data transformations that may be required to move data into or out of the QlikView Expressor application. Mapping formats provide directives to these transformations.

When you create a schema artifact, the schema wizard will create default mappings and formatting directives that, in many situations, may be completely suitable for your purposes. In other situations, you may want to alter the schema, perhaps by creating or selecting a different composite type or by changing the data type of an attribute. In these situations, you may need to modify the mapping format.

- Mapping formats are characteristics of schema artifacts and are therefore only relevant to those operators involved in moving data into or out of a QlikView Expressor application, i.e., input and output operators.
- A mapping format may only be applied when either the field in the external data or the attribute in the semantic type corresponding to that field is a string type.
- Think of a schema artifact as having an upstream and downstream orientation.
- For the input and output operators, upstream represents the data received by the operator and downstream represents the data emitted by the operator.
    - For input operators, upstream data is the external data read by the operator and downstream data are the attributes emitted by the operator. The mapping format is applied to the data as it moves left-to-right across the schema across the schema as represented in the editor.
    - For output operators, upstream data are the attributes received by the operator and downstream data is the data written by the operator to an external resource. The mapping format is applied to the data as it moves right-to-left across the schema as represented in the editor.
- A format always describes how upstream data will be transformed into the downstream data.
- If one endpoint of the mapping is a non-string type the format is a description of how the string type represents the non-string type.

    - With a datetime endpoint, the format describes how the characters in the string correspond to the datetime entities such as century, year, month, day, hour, minute, seconds.
    - With a decimal endpoint, the format describes how the characters in the string correspond to a number and which characters (currency or grouping symbols) should be removed from or added to the string representation.
- If both endpoints of the mapping are string types, the format describes the modifications to the upstream data that must be applied before emitting the downstream data.

If you are using the same Schema artifact for both input and output operators, you may find it useful to create two mappings, one for when the Schema is used by an input operator and the second for when the Schema is used by an output operator. This will allow you to read data with minimal formatting and write it with enriched formatting such as a currency sign or numeric grouping.

Constraints are business rules that are applied at the level of an attribute in a composite type. Since within a composite type each attribute represents either a local or shared Atomic Type, a constraint is actually a business rule that becomes part of the definition of an Atomic Type.

Depending on the data type underlying an Atomic Type, the possible constraints vary. For example, if string is the underlying type, you may specify the length of the value, allowed values, or a character pattern that the value must match, while for a numeric type, you can enforce the position of a decimal point, maximum and minimum values, the number of significant digits, or allowed values.

Once you specify one or more constraints for an Atomic Type, QlikView Expressor will test the attribute's value against these specifications as an input, transforming, or output operator emits a record. If one of the constraints is violated, a corrective action will be applied. Again, the type of corrective action depends on the data type underlying the attribute. Corrective actions available to all types are to throw (escalate) the error up to the operator for resolution, to replace the offending value with null, or to replace the offending value with a default value. With string values, you may alter the length of the value by either truncating characters from the left or right ends or by padding the left or right ends, and numeric values give you the additional option of rounding.

If the corrective action is to be applied by the operator, there are multiple options that vary depending on the type of operator.

- For the Read File, Read Table, Read Custom, Read Excel, Read Salesforce, Read SOQL, Read QlikView, Read QVX Connector, SQL Query, Pivot Row, Transform, and all output operators, there are five options:
    - Abort the dataflow
    - Skip the record containing the offending value
    - Reject the record containing the offending value
    - Skip the offending record and all following records
    - Reject the offending record and all following records

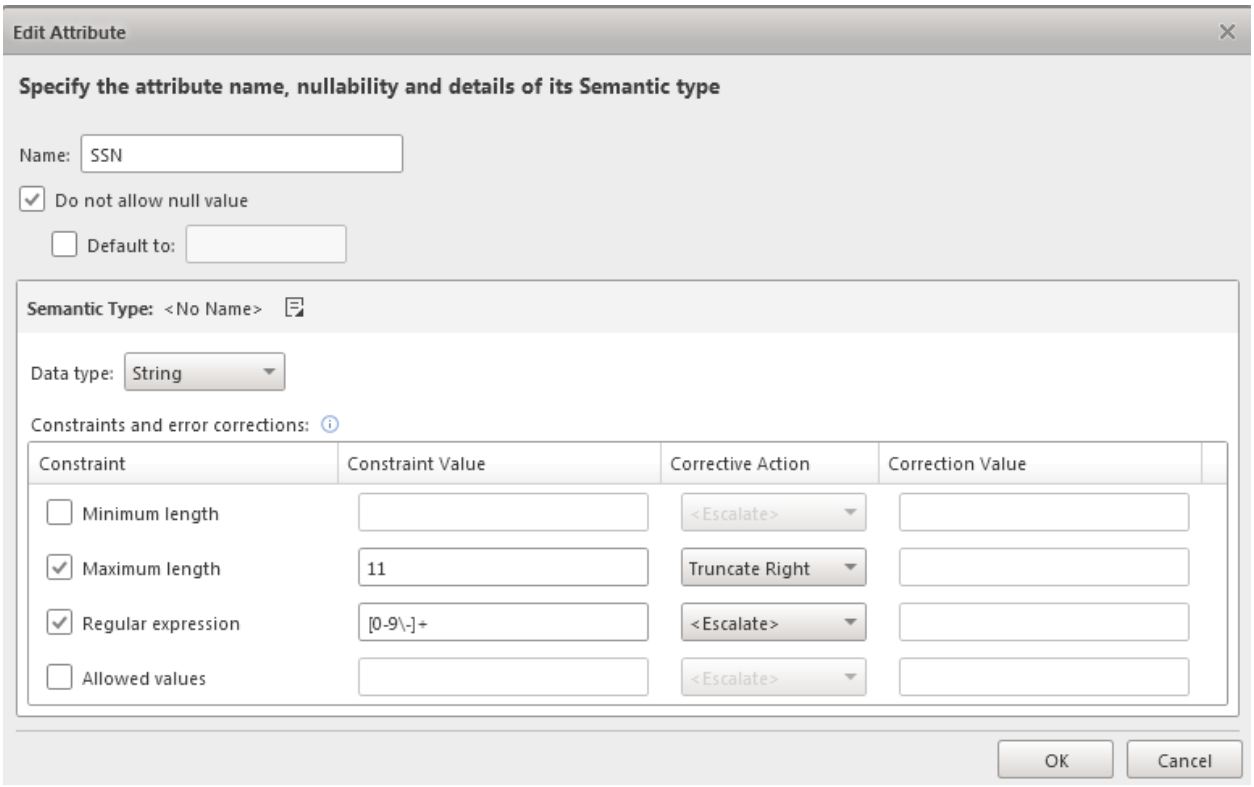
- For Pivot Column, Aggregate and Join operators, there are three options:
    - Abort the dataflow
    - Skip the record containing the offending value
    - Skip the offending record and all following records

The output operator Write Table will also raise a constraint exception if the target database table rejects the record for some reason.

Before selecting a corrective action, give some thought to the impact your choice will have on the processing.

- Aborting the entire dataflow has a very serious impact as all processing will immediately cease. Records that have completed processing will not be rolled back, processing will not continue for records that are currently being processed, and no further records will enter the processing stream.
- Skipping a record will allow processing to continue but the information contained in the skipped record will not be processed; this may, or may not, affect the validity of your application, which is something you must determine.
- Rejecting a record will also allow processing to continue but the offending record can be captured, reanalyzed, and perhaps resubmitted for processing. This option may allow you to recover from the constraint violation.
- If you choose to skip or reject the offending record and all following records, the records that are currently being processed will run to completion and you may also be able to recover from the constraint violation.
- When it is the Aggregate operator that identifies an offending record, you need to consider how the absence of this record will impact the ongoing calculation.

You may set multiple constraints for the same attribute. If you do, the constraints are evaluated in parallel. For example, suppose a database column defined as char(15) contains social security numbers of the format 123456789 and 123-45-6789. When these values are retrieved from the table they will be 15 characters wide, padded on the right with space characters. As constraints, you want to restrict the length to no more than 11 characters, limited to numeric characters and the dash. You will use both the Maximum Length and Regular expression constraints. Both constraints will be evaluated.



Exercise: Types, Constraints and Schemas

In this exercise, you will develop an application that processes revenue information from the slot machines at a casino/hotel to learn about semantic types, how constraints are associated with types, and how you can use constraints and the attribute mappings within a schema to control how data is read into and written from a data flow application.

Your customer, a large casino/hotel has many slot machines located throughout the complex. You have been asked to develop an application demonstrating how QlikView and QlikView Expressor can process revenue information from these machines and prepare summary reports that present the revenue partitioned by machine location and type of game. For simplicity, you are only going to develop the application for two of the machine locations, the hotel lobby and casino (there are also machines in the restaurants and pool area).

The manufacturers of the machines all provide a mechanism to extract usage information but unfortunately they do not provide exactly the same information. For example, the machines in the hotel lobby allow a guest to swipe their room card, for which they then earn reward points that reduce their room bill. The manufacturer of the machines in the casino did not provide for a similar functionality in their machines. Consequently, the raw data obtained from the machines is different and your application must take this into consideration.

All of the machines in a distinct location, e.g., the hotel lobby, are networked so that their usage information is forwarded to a computer that writes the usage information to a comma separated delimited file. So your application will need to combine the data in multiple files before it can proceed with the analysis.

The File Formats

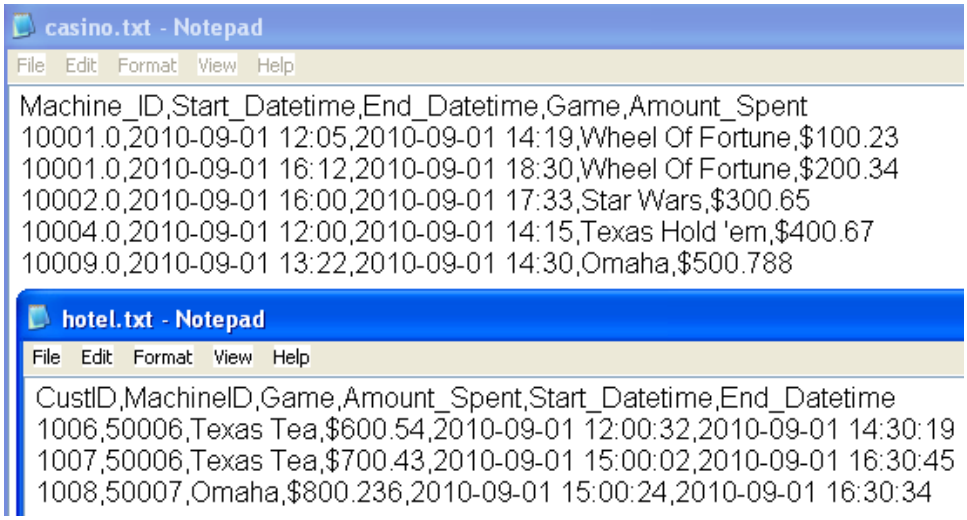
Data Field	Data Type
CustID	Integer
MachineID	Integer
Game	String
Amount_Spent	Decimal
Start_Datetime	Datetime
End_Datetime	Datetime

The summary file generated for the machines in the hotel lobby has six fields of information for each usage of a machine. In the actual file, the Amount_Spent field includes the dollar sign and the datetime fields include both the calendar date and the time, in 24 hour format, including the seconds (i.e., CCYY-MM-DD HH24:MI:SS).

Data Field	Data Type
MachineID	Decimal
Start_Datetime	Datetime
End_Datetime	Datetime
Game	String
Amount_Spent	Decimal

The summary file generated for the machines in the casino have five fields of information for each usage of a machine. In the actual file, the Amount_Spent field includes the dollar sign and the datetime fields include both the calendar date and the time, in 24 hour format, without the seconds (i.e., CCYY-MM-DD HH24:MI).

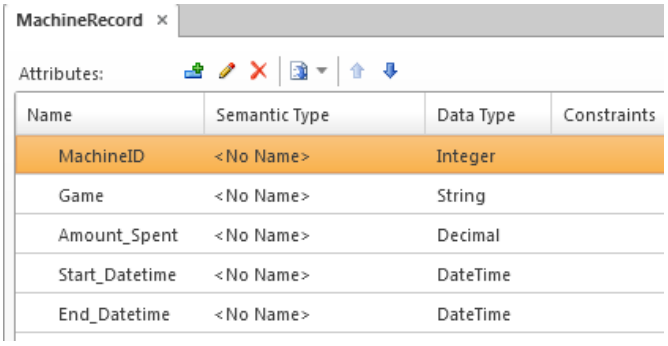
For your analysis, you do not need the CustID information as this is only specific to the machines in the hotel lobby. But what you must do is describe a composite type that can be used to hold the data from either group of machines so that all the usage data can be processed in a single application.



The Consensus Composite Type

You will accomplish three objectives by creating a single composite type that can hold the data from either group of machines.

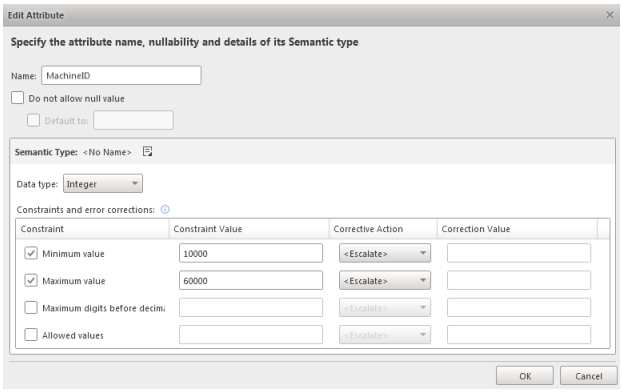
1. You will be able to rearrange the order of the data fields.
2. You will be able to specify type conversions, which will make your subsequent coding easier.
3. You will be able to assign constraints, that is, business rules, to the individual data fields, which provides a mechanism to implement data cleansing.

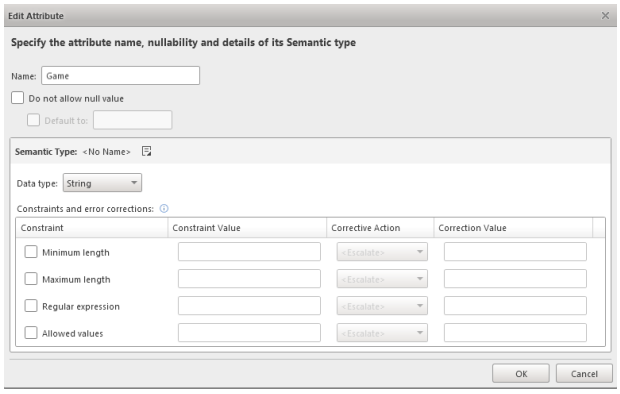


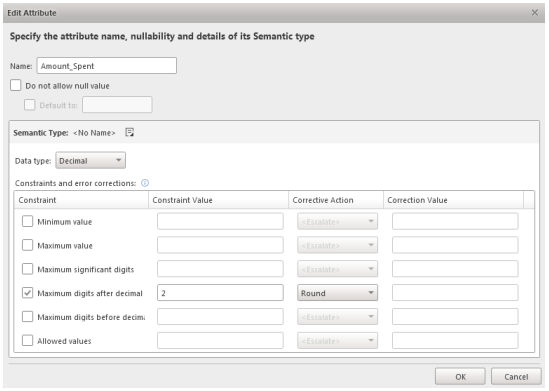
This composite type, which you should name MachineRecord, includes five attributes. Note that the data type of the MachineID attribute is Integer, which means that a data transformation will be needed when reading the data files. This is actually very easy to implement as the attribute mapping within the schema will provide QlikView Expressor with all the information it needs to affect this change. Also, the attribute mapping will be able to reformat the differing datetime representations of the source files. Within an QlikView Expressor dataflow, all datetime values are represented as CCYY-MM-DD HH24:MI:SS, but your mapping will allow other formats to be read and automatically converted into the default representation.

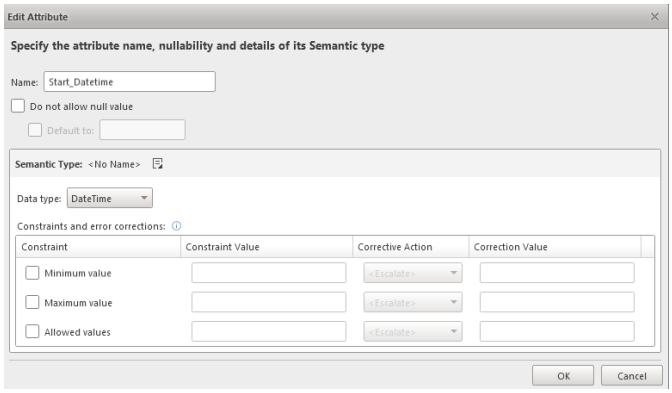
To get started, create this composite type.

1. Work in the QVE_Labs workspace within Project1.0.
2. In the Desktop Explorer panel, expand the project, highlight **Types**, right-click and select **New – Composite** Type from the popup menu.
3. Name the type **MachineRecord** and click **Create**, which open the Types Editor.
4. For each attribute, click **Add Attribute** in the Add Items grouping on the Edit tab of the editor's ribbon bar.
    a. Use the following screen shots as guides when creating the individual attributes.
        i. <u>The properties of the start and end datetime attributes are the same except for their names</u>.
    b. For some of the attributes, you will specify constraints.
        i. For the MachineID, set a minimum value of 10000 and a maximum value of 60000; machines in the casino all have identifiers in the 10000 range and machines in the hotel lobby have identifiers in the 50000 range; machines in other locations have different MachineID ranges and there are no machine identifiers below 10000 or above 60000. Select **<escalate>** as the Corrective Action.
        ii. For Amount_Spent, specify a maximum of 2 digits after the decimal point and select **Round** from the Corrective Action dropdown. If a value has more than 2 decimal digits, QlikView Expressor will automatically round the value.









5. Save your type and close the editor.

The Schema Artifacts

You will need four Schema artifacts for this example.

- A Schema used when reading the data file from the machines in the hotel lobby.
- A Schema used when reading the data file from the machines in the casino.
- A Schema used to write the output file.
- A Schema used to write output data to a database table (created in a following exercise).

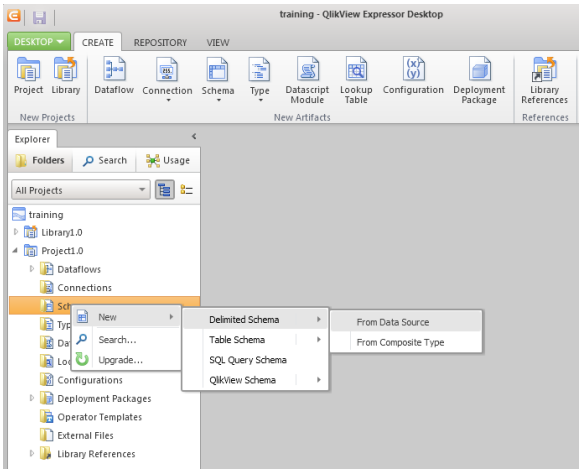
You can use the source files hotel.txt and casino.txt to create the Delimited Schemas needed to read these files. You will use the **MachineRecord** composite type to create the schema for the output file.

In order to create schemas from the source files, you need a Connection artifact that points to the directory containing the files. You have already created this connection, so there is nothing more you need to do.

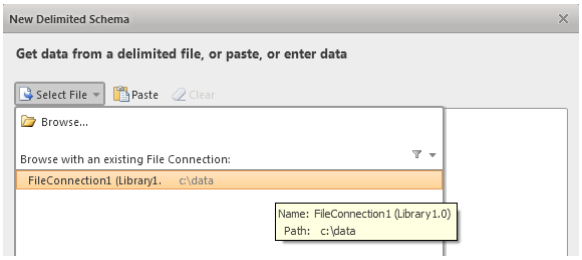
NOTE: Extract the file demo.zip into the directory C:\data.

Hotel Lobby Data File Schema

1. In the Desktop Explorer, expand Project1.0, highlight the **Schemas** folder, right-click, and select **New – Delimited Schema – From Data Source** from the popup menu.

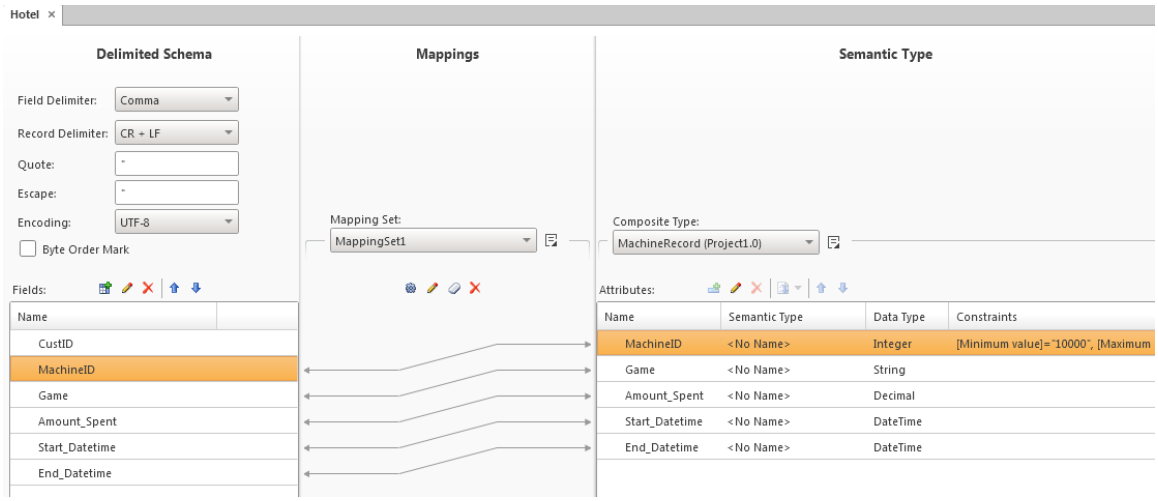


2. In the first window of the New Delimited Schema wizard, click **Select File** and click on your file connection.

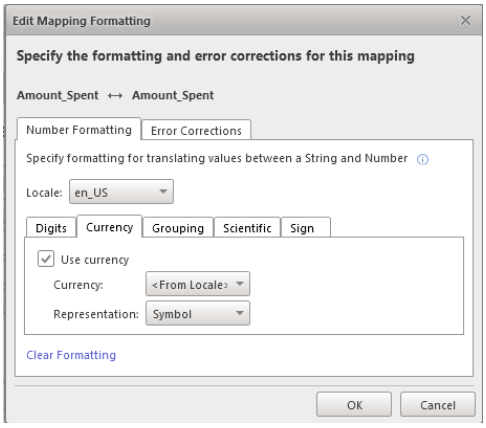


   a. This opens a browse window to the file system location pointed to by the connection.
3. Highlight the **hotel.txt** file and click **Open**.
   a. The first 10 lines of the file are copied into the wizard's panel (however, this file only includes a header and 3 data rows).
4. Examine the file's content and observe that the field delimiter is the comma character.
   a. Confirm that the selection in the Field Delimiter drop down control is **Comma**.
   b. Since this is a Windows text file, the appropriate selection in the Record Delimiter drop down control is **CR+LF**.
   c. Click **Next**.
5. In the following window, you specify the names for each field in the data record. Since this file includes a header row, and the header names are appropriate,
   a. Click **Set All Names from Selected Row**, which copies the header values into the field names.
   b. Click **Next**.
6. In the last window, select where you want to store this artifact (Project1.0), give the artifact a meaningful name (e.g., Hotel), and click **Finish**.
   a. The Schema artifact appears under the Schemas folder.
7. Double-click on the Hotel schema to open it in the Schema Editor.
8. Click **Assign – Add to Schema – Shared** in the Semantic Type grouping of the Schema – Edit tab.
   a. Select the **MachineRecord** composite type.

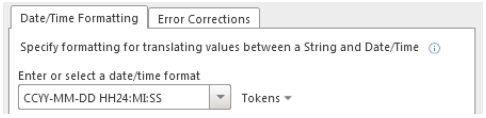
b. Accept the offer to automatically create mappings.



9. Highlight the **MachineID** mapping line and click on the pencil icon to open the Edit Mapping Formatting window.
    a. Select the **Digits** tab and specify 0 as the maximum number of digits after the decimal.
10. Open the **Amount_Spent** mapping and
    a. On the **Digits** tab, specify 2 as the number of minimum and maximum digits after the decimal.
    b. On the **Currency** tab, check Use currency and select **<From local>**[2] and **Symbol** from the drop down controls.



    i. QlikView Expressor will now understand how to handle the dollar sign.
11. Open the **Start_Datetime** and **End_Datetime** mappings and specify CCYY-MM-DD HH24:MI:SS as the date/time format.



[2] With non-US operating systems, you may need to explicitly select USD as the currency type or change the Locale to en_US.

12. Save the modified schema and close the editor.

Casino Data File Schema

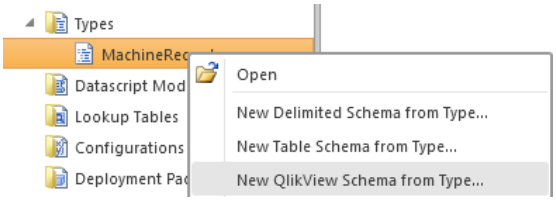
1. Use the New Delimited Schema wizard to create a schema for the file **casino.txt**.
2. Double-click on the Casino schema to open it in the Schema Editor.
3. Click **Assign – Add to Schema – Shared** in the Semantic Type grouping of the Schema – Edit tab.
    a. Select the **MachineRecord** composite type.
    b. Accept the offer to automatically create mappings.
        i. Note that the mapping between the field Machine_ID and the attribute MachineID was not automatically created as the names differ.
4. Using your mouse, drag from Machine_ID to MachineID tocreate a mapping.
5. Highlight the **MachineID** mapping line and click on the pencil icon to open the Edit Mapping Formatting window.
    a. Select the **Digits** tab and specify 0 as the maximum number of digits after the decimal.
6. Open the Amount_Spent mapping and
    a. On the **Digits** tab, specify 2 as the minimum and maximum number of digits before and after the decimal.
    b. On the Currency tab, check Use currency and select **<From local>** and **Symbol** from the drop down controls.
7. Open the Start_Datetime and End_Datetime mappings and specify CCYY-MM-DD HH24:MI as the date/time format.
8. Save the modified schema and close the editor.

Output File Schema

You have a choice of file types for the output file. You could write to a delimited file, an Excel worksheet, or a QlikView .qvx or .qvd file. Any of these choices would be suitable input to QlikView and the process of creating a Schema artifact from the composite type that describes the data extracted from the source files is identical. So let's create a schema for a QlikView file. **The same schema is used to read and write .qvx and .qvd files.**

As with many graphical applications, there are multiple ways to accomplish most tasks. When you have a shared composite type, the easiest approach is to work directly with the type.

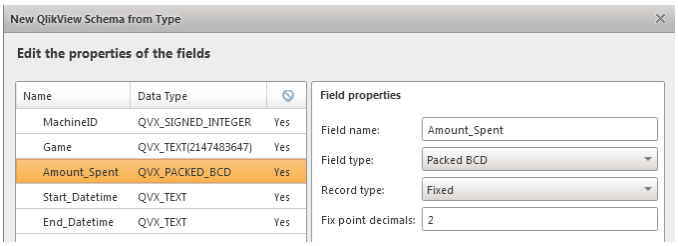
1. Expand the **Types** subdirectory to expose the entry for the **MachineRecord** composite type.
2. Highlight this entry, right-click and select **New QlikView Schema From Type…** from the popup menu.



    a. Notice that you have just as easily chosen a delimited file or database table schema.

b. If you had enabled the Excel and Salesforce extension, you would have been able to create schemas for these resources.

3. The New QlikView Schema from Type wizard starts and gives you an opportunity to confirm that you have selected the correct type.
    a. Click **Next**.
4. In the next window, you can review, and if necessary change, the properties of each of the fields in the QlikView file.



5. In the final screen of the wizard, give the Schema artifact a meaningful name (e.g., MachineRecord) and select your project as the storage location.
    a. Click **Finish**.

Remember that the same schema artifact can be used to read and write .qvx and .qvd files. Although QlikView Expressor can read .qvd files generated by a QlikView load script, only .qvx files generated by Expressor can be read by Expressor.

If you want to read an existing .qvd file, you can create a schema by selecting **New > QlikView Schema > From Data Source** from the popup menu. You can also use the same approach to create schemas from the table definitions included within a .qvw file.

6 Funnel Operator – Concatenate Load

The Funnel operator combines data from multiple inputs that have the same record structure into a single output stream. When this output becomes the source for a LOAD statement in a QlikView script, the Funnel operator accomplishes the same objective as a series of LOAD and CONCATENATE LOAD statements.

Use this operator to combine data from two or more pathways into a stream that can be delivered to an operator that accepts data on only one input. This operator emits the incoming record from whichever input port that has an available record so that it cannot become blocked or deadlocked waiting for input from a specific port. The order in which data is selected from the input ports is non-deterministic and may vary from run to run of a dataflow, even with the same data being processed.

When you configure this operator, you only need to specify the number of input ports, which may be between two and ten. Note that the structure of the record associated with the multiple inputs must be the same. The Funnel operator simply transfers records from one of the input ports to the output port; it does not transform the records.

Even when you want to concatenate incoming records with slightly different complement of fields, the structure of the emitted record is determined by the composite type used within the Schema. Consequently, implementation of a forced concatenation is no more involved than an unforced concatenation.

7 The Dataflow

The QlikView Expressor dataflow is a graphical representation of your data integration application. Dataflows are constructed in the QlikView Expressor Desktop dataflow editor, which is active when a dataflow has been created and opened for editing within a project or library.

Operators are the building blocks of a dataflow. The operators representing the various operations are listed in the left hand panel when the Operators tab is selected. Each operator performs a well-defined operation in a data integration application. After the operators are placed onto the dataflow panel, they must be connected and configured. Configured operators can be saved as templates and reused.

Dataflows are run from within Desktop and in a deployment or production environment. Desktop is usually the first test environment for an application. When a dataflow is complete and ready to be deployed into a production environment, it is added to a Deployment Package. In a Deployment Package, a dataflow is compiled with all the other artifacts it uses (Connections, Schemas, etc). The Deployment Package can be placed on a system running the QlikView Expressor Data Integration Engine, and individual dataflows are executed there. A dataflow in a Deployment Package may also be run from a QlikView script through the functionality of the QlikView Connector (described in a following section).

Dataflows, and all other project artifacts, can also be stored in a QlikView Expressor Repository. There they are placed under version control and can be shared with other developers and users. To be stored by Repository, the dataflow must be contained within a Repository Workspace.

Dataflows start with input operators, such as Read File and Read Table, which bring data in from an outside source. Input operators pass the data on to operators that perform actions on the data, such as Transform, Sort, Filter, and Join, and finally the data goes to an output operator, such as Write File or Write Table. The data flows from operator to operator through channels that are represented by links in the dataflow that connect the operators to one another.

Input operators, which receive data from an external source rather than another operator, read data whose structure and format is defined by a Schema. The QlikView Expressor schema artifact assigned to an input operator must match the structure of the data in the external source. For example, databases have schemas that define the structure of tables or views in the database, and the QlikView Expressor schema assigned to an input or output operator must be consistent with that structure.

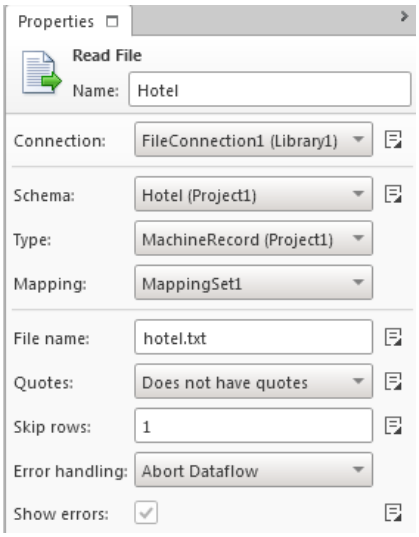
Once an input operator has read data that matches its schema, it then maps the schema's fields to composite type attributes that specify the format of the data as it flows "downstream" to the next operator.

Output operators must know the composite type attributes of data they will receive, but instead of passing data to another operator, they take the data received from the "upstream" operator and map from its attributes to the schema fields that defines its format in the external system.

Certain operators, such as Transform and Aggregate, modify the data in ways that require their operation be precisely specified for the particular dataflow. In addition to property configuration, transformation rules must be written for those operators that transform or aggregate data. Those rules are specified by mapping input and output in the QlikView Expressor Rules Editor. Expressor Datascript is used to construct transformations in the rules editor.

Exercise: Designing an Expressor Dataflow

1. In Desktop Explorer, in Project1.0, right-click on the **Dataflows** subdirectory and select **New…** to open the New Dataflow dialogue box.
2. Name the dataflow **SlotMachineRevenue**.
3. From the Inputs grouping, drag a **Read File** operator onto the canvas and configure it to process the **hotel.txt** source file.
    a. In this operator's Properties sheet make the following entries.

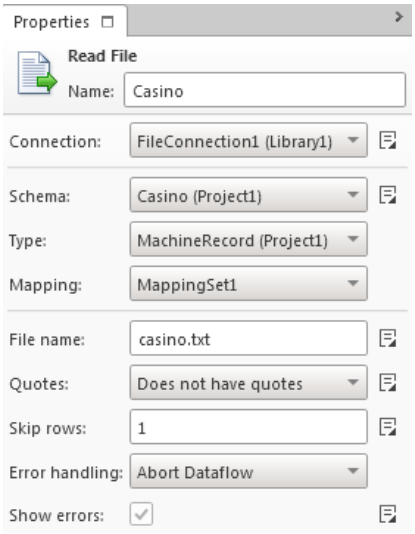


b. You must select the proper type – **MachineRecord** – from the Type drop down control.

c. Be sure to enter 1 into the Skip rows text box.

   i. This ensures that the header row will be skipped.

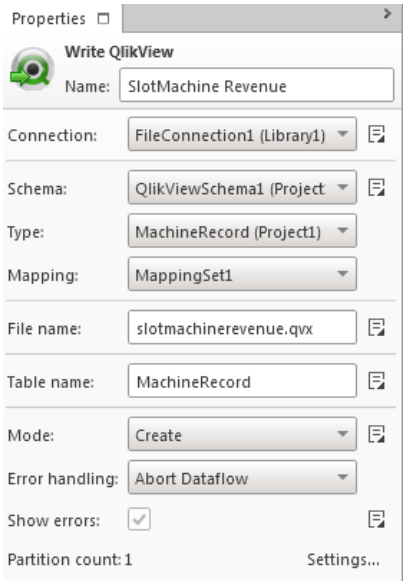
4. Drag another **Read File** operator onto the canvas and configure it to process the casino.txt source file.



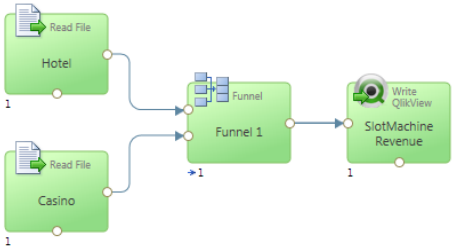
5. From the Utility grouping, drag a **Funnel** operator onto the canvas and connect the outputs from the Read File operators to the Funnel operator's inputs.

6. Finally, from the Output grouping, drag a **Write QlikView** operator onto the canvas and connect the Funnel operator's output to the Write QlikView operator's input.

   a. In this operator's Properties sheet make the following entries.



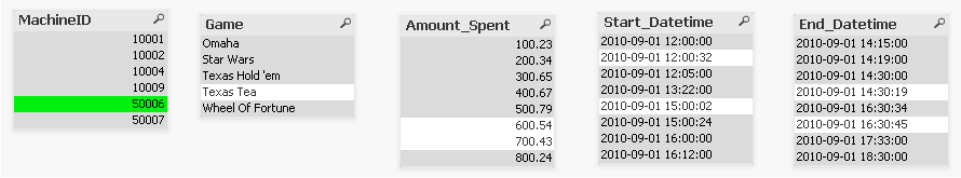
7. Save the completed dataflow.



   a. Note that when properly configured, all the operators turn green while a yellow color indicates that an operator is not yet properly configured.

8. To run the dataflow, click the **Start** button in the **Run** grouping of the **Dataflow – Build** tab of the ribbon bar.

9. Use the output file as a data source in a QlikView application. Do not enable the relative path option.



MachineID	Game	Amount_Spent	Start_Datetime	End_Datetime
10001	Omaha	100.23	2010-09-01 12:00:00	2010-09-01 14:15:00
10002	Star Wars	200.34	2010-09-01 12:00:32	2010-09-01 14:19:00
10004	Texas Hold 'em	300.65	2010-09-01 12:05:00	2010-09-01 14:30:00
10009	Texas Tea	400.67	2010-09-01 13:22:00	2010-09-01 14:30:19
50006	Wheel Of Fortune	500.79	2010-09-01 15:00:02	2010-09-01 16:30:34
50007		600.54	2010-09-01 15:00:24	2010-09-01 16:30:45
		700.43	2010-09-01 16:00:00	2010-09-01 17:33:00
		800.24	2010-09-01 16:12:00	2010-09-01 18:30:00

10. Note the 800.24 entry in the Amount_Spent text box. This value was included in the third record in hotel.txt as $800.236.
    a. The original value failed the 2 digits after decimal point constraint and the rounding corrective action was applied to the value.
    b. The same corrective action was applied to the value $500.788 in casino.txt, resulting in the entry 500.79 in the text box.
11. Change the Write QlikView operator so that it emits a .qvd file.
    a. Rerun the dataflow and display the results in a QlikView application.

We will investigate the effect of the Escalate corrective action in a later exercise on Error Handling.

Exercise: A Comparable QlikView Load Script

You can combine the data from the two slot machine locations using a CONCATENATE LOAD in a QlikView script. Just be certain to format the data as in the Expressor example.

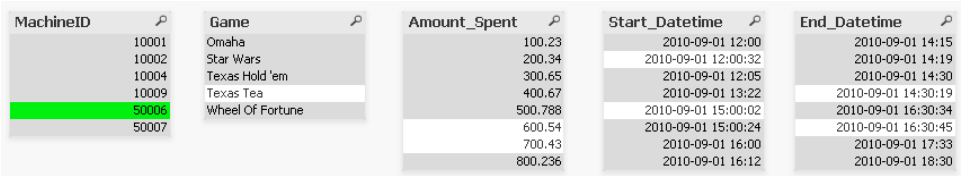
- Remove the dollar sign from the Amount_Spent value.
- Remove the decimal digit from the casino Machine_ID values.

Main

```
SET ThousandSep=',';
SET DecimalSep='.';
SET MoneyThousandSep=',';
SET MoneyDecimalSep='.';
SET MoneyFormat='$#,##0.00;($#,##0.00)';
SET TimeFormat='h:mm:ss TT';
SET DateFormat='M/D/YYYY';
SET TimestampFormat='M/D/YYYY h:mm:ss[.fff] TT';
SET MonthNames='Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec';
SET DayNames='Mon;Tue;Wed;Thu;Fri;Sat;Sun';

LOAD num(Machine_ID,'#####') AS MachineID,
     Start_Datetime,
     End_Datetime,
     Game,
     num(Amount_Spent) AS Amount_Spent
FROM
C:\data\casino.txt
(txt, codepage is 1252, embedded labels, delimiter is ',', msq);

CONCATENATE LOAD //CustID,
     MachineID,
     Game,
     num(Amount_Spent) AS Amount_Spent,
     Start_Datetime,
     End_Datetime
FROM
C:\data\hotel.txt
(txt, codepage is 1252, embedded labels, delimiter is ',', msq);
```



MachineID	Game	Amount_Spent	Start_Datetime	End_Datetime
10001	Omaha	100.23	2010-09-01 12:00	2010-09-01 14:15
10002	Star Wars	200.34	2010-09-01 12:00:32	2010-09-01 14:19
10004	Texas Hold 'em	300.65	2010-09-01 12:05	2010-09-01 14:30
10009	Texas Tea	400.67	2010-09-01 13:22	2010-09-01 14:30:19
50006	Wheel Of Fortune	500.788	2010-09-01 15:00:02	2010-09-01 16:30:34
50007		600.54	2010-09-01 15:00:24	2010-09-01 16:30:45
		700.43	2010-09-01 16:00	2010-09-01 17:33
		800.236	2010-09-01 16:12	2010-09-01 18:30

8 Managing Log File Output

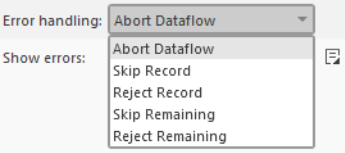
When Desktop runs a dataflow, logging information is displayed in the Results tab of the Status panel. Currently the detail of the logging messages is not managed from within Desktop but by setting an environment variable in the Environment Variables window accessed from the **Start > Control Panel > System > Advanced > Environment Variables** button.

Create a System variable named **EXP_TRACE_LEVEL** and set it to a value between 1 and 7, which has the effects summarized in the following table. Specifying a value for this variable is completely optional and should only be done if you need to alter the detail of the logging messages, for example when tracking down a connectivity issue. If you do not explicitly set this variable, the default setting is 4.

EXP_TRACE_LEVEL	Description
1	Logs critical errors and failures and a description of why the error occurred.
2	Logs noncritical errors and failures such as the fact that the license file is about to expire.
3	Logs noticeable events.
4	Logs standard operational information such as initialization of operators or anything out of the ordinary.
5	Logs extended operational information.
6	Logs standard debugging information.
7	Logs extended debugging information.

9 Error Handling

In the QlikView Expressor example you just developed, you set maximum and minimum value constraints for the MachineID attribute in the MachineRecord composite type. If a value is outside of this range, it will fail the constraint and Expressor will apply the corrective action. In this example, you set the corrective action to Escalate, which means the choice of how to handle the offending record will be made by the operator. Alternatives, which would have been carried out directly on the offending record, were to set the field to null or to a default value or to modify the value for example by rounding or truncating.



When you configured the Read File operators in the dataflow, you left the **Error handling** property set to its default – Abort Dataflow.

Consequently, when a field's value fails a constraint the operator will terminate the dataflow. This may be what you want to do, but it is a little harsh.

You can see in the above image that there are alternative error handling options that will allow the processing to continue.

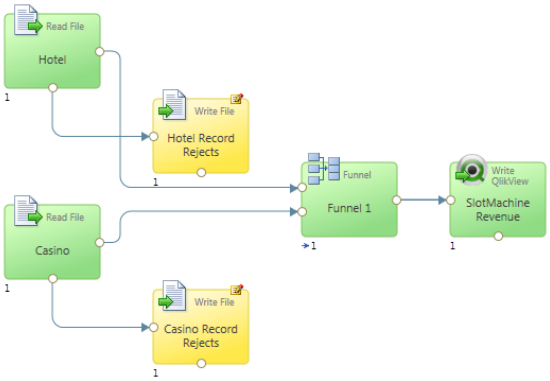
Error Handling Option	Effect
Abort Dataflow	Immediately terminates execution of the dataflow.
Skip Record	Skips the offending record. Continues processing.
Reject Record	Rejects the offending record. Continues processing.
Skip Remaining	Skips the offending record and all following records. Continues processing records that have already been read.
Reject Remaining	Rejects the offending record and all following records. Continues processing records that have already been read.

Regardless of which option you select, the log generated while the dataflow executes will include information that documents how the offending record was handled.

Reject ports are also available with the Transform and all output operators.

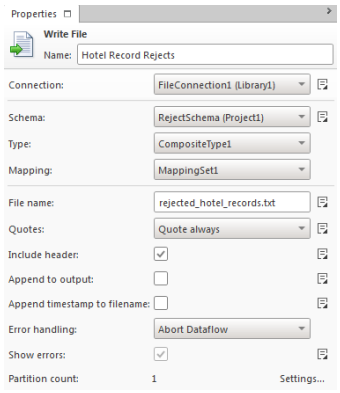
Exercise: Handling Errors

1. Open one, or both, of the data files: hotel.txt, casino.txt.
2. Insert additional records where the MachineID value will fail the constraint test, i.e., less than 10000 or greater than 60000.
    a. Simply copy an existing record then change the value of the MachineID.
3. Rerun the dataflow.
    a. Note that the processing aborts.
4. View the log.
    a. You may find it easier to copy the log into a text file so you can search for the character string 'Exception.'
    b. Notice that the log details why the record failed the constraint.
    c. If you have multiple offending records, processing terminates when the first is encountered.
        i. All offending records need to be corrected before the dataflow will successfully run.
5. Open the dataflow and change the **Error handling** property of the Read File operators to **Skip Record**.
6. Rerun the dataflow.
    a. View the log.
        i. Notice that the log documents the number of skipped records.
7. Open the dataflow and change the **Error handling** property of the Read File operators to **Reject Record**.
    a. If a record fails a constraint test, it will be directed out the operator's reject port, which is on its bottom edge.
    b. You will be able to collect any rejected records, which gives you the opportunity to correct the record and resubmit for processing.
8. Connect Write File operators to the reject ports on both Read File operators.



    a. Select the same File Connection artifact as the Read File operators.
    b. Enter a unique name into each operator's File name property, for example, rejected_casino_records.txt and rejected_hotel_records.txt.
    c. Select **Quote always** from the **Quotes** property drop down control.
        i. You will find it very useful to select the Quote as needed or Quote always option as it will make reviewing the contents of the file containing the rejected record information easier.
    d. Check the **Include header** box.

9. Now you need to specify a schema. Fortunately, the Read File operator reject port already has an assigned composite type and this type can be used to create the necessary schema.
    a. Select one of the new Write File operators and click the **Actions** button next to the schema drop down control.
        i. Select **New Delimited Schema from Upstream Output…** from the popup menu.
        ii. Step through the wizard, giving the schema a descriptive name, for example RejectPortSchema.



        iii. Once you have created the schema, you can also select this schema for the other Write File operator's schema property.
10. Rerun the dataflow.
    a. View the log.
        i. Notice that the log documents the number of rejected records.
11. Examine the contents of the file containing the rejected records.

Each rejected record is encapsulated in a larger, 5 field record that also includes the reason the record was rejected.

```
RejectType,RecordNumber,RecordData,RejectReason,RejectMessage
```

The actual record that failed the constraint test is contained in the third field (Note that the entire original record including the fields not represented in the MachineRecord composite type is included in the reject record.)

```
"1008,60007,Omaha,$800.236,2010-09-01 15:00:24,2010-09-01 16:30:34"
```

and an explanation for the failure is in the last field.

```
"The value '60,007' for attribute 'MachineID' is higher than the
defined limit of '60,000'"
```

Also observe that the quotation marks make it much easier to interpret the file's content as some of the fields in the reject record include comma delimiters and/or embedded quotation marks.

In the previous exercise, you explicitly set constraints against which a record's attribute values were tested. But with the Read File operator there is another situation that could raise an error that you need to handle. Specifically, what to do if an incoming record does not adhere to the structure defined in the schema.

For example, suppose your schema describes a delimited file that employs the comma character as the field delimiter but some rows are terminated with additional commas. In this case, the operator will not know how to handle these rows as the extra field delimiters specify additional fields that the schema does not describe.

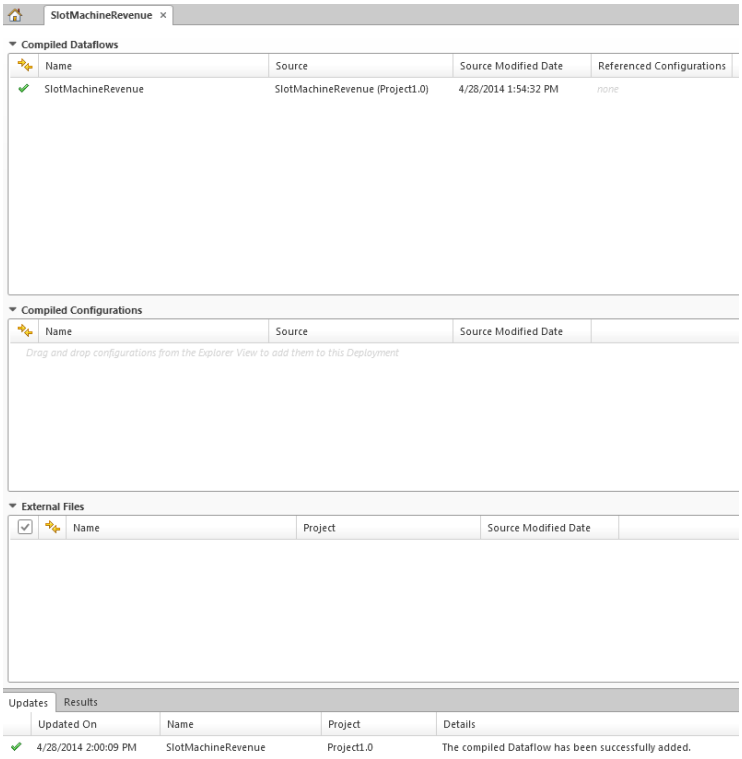
1. Open one of the data files and add two or more commas to the end of any of the original rows or replace one of the commas in the middle of the row with two adjacent commas.
2. Rerun the dataflow.
3. Examine the contents of the file containing the rejected records.
    a. Note that the RejectMessage describes the problem with the modified record.

10 The Deployment Package

Up to this point, you have been running your data integration applications from within Desktop. However, Desktop is not available on a computer on which the Expressor Data Integration Engine has been installed. To run your application in the Engine or to access your application from a QlikView load script via the QlikView Connector, you need to create a deployment package, a self-contained compiled version of your application.

Each dataflow, and its associated Connection, Schema, etc. artifacts are compiled into an operating system neutral form that contains everything needed to run the application.

If the compilation process raises any errors, a descriptive message will be displayed. You must resolve all issues before a valid deployment package can be created.



Exercise: Creating a Deployment Package

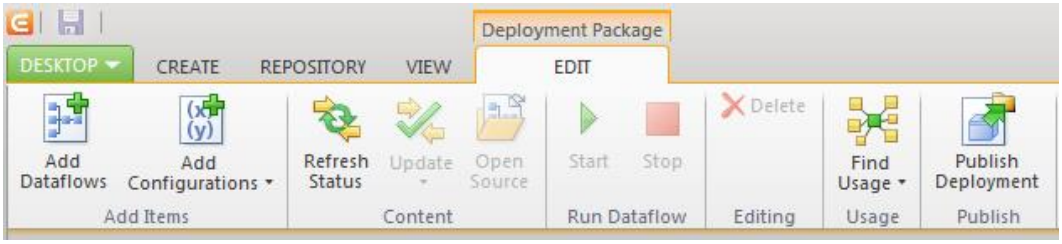
1. Expand the project (Project1.0) that includes the dataflow, or dataflows, you want to include in the deployment package.
    a. Note the subdirectory Deployment Packages.
    b. This subdirectory is only included within a project.
2. Right-click on the **Deployment Packages** subdirectory and select **New…** from the popup menu.

3. In the New Deployment Package window, give the package a meaningful name, enter a description, and confirm the desired storage location for the package.
4. Click **Create** to open the deployment package editor.
5. Drag and drop one or more dataflow entries into the editor's upper panel.
    a. Add the **SlotMachineRevenue** dataflow to the deployment package.
    b. The compilation process may take a minute or so depending on the size, complexity, and number of dataflows included in the package.
    c. The deployment package is automatically saved.

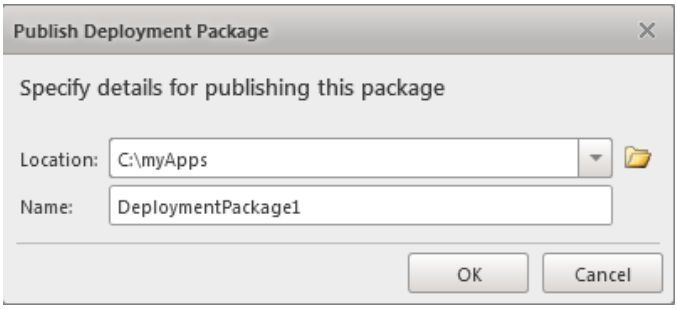
Exercise: Moving the Deployment Package

While there is nothing wrong with leaving the deployment package in its default location within the directory hierarchy of your project, its location is pretty far down in the hierarchy, which will be inconvenient when integrating the package with a QlikView application. While you could open Windows Explorer, find the deployment package directory and copy (do not move) it into a different file system location, Expressor Desktop will carry out these steps for you.

1. Create a target directory into which the deployment package will be placed, for example, `C:\myapps`.
2. Open the deployment package and in the **Deployment Package – EDIT** ribbon bar, click the **Publish Deployment** button in the Publish grouping.



    a. This opens the **Publish Deployment Package** window.
3. Browse to or enter the location of your target directory and give a name to the subdirectory that will hold the package and click **OK**.

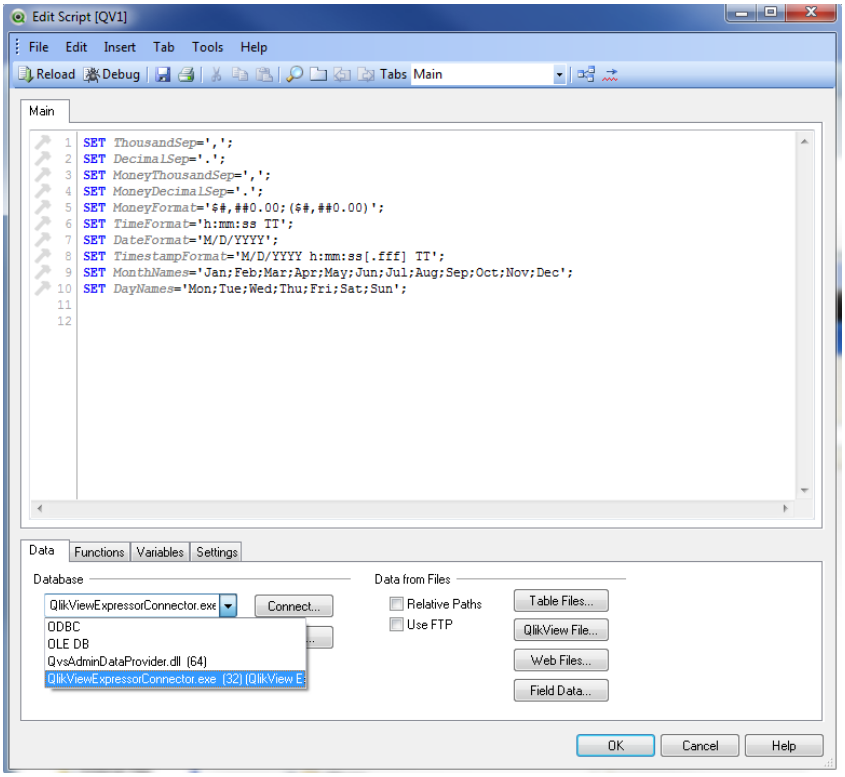


4. After the process completes, a confirmatory message will be displayed.

11 QlikView Connector

The QlikView Connector, which is a feature of QlikView Expressor installed with Desktop or the Data Integration Engine, is a collection of libraries that allow a dataflow to be used as a QlikView data source in the same way that a relational database can be used. Once you have created your QlikView script, each command to reload will first run the dataflow so that the data displayed in the dashboard is current.

When you install either Desktop or the Engine, the C:\Program Files (x86)\Common Files\QlikTech\Custom Data\QlikViewExpressorConnector directory is created. This directory, which contains the executable applications and DLLs comprising the QlikView Connector, will be visible to QlikView and the QlikView Connector will appear as an option in the Database drop down in the QlikView Edit Script window.



In order to use the QlikView Connector as a data source, the dataflow must contain at least one Write QlikView operator that generates a .qvx file. The dataflow may have other output operators including additional Write QlikView operators, but the script must load data from a .qvx file before loading the data from other output operators.

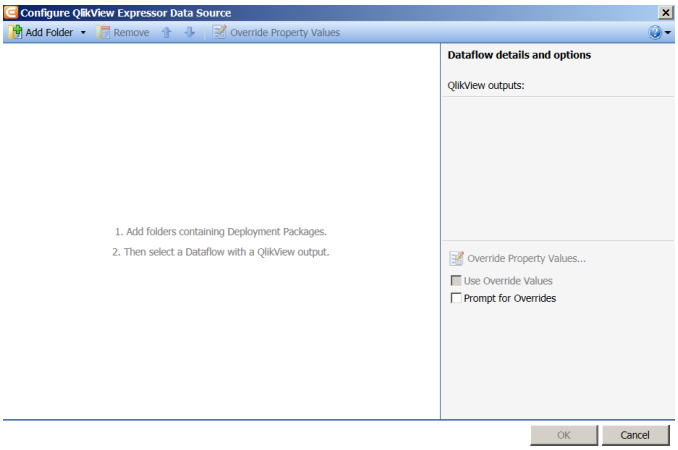
Before you can use the QlikView Connector in a QlikView script, you must first run the dataflow so that the .qvx output file needed to describe the data to be loaded through the connector already exists.

In order to run a dataflow from the QlikView script, the dataflow must be contained in a Deployment Package and the directory holding the Deployment Package and Desktop or the Expressor Data Integration Engine must be on the same computer as QlikView Personal Edition, QlikView Desktop, or QlikView Server/Publisher.

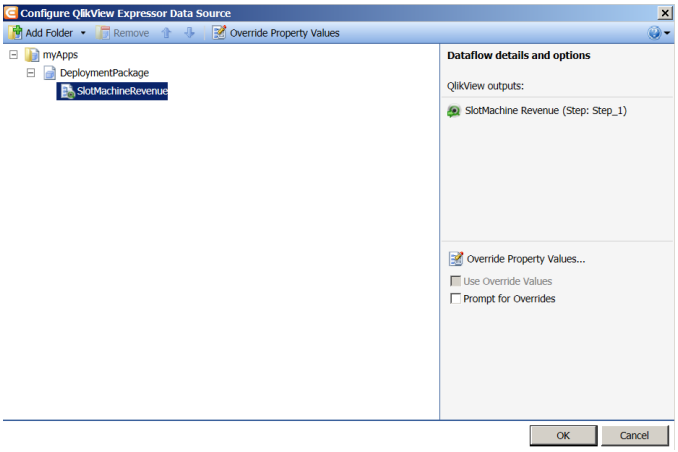
Exercise: Using the QlikView Connector[3]

The QlikView Connector requires that your application is in a deployment package.

1. Open the QlikView Edit Script window.
2. Select the **QlikView Connector** from the Database drop down control
3. Click **Connect…**, which opens the Configure expressor Data Source window.

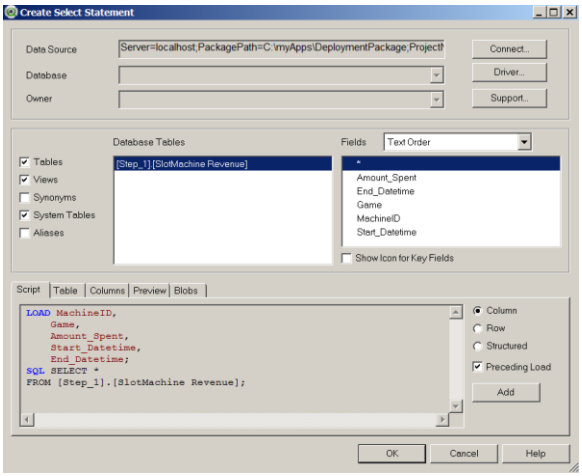


4. Click **Add Folder**, which opens the Browse For Folder window.
    a. Select the folder that holds your deployment package `C:\myApps\SlotMachinePackage`.
5. Expand the tree and highlight the name of the dataflow.



[3] With the QlikView Expressor3.11 and 3.12 releases, the QlikView Expressor Connector only works with .qvx files. A later point release will provide for functionality with .qvd files.

6. Click **OK**.
    a. This places a CUSTOM CONNECT statement into the QlikView Script.
7. Click **Select…**, which opens the Create Select Statement window.
8. In the Database Tables list box, select the desired Write QlikView operator.
    a. In this example, there is only one entry: **SlotMachine Revenue**.
9. In the Fields list box, select ***** then click **OK**.



    a. You cannot select individual fields even if you select all fields; you must highlight the * entry.
    b. A LOAD/SQL SELECT statement appears in the QlikView script.

```
LOAD MachineID,
    Game,
    Amount_Spent,
    Start_Datetime,
    End_Datetime;
SQL SELECT *
FROM [Step_1].[SlotMachine Revenue];
```

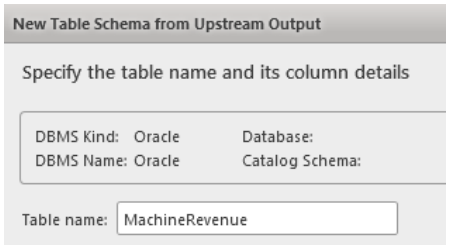
        i. The FROM clause references the Write QlikView operator.
    c. If your QlikView dashboard does not require all of the data retrieved from this .qvx file, comment out the unneeded fields in the LOAD statement.
10. If your dataflow includes other output operators, add the corresponding LOAD statements to the QlikView script.
    a. Only one set of CONNECT/SQL SELECT statements may use the QlikView Connector. Other files must use a standard LOAD statement.
    b. It is required that the CONNECT/SQL SELECT statement pair be the first data loading statement in the QlikView script.
        i. This ensures that the dataflow runs before any of the data generated by the dataflow is loaded into the QlikView application.
11. Save the script.
12. In the Edit Script window, click **Reload**.
    a. The dataflow runs, regenerating all of the output.
13. Make the desired field selections in the Sheet Properties window and click **OK**.
14. Look in the directory C:\data for the file named slotmachinerevenue.qvx and confirm that the timestamp indicates that this file was recently generated.

12 Database Access

QlikView Expressor has the ability to read from and write to database tables using the Read Table, SQL Query and Write Table operators. While using the Read Table operator is quite similar to using the Read File operator, the SQL Query operator gives you much fuller control over the SELECT statement by allowing specification of any valid SELECT statement. The SQL Query operator allows you to utilize the processing capabilities of the database management system. And the Write Table operator may be configured to perform record deletions, updates and merges as well as insertions.

Exercise – Using the Write Table Operator

1. Open the dataflow **SlotMachineRevenue**.
2. Select and delete the connector between the **Funnel** and **Write QlikView** operators.
3. Place a **Copy** operator onto the canvas and connect its input to the output from the **Funnel** operator.
    a. On the **Copy** operator's Properties sheet, set the number of outputs to 2.
4. Connect the **Write QlikView** operator to one of the outputs from the **Copy** operator.
5. Place a **Write Table** operator onto the canvas.
6. Connect the second output port from the **Copy** operator to the input port of the **Write Table** operator.
7. Click on the **Write Table** operator to display its Properties sheet.
    a. Enter **MachineRevenue** into the **Name** text box.
        i. Every operator in a dataflow step must have a unique name.
    b. From the **Connection** drop down control, select the database connection artifact you previously created.
    c. Click on the **Actions** button  next to the **Schema** drop down control.
        i. Select **New Table Schema from Upstream Output…** from the popup menu.
        ii. Step through the wizard, giving the schema a descriptive name, for example RevenueSchema.
            1. **Be certain to enter a meaningful name for the target table, otherwise the table will have the same name as the default composite type**.



            2. Specify a realistic size for each of the `varchar` fields (the default of 4000 characters is too large).
    b. The default **Type** entry already corresponds to the **MachineRecord** type, so there is no need to add this type to the schema, although doing so would be a best practice since it documents how the schema will be used.
    c. Select the **Truncate** and **Create Missing Table** options.

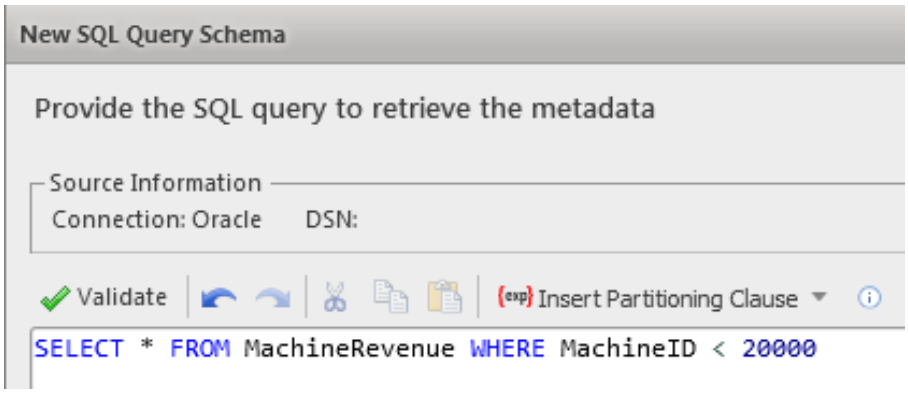
i. Each time the dataflow runs, the table will be emptied and repopulated.
ii. If the target table does not exist, it will be created.

d. In the drop down **Mode** control, select **Normal**. This specifies a table insert.

8. Save and rerun the dataflow.
9. Use the utilities provided with the RDBMS to view the populated table.
10. Now that you have a database table populated with the revenue data, develop another dataflow that deletes the entries from the casino from the database table.

Exercise – Using the SQL Query Operator

1. Create a new dataflow, perhaps named **SQLQuery** within a new project.
2. Place **SQL Query** and **Write QlikView** operators onto the canvas.
    a. Connect the input operator to the output operator.
3. Select the **SQL Query** operator to display its Properties sheet.
    a. From the **Connection** drop down control, select the database connection artifact you previously created.
    b. Click on the **Actions** button next to the **Schema** drop down control.
        i. Select **New SQL Query Schema…** from the popup menu.
        ii. In the first step of the wizard, select the database connection then click **Next**.
        iii. In the following step, enter your SELECT statement. For example, the following statement will retrieve only the records from the slot machines located in the hotel.



        iv. Click the **Validate** button to check the validity of your statement.
            1. You cannot create a schema from a statement that is not valid.
            NOTE: If you are using Oracle, and the Expressor Write Table operator created the table, you will need to put " around both the table and column names.
        v. Click **Next** and complete the remaining wizard steps.
    c. The default **Type** entry already corresponds to the **MachineRecord** type, so there is no need to add this type to the schema, although doing so would be a best practice since it documents how the schema will be used.
4. Configure the **Write QlikView** operator.
    a. You already have the required Connection and Schema artifacts.

b. Specify a .qvx file as output by using the .qvx extension in the output file's name, or specify a .qvd file as output by using the .qvd extension in the output file's name.

5. Save and run the dataflow and examine the content of the output file.

13 Rules Editor – Expression Rules

The Aggregate, Join, and Transform operators in the Transformers operator grouping are the operators into which you add business logic such as aggregations or data transformations. You can also use these operators to drop attributes from or add attributes to the record being processed.

In order to specify the business logic to perform, you use the Rules Editor to define an Aggregation, Expression, Function, or Lookup Rule. In a Join or Transform operator, an Expression Rule is equivalent to the right hand side of an assignment statement. That is, you write an expression, which may include nested function calls, that returns a single value. This expression may use any number of the input attributes or Expressor runtime parameters as arguments and its return value is used to initialize a single output attribute.

If your business logic cannot be implemented as an assignment statement, you use a Function Rule. With a Function Rule you are able to use any sort of programming construct, for example, loops or `if..then..else` clauses, to carry out your data transformations. If necessary, you can always convert an Expression Rule into a Function Rule without losing any of the work you have already done.

While Expression Rules will most likely support a majority of your processing objectives, Function Rules give you the most control over the processing and are therefore significantly more powerful. The initial exercises in this course works with Expression Rules; Function Rules are covered later in the course after the introduction to Expressor Datascript.

The Filter operator, in the Utility operator grouping, also offers a Rules Editor. Code that you include in this operator may only return the values true or false, which then determine how the record will be processed.

Although you are free to simply enter your processing logic, the Rules Editor provides many programming aids such as auto-completion and drop down menus, which help ensure that there are minimal spelling and logic errors. You can also quickly frame out a complex statement such as a loop by selecting a pre-designed code block. And as your scripting becomes more complex, the find/replace editing features will prove useful. All of these coding aids are located on the Edit tab of the Rules Editor.

In the Aggregate operator, the Aggregate Rule requires no scripting. With these rules, you may only specify one of the incoming attributes as an argument. Before you can select what type of aggregation to perform, you must also associate an output argument with the rule. Once you associate the input and output arguments with the rule, a drop down list is populated with valid aggregations for these

argument types. For example, if the input parameter is a numeric type, the output parameter must also be numeric and the possible aggregations will differ depending on whether the output attribute's type is integer or decimal. With an Aggregate Rule, the aggregations presented in the drop down list are your only options. If you want to perform more complex processing, you must use a Function Rule.

As with the Join and Transform operators, Function Rules in the Aggregate operator give you full control over the processing logic such as retaining each record for additional processing or performing multiple aggregation calculations in a single rule.

14 Join Operator – Join Load

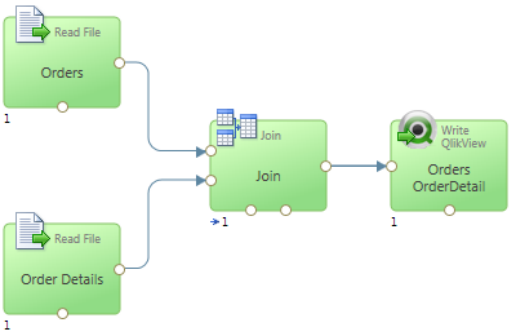
In QlikView loads, the use of the JOIN keyword before a LOAD statement allows the contents of a data source to be combined with the data in an existing table. In this processing, QlikView uses all fields in the two data sources with identical names as a composite join key. Depending on how the join operation is specified (i.e., INNER, LEFT, RIGHT, OUTER) it is possible that additional records will be generated. When the INNER or LEFT qualifiers are specified, the final table will contain one row for each row in the existing table with added fields from the data source referenced in the LOAD statement. When the RIGHT qualifier is specified, the final table may not contain every record from the existing table and may contain rows derived entirely from the data source referenced in the LOAD statement. And when the OUTER qualifier is specified, the final table will contain a row for each row in the existing table and may contain rows derived entirely from the data source referenced in the LOAD statement.

The Expressor Join operator provides the same capabilities. However, with the Expressor Join operator it is possible to have full control over the composition of the join key. If there is more than one identically named field in the two sources, the key does not need to include all these fields. Additionally, when an INNER, LEFT (join type 1), or RIGHT (join type 0) join is specified, it is possible to isolate the unjoined records, which may then be written to a separate output file and analyzed independently from the joined records.

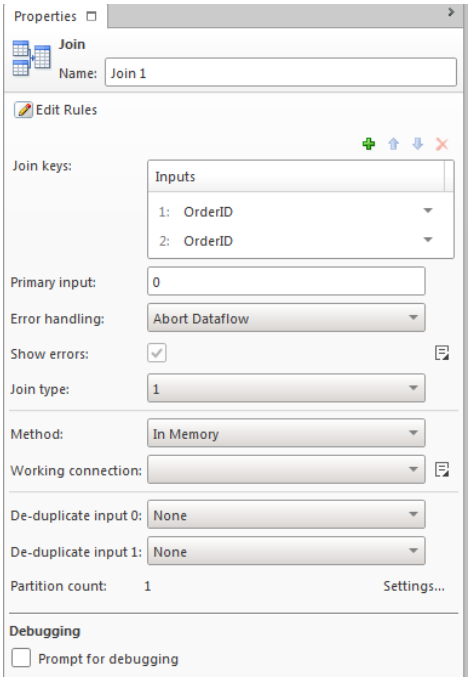
Exercise: The Join Operator – Join Load

1. Working in the QVE_Labs workspace, create a new project.
2. Create a File Connection artifact to the directory **C:\data** or create a reference to Library1.
3. Create a Delimited Schema for the file **orders.txt**.
4. Create a Delimited Schema for the file **order_details.txt**.

5. Create the following Dataflow named **TableJoin**.



   a. Configure the Join operator's properties as shown below.

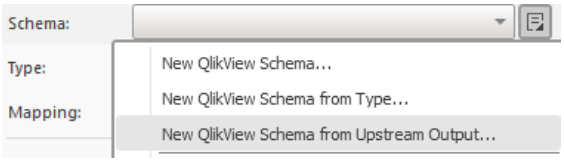


      i. In this example, the only field that is common to both inputs has the name OrderID.
         1. It is not required that the join fields have the same name but they must be the same type.
      ii. Join type 1 corresponds to a LEFT outer join.

   b. Use the same Connection for the **Read File** and **Write QlikView** operators.

   c. To create the schema used by the **Write QlikView** operator, connect its input to the upstream Join operator then click the **Actions** button adjacent to the Schema drop down control.

      i. Select the **New QlikView Schema from Upstream Output…** menu item.



ii. Accept all the defaults offered by the New QlikView Schema from Upstream Operator wizard.
    1. Enter **TableJoinQV** as the schema's name.

d. For the Write QlikView operator:
    i. Enter **orders_orderdetail.qvx** or **orders_orderdetail.qvd** as the File name.
    ii. Enter **QVE** as the Table name property.
    iii. Select **Create** for the Mode.

6. Run the Dataflow.

Now compare the output from the Expressor Dataflow to the table created through a LEFT JOIN in a QlikView script.

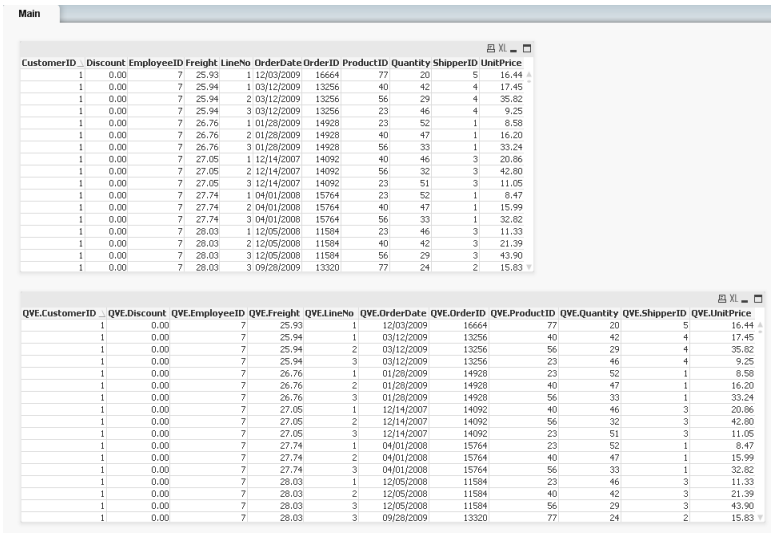
1. Open a new QlikView application.
2. In the script, load the file created by Expressor and the two source files.

```
Orders:
LOAD OrderID,
     OrderDate,
     CustomerID,
     EmployeeID,
     ShipperID,
     Freight
FROM
C:\data\orders.txt
(txt, codepage is 1252, embedded labels, delimiter is ',', msq);

Left Join (Orders)
LOAD OrderID,
     LineNo,
     ProductID,
     Quantity,
     UnitPrice,
     Discount
FROM
C:\data\order_details.txt
(txt, codepage is 1252, embedded labels, delimiter is ',', msq);

QUALIFY *;
LOAD OrderID,
     OrderDate,
     CustomerID,
     EmployeeID,
     ShipperID,
     Freight,
     LineNo,
     ProductID,
     Quantity,
     UnitPrice,
     Discount
FROM
C:\data\orders_orderdetail.qvx
(qvx);
UNQUALIFY;
```

3. Add a Table Box Sheet Object and include all the fields from the Expressor output file.
4. Add another Table Box Sheet Object and include all the fields from the QlikView Join Load.


Main
CustomerID_1 Discount EmployeeID Freight LineNo OrderDate OrderID ProductID Quantity ShipperID UnitPrice
1 0.00 7 25.93 1 12/03/2009 16664 77 20 5 16.44
1 0.00 7 25.94 1 03/12/2009 13256 40 42 4 17.45
1 0.00 7 25.94 2 03/12/2009 13256 56 29 4 35.82
1 0.00 7 25.94 3 03/12/2009 13256 23 46 4 9.25
1 0.00 7 26.76 1 01/28/2009 14928 23 52 1 8.58
1 0.00 7 26.76 2 01/28/2009 14928 40 47 1 16.20
1 0.00 7 26.76 3 01/28/2009 14928 56 33 1 33.24
1 0.00 7 27.05 1 12/14/2007 14092 40 46 3 20.86
1 0.00 7 27.05 2 12/14/2007 14092 56 32 3 42.80
1 0.00 7 27.05 3 12/14/2007 14092 23 51 3 11.05
1 0.00 7 27.74 1 04/01/2008 15764 23 52 1 8.47
1 0.00 7 27.74 2 04/01/2008 15764 40 47 1 15.99
1 0.00 7 27.74 3 04/01/2008 15764 56 33 1 32.82
1 0.00 7 28.03 1 12/05/2008 11584 23 46 3 11.33
1 0.00 7 28.03 2 12/05/2008 11584 40 42 3 21.39
1 0.00 7 28.03 3 12/05/2008 11584 56 29 3 43.90
1 0.00 7 28.03 3 09/28/2009 13320 77 24 2 15.83
QVE.CustomerID_1 QVE.Discount QVE.EmployeeID QVE.Freight QVE.LineNo QVE.OrderDate QVE.OrderID QVE.ProductID QVE.Quantity QVE.ShipperID QVE.UnitPrice
1 0.00 7 25.93 1 12/03/2009 16664 77 20 5 16.44
1 0.00 7 25.94 1 03/12/2009 13256 40 42 4 17.45
1 0.00 7 25.94 2 03/12/2009 13256 56 29 4 35.82
1 0.00 7 25.94 3 03/12/2009 13256 23 46 4 9.25
1 0.00 7 26.76 1 01/28/2009 14928 23 52 1 8.58
1 0.00 7 26.76 2 01/28/2009 14928 40 47 1 16.20
1 0.00 7 26.76 3 01/28/2009 14928 56 33 1 33.24
1 0.00 7 27.05 1 12/14/2007 14092 40 46 3 20.86
1 0.00 7 27.05 2 12/14/2007 14092 56 32 3 42.80
1 0.00 7 27.05 3 12/14/2007 14092 23 51 3 11.05
1 0.00 7 27.74 1 04/01/2008 15764 23 52 1 8.47
1 0.00 7 27.74 2 04/01/2008 15764 40 47 1 15.99
1 0.00 7 27.74 3 04/01/2008 15764 56 33 1 32.82
1 0.00 7 28.03 1 12/05/2008 11584 23 46 3 11.33
1 0.00 7 28.03 2 12/05/2008 11584 40 42 3 21.39
1 0.00 7 28.03 3 12/05/2008 11584 56 29 3 43.90
1 0.00 7 28.03 3 09/28/2009 13320 77 24 2 15.83

15 Lookup Table – Mapping Load

In QlikView loads, a Mapping Load is used to add a single field of data to a table. Additionally, you can specify an alternate value if there is no matching value in the mapping table. And since the mapping table is created independently of other tables, it is available throughout the entire script and may be referenced in multiple load statements. However, in the QlikView mapping table each mapped field must resolve to a single value.

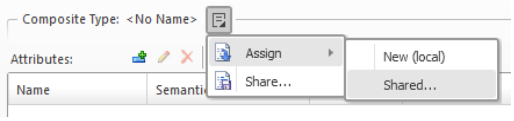
Expressor provides this same functionality and more through the Lookup Table. An Expressor Lookup Table may use one or more fields in determining the match, may emit one or more values for each match, and may be used for one-to-many relationships. If the Lookup Table does not contain a matching entry, you may provide alternative values for the output.

In a Dataflow, the Lookup Tables must be initialized before they are used. This is generally done in the first step of a multi-step Dataflow. In the following steps, values are extracted from Lookup Tables through a Lookup Expression Rule or Lookup Function Rule from within a Transform operator.

Exercise: Lookup Table – Mapping Load

1. Working in the QVE_Labs workspace, create a new project.
2. Copy the File Connection artifact from Library1 into the new project.
    a. Highlight the connection under Library.
    b. Right-click and select **Copy To – New project name** from the popup menu.
3. Copy the Delimited Schema for the file **orders.txt** into the new project.
    a. Highlight the schema.
    b. Right-click and select **Copy To – New project name** from the popup menu.
    c. Open this schema in the editor and change the type of the **ShipperID** attribute from String to Integer.
4. Create a Delimited Schema for the file **shippers.txt**.
    a. Name this schema **ShippersSchema**.
    b. Open this schema in the editor and change the type of the **ShipperID** attribute from String to Integer.
    c. Rename the attribute **CompanyName** to **ShipperName**.
5. In the Desktop Explorer panel, under this new project, highlight **Lookup Tables**, right-click, and select **New…** from the popup menu.
    a. Enter **Shippers** as the name.
    b. Click **Create**.
        i. The Lookup Table editor opens.
    c. You must now define the composite type that describes the structure of this lookup table.
        i. You may add attributes manually by clicking the  icon above the Attributes box.
            1. Add an integer attribute named **ShipperID**.
            2. Add a string attribute named **ShipperName**.

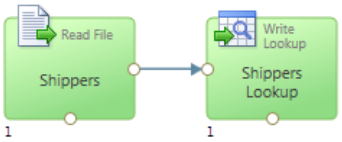
ii. Alternatively, you may import the composite type contained in the **ShippersSchema**.
    1. Click the **Actions** button and select **Assign – Shared…** from the popup menu.



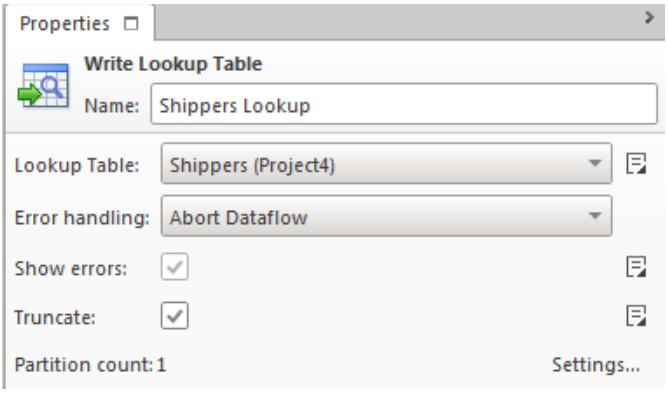
    2. Confirm that the Project drop down lists the current project.
    3. Select **Used by a Schema** from the Scope drop down and **ShippersSchema** from the adjacent drop down.

d. Now specify a unique key.
    i. In the Attributes box, highlight the **ShipperID** attribute and click **Add to New Key…**.
    ii. In the Add Key window, give your key a descriptive name, such as **Shipper**.
        1. Be certain the **Unique** checkbox is selected.
        2. Click **OK**.
    iii. An entry appears in the Keys box.
e. Close the editor, which saves the table definition.

6. Create a two-step Dataflow named Lookup.
    a. Step 1:
        i. Place the Read File and Write Lookup operators onto the canvas.

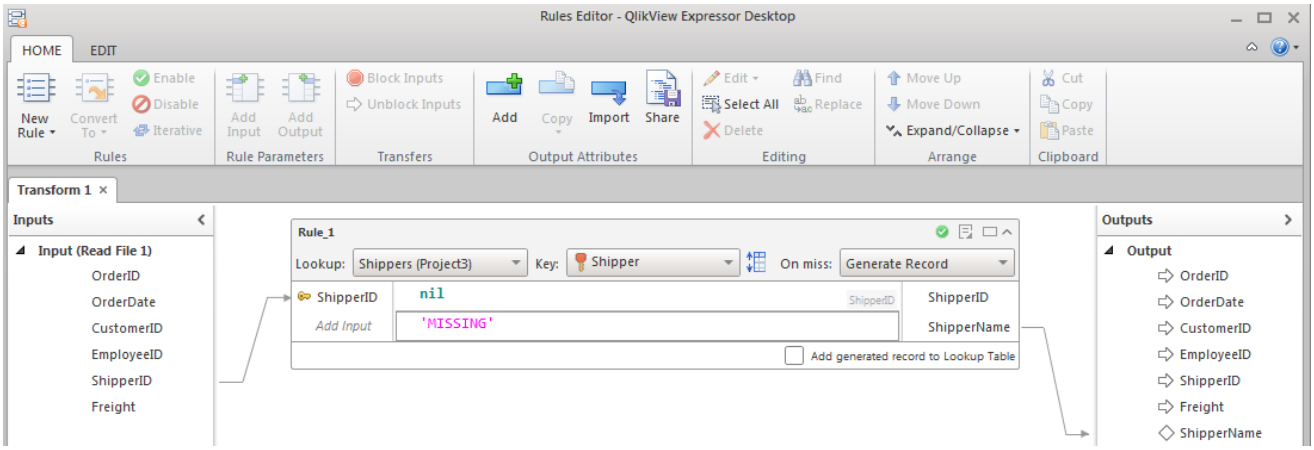


        ii. Configure the Read File operator to read the file **shippers.txt**.
        iii. Configure the Write Lookup operator to write to the Shippers lookup table.



    b. Step 2:
        i. In the **Dataflow – Build** ribbon bar, click **Add Step** in the Steps grouping.
            1. This adds a second tab, Step 2, to the canvas.
        ii. Place the Read File, Transform, and Write QlikView operators onto the canvas.

iii. Configure the Read File operator to read the file **orders.txt**.
iv. Double-click on the Transform operator to open the Rules Editor.
v. In the Home tab of the ribbon bar in the Rules grouping, click **New Rule** and select **Lookup Expression Rule** from the drop down menu.
    1. Select **Shippers** from the Lookup drop down.
    2. Since it is the only key, the name **Shipper** appears in the Key drop down.
    3. Select **Generate Record** from the **On miss** drop down.
vi. Drag your cursor from the **ShipperID** attribute in the Input panel to the **ShipperID** input parameter in the rule.
vii. Add another attribute to the Output panel.
    1. Click **Add** in the **Output Attributes** grouping of the Home tab in the ribbon bar.
    2. Enter **ShipperName** into the Name text box.
    3. Click **OK**.
    4. The attribute is added to the Output panel.
viii. Drag your cursor from the **ShipperName** output parameter in the rule to the **ShipperName** attribute in the Output panel.
ix. Enter **'MISSING'** into the rule adjacent to the **ShipperName** output parameter.



x. Configure the Write QlikView operator.
    1. Use the same connection.
    2. Create a schema from the output of the Transform operator.
    3. Write to either a .qvx or .qvd file.
    4. Give the QlikView table any name you want.

7. Run the Dataflow.

Now check the output in QlikView.

1. Open a new QlikView application.
2. Load the output file from the Expressor Dataflow.
3. Create a List box for the ShipperName field.
4. Create a Table Box for all fields.

5. Observe the entries in the ShipperName List box.
    a. Note that 'MISSING' is an entry.
6. Select the various ShipperName entries and note the associated orders.

16 Scripting Operators

QlikView Expressor includes many preprogrammed operators as summarized in the following list.

Category	Operators
Inputs	Read File Read Table Read Lookup Table SQL Query Read Custom Read Excel*** Read QlikView** Read Salesforce* Read SOQL*
Outputs	Trash Write File Write Table Write Lookup Table Write Parameters Write Custom Write Excel*** Write QlikView** Write Salesforce*
Transformers	Aggregate Join Transform Multi-Transform Pivot Row Pivot Column
Utility	Buffer Copy Filter Funnel Sort Unique
*** Requires installation of the Excel extension. ** Requires installation of the QlikView extension. * Requires installation of the Salesforce extension	

This module discusses operators that are a little more complex, such as Aggregate, Join, and Unique. In order to use these operators in their more advanced form, you must provide the processing logic. In many situations, you will be able to develop this logic using Expression Rules. In other situations, your logic will require that you use Function Rules, which allow you to develop logic that employs looping or coding dependent on complex data constructs such as tables.

Additionally, each of these operators has multiple optional helper functions that you may use to develop more powerful processing logic that will frequently allow you to simplify the design of your dataflow. When you understand how QlikView Expressor coordinates invocation of these helper functions with the invocation of the operator's mandatory functions, you will be able to write more efficient applications that will require less maintenance and demonstrate better performance.

16.1 The Aggregate Operator

This operator has two mandatory functions that must be implemented. When you use an Aggregator Rule, code is automatically included within these functions and you may not need to take any further actions. When you use Function Rules to add you own logic, you take responsibility for ensuring that these functions are properly implemented. In addition, this operator has five optional helper functions that you may use from within a function rule.

Let's review how the data processing engine interacts with the Aggregate operator. When you configure this operator, there are three properties that affect how the operator performs the data collations.

- Aggregate Keys – This property specifies the attribute(s) that will be used to group the records. For example, selecting an attribute named order_number will create a separate group for each distinct order number.
- Method – This property specifies whether the operator will use random access memory or disk when grouping the records.
    - If sorted is selected, the operator assumes that all records are ordered ascending by the value in the aggregate key(s) and the records within one group will be retained in memory until processing of the group completes.
    - When in memory is specified, the operator must store all records in memory before it can organize them into groups and perform the aggregations.
    - Specifying on disk tells the operator that the number of records will be too large to hold in memory before grouping and the operator then uses disk as the storage medium.
- Use Change Function – Setting this property means that you will manage the record grouping through the logic that you write in the change rule. Within the change function, you implement the logic that determines when a group should be closed and its summary information emitted.

Regardless of how you set these properties, the data processing engine invokes the operator's mandatory and optional functions in a specified order.

- If change function usage has not been specified, the operator uses the aggregate key choice and the method setting to divide the incoming records into groups.
- As processing of each group begins, the engine invokes the prepare function. Use this function to set up the operator to process a new group, for example, initialize variables to starting values.
- As each record is processed, the engine invokes the aggregate function. Use this function to update your running calculations.
- When processing of the group completes, the engine invokes the result function. Use this function to initialize the attributes in the outgoing record.
- Immediately before the operator emits the record, the engine invokes the sieve function. Use this function to examine the contents of the outgoing record and suppress its emission if desired.
- If the change function has been specified, the engine invokes this function immediately after the aggregate function completes processing. This function's return value determines whether to close the group's processing. If the group is to be closed, the engine invokes the result and sieve functions to close out the group then the prepare function to begin processing of the next group.

Also, within the Aggregate operator, a global variable named `work` is initialized as an empty Datascript table when the operator begins processing each group of records. Code uses this variable to store intermediate values that will be used later to initialize the output record.

The following table summarizes each function's application programming interface.

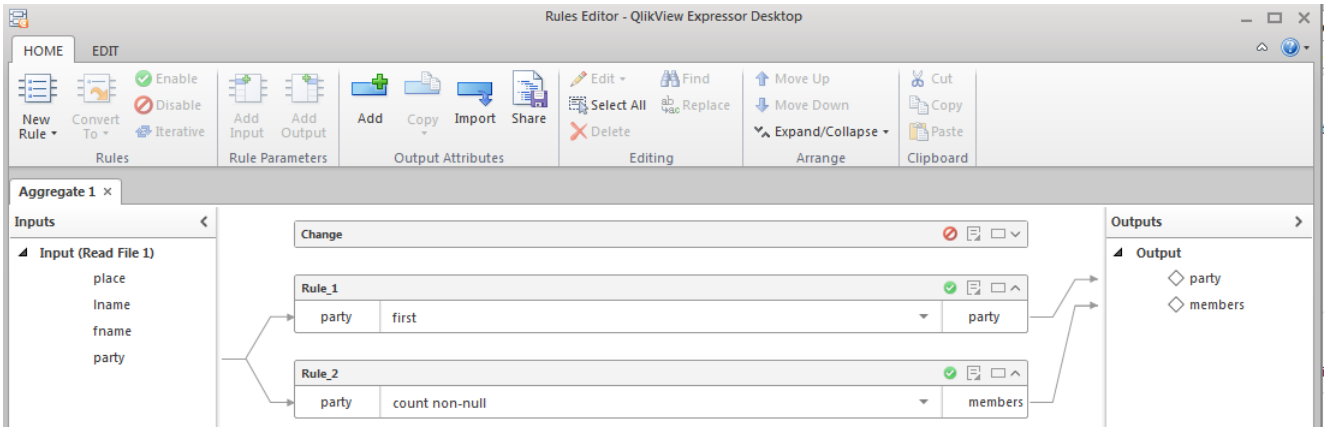
Function Type	Function Name	Description
Mandatory Functions	aggregate	The data processing engine invokes this function as each record is processed. Include in the method body QlikView Expressor Datascript that performs the desired data manipulation. Arguments: • input – the record to be processed • index – which record in the group that is being processed
	result	The data processing engine invokes this function after the last record within the group has been processed. Include in the method body QlikView Expressor Datascript that finalizes the processing and returns the output record. Arguments: • input – the last record in the group • count – the total number of records in the group
Optional Helper Functions	change	Use this function when you want to explicitly control how records are divided into groups for processing. If this function returns true, it forces invocation of the result and prepare functions and then begins processing the next group. If this function returns false, processing of the current group continues. Arguments: • input – the record to be processed • previous – the immediately previous record

prepare	The data processing engine invokes this function before processing any records within a group. Include in the method body QlikView Expressor Datascript that initializes variables needed for the subsequent processing of the group. Argument: • input – the first record in the group
sieve	Use this function when you want to control whether a summary record for a group is emitted from the operator. Argument: • output – the summary record from a group If this function returns true or a non-nil value, the summary record is emitted by the operator. If this function returns false or a nil value, the summary record is not emitted by the operator.
initialize	The data processing engine invokes this function one time before the operator begins to process records. This function has no arguments or return values.
finalize	The data processing engine invokes this function one time after the operator has completed processing all records. This function has no arguments or return values.

Exercise: The Aggregate Operator

1. Create a new project named **Aggregate**.
2. Add a File Connection to the project.
    a. Set the connection's path to **C:\data**.
3. Add a Delimited Schema artifact for the file **C:\data\presidents.dat** to the project.
4. Create a dataflow named **Aggregate** containing the Read File, Aggregate, and Write File operators.
5. Configure the Read File operator to read the file **presidents.dat** with the connection and schema artifacts created in steps 2 and 3.
6. In the Aggregate operator:
    a. Choose the attribute **party** as the Aggregate key.
        i. To add an entry to the **Aggregate key** control, click  then select **party** from the drop down.
    b. Set the Method property to **In Memory**.
    c. Define the structure of the output record.
        i. Open the Rules Editor.
        ii. Click **Add** in the **Output Attributes** grouping.
            1. Define a **string** attribute named **party**.
            2. Define an **integer** attribute named **members**.
    d. Use two aggregator rules to initialize the output attribute party with the value from the input attribute of the same name from the first record processed, and initialize the output attribute members with the count of the number of presidents who were members of a specific party.

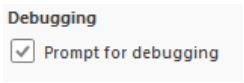
i. To add each rule, click **New Rule – Aggregate Rule** in the Rules grouping.



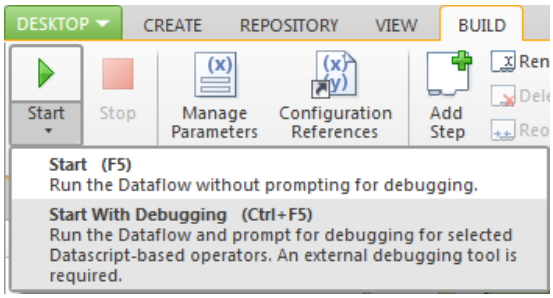
7. Configure the Write File operator.
   a. Create a schema by selecting **New Delimited Schema from Upstream Output…**.
   b. Write the output file to the same directory that contains the input file.
8. Save and run the dataflow.
9. Examine the output file.
   a. Why do you think the summary records are ordered as they are? _____

Exercise: Using the Decoda Debugger

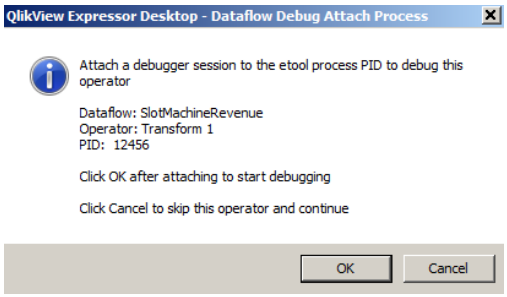
1. Select the **Start > All Programs > Decoda > Decoda** menu item to start the Decoda IDE.
2. In Expressor Desktop, open, if necessary, the dataflow you want to examine.
3. Select the aggregate operator and in its **Properties** panel check the **Prompt for debugging** property.



   a. Save the dataflow.
4. In the **Dataflow > Build** ribbon bar, carefully click on the ▾ within the **Start** button in the Run grouping.
   a. This opens a drop down menu.

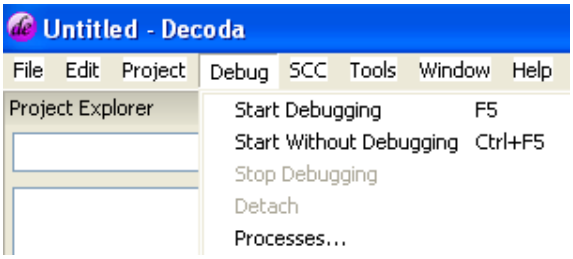


5. Select the **Start with Debugging (Ctrl+F5)** menu item.
    a. The dataflow begins processing.
    b. The Dataflow Debug Attach Process window opens.



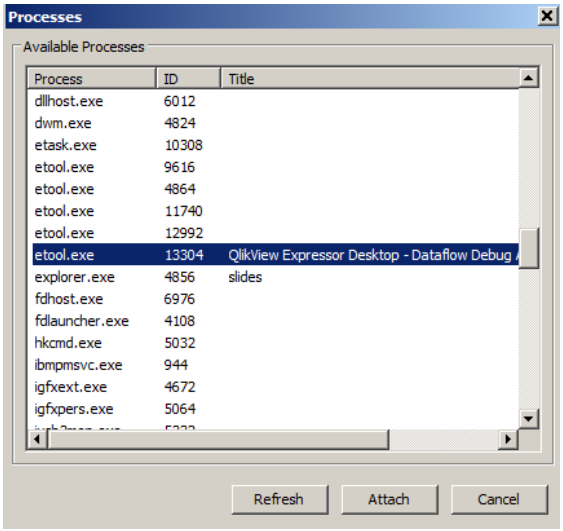
        i. **DO NOT click OK**. You must first attach the process to Decoda.
        ii. In the Dataflow Debug Attach Process window, **note the PID**.

6. Return to the Decoda IDE and select the **Debug > Processes…** menu item.

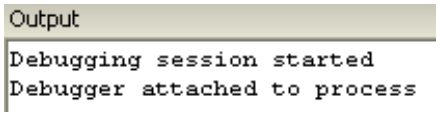


    a. This opens the Processes window.

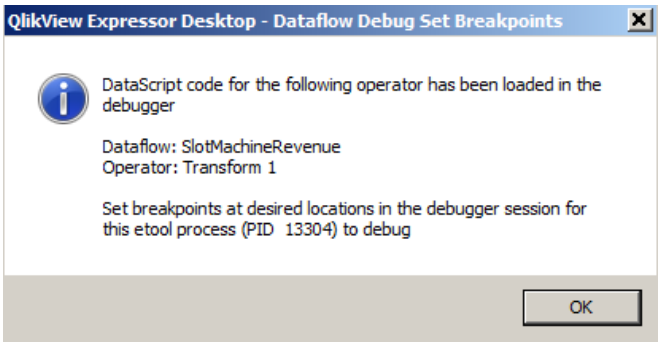
7. In the Processes window, click on the ID header label to sort the entries by PID number then select the entry corresponding to the PID displayed in the Dataflow Debug Attach Process window and click **Attach**.



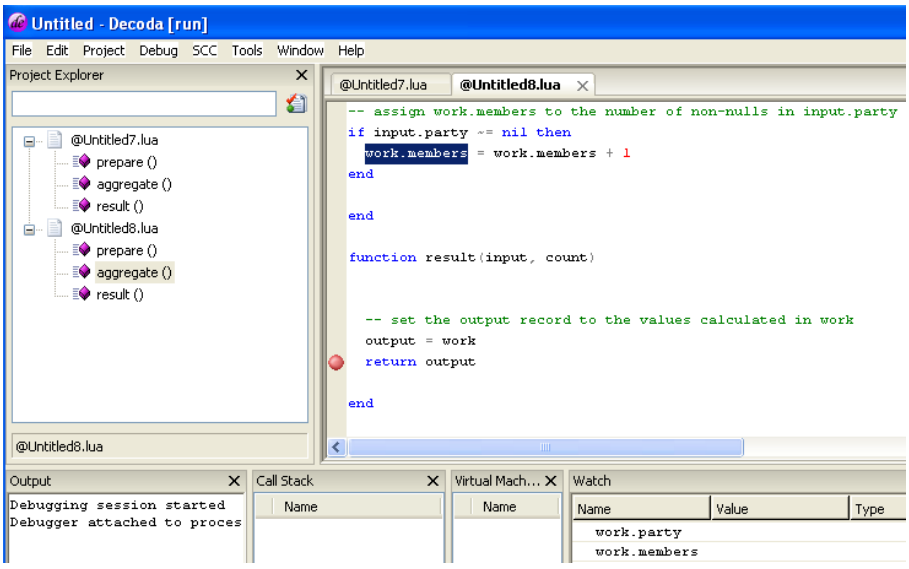
    a. In a moment or two, Decoda attaches to the process and displays confirmation (in lower left-hand corner).



8. Now return to the Dataflow Debug Attach Process window and click **OK**.
    a. The Dataflow Debug Set Breakpoints window opens.

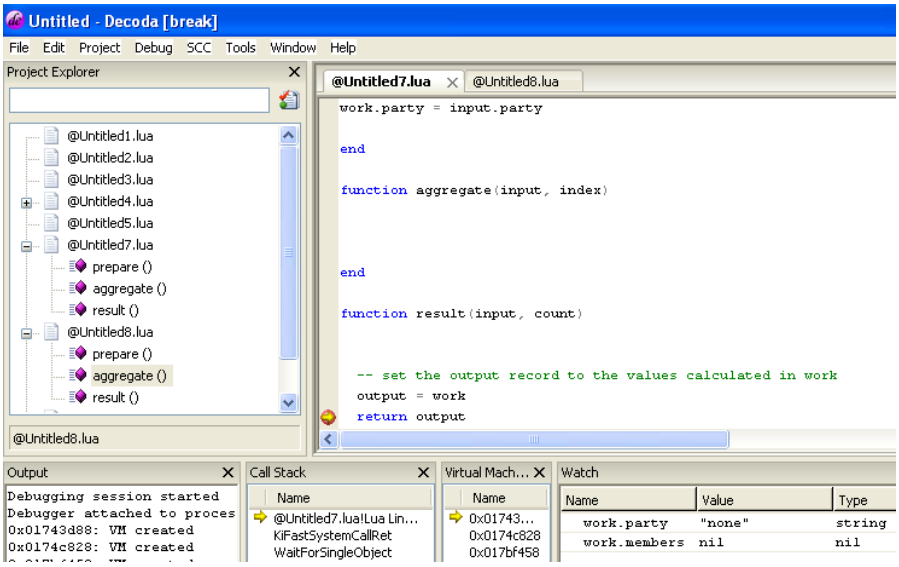


    b. **DO NOT click OK**. You must first set break points in the code and select watch variables.
9. Return to Decoda and in the Project Explorer panel select and open the entries corresponding to the code in the operator being examined.
    a. The explorer entry corresponding to the operator code is generally the last entry named Untitled.
        i. However, since this aggregate operator includes two rules, the operator code will be found within the last two entries.
    b. Double click on any function name to display the code for that rule.
10. Place the cursor at the beginning of the statement where you want to position a break point.
    a. In this example, place the cursor on the **`return output`** statement in the result function.
    b. Press **F9** to set a break point.
        i. Also use F9 to remove a break point.
    c. Set the same break point in both rules.
11. To select a watch variable, highlight the variable and drag-and-drop into the Watch panel (lower right-hand corner).
    a. Select the **`work.party`** and **`work.members`** variables.



12. Return to the Dataflow Debug Set Breakpoints window and click **OK**.
    a. The processing begins and stops at the first break point.

13. Return to Decoda and note the watch variable values.

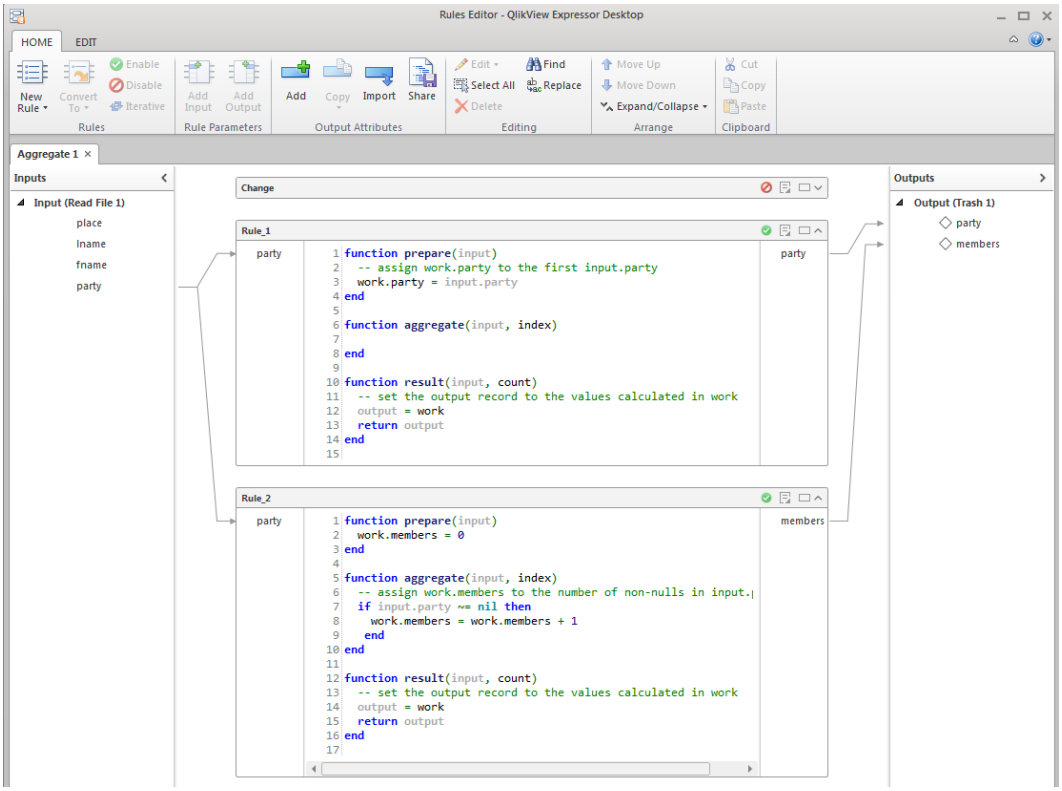


14. While keeping Decoda as the active window, click **F5** to begin processing to the next break point.
    a. Note the changing values in the watch variables as you continue to click **F5**.
    b. Explain what you see.

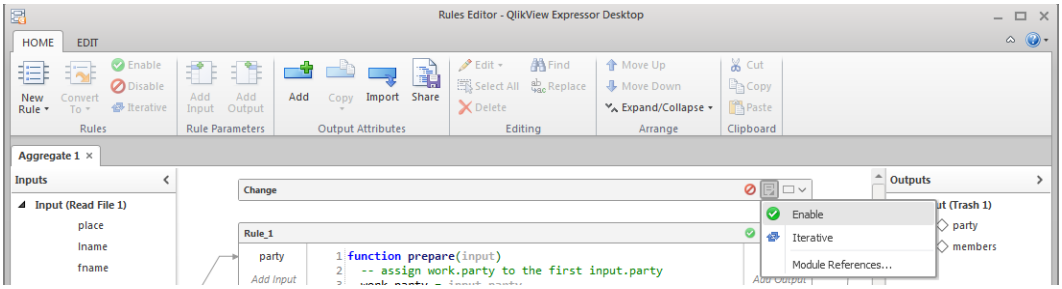
    _____
    _____
    _____
    _____

Exercise: Using the Change Function

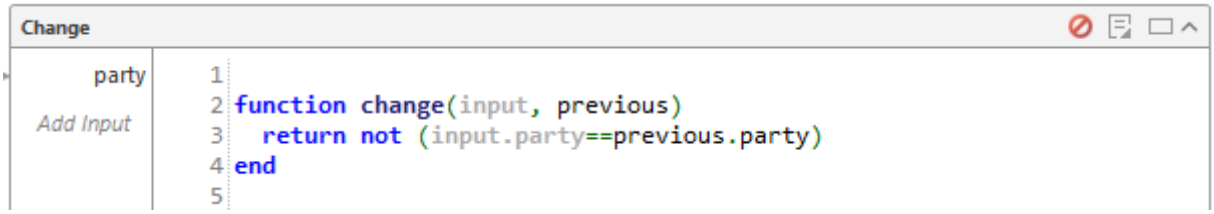
1. Open the Aggregate operator Rules Editor and convert both of the rules into function rules.
    a. Click on each rule box then click **Convert To – Function Rule** in the Rules grouping.
    b. For the party output attribute, the required information is captured in the prepare function whose argument is the first record in each party grouping.
    c. For the members output attribute, the aggregate function keeps a running count of the number of records in each party grouping.
    d. For both output attributes, the result function copies data from its temporary location in the table work to the output.



2. Return to the Aggregate operator properties panel and check the **Use Change Function** property.
3. Reopen the Rules Editor.
    a. Enable the change function.



    b. Click  to expand the rule so that you can enter coding.
    c. Drag the input attribute party into the input parameter listing.
    d. Enter the following code into the change function.



4. What is this function doing? How does this affect how the operator performs grouping?

   _____

5. Close the Rules Editor, save and run the data flow.
6. Review the output.
   a. Explain the grouping.

      _____
      _____
      _____
      _____

7. Once again, open the Rules Editor.
   a. Add an implementation of the sieve function to the first rule, the rule that initializes party.

```
16 function sieve(output)
17   return output.party~='Democratic'
18 end
```

   b. Why was the sieve function implementation added to this rule?
8. Close the Rules Editor.
9. Save and run the dataflow.
10. Review the output.
    a. Explain the grouping.

       _____
       _____
       _____
       _____

16.2 The Join Operator

It takes a little thought to fully appreciate the Join operator. Like a join in a relational database or QlikView script, this operator will execute an inner, left, right or outer join. In addition, the operator allows you to collect and process the records from either input that did not join with another record. You can put this to good use in distinguishing between updated, new and unchanged records when performing an incremental load (or Type 2 slowly changing dimension processing).

The two ports on the left-hand side of the operator are the inputs. The structures of the incoming records do not need to be identical and the fields used as the join keys do not need to have identical names, although they must be the same data type. You have complete control over which fields from the incoming records are contained in the emitted record.

The port on the right-hand side is the output, which emits records that satisfy the join type. And the two ports on the bottom edge emit those records that did not participate in a join. The port on the bottom left emits records that entered through the upper left-hand port while the port on the bottom right emits records that entered through the lower left-hand port.

This operator has one mandatory function that must be implemented. When you use an Expression Rule, code is automatically included within this function and you may not need to take any further actions. When you use a Function Rule to add your own logic, you take responsibility for ensuring that this function is properly implemented. In addition, this operator has three optional helper functions. An Expression Rule will not add any code to these functions. If you want to use one of these helper functions, you must provide the implementation through a Function Rule.

Let's review how the data processing engine interacts with the Join operator. When you configure this operator, there are five properties that affect how the operator performs the join operation.

- Join Keys – This property specifies the attribute(s) that will be used to select the records for joining. For example, selecting an attribute named order_number will join records that share this value.
- Primary Input – Specifies the driving input. The operator takes each record from this input and attempts to perform a join against the records in the secondary input.
- Join Type – This property specifies the type of join to perform.
    - Selecting inner means that only when both inputs include a record with the same key value(s) will a record be emitted from the primary output port on the operator's right side. This record may contain values derived from attributes in either or both incoming records. Records from each input that do not have a matching record on the other input will be emitted from the secondary output ports on the operator's bottom edge.
    - Selecting outer means that all matched and unmatched records will be emitted from the primary output port. This record may contain values derived from attributes in either or both incoming records and any values that would have been derived from the missing record will be set to nil. No records will be emitted from the secondary output ports.
    - Selecting 0 means that records arriving on input 2 for which there is no matching record on input 1 will still be emitted on the primary output port. Unmatched records arriving on input 1 for which there are no matching record on input 2 will be emitted on the corresponding secondary output port.
    - Selecting 1 means that records arriving on input 1 for which there is no matching record on input 2 will still be emitted on the primary output port. Unmatched records arriving on input 2 for which there are no matching record on input 1 will be emitted on the corresponding secondary output port.
- Method – This property specifies whether the operator will use random access memory or disk when grouping the records.
    - If sorted is selected, the operator assumes that all records are ordered ascending by the value in the aggregate key(s) and records will remain in memory only as long as it takes to perform any joins.
    - When in memory is specified, the operator must store all records in memory before it can begin performing joins.
    - Specifying on disk tells the operator that the number of records will be too large to hold in memory before joining and the operator then uses disk as the storage medium.

- De-duplicate Input 0, De-duplicate Input 1 – These properties allow selection of only the first or last record for a specific join key value on the specified port. These settings are used to select a single record from the corresponding input when more than one record has the same join key value. If a selection is made for both properties, at most one record is emitted for each join key value.

Regardless of how you set these properties, the data processing engine invokes the operator's mandatory and optional functions in a specified order.

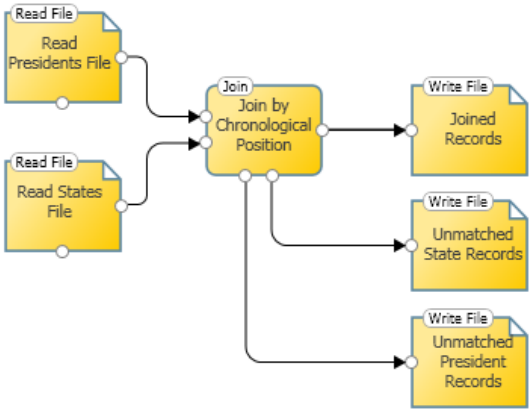
- The operator accepts a record from the primary input and runs through the records on the secondary input looking for valid joins as defined by the join type, method and de-duplicate input properties.
- For each valid join, the operator invokes the joiner function and then the optional sieve function.
- Unmatched records are either dropped by the operator or emitted from one of the secondary output ports positioned along the bottom of the operator shape.

The following table summarizes each function's application programming interface.

Function Type	Function Name	Description
Mandatory Functions	joiner	The data processing engine invokes this function as each pair of matched records is processed. Include in the method body QlikView Expressor Datascript that performs the desired data manipulation. The function returns the output record. Arguments: • input – attributes from the two incoming records
Optional Helper Functions	sieve	Use this function when you want to control whether an output record should be emitted from the operator. Argument: • output – the record resulting from a join If this function returns true or a non-nil value, the record is emitted by the operator. If this function returns false or a nil value, the record is not emitted by the operator.
	initialize	The data processing engine invokes this function one time before the operator begins to process records. This function has no arguments or return values.
	finalize	The data processing engine invokes this function one time after the operator has completed processing all records. This function has no arguments or return values.

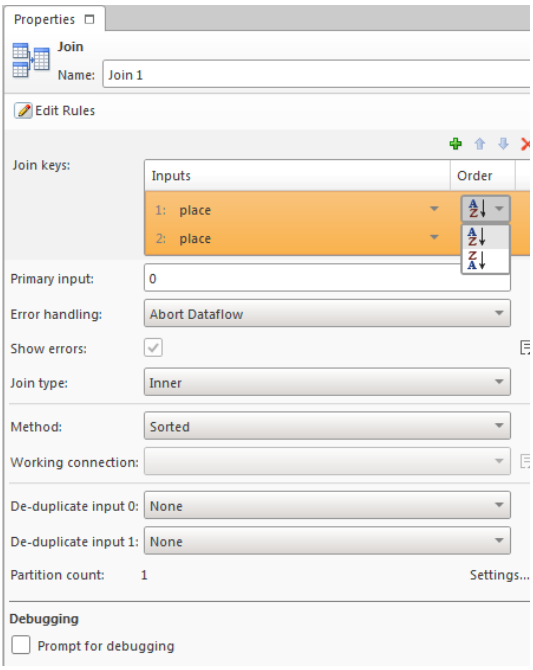
Exercise: The Join Operator

1. Create a new project named **Join**.
2. Add a File Connection to the project.
    a. Set the connection's path to C:\data.
3. Add a Delimited Schema for the file C:\data\presidents.dat to the project.
    a. This file includes a listing of the Presidents of the United States sorted chronologically.
    b. This file contains a header row.
    c. In the schema's default composite type, change the data type of the place attribute to integer.
        i. Edit the associated mapping specifying zero digits after the decimal point.
4. Add a Delimited Schema for the file C:\data\states.dat to the project.
    a. This file includes a listing of the birth states of the Presidents sorted chronologically by president.
        i. Entries for several of the presidents are missing.
    b. The file does not contain a header row.
        i. Give the fields meaningful names.
        ii. Think about what name to give to field1. It too is the chronological position of the president.
    c. In the schema's default composite type, change the data type of the attribute that holds the field1 value to integer.
        i. Edit the associated mapping specifying zero digits after the decimal point.
5. Create the dataflow pictured below.

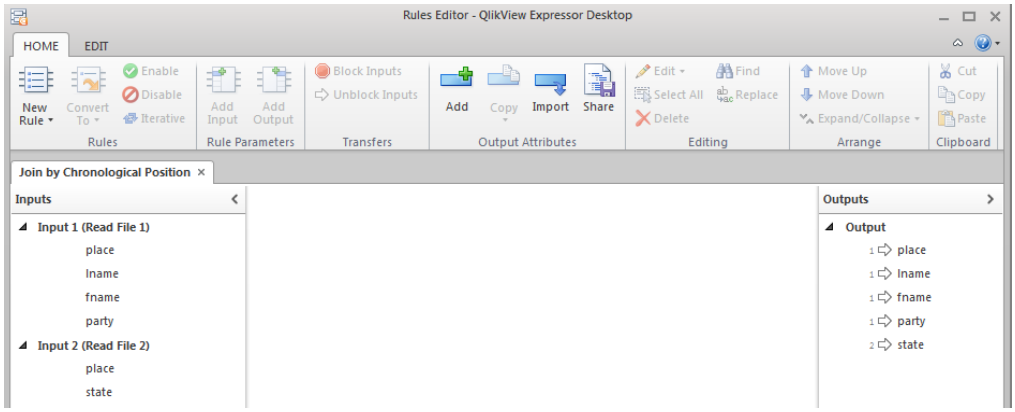


6. Configure the two Read File operators to read their respective data files.
    a. Use the same connection for both operators.
7. Configure the Write File operator "Unmatched State Records" using the same schema as the Read File operator "Read States File"; name the output file unmatched_states.txt.
    a. Use the same connection as the Read File operators.
8. Configure the Write File operator "Unmatched President Records" using the same schema as the Read File operator "Read Presidents File"; name the output file unmatched_presidents.txt.
    a. Use the same connection as the Read File operators.
9. Configure the Join operator to perform an inner join using the chronological position attribute as the join key. Since the records in both files are sorted chronologically by president, and the data type of the place

attribute is integer, you may use the sorted method, and since neither file includes duplicate entries, there is no need to specify a de-duplicate setting. Note that you must specify the join key attribute from both inputs, so they do not need to have the same name.



10. Now open the Join operator Rules Editor.
    a. Observe how all of the attributes from input 1 and the state attribute from input 2 are already represented in the output and that the source input for each attribute is identified.
        i. In the output panel, the right facing arrows indicate that values in these incoming attributes will be transparently moved into identically named attributes in the outgoing record.
            1. If you do not need to perform transformations on any of these values, there is no need to include Expression or Function Rules. The operator will perform the join and emit the combined record.
    b. Close the Rules Editor.



11. Configure the Write File operator “Joined Records.”

a. Use the same connection as the Read File and other Write File operators.
b. You will need to create a Delimited Schema from the upstream output that describes the record emitted by the Join operator.
c. Name the output file joined_records.txt.

12. Save and run the dataflow.
13. Examine the content of the output files.
    a. Each record in the file joined_records.txt contains the president's birth state.
    b. Several presidents are missing from this listing.
        i. Since you know that all the presidents were in the file presidents.dat, the missing entries must be due to the fact that there was no entry for these presidents in the file states.dat.
    c. View the file unmatched_presidents.txt.
        i. This file includes the four presidents missing from the file joined_records.txt.
    d. The file unmatched_states.txt is empty.
14. Reconfigure the Join operator, specifying 0 as the Join Type.
15. Save and run the dataflow.
16. Examine the content of the output files.
    a. Explain the results.

    _____

    _____
17. Reconfigure the Join operator, specifying 1 as the Join Type.
18. Save and run the dataflow.
19. Examine the content of the output files.
    a. Explain the results.

    _____

    _____
20. What happens if you try to reconfigure the Join operator to use Outer as the Join Type?
21. Look at the Messages tab in the Status panel for a hint.
    a. Explain the results.

    _____

    _____

16.3 The Transform Operator (Data Transformations)

When used to transform incoming attribute values, this operator has one mandatory function that must be implemented. When you use an Expression Rule, code is automatically included within this function and you may not need to take any further actions. When you use a Function Rule, you take responsibility for ensuring that this function is properly implemented. In addition, this operator has four optional helper functions. If you want to use one of these helper functions, you must provide the implementation through within a Function Rule.

When used with an Expression or Function Rule, the data processing engine invokes the operator's mandatory and optional functions in a specified order.

- As each record is processed, the engine first invokes the optional filter function.
- The engine next invokes the mandatory transform function.
- Finally, the engine invokes the optional sieve function.

Function Type	Function Name	Description
Mandatory Functions	transform	The data processing engine invokes this function as each record is processed. Include in the method body QlikView Expressor Datascript that performs the desired data manipulation. The function returns the output record. Arguments: • input – the record to be processed
Optional Helper Functions	filter	The data processing engine invokes this function before invoking the transform function. Include in the method body QlikView Expressor Datascript that determines whether the record should be processed by the transform function. Arguments: • input – the record to be processed If this function returns true or a non-nil value, the transform function processes this record. If this function returns false or a nil value, the transform function does not process this record.
	sieve	Use this function when you want to control whether a summary record for a group is emitted from the operator. Argument: • output – the summary record from a group If this function returns true or a non-nil value, the summary record is emitted by the operator. If this function returns false or a nil value, the summary record is not emitted by the operator.
	initialize	The data processing engine invokes this function one time before the operator begins to process records. This function has no arguments or return values.
	finalize	The data processing engine invokes this function one time after the operator has completed processing all records. This function has no arguments or return values.

16.3.1 The Initialize and Finalize Functions

The initialize and finalize functions are available in operators that support scripting: Aggregate, Join, and Transform (as well as the Read Custom and Write Custom operators). Using these functions is completely optional.

16.3.1.1 The initialize Function

This function is invoked one time as the operator initializes before any records are processed. You can use this function to set up an operator, for example, by initializing variables or loading data from an external data resource.

One possible use is to populate variables that will then remain accessible to the operator while it processes data records.

16.3.1.2 The finalize Function

This function is invoked one time as the operator after the operator has processed all records. You can use this function to update any external data resources with data derived from the data integration application.

One possible use is to transfer data from the operator to external storage so that it can be read in during subsequent executions of the dataflow.

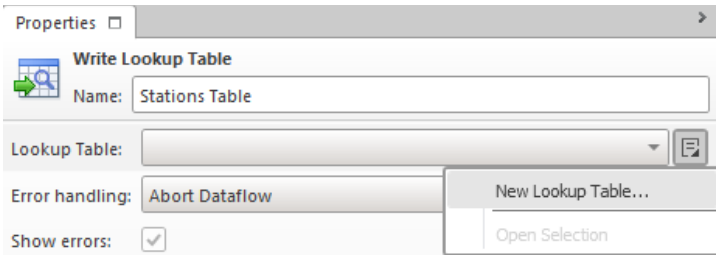
16.4 The Transform Operator (Data Lookups)

When this operator is used to implement Lookup Expression or Function Rules, the mandatory and helper transform related functions described in the preceding section are not involved. A Lookup Expression Rule is described within a dedicated 'wizard' that leads the developer through the process of selecting a lookup table, choosing appropriate lookup keys, and selecting the desired output attributes. Optionally, Expression Rule-like coding can be used to add rows to the lookup table.

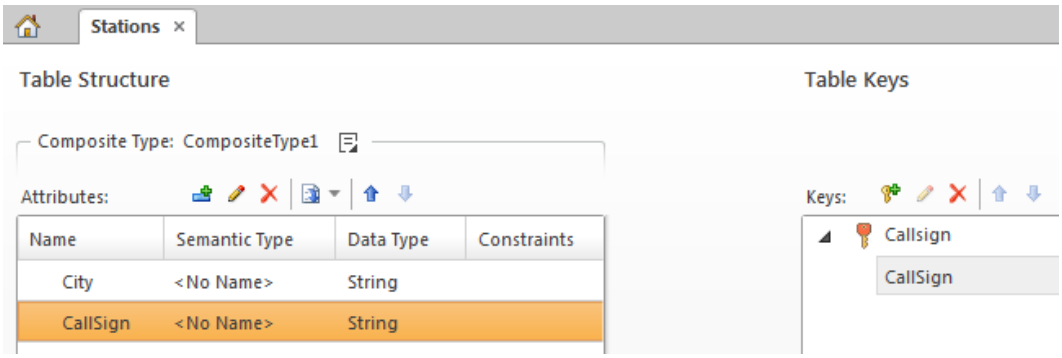
Lookup Function Rules are presented in a later section of this document.

Exercise – Lookup Expression Rule

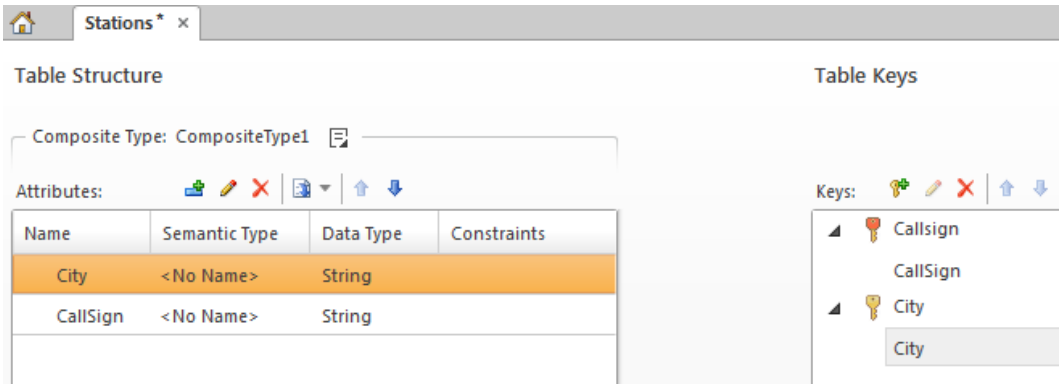
1. Create a new project named Lookup.
2. Create a file connection artifact to the C:\data directory.
3. Using the file stations.txt (a pipe | delimited text file), create a schema named Stations.
4. Open a new dataflow named ExpressionLookup and in Step 1, load the contents of the stations.txt file into a lookup table.
    a. Place the Read File and Write Lookup Table operators onto the step and connect.
    b. Configure the Read File operator input the content of the file stations.txt.
    c. Select the Write Lookup Table operator and in its properties sheet click the actions button  and select **New Lookup Table…** from the popup menu.



5. In the New Lookup Table window, name the lookup table **Stations** and click **Create**.
6. In the lookup table wizard, click the actions button and select **Assigned > Shared…** from the popup menu to open the Select Shared Composite Type window.
    a. Confirm that the Project dropdown displays Lookup.
    b. In the Scope dropdown, select Used by a Schema, which will list all composite types in the schemas contained in the Lookup project.
        i. There is only one composite type, the one in the delimited schema prepared in step 3.
    c. Select this composite type and click **OK**.
        i. This will copy the attributes from the schema into the lookup table's structure.
    d. Highlight the CallSign attribute and click **Add to New Key…** to create a unique key based on each station's call sign



    e. Create a second, non-unique key, from the City attribute.

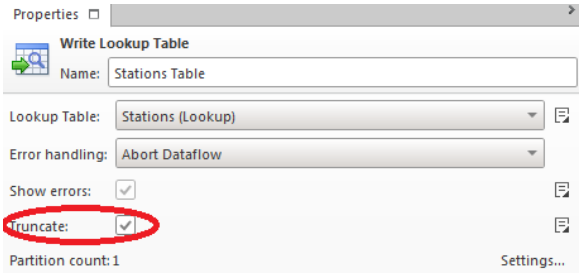


    f. At the bottom of the lookup table wizard, click the actions button adjacent to the File Connection link and select Assign in the popup menu.
        i. Select the existing connection artifact, which becomes the file system location in which the file representing the lookup table will be stored, then click **OK**.

ii. Alternatively, click **New File Connection…** and create a new file connection artifact for a different location.

g. Save the lookup table and close the wizard.

7. On step 1 of the dataflow, select the Write Lookup Table operator and in its properties sheet select the Truncate property.



a. This ensures that each time you run the dataflow, the lookup table is repopulated.

8. In the C:\data directory, create a text file containing a listing of a few selected call signs, one call sign per row.

a. Save this file with the name callsigns.txt.

b. This file includes call signs for which you want to look up the associated city.

9. In the **Steps** section of the **Dataflow > Build** ribbon bar, click **Add Step** to add a second step to the dataflow.

a. Place and connect Read File, Transform, and Write File operators onto this step.

b. Configure the Read File operator to extract data from the callsigns.txt file.

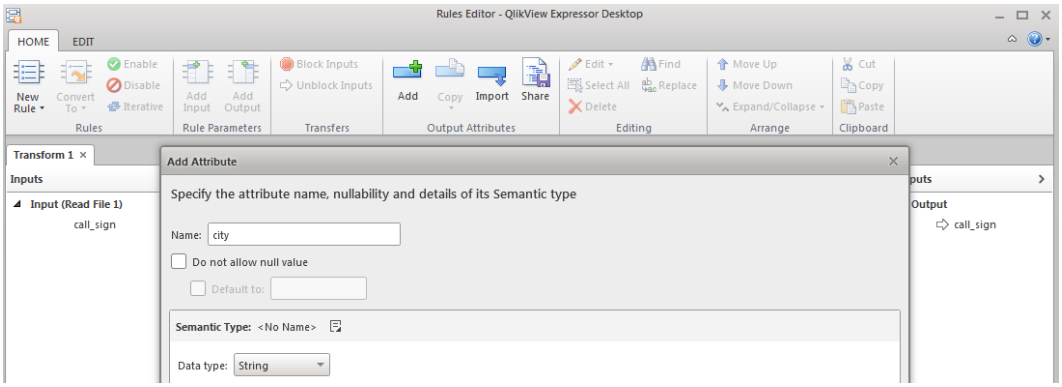
i. You will need to create a schema for this file.

ii. Give the attribute representing the call sign a meaningful name.

c. In the Transform operator, open the Rules Editor

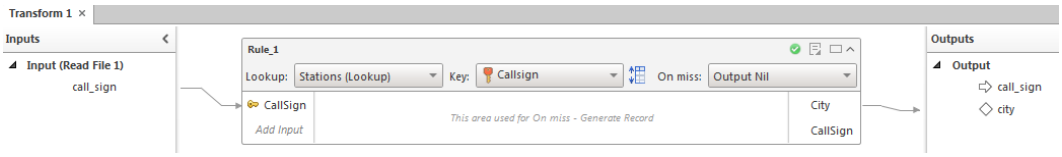
i. Add a string attribute to the output.

1. In the Output Attributes grouping of the HOME ribbon bar, click Add and define the new output attribute.



ii. Add a Lookup Expression Rule.

1. Configure this rule to use the Stations lookup table and the unique CallSign key.
2. Connect the input attribute to the rule's input parameter.
3. Connect the rule's City output parameter to the new output attribute.



4. Note that the value of the input attribute is automatically transferred to the output record.
5. Close the Rules Editor.

d. Configure the Write File operator to write an output file
   i. You will need to create a schema from the output of the upstream Transform operator.

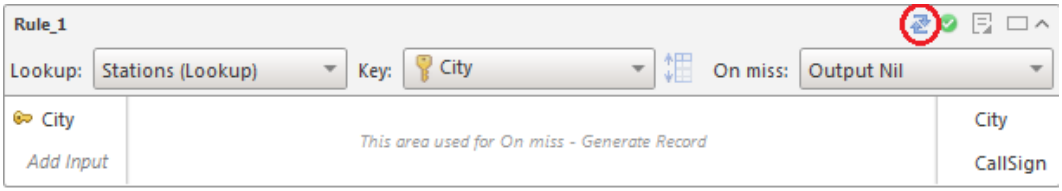
10. Run the application and note the listing of call signs and associated cities.
    a. If the input file contains the call signs WCAU, SAGA, SHYY, SPVI, KYW, KXTX the output file will contain

```
Philadelphia,WCAU
Atlanta,WAGA
Philadelphia,WHYY
Philadelphia,WPVI
Philadelphia,KYW
Dallas,KXTX
```

But what happens if the input file includes a call sign, perhaps ZZZZ, for which there isn't a table entry? The lookup rule gives you control over the outcome.

1. Open the Transform operator's Rules Editor and focus on the On miss dropdown control in the upper right-hand corner of the Lookup Expression Rule.
   a. The rule's output parameter city could be initialized with nil (the default action).
   b. An error could be raised and passed to the Transform operator for a response.
      i. The Transform operator could respond by aborting the dataflow, skipping this one record, rejecting this one record, skipping this record and all following records, or rejecting this one record and all following records.
   c. The rule could generate a record in which the rule output parameter city is initialized with whatever value you choose.
      i. Optionally, this new record could also be written to the lookup table.

What happens with the non-unique key is used? In this case, more than one record could be returned from the lookup table, for example, New York has seven stations. The Lookup Expression Rule can handle this situation as well.



The circled symbol in the upper right-hand corner of the rule indicates that multiple matches may be found in the lookup table. In this situation, the Transform operator will emit multiple records.

1. Rework the step 2 to read a list of cities and emit all stations associated with each city.

If you ever need to read the entire contents of a lookup table, use the Read Lookup Table operator. The only property you need to specify is the name of the lookup table you want to read.

16.5 The Unique Operator

Although the Unique operator's underlying functionality derives from the ability to group records based on the value of a key field, similar to the Aggregate operator, it does not require any coding and its operation can be completely controlled through its properties.

The Unique operator selects data based on the presence, or absence, of records with the same keys. When you configure this operator, there are three properties that affect how the operator groups the incoming records.

- Aggregate keys – This property specifies the key field(s) used to group the incoming records.
- Mode – This property specifies the selection criteria to apply to each grouping of records.
    - First record directs that the first record within a group with matching keys should be emitted from the operator. With this setting, a single record from each group is selected and emitted.
    - Last record directs that the last record within a group with matching keys should be emitted from the operator. With this setting, a single record from each group is selected and emitted.
    - Unique records directs that only those groups that contain a single member should emit a record from the operator.
    - Duplicate records directs that only those groups that have multiple members should emit records from the operator. With this setting, each group emits a collection of records all of which have the same aggregate key value.
- Method – This property specifies whether the operator will use random access memory or disk when grouping the records.
    - When in memory is specified, the operator must store all records in memory before it can organize them into groups.
    - Specifying on disk tells the operator that the number of records will be too large to hold in memory before grouping and the operator then uses disk as the storage medium.

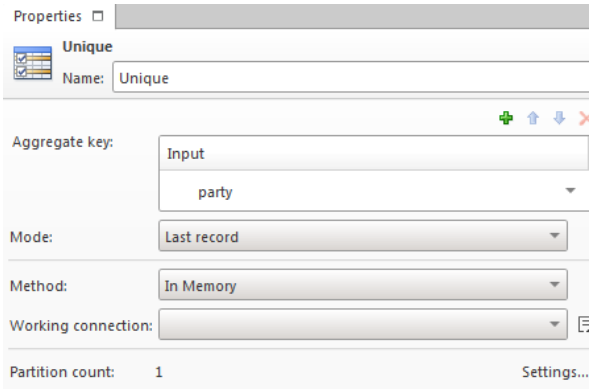
Exercise: The Unique Operator

1. Create a new project named **Unique**.
2. Add a File Connection to the project.
    a. Set the connection's path to C:\data.
3. Add a Delimited Schema for the file C:\data\presidents.dat to the project.
    a. This file includes a listing of the Presidents of the United States sorted chronologically.

b. This file contains a header row.

4. Create the dataflow pictured below. Name the dataflow Unique.

5. Configure the Read File operator to read the file presidents.dat.
6. Configure the Write File operator.
   a. Use the same connection and schema as the Read File operator.
   b. Give the output file a unique name.
7. Configure the Unique operator, using **party** as the key.
   a. Initially configure this operator to select the last member in each party, using the **In Memory** method.

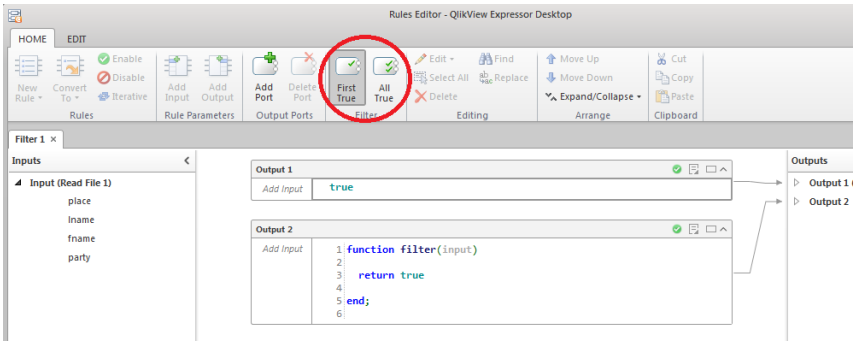


8. Save and run the dataflow.
   a. Review the content of the output file.
9. Change the **Mode** to **First** record.
   a. Save and run the dataflow.
   b. Review the content of the output file.
10. Change the **Mode** to **Unique records**.
    a. Save and run the dataflow.
    b. Review the content of the output file.
11. Finally, change the **Mode** to **Duplicate records**.
    a. Save and run the dataflow.
    b. Review the content of the output file.

Why do you think the records are ordered as they are? _____

16.6 The Filter Operator

This QlikView Expressor operator is extremely powerful in that it allows you to specify one or more conditional statements each associated with its own output port. If the conditional statement returns true, the record will be emitted from the corresponding output port. If none of the conditional statements return true, the record will be emitted from a separate 'false' port. Moreover, you can configure the operator to emit the record for all conditional statements that return true or for only the first true statement.



Consequently, the Filter operator can be used to select the records containing new data (for example, has a created or modified date after some specified date) as well as all the records. Therefore, this operator is perfect for isolating records with updates.

All of the QlikView Expressor operators that support Datascript coding have a number of helper functions that you may use to implement your processing logic. The operator has one required function and two optional helper functions.

Function Type	Function Name	Description
Mandatory Function	filter	The data processing engine invokes this function as each record is processed. Include in the method body QlikView Expressor Datascript code that uses field values from the record to return true or false. If the code returns true, the record is emitted from the corresponding port. Additional output ports may be added so that more than one rule can be used to test the incoming record. The lowest output on the right-hand side always emits records that fail all of the conditional tests. Arguments: • input – the record to be processed
Optional Helper Functions	initialize	The data processing engine invokes this function one time before the operator begins to process records. This function has no arguments or return values.
	finalize	The data processing engine invokes this function one time after the operator has completed processing all records. This function has no arguments or return values.

17 QlikView Expressor Datascript

In many situations, the graphical editors that are part of QlikView Expressor Desktop will provide sufficient support for you to develop your coding. But in other situations, you may need to write more extensive processing logic that needs to be developed and tested outside the scope of a running application. While you could write this code using Desktop's Rule Editor, you may find it helpful to work outside of this editor for this type of work. Additionally, external scripts – called Datascript Modules – that you may want to develop are written in a dedicated editor and cannot be debugged from within the editors used to configure the programmable operators. QlikView Expressor provides a command line utility – **datascript** – that you can use to develop and test your code.

To use this utility, you must open a command window from the **Start – All Programs – QlikView – expressor3 – expressor command prompt** menu item. Opening a command window using other approaches will not properly set the environment to use **datascript**. Use **Ctrl-C** to close the command window.

The **datascript** utility may be run interactively, in which case each line of a script being developed or tested is entered individually into the command window, or a script may be contained in a file and executed as a single entity.

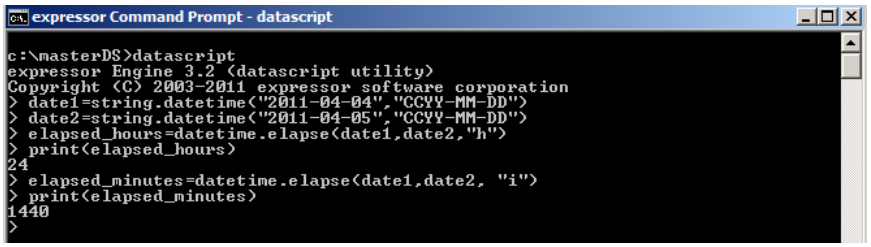
To start the utility in interactive mode, open a command window and enter **datascript** followed by **Enter**. Then type each line of code to be tested. Pressing **Enter** before completing a statement is permitted. The interpreter will wait until the statement is complete and followed by **Enter** before processing.

Alternatively, you can place multiple lines of code into a text file and execute within a command window by entering **datascript path/file_name** followed by **Enter**.

Running the **datascript** utility in a command window is the best way to master each of the QlikView Expressor Datascript functions and learn to use the various control structures.

Be certain to open a command window using the **Start – All Programs – QlikView –expressor3 – expressor command prompt** menu item, then change directories into a directory you will use to hold any script files you may be developing. Then issue the **datascript** command to enter interactive mode or use this utility to execute the code in your script file.

To experiment with a function, take it step-by-step until you feel confident enough to build up a single statement that accomplishes the entire objective. The **datascript** utility also includes a `print` function that you can use to display the value in any variable or the return from a function call, as shown in the following figure. The `print` function accepts a comma separated list of values to display.



Even when your coding skills have improved and you feel confident enough to code within an operator's Rule Editor it's useful to have a **datascript** window available to test your syntax and logic.

17.1 QlikView Expressor Tables

QlikView Expressor Datascript tables are extremely powerful and useful. Fortunately, basic uses of the table are easy to master and the more complex usages are not that much more involved.

In QlikView Expressor Datascript, tables are the only data structure. Tables have no fixed size, can be indexed using numbers, strings, or a combination of both numbers and strings, and can change size as needed. Table entries evaluate to nil when they are not initialized.

Let's consider the basic numeric indexed table. In this usage, index values start, by default, at one, although you can specify zero or negative numbers as the index value. If you want to accept the default indexing scheme, you can declare and initialize the table in a single statement.

```
myTable = {val1, val2, val3, ... }
```

If `val` is a numeric, you simply enter it. If a table value is a string, it must be enclosed in quotation marks (“”). To retrieve a value, you specify the index of the value you want.

`myTable[1]` returns `val1`, while `myTable[2]` and `myTable[3]` return `val2` and `val3`

If you want to set the index value, you must explicitly provide it.

```
myTable = {[-1]=val1, [-2]=val2, val3, ... }
```

In this case, `val1` is associated with the index -1, `val2` with the index -2, and `val3` with the index 1. You are free to intermix specified and default indices.

```
myTable = {[-1]=1, [-2]=2, 3, [-3]=4, 5}
```

Although you can use negative numbers as indices, this is not a common practice as it makes retrieving values from the table more difficult.

Table indices are not limited to numbers, strings are acceptable too. But there are a few rules you need to observe.

- If the index starts with an alphabetic character and includes only alphanumeric characters, it may be entered without surrounding quotation marks (`first` and `s2` in the following example).

```
myTable = {first=val1, s2=val2}
```

  - To retrieve the value you can use one of two notations:

```
myTable["first"] or myTable.first
```

- If the index starts with a numeric or non-alphabetic character, or includes a non-alphanumeric character, it must be entered with surrounding quotation marks within a set of square braces.

```
myTable = {["1"]=val1, ["2!"]=val2}
```

  - To retrieve the value you must quote the index value:

```
myTable["1"] or myTable["2!"]
```

Of course, you can always declare the table first and then add the elements.

```
myTable = { }
myTable[1] = val1
myTable["2!"] = val2
```

Note that with this approach, you must always provide the index. Note also, that you are free to mix numeric and string indices in the same table.

A table may hold complex types, for example other tables.

```
Table1 = {1,2}
Table2 = {first=Table1, second = {3,4}}
```

In `Table2`, the index `first` references `Table1` while the index `second` references an unnamed table. To retrieve one of the numeric values, you must specify two indices.

`Table2.first[1]`, which returns the value 1
`Table2.second[2]`, which returns the value 4

Or consider

```
Table2 = {Table1}
```

where 1 is the index in Table2 that corresponds to Table1. To retrieve values, use a two-dimensional index:

`Table2[1][2]`, which returns the value 2

Another way to quickly initialize a table is to assign the return values from a function that generates multiple return values. For example, the `string.match` function can be used to parse an IP address into its constituent parts and each part used to initialize an element of the table.

```
Table3 = {string.match("127.0.0.1","(%d+)%.(%d+)%.(%d+)%.(%d+)")
```

The pattern `(%d+)%.` looks for a sequence of one or more digits followed by a dot and the parentheses capture the digits as a return value. `Table3[1]` now contains 127, `Table3[2]` contains 0, `Table3[3]` contains 0, and `Table3[4]` contains 1, and you can easily retrieve these values from the table.

And perhaps one of the most interesting uses of a table is to hold functions.

```
myTable = { }
function myTable.larger(a,b)
  return (a>b) and a or b
end
```

`myTable.larger(4,5)` returns 5
`myTable.larger(4,3)` returns 4

Here, the table `myTable` uses the index `larger` to locate the code that returns the greater of two numbers. This is a nice technique to use to build up your own named library of functions. In fact, this is exactly how the functions in the String, Datetime, or other libraries have been defined.

So, how do you retrieve values from a table? As you saw above, you can always provide the index of the element you want to retrieve. But what if you don't know the index for your desired entry, Is there a way to work through a table, retrieving values? The answer is Yes!

QlikView Expressor Datascript includes two iterator functions that are designed to walk through a table returning each index and its associated value.

With a numeric indexed table that uses the default indexing scheme, this iterator function is named `ipairs`.

```
for index, value in ipairs(myTable) do
  -- manipulate index and value in some way
end
```

This code block will retrieve each index and value from the table, starting at index 1, and continue until all sequentially indexed elements have been retrieved. Values associated with zero or a negative numeric index or with a string index will not be retrieved. The iterator will stop retrieving values as soon as an out of order numeric or string index is retrieved.

With a table that uses string and/or numeric indices, even negative numeric indices, the iterator function `pairs` will walk through the table.

```
for index, value in pairs(myTable) do
  -- manipulate index and value in some way
end
```

You will find many uses for tables when working with QlikView Expressor Datascript. In fact, any time you need to create a collection or want to implement an efficient way to carry out a series of comparisons, such as nested `if..then..else` statements, or perform a lookup, think table.

In order to optimize memory usage, tables have their own memory management rules. Normally when you assign a value to a variable, the variable holds a private copy of the value.

```
var1 = "hello"
var2 = var1       --both variables hold the same value
var2 = "goodbye"  --var1 holds "hello"; var2 holds "goodbye"
```

This is not true with tables.

```
var1 = {"first","second"}
var2 = var1
var2[1] = "third" --var1[1] and var2[1] hold "third"
```

That is, the assignment statement `var2 = var1` does not copy the values within `var1` into `var2`; rather the two variables point to the same location in memory and any changes to a value in `var1` will be reflected in `var2` and *vice versa*. If you want to copy a value, you must explicitly reference the element.

```
var1 = {"first","second"}
var2 = {var1[1], var1[2]}
var2[1] = "third"   --var2[1] holds "third"; var1[1] holds "first"
```

Exercise: QlikView Expressor Function Exercises

Each of the following exercises will require use of functions detailed in the API appendix to this manual. Use the `datascript` utility to develop your solutions. Sometimes you may use this utility in interactive mode and for other exercises, placing your code in a file so it executes as a continuous block of code will be the way to go.

In many cases there will be more than one correct solution. Feel free to look through the product documentation as well as the information in this document in planning your solutions.

1. Datetime (and string.datetime) functions
    a. Using your birthday and today's date, determine how many days old you are.
    b. How many minutes and seconds are there in an eight hour workday?

c. Determine the day of the week (Sunday – Saturday, not 0 to 6) for your birthday in the year you were born and next year.
d. What day of the week (Sunday – Saturday) will it be 30 days from today.

2. Is functions
    a. Set some variables to the following values.
        i. nil
        ii. “”
        iii. “ “
        iv. 25
        v. true
        vi. “expressor software”
    b. Test each value with the various is functions and summarize your findings.

	is.blank	is.decimal	is.datetime	is.empty	is.future	is.integer	is.null	is.number	is.past
nil									
“”									
“ “									
25									
true									
“expressor software”									

    c. Use is.pattern to extract “expressor” from the value “expressor software”.

3. String functions
    a. Using the find and/or match functions, extract the top-level domain from an email address. Be sure to consider the fact that there may be periods before the @ character in an email address
        i. Your solution should be able to handle top-level domains of 2 to 4 characters.
        ii. How would you extract a multi-part domain such as `.co.in`.
            1. Will this pattern also work for example 3.a.i?
    b. Suppose you have a variable set to a string value that includes characters that you want to remove. For example:

```
123 Main Street 'Apt 25'
```

        i. Investigate various ways of removing the single quote characters from the string.
            1. There are three different functions you could use.
            2. Investigate the differences between these functions.
    c. Write a routine that determines how many times the characters “ss” appear in the word Mississippi.

4. OS functions
    a. Retrieve and display the values of the following environment variables set on your computer:
        i. USERNAME
        ii. USERPROFILE
        iii. PATH

5. IO functions
    a. In some convenient file system location, create a simple text file in which each line holds a single word or person's name. Add 5 or so lines to the file.
    b. Read and display the contents of the file.
    c. Read the file's contents and write to another file.

Exercise: QlikView Expressor Table Exercises

1. Initialize a numerically indexed table with the three letter airport abbreviations; five or six will be sufficient.
    a. Using the `ipairs` iterator function, print out the contents of the table, one airport abbreviation on each line.
        i. Print out each index value and its associated airport abbreviation, one pairing per line.
            1. The print function may take multiple comma separated arguments.
        ii. Now, print a single line with all the airport abbreviations where the individual entries are separated by a comma, pipe character, or tab.
2. You have an application in which three letter abbreviations are used for airport identifiers (assume that there are no spelling errors in the incoming data, but the case of the characters is not necessarily consistent).
    a. You need to convert each airport abbreviation into the full name; for example, BOS needs to be converted to Boston.
        i. Develop a routine that will look up each airport's full name when supplied with the airport abbreviation.
            1. Again, five or six airport name, abbreviation pairings would be sufficient.
        ii. Investigate what happens when a meaningless abbreviation is submitted to your routine.
            1. How can you exploit this situation?
    b. Revise your routine so that the input could be either the airport abbreviation or name but the return is always the airport name.

18 Datascript Modules

So far, all of the coding within a dataflow you have done has been written within the Rules Editor of a scriptable operator. Yet one of the competitive advantages of QlikView Expressor is extensive support for writing reusable code that you can incorporate into multiple projects or workspaces. These external script files are considered top-level artifacts just like the connection and schema artifacts and they will be managed in the same way. That is, QlikView Expressor will track the code included in these files and where that code is used and you can be guaranteed that the script files will be packaged with your applications during deployment.

Expressor Datascript, while it includes many versatile functions, does not completely overlap the functionality of the QlikView script expressions. But there is nothing stopping you from implementing equivalent functionality within Expressor, placing your code into a Datascript Module within a library that can be referenced by many projects.

Note: Expressor Datascript function names are case sensitive.

Exercise: Datascript Implementations of QlikView Expressions

In this exercise, you will create a Datascript Module that includes several functions that duplicate the processing of QlikView script functions. Although you are free to select the functions you would like to implement, consider the following as suitable examples. In developing your implementations, use the following descriptions as your objective, not the possibly more complex descriptions in the QlikView reference manual. The Appendix in this manual and the QlikView Expressor product documentation should be used as coding resources.

You might find it easier to develop and test your implementations in a command window running the **datascript** utility and then move them into the Datascript Module.

month (date)

This function returns the three character abbreviation for the month represented by date. The argument date is a datetime type. How would the code change if the argument date were a string type?

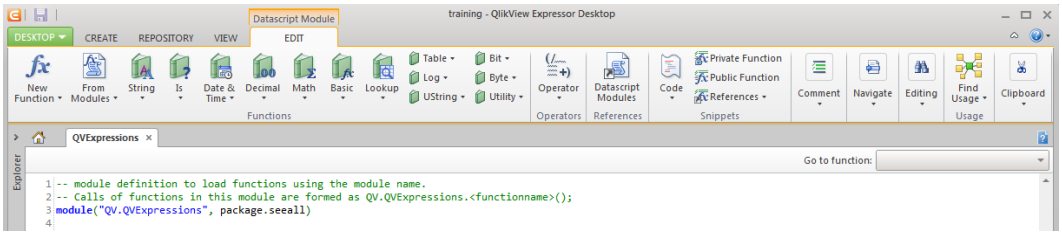
isyeartodate (date, basedate, shift)

This function returns true if date is an earlier date in the year containing basedate. The year can be offset by shift. Date and basedate are datetime types. Shift is an integer, where the value 0 indicates the year that contains basedate. Negative values in shift indicate preceding years and positive values indicate succeeding years.

mid (s, n1, n2)

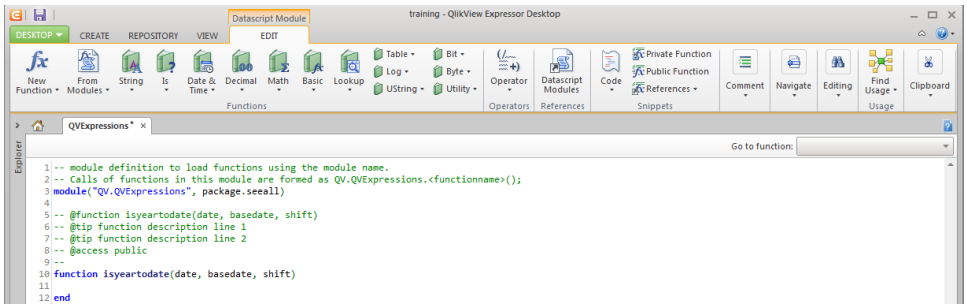
This function returns a substring of string s of length n2 characters starting at character n1. Note, the Expressor Datascript string.substring function returns the substring of string s beginning at character n1 and ending at character n2. The mid function would therefore be an alternative implementation of the substring function.

1. Create a new library named **QV**.
2. In the Desktop Explorer panel, expand the library and note the subdirectory name Datascript Moduels.
    a. This is where you will create the files that contain the code you want to write independent of where it will be used.
3. In this library, select the **Datascript Module** subdirectory, right-click, and select **New…** from the popup menu.
    a. Placing your Datascript Module into a library makes it easier to use it with multiple projects and/or workspaces.
4. In the New Datascript Module window, confirm that the module will be added to the library, enter a distinctive name (e.g., QVExpressions), then click **Create**.
    a. The Datascript Module opens in a text editor and the Datascript Module – Edit tab displays in the ribbon bar.
    b. The Datascript Module – Edit ribbon bar tab has many similarities to the ribbon bar that appears when developing rules in that you can quickly include one of the pre-existing QlikView Expressor Datascript functions by simply clicking on the appropriate function class button and selecting the desired function from the drop-down menu. You can also easily include a skeleton for a public or private function that you would like to write.
    c. To give yourself more room to code, collapse the Explorer panel by clicking the  icon in the panel's upper right corner.

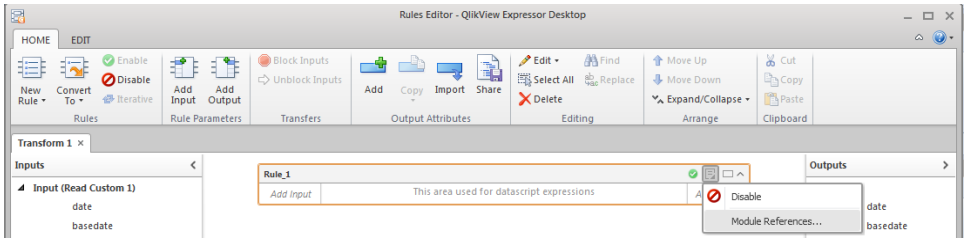


5. The first line in a Datascript Module should be the module declaration, which defines a namespace for your code.
   The namespace is the name of the project or library (without the version number) pre-pended to the name of the Datascript Module.
   The argument package.seeall is required; it makes all of the pre-existing QlikView Expressor Datascript functions accessible to the code you will write.
    a. The function class names that are part of QlikView Expressor Datascript (that is, string, datetime, etc.) are also namespaces.

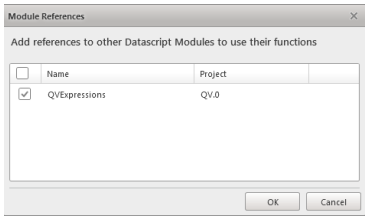
6. To write a public function, that is, a function that will be visible when you are working with the Rules Editor, click either **Public Function** in the **Snippets** category on the Datascript Module – Edit tab of the ribbon bar or click **New Function – Public Function** in the **Functions** category.
    a. Either approach inserts a comment/annotation into your code.
    b. Immediately below this comment/annotation, enter your function implementation then flush out the comment details.



7. Write implementations of the functions described above.
    a. Your implementations may utilize coding you choose to include in a private function.
        i. A private function will be visible only to the code in your Datascript Module not within the Rules Editor.
8. Develop an application to test your logic.
    a. You can develop and test this application in the library named QV.
    b. You will need to create File Connection, Schema, and Dataflow artifacts.
        i. The Dataflow will probably require the Read File, Transform, and Write File operators.
    c. In the Transform operator,
        i. Add an expression rule then add a reference to your module. Click on **Module References…** to open the Module References window.

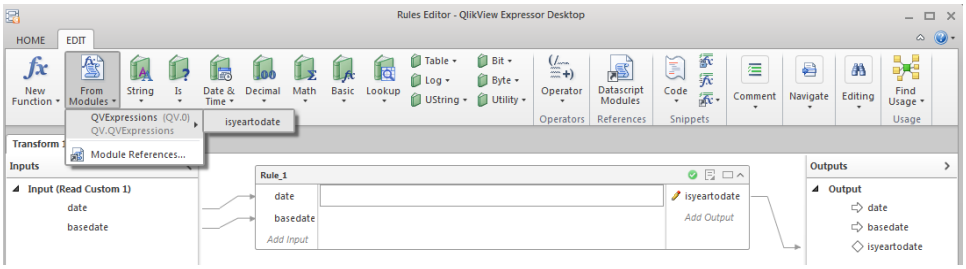


        ii. Select the Datascript Module and click **OK**.

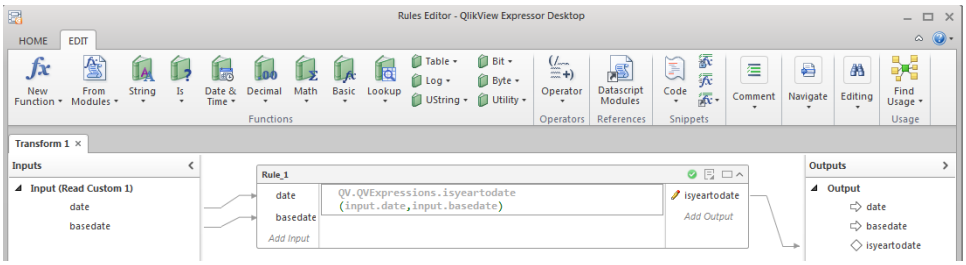


        iii. The code in your module is now available in this rule.

iv. To use a function from the module, place your cursor into the expression rule then click **From Modules** in the Edit tab of the rules editor ribbon bar.



v. The function name is prefaced with the module name. Complete the coding by filling in the function arguments.

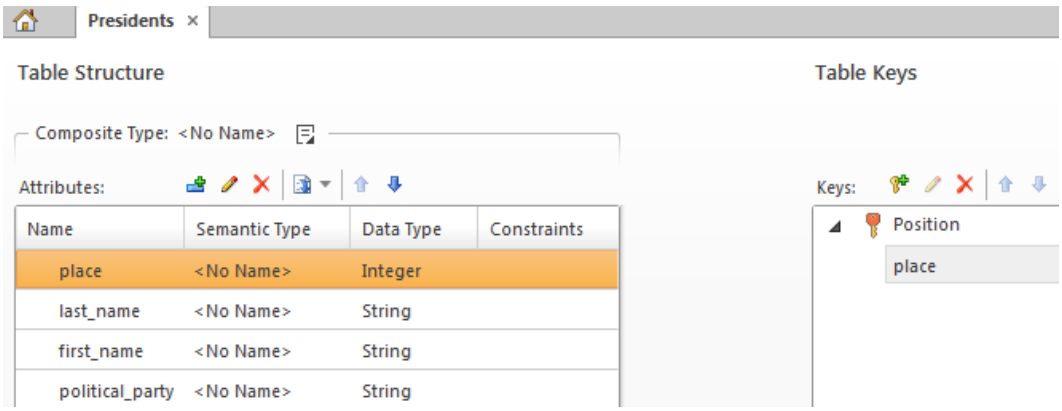


19 Transform Operator – Lookup Function Rule

A Lookup Function Rule requires that the developer implement the mandatory generate function, which is invoked when the lookup operation fails to return a record. As with the Lookup Expression Rule, a 'wizard' leads the developer through the process of selecting a lookup table and choosing appropriate keys. The developer then codes the processing that is executed when the lookup table fails to return a record.

Function Type	Function Name	Description
Mandatory Functions	generate	The data processing engine invokes this function whenever the lookup table fails to return a record. Include in the method body QlikView Expressor Datascript that generates a new output record. Arguments: • input – the record to be processed • output – the record to be emitted
Optional Helper Functions	initialize	The data processing engine invokes this function one time before the operator begins to process records. This function has no arguments or return values.
	finalize	The data processing engine invokes this function one time after the operator has completed processing all records. This function has no arguments or return values.

Before discussing the lookup functions and how they are used, let's start by creating a lookup table.



If you access a lookup table using a Lookup Expression Rule you will use a key value to retrieve the other attributes. If there is no entry in the lookup table with a specific key value, the Lookup Expression Rule will allow you to write a new entry into the lookup table. But the Lookup Expression Rule does not give you the ability to modify or delete entries from the lookup table. With a Lookup Function Rule, you have full control over the lookup table and the lookup functions are what you use to manipulate the contents of the lookup table.

The lookup functions are only available from within a Lookup Rule, although to use them from within a Lookup Expression Rule you would probably need to call a function defined within a Datascript Module, as no matter what you want to do with these functions you will need to write more than a single statement of code (which is all you can include in a Lookup Expression Rule).

The `expressor.lookup.connection` is the top-level object that you use to create the other lookup function objects. It is created using the function `get_connection` on the lookup object.

```
conn = lookup.get_connection("name of lookup table")
```

The name of the lookup table, the argument to the `get_connection` function, is comprised of the name of the project or library that contains the lookup table artifact concatenated to the name of the lookup table artifact, separated by an underscore. For example, if the Presidents lookup table were in a project (or library) name Government, the argument to the `get_connection` function would be `Government_Presidents`.

Once you have the connection object, you can get the reader, range_reader, writer, updater and deleter objects with functions such as:

```
writer  = conn:get_writer()
updater = conn:get_updater()
deleter = conn:get_deleter()
```

The `get_reader` and `get_range_reader` functions require an argument, which is the name of the key. For the Presidents lookup table, this would be `Position`, the name of the key and not `place` the name of the attribute comprising the key.

```
reader = conn:get_reader("Position")
range_reader = conn:get_range_reader("Position")
```

Each of the lookup objects implements the method `execute`. The arguments to this method differ depending on which object the method is called.

For the reader and range_reader objects, the Key argument is a string indexed datascript table in which the index name is the name of the key column(s) (not the key name), and the value is the key value, which you most likely obtain from an input parameter to the Lookup Function Rule. In the Lookup Function Rule, the input and output parameters are populated once you select the lookup table and key although you may add additional input parameters corresponding to the attributes in the incoming record.

To retrieve data related to a specific president, your function call would appear something like the following.

```
reader:execute({place=input.place})
```

If the key is comprised of multiple attributes, you simply create a table argument with multiple elements.

```
reader:execute({key1=val1,key2=val2,...})
```

Note that this invocation does not return a value. To obtain the return, you call the `reader:next` function, which returns a string indexed table with the data from the lookup and the `expressor.lookup.rowid`, the unique identifier for the data row in the lookup table.

```
president, rowid = reader:next()
```

The variable president will contain content similar to the following.

```
{place=16,last_name="Lincoln",first_name="Abraham",
 political_party="Republican"}
```

To obtain each value, use the standard table dot notation.

```
FirstName = president.first_name
```

The `rowid` is not something you can view or print; you use it as an argument when updating or deleting a row in the lookup table. If you have used a non-unique lookup table key, you can call `reader:next` multiple times to retrieve all of the entries returned from the lookup table.

To write new content to the lookup table you invoke execute method on the writer object. In this usage, the argument is a string indexed table that has the same structure as the lookup table. For example:

```
rowid, str = writer:execute(
  {place=44,last_name="Obama",first_name="Barack",
   political_party="Democratic"}
)
```

If the row is successfully written, the `rowid` is returned. If there is a problem, `rowid` is `nil` and an error message (`str`) is returned.

In order to update or delete a row from the lookup table, you must first read the row to obtain the `rowid`, which identifies the row you want to update or delete. To update a row in the lookup table, you invoke the execute method on the updater object. This function call requires two arguments; the first is the `rowid` of the row you want to update; the second is a string indexed table with the updated content. This table must include an element for every attribute in a lookup table entry.

```
rowid, str = updater:execute(
  rowid_of_row_to_update,{place=1,last_name="Washington",
  first_name="George",political_party="not affiliated"}
)
```

And to delete content from the lookup table, you pass the `rowid` as the argument to `execute`.

```
deleter:execute(rowid_of_row_to_delete)
```

Calling `execute` on the range_reader object may have surprising results. This call will return a lookup table entry that matches the key, but if there is no such record it returns the record with the next higher key value. For example,

```
reader:execute({place=1})
president, rowid = reader:next()
```

Will return

```
{place=1,last_name="Adams",first_name="John",
 political_party="Federalist"}
```

if George Washington's entry is not in the lookup table. Note that the value of place is inconsistent with the rest of the data.

Exercise – Lookup Function Rule

1. Create a new project named Lookup2.
2. Add a file connection artifact to the project.
3. Using the file containing the listing of presidents (presidents.dat), create a schema artifact.
    a. Be sure to change the type of the place attribute to integer.
4. Create a dataflow named FunctionLookup.
    a. On step 1 of the dataflow, initialize a lookup table with the listing of presidents.
        i. Use the place attribute for a unique key and the last name attribute for a non-unique key.
        ii. When configuring the Write Lookup Table operator, remember to select the Truncate property.
    b. On step 2 of the dataflow, add the Read File, Transform, and Write File operators.
        i. Configure the Read File operator to read a small text file (perhaps only 5 entries) where each row contains an integer number between 1 and 44 inclusive.
        ii. Add attributes for first name and last name to the output.
        iii. Add a Lookup Function Rule to the Transform operator and configure it to use the unique key in the lookup table created in step 4a.
            1. Set Output Nil as the On miss selection.
        iv. Set up the Write File operator to write the first and last names of each president from information retrieved from the lookup table.
    c. Test, and if necessary debug, the dataflow.
5. In step 2 of the dataflow,
    a. Reconfigure the Read File operator to read the presidents.dat file.
    b. In the Transform operator's Lookup Function Rule, reset the On miss selection to Generate Record.
    c. In the Rules Editor, block the first and last name and political party attributes so that they are not passed to the output.
    d. Confirm that the dataflow still successfully runs.
6. Open the presidents.dat file and add an entry for president number 45.
7. In the Rules Editor, using the preceding discussion and the boiler plate code in the Lookup Function Rule as guides, implement the initialize and generate functions of the Lookup Function Rule.
    a. Your code should add the 45th president to the lookup table and create an output record with the individual's first and last names.
8. Add a third step to the dataflow.
    a. Use the Read Lookup Table operator to read the contents of the lookup table.
    b. Use a Write File operator to write the retrieved values to a text file.
    c. Confirm that personal details of the 45th president are in the file.

Exercise – Lookup Function Rule (Range Reader)

1. Make a copy of the dataflow you created in the preceding exercise.
2. Reconfigure the Read File operator to use the input file containing the small listing of numbers (step 4.b.1 above).
3. Modify the generate function code you developed in the preceding exercise to use the range_reader interface.
4. Delete one or more entries from the presidents.dat file that correspond to entries in the small number listing file.
5. Run the dataflow and confirm that the first and last names of the president following each missing entry are emitted.
    a. For example, if your small listing file includes 16,
        i. Delete the 16th president (Abraham Lincoln) from presidents.dat so that the lookup table does not include this entry.
        ii. When the data flow runs, the output record for president 16 should include Andrew Johnson in place of Abraham Lincoln.

Can you think how to modify your dataflow such that the entry before, rather than after, the missing entry is emitted?

20 The Read Custom Operator

QlikView Expressor includes many fully implemented input operators such as Read File, Read Table, and SQL Query. While these three operators are suitable for many applications, you may encounter a situation in which you need to access a data resource not supported by these operators. In this situation, the Read Custom operator allows you to create your own input operator. To use this operator, you use QlikView Expressor Datascript to read and parse the data into the individual records emitted by the operator. The operator has one required function and two optional helper functions, but it is likely that you will need to provide implementations for all three functions.

Function Type	Function Name	Description

Mandatory Functions	read	The data processing engine invokes this function repeatedly to produce each record emitted by the operator. Include in the method body QlikView Expressor Datascript that initializes the outgoing record.
		
		This function has no input arguments. It returns a status value, which determines whether invocations of this function continue, and the output record or a message. The status values are defined in the datascript module named expressor.ScriptSupport.
		
		• The QlikView Expressor runtime will continue to invoke this function as long as the previous invocation returns the output record or two return values – the status value expressor.ScriptSupport.OK and the output record.
		• Repeated invocations of this function will cease if the previous invocation returns true or the status value expressor.ScriptSupport.FlowComplete.
		• This function may also return the status values expressor.ScriptSupport.SkipRecord or expressor.ScriptSupport.RejectRecord.
		  ○ If SkipRecord is returned, the read function continues to be invoked after skipping the current record.
		  ○ If RejectRecord is returned, the record and optional messages are emitted from the operator's reject port.

Optional Helper Functions	initialize	The data processing engine invokes this function one time before the operator begins to invoke the read function. This function has no arguments or return values. Use this function to set up a connection to your data source, e.g., establish a connection to an FTP server or obtain a handle to a file that will be read during each invocation of the read function.
	finalize	The data processing engine invokes this function one time after the operator has completed emitting all records, that is, after the read function has returned expressor.ScriptSupport.FlowComplete or expressor.ScriptSupport.SkipRemaining. This function has no arguments or return values. Use this function to free any resources obtained during execution of the initialize function.

Exercise: The Read Custom Operator

In this example, you will add code to a Read Custom operator so that it reads a file and copies each line to an output file. While very basic, this example will show you how to use this operator's functions. Since the source file includes a header row, your implementation must be able to identify this row and not emit it to the downstream operator.

1. Create a new project named **CustomOperators**.
2. Add a File Connection with the path C:\data to the project.
3. Create a dataflow that includes the Read Custom and Write File operators.
    a. Name this dataflow **ReadCustom**.
4. Select the Read Custom operator and open its Rules Editor and note the starting point code.
5. Rework the code as shown in the following screen shot.

Read Multiple Files

```
require "expressor.ScriptSupport"
file_handle = nil

function initialize()
  -- read multiple files from the same file system location
  file_handle = io.popen(
    "type c:\\data\\republicans.dat c:\\data\\democrats.dat c:\\data\\whigs.dat")
end;

function read()
  -- retrieve a line
  line = file_handle:read("*l")
  if line then
    -- determine if record is for a member of the Whig party
    if string.find(line,"Whig") then
      -- drop members of Whig party
      return expressor.ScriptSupport.SkipRecord
    else
      -- emit records for Republicans and Democrats
      output.line = line
      return expressor.ScriptSupport.OK, output
    end
  else
    -- no more records to process
    return expressor.ScriptSupport.FlowComplete
  end
end

function finalize()
  file_handle:close()
end
```

Outputs

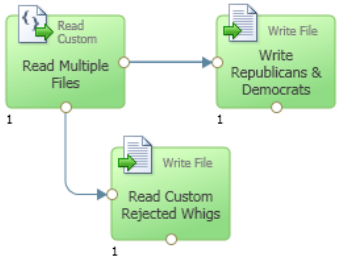
Output (Write File 1)

line

    a. The initialize function is used to obtain a handle to the file that will be processed.
        i. This function is called before the read function.
    b. The finalize function is used to close the file handle.
        i. This function is called after the read function has returned the indicator value true.
    c. The read function uses the file handle to read each line of the file.
        i. When this function returns expressor.ScriptSupport.SkipRecord (or false), no record is emitted and the read function is invoked again.
        ii. When this function returns expressor.ScriptSupport.OK and the initialized output (or just the initialized output), a record is emitted and the read function is invoked again.
        iii. When this function returns expressor.ScriptSupport.FlowComplete (or true), the Expressor runtime shuts down the Read Custom operator.
            1. This indicates that there are no more records to emit.

2. The read function is not invoked.

6. Since the output record contains a single attribute named line, you need to create a corresponding entry within the output panel.
    a. Click Add in the Output Attributes grouping on the Home tab of the ribbon bar.
    b. Create a string attribute named line.
    c. Click OK then close the Rules Editor.
7. Set the Read Custom operator's **Error Handling** property to **Skip Record**.
8. Select the Write File operator.
    a. Select the File Connection.
    b. Click the button to the right of the Schema drop down control and select **New Delimited Schema from Upstream Output…**.
        i. Work through the steps of the wizard.
        ii. Name the schema LineSchema.
    c. Make an entry into the **File name** control.
    d. Choose whatever options for the other properties that you want.
9. Save and run your dataflow.
    a. Observe that the contents of the source files democrats.dat and republicans.dat have been copied to your output file but that the records in the file whigs.dat were not written to the output.



What if you want to capture the records for members of the Whig party by redirecting them to the Read Custom operator's reject port? In this case, rather than returning expressor.ScriptSupport.SkipRecord, your code returns expressor.ScriptSupport.RejectRecord and one or more additional values that include the content of the rejected record and a message describing why the record was rejected. To capture the rejected records, you must connect a Write File operator to the Read Custom reject port (create the schema from the upstream output) and change the **Error Handling** property of the Read Custom operator to **Reject Record**.

In the Read Custom operator, change line 17 to

```
17      return expressor.ScriptSupport.RejectRecord,line,"some message","a reason the record was rejected"
```

The status value causes the record to be rejected and the reject record will include the entire content of the record that was rejected. Two other optional return values, which give reasons for the rejection, could be added.

21 The Write Custom Operator

QlikView Expressor includes many fully implemented output operators such as Write File and Write Table. While these three operators are suitable for many applications, you may encounter a situation in which you need to write to a data resource not supported by these operators. In this situation, the Write Custom operator allows you to create your own output operator.

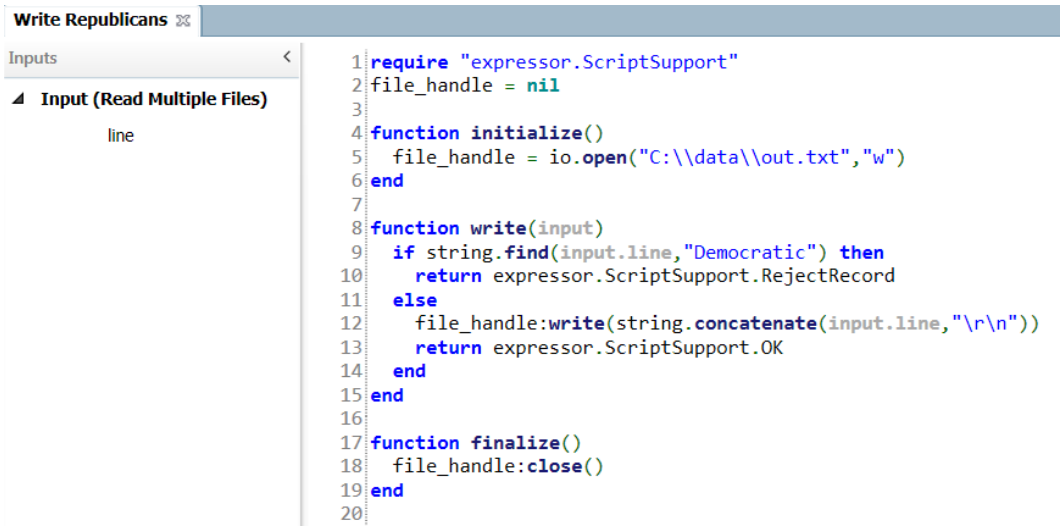
To use this operator, you use QlikView Expressor Datascript to extract data from each incoming record and write to the external resource.

The operator has one required function and two optional helper functions, but it is likely that you will need to provide implementations for all three functions.

Function Type	Function Name	Description
Mandatory Functions	write	The data processing engine invokes this function as each record enters the operator from its upstream operator. Include in the method body QlikView Expressor Datascript that writes the record to the external data resource. The record received from the upstream operator is the method's argument. This function returns one of three indicator values that are defined in the expressor.ScriptSupport datascript module. • The record received as the argument will be emitted from the operator's reject port when this function returns the indicator value expressor.ScriptSupport.RejectRecord. • Processing of the record received as the argument will be skipped when this function returns the indicator value expressor.ScriptSupport.SkipRecord. Since formatting and writing the output is performed by developer supplied code, these return values are independent of, and do not affect whether, a record is written to the external data resource.
Optional Helper Functions	initialize	The data processing engine invokes this function one time before the operator begins to invoke the write function. This function has no arguments or return values. Use this function to set up a connection to your data source, e.g., establish a connection to an FTP server or obtain a handle to a file that will be written during each invocation of the write function.
	finalize	The data processing engine invokes this function one time after the operator has completed emitting all records, that is, after the operator stops receiving records from its upstream operator. This function has no arguments or return values. Use this function to free any resources obtained during execution of the initialize function.

Exercise: The Write Custom Operator

1. Make a copy of the dataflow used in the previous exercise.
    a. Name the dataflow **WriteCustom**.
2. Delete the Write File operator.
3. Add a Write Custom operator to the dataflow and connect to the Read Custom operator.
4. Select the Write Custom operator, open its Rule Editor, and enter the following coding.



    a. Note in line 1 the reference to the datascript module expressor.ScriptSupport. Although use of this module is not necessarily required, it does provide the ability to reject a record as well as skipping or emitting a record.
    b. In line 10, records for Presidents who are members of the Democratic party are directed to the Write Custom operator's reject port rather than to the output port.
    c. Line 13 is optional as the write function will be invoked when the next record arrives from the upstream operator.
    d. Select **Reject Record** as the **Error Handling** property.
5. Connect a Write File operator to the reject port of the Write Custom operator.
    a. Select the File Connection.
    b. Create a Schema artifact by selecting **New Delimited Schema from Upstream Output…**.
    c. Give the file a unique name.
6. Save and run the dataflow.
7. Examine the contents of the output file out.txt and the file written by the Write File operator.

API Manual

Appendix – QlikView Expressor Datascript API Summary

The functions you will most likely use are in the basic, datetime, io, is, os, string, and table categories. In addition, QlikView Expressor Datascript includes the standard arithmetic, relational, and logical operators that are typical of other programming languages.

Basic Functions

Many of these functions are used to convert a value of one type into another type and will not be reviewed in this document. The functions ipairs, next, and pairs operate on QlikView Expressor Datascript tables and will be detailed in a later section of this document. Within the scope of this training course, the following basic functions are important.

Function	Purpose
decision	This function can be used to create a single statement that implements the same sort of processing as a case statement.
decode	This function can be used to create a single statement that implements the same sort of processing as an if..then..else or an if..elseif..else statement.
ipairs	This function returns an iterator over a numerically indexed QlikView Expressor Datascript table.
pairs	This function returns an iterator over any QlikView Expressor Datascript table.
pcall	This function invokes a function in protected mode, returning true and the function's return values if execution completes normally or false and an error message if execution did not complete normally.
select	This function can be used to select specific return values from a function.
todecimal	Converts a number or numeric string to a decimal type.
tointeger	Converts a number or numeric string to an integer type.

decision function

usage	**`decision(comp1, ret1 [,comp_n, ret_n]* [,default])`**	
arguments	Comp1	Required logical expression.
	ret1	The return value associated with the required logical expression.
	comp_n, ret_n	Optional additional logical expressions and associated return values.
	default	Optional default return value.
return	true and return value(s) from function or false and error message	
usage	`decision(comp1, ret1 [,comp_n, ret_n]* [,default])`	

When you use the decision function, you do not need to explicitly code a return statement or an assignment statement. Rather, the decision function will return a value that you specify. Consequently, you can use the decision function to implement the equivalent of a complex block of code while adhering to the guidelines required of the Expression Editor.

The API of the decision function sets up a series of logical comparisons, each associated with a return value. The function evaluates each comparison in sequence and returns the return value for the first comparison that evaluates to true.

As an example, let's assume that each incoming record includes four attributes: the president's chronological position, last name, first name, and party. We want to rework the data in the input record, concatenating a president's first and last names and identifying in which century he served. Each outgoing record includes a single attribute named summary containing a statement such as "George Washington is an eighteenth century president." You could write QlikView Expressor Datascript to implement this logic but since a case statement is not part of the QlikView Expressor Datascript language, you would be forced to use an if..elseif..else block.

But you could implement the same logic as a single statement, replacing the if..elseif..else block with the decision function.

```
output.summary =
  string.concatenate(
    input.fname, " ", input.lname,
    decision(
      input.place<=2, " is an eighteenth century president",
      input.place>2 and input.place<26, " is a nineteenth century president",
      input.place>25 and input.place<43, " is a twentieth century president",
      " is a twentyfirst century president")
  )
```

Note how the each comparison argument to the decision function performs the same logical discrimination as a case block within a case statement and that by using the decision function as an argument to the string.concatenate function, it is possible to generate the entire output value within a single statement.

The code could also be entered into an expression rule as its syntax adheres to the guidelines for using this type of rule.

decode function

usage	`decode(val1, val2, ret2 [, val_n, ret_n]* [,default])`	
arguments	val1	The value to be compared.
	val2	Required comparison value.

	ret2	Return value associated with val2.
	val_n, ret_n	Optional comparison values and associated returns.
	default	Optional default return value.
return	true and return value(s) from function or false and error message	

When you use the decode function, you do not need to explicitly code a return statement or an assignment statement. Rather, the decode function will return a value that you specify. Consequently, you can use the decode function to implement the equivalent of a complex block of code while adhering to the guidelines required of an expression rule.

The API of the decode function sets up a series of logical comparisons, each associated with a return value. The function evaluates each comparison in sequence and returns the return value for the first comparison that evaluates to true.

As an example, let's assume that each incoming record includes four attributes: the president's chronological position, last name, first name, and party. We want to rework the data in the input record, concatenating a president's first and last names and his party affiliation into a single statement. Each outgoing record includes a single attribute named summary containing a statement such as "George Washington is neither a Democrat nor a Republican." You could write QlikView Expressor Datascript to implement this logic using an if..elseif..else statement.

```
name=
  string.concatenate(input.fname, " ", input.lname)

if input.party=="Democratic" then
  output.summary = string.concatenate(
                      name, " is a Democrat")
elseif input.party=="Republical" then
  output.summary = string.concatenate(
                      name, " is a Republican")
else
  output.summary = string.concatenate(
                      name, " is neither a Democrat nor a Republican")
end
```

But you could implement the same logic as a single statement, replacing the if..elseif..else block with the decode function.

```
output.summary =
  string.concatenate(
    input.fname, " ", input.lname,
    decode(input.party,
      "Democratic", " is a Democrat",
      "Republican", " is a Republican",
      " is neither a Democrat nor a Republican")
  )
```

Note how the four arguments to the decode function perform the same logical comparisons as the if..elseif..else block and that by using the decode function as an argument to the string.concatenate function, it was possible to generate the entire output value within a single statement.

The implementation that employs the decode function could be entered into an expression rule as its syntax adheres to the guidelines for using this type of rule.

pcall function

usage	**pcall(function [,arguments])**	
arguments	function	A string containing the name of the function to execute.
	arguments	A comma separated list of the arguments to the function.
return	true and return value(s) from function or false and error message	

You use this function when you fear that a function may return a nil or false value, or throw a fault, which you cannot subsequently process. For example, suppose you want to convert a string representation of a datetime into a datetime value. The string.datetime function will perform such a conversion, but if the string cannot be converted into a valid date, the function will throw a fault. By wrapping the string.datetime function call within the pcall function, you can suppress the fault and evaluate an indicator return value in order to decide how to continue your processing.

```
indicator, value = pcall(string.datetime,"02/31/2011",
"MM/DD/CCYY")
```

Rather than throwing a fault, indicator will contain the value false and value will contain the fault message. Your code can test the value of indicator and make a decide how to proceed.

If the string.datetime function executes without error, indicator will contain the value true and value will contain the datetime value.

select function

usage	**select(index, …)**	
arguments	index	The first argument to return.
	…	Function or comma separated listing of arguments to evaluate.
Return	return value at index and	

For example: `print(select(3,"a","b","c","d"))` prints c and d, while enclosing the select function in parentheses limits the return to the value at index.

`print((select(3,"a","b","c","d")))` prints just c

Datetime Functions

Within the scope of this training course, the following datetime functions are important.

Function	Purpose
adjust	Operates on a datetime value returning that value adjusted by a specified interval.
elapse	Operates on two datetime values returning the difference between them.
moment	Operates on a datetime value returning a specific element (seconds, minutes, hours, day, month, year, century).
string	Operates on a datetime value returns a string representation.

adjust function

usage	`datetime.adjust(value, interval [,format [,exact]])`	
arguments	value	The datetime value to adjust.
	interval	The adjustment to be applied.
	format	The interpretation of the adjustment interval. format (case insensitive) / interpretation: none – seconds s – seconds i – minutes h – hours d – days y – years c – centuries
	exact	If false (the default), intervals are calculated using a 365.25 day year. If true, intervals are calculated using a 365 day year.
return	datetime	

elapse function

usage	`datetime.elapse(value1, value2 [,format])`	
arguments	value1	The starting datetime value.
	value2	The ending datetime value.
	format	The units in which to express the differential.

format (case insensitive)	interpretation
none	seconds
s	seconds
i	minutes
h	hours
d	days
y	years
c	centuries

Note that months (m) is not a valid format.

return number

moment function

usage `datetime.moment(value [,format])`

arguments

value — The datetime value to analyze.

format — The units in which to express the differential.

format (case insensitive)	interpretation
none	seconds
s	seconds
i	minutes
h	hours
d	days
j	Julian day (leap year has 366 days)
w	day of week (Sunday is 0)
m	month
y	year
c	century

return number

string function

usage `datetime.string(value [,format])`

arguments

value — The datetime value to convert into a string.

format — The format of the string representation.
Default is `CCYY-MM-DD HH24:MI:SS`.

format	interpretation

HH24	hours in 24 hour format
H*24	hours in 24 hour format; single digit hour formatting when appropriate
HH12	hours in 12 hour format
H*12	hours in 12 hour format; single digit hour formatting when appropriate
HH	hours in 24 hour format
H*	hours in 24 hour format; single digit hour formatting when appropriate
MI	minutes
SS	seconds
S[ssssss]	fractional seconds
AM or PM	used with HH or HH12 to indicate whether hour values are AM or PM; only valid if a full time format, including fractional seconds, is specified; this value is included in the output
DD	day in numeric format
D*	day specified as either one or two digits format pattern must be delimited, i.e., MM-D*-CCYY or MM/D*/CCYY, not MMD*CCYY; valid format delimiters are space, hyphen, forward slash, comma and period
D?	invalid day specification accepted; converts the day to either 01 or the last day of the month based on the input value
DM	allows processing of mixed day/month, giving precedence to day; used in conjunction with the MD format
DDD	day of week abbreviated (e.g., MON)
DAY	day of week abbreviated (e.g., MON)
DDDD	day of week in long format (e.g., MONDAY)
DDAY	day of week in long format (e.g., MONDAY)
JJJ	Julian day of year
MM	month in numeric format
M*	month specified as either one or two digits; format pattern must be delimited, i.e., M*-DD-CCYY or M*/DD/CCYY, not

			M*DDCCYY; valid format delimiters are space, hyphen, forward slash, comma and period
		M?	invalid month specification accepted
		MD	allows processing of mixed month/day, giving precedence to month; used in conjunction with DM format
		MMM	month in short format (e.g., JAN)
		MMMM	month in long format (e.g., January)
		YY	years
		YNN	Forces a century designation anchored to NN; In a date field, a two character year is interpreted as the current century if less than NN and the previous century if greater than NN
		CC	century
return	string		

IO Functions

Within the scope of this training course, the following IO functions are important.

Function	Purpose
io.open	Opens a file for reading, writing, etc.
file_handle:close	Closes a file.
file_handle:read	Reads a file's contents.
file_handle:lines	Returns an iterator function for reading a file line by line.
file_handle:write	Writes to a file.
file_handle:flush	Flushes data to a file.

open

usage	`io.open(fn [,mode])`	
arguments	fn	A string containing the name of the file.
	mode	A string containing the mode. "r", read; "w", write; "a", append; "r+", update-preserve, "w+", update-erase; "a+", update-append
return	file handle used to invoke other IO functions	

close

usage	`file_handle:close()`	
arguments	none	
return	none	

read

usage	`file_handle:read(format)`	
arguments	format	A string specifying what to read. "*n", number; "*a", entire file (returns "" at end of file); "*l", a line (returns nil at end of file); n, string of up to n characters (returns nil at end of file)
return	varies	

lines

usage	`file_handle:lines()`	
arguments	none	
return	iterator function for reading file line-by-line	

write

usage	`file_hande:write(values)`	
arguments	values	Values to write to file. Values may be numbers or strings. No separators are added between values
return	none	

flush

usage	`file_handle:flush()`	
arguments	none	
return	none	

IS Functions

Within the scope of this training course, the following is functions are important.

Function	Purpose
decimal, datetime, integer, number, string	Return true if the value being tested is of the corresponding type.
blank	Returns true if the value being tested consists entirely of space characters.
empty	Returns true for a nil value, zero numeric value, or empty string value.
future, past	Returns true if datetime value is in the future or past.
null	Returns true if value is nil.
pattern	Returns a character pattern if it is present in the value being tested; otherwise returns nil.

blank, decimal, datetime, empty, future, integer, null, number, past, string functions

usage	`is...(value)`	
arguments	value	The value to test
return	boolean	

pattern

usage	`is.pattern(value, pattern[, begin])`	
arguments	value	The value to test.
	pattern	The pattern to find.
	begin	Optional start position at which to begin the search. Default is 1. Negative values are offsets from the right end of value.
return	string	

Lookup Functions

These lookup functions may only be used from within a Lookup Function Rule within a Transform operator.

Function	Purpose
lookup.get_connection	Opens a connection to a lookup table.
connection:get_deleter **connection:get_reader** **connection:get_range_reader** **connection:get_updater** **connection:get_writer**	Creates on object used to • delete records from a lookup table • extract records from a lookup table • update records in a lookup table • write new records into a lookup table
connection:lock	Prevents access to a lookup table (executes a lock) except by objects created by this connection.
connection:unlock	Opens access to a lookup table from objects other than the ones created by this connection.
execute	Function on delete, reader, range_reader, updater and writer objects to act against lookup table.
next	Function to obtain each record extracted by the execute function on a reader object.

lookup:get_connection

usage	`lookup:get_connection("lookup")`	
arguments	lookup	Name of the lookup table, which is a concatenation of the name of the project or library containing the lookup table and the lookup table's name, separated by an underscore. Quotation marks are required part of the syntax.
return	A connection object used to create deleter, reader, range_reader, updater and writer objects.	

connection:get_…

usage	`connection:get_deleter()`
	`connection:get_reader("key")`
	`connection:get_range_reader("key")`
	`connection:get_updater()`
	`connection:get_writer()`	

arguments	key	Name of a unique or non-unique table key. A key may be composed of one or more attributes. Quotation marks are required part of the syntax.
return	A delete, reader, range_reader, updater, or writer object.	

connection:lock

usage	`connection:lock()`
arguments	none
return	A return of true indicates that a lock was successfully placed on the lookup table. A return of false indicates that another process currently holds a lock.

connection:unlock

usage	`connection:unlock()`
arguments	none
return	A return of true indicates that the lock was successfully removed from the lookup table.

object:execute

usage	`deleter:execute(rowid)`
	`reader:execute({columnName})`
	`range_reader:execute({columnName})`
	`updater:execute(rowid,{attributeName})`
	`writer:execute({attributeName})`	

arguments	rowid	The row identifier of the target row. You must first use reader:execute to obtain the identifier.
	{columnName}	A string indexed datascript table in which the index name is the name of the key column (not the key name), and the value is the key value. If the key is composed of multiple attributes, create a table argument with multiple elements. `{keyColumnName=columnValue[,…]}`
	{attributeName}	A string indexed datascript table in which the element name is the name of a table attribute and the value is the updated attribute value. This table must include an element for each table attribute in the lookup table. `{attribute=val[,…]}`
return	None: deleter, reader, range_reader. rowid, message: updater and writer. If successful, row identifier of the	

	updated or inserted row and an empty message string. If unsuccessful, a nil row identifier and an error message.

object:next

usage	**`reader:next()`**
arguments	none
return	row, rowid • If the result set returned by the `reader:execute` function is not empty, returns a row from the result set and the row identifier. • If a non-unique key was provided to the `reader:execute` function, the result set may include multiple entries. o Invoke `next` to obtain each row.

OS Functions

Within the scope of this training course, the following OS functions are important.

Function	Purpose
execute	Executes a command returning a system dependent status code.
getenv	Returns a string with the value of an environment variable or nil if the variable does not exist.
remove	Deletes a file.
rename	Renames a file.

execute

usage	`os.execute(cmd)`	
arguments	cmd	A string containing the command to execute.
return	number	

getenv

usage	`os.getenv(var)`	
arguments	var	String containing the name of an environment variable.
return	string	

remove

usage	`os.remove(fn)`	
arguments	fn	String containing name of the file to delete.
return	In case of error, nil and error description	

rename

usage	`os.rename(of, nf)`	
arguments	of	String containing current name of file.
	nf	String containing new name of file.
return	In case of error, nil and error description	

String Functions

Within the scope of this training course, the following string functions are important. These functions all operate on a string value, generally returning an altered string.

Function	Purpose
allow	Returns a string containing only the allowed characters.
concatenate	Operates on a list of values returning a concatenated string.
datetime	Converts a string representation of a datetime into an unformatted datetime type.
filter	Returns a string from which the filtered characters have been removed.
find	Returns the starting, ending, and optional capture of a specified character pattern.
frequency	Returns the number of times a specified character pattern is found.
iterate	Returns all occurrences of a specified character pattern.
length	Returns the length of a string.
match	Returns, if present, the characters corresponding to a specified character pattern.
replace	Returns a string with all occurrences of a specified character sequence replaced with another character sequence.
substring	Returns a substring of the string.
trim	Removes space characters from both ends of the string.

allow function

usage	`string.allow(value, allowed)`	
arguments	value	String value to modify
	allowed	Subset of characters to return
return	string	

concatenate function

usage	`string.concatenate(value [,value_n])`	
arguments	value	First value to be concatenated
	value_n	Additional values to be concatenated
return	string	

usage `string.datetime(value [,format])`

arguments

value — String to be converted into a datetime.

format — The format of the string representation. The format is optional only when the string representation has the default format. Default format is `CCYY-MM-DD HH24:MI:SS`.

format	interpretation
HH24	hours in 24 hour format
H*24	hours in 24 hour format; single digit hour formatting when appropriate
HH12	hours in 12 hour format
H*12	hours in 12 hour format; single digit hour formatting when appropriate
HH	hours in 24 hour format
H*	hours in 24 hour format; single digit hour formatting when appropriate
MI	minutes
SS	seconds
S[ssssss]	fractional seconds
AM or PM	used with HH or HH12 to indicate whether hour values are AM or PM; only valid if a full time format, including fractional seconds, is specified; this value is included in the output
DD	day in numeric format
D*	day specified as either one or two digits format pattern must be delimited, i.e., MM-D*-CCYY or MM/D*/CCYY, not MMD*CCYY; valid format delimiters are space, hyphen, forward slash, comma and period
D?	invalid day specification accepted; converts the day to either 01 or the last day of the month based on the input value
DM	allows processing of mixed day/month, giving precedence to day; used in conjunction with the MD format
DDD	day of week abbreviated (e.g., MON)
DAY	day of week abbreviated (e.g., MON)

	DDDD	day of week in long format (e.g., MONDAY)
	DDAY	day of week in long format (e.g., MONDAY)
	JJJ	Julian day of year
	MM	month in numeric format
	M*	month specified as either one or two digits; format pattern must be delimited, i.e., M*-DD-CCYY or M*/DD/CCYY, not M*DDCCYY; valid format delimiters are space, hyphen, forward slash, comma and period
	M?	invalid month specification accepted
	MD	allows processing of mixed month/day, giving precedence to month; used in conjunction with DM format
	MMM	month in short format (e.g., JAN)
	MMMM	month in long format (e.g., January)
	YY	years
	YNN	Forces a century designation anchored to NN; In a date field, a two character year is interpreted as the current century if less than NN and the previous century if greater than NN
	CC	century
return	datetime	

filter function

usage	`string.filter(value, filter)`	
arguments	value	String value to modify
	filter	Subset of characters to remove
return	string	

find function

usage	**string.find(value, pattern [, begin [, offset]])**	
arguments	value	The string value to examine.
	pattern	Character pattern to find
	begin	Optional starting character in value. If negative, offset search from right end of value and search from left to right.
	offset	If true, turn off character class capabilities in pattern and the function does a basic find substring operation.
return	Integer, integer [, string]	

iterate function

usage	**string.iterate(value, pattern)**	
arguments	value	The string to examine
	pattern	A character pattern to find in the value
return	function	

Since the iterate function returns an iterator, you must capture the possible multiple return character strings in a numerically indexed table. To determine if the iterator function returned any captures, check the length of the table.

```
return_strings = {}
for ret in string.iterate(value, pattern) do
  return_strings[#return_strings+1] = ret
end
```

frequency function

usage	**string.frequency(value, pattern)**	
arguments	value	String value to examine
	pattern	A character pattern to find in the value
return	number	

length function

usage	`string.length(value)`	
arguments	value	The string to examine
return	integer	

match function

usage	`string.match(value, pattern [, begin])`	
arguments	value	The string value to examine
	pattern	Character pattern to find
	begin	Optional starting character position. If negative, offset search from right end of value and search from left to right.
return	String	

replace function

usage	`string.replace(value, old, new[,count])`	
arguments	value	String value to modify
	old	Character sequence to be replaced
	new	Character sequence to be inserted
	count	Number of times to replace the character string old
return	string (the revised string), integer (number of replacements)	

substring function

usage	`string.substring(value [, start [, end]])`	
arguments	value	String value to modify
	old	Optional positional offset into value; default is 1. If negative, offset from right end of value.
	new	Optional positional offset of last character to return. Default is the last character of value.
return	string	

trim function

usage	`string.trim(value)`	
arguments	value	String value to modify
return	string	

Table Functions

Within the scope of this training course, the following table functions are important.

Function	Purpose
concat	Concatenates the values of a numerically indexed table into a single string.
insert	Inserts an element into a numerically indexed table.
remove	Removes an element from a numerically indexed table.
sort	Sorts a numerically indexed table.
#	Returns the number of elements in a numerically indexed table.

concat

usage	`table.concat(table [, separator [, first [, last]]])`	
arguments	table	A numerically indexed table where all values are numbers or strings.
	separator	The character to insert between each value extracted from the table. The default separator is a space character.
	first	The index of the first element to extract from the table. Default is 1.
	last	The index of the last element to extract from the table. Default is the last element in continuous numerical order.
return	string	

insert

usage	`table.insert(table, [position,] value)`	
arguments	table	A numerically indexed table into which to insert an element.
	position	The index at which to insert the element. Default is at the end of the table.
	value	The value to insert
return	The table is modified in place.	

remove

usage	**table.remove(table [,position])**	
arguments	table	A numerically indexed table from which to remove an element.
	position	The index at which to remove the element. Default is at the end of the table.
return	file handle used to invoke other IO functions	

sort

usage	**table.sort(table [,order])**	
arguments	table	A numerically indexed table to sort.
	order	A function that takes two arguments and returns true when the table element represented by the first argument should come before the table element represented by the second element.
return	The table is modified in place.	

For example, this order function, where a and b represent values within each table element, will result in the table being sorted descending.

```
table.sort(table, function(a,b) return (a>b) end)
```

While this order function will result in the table being sorted ascending.

```
table.sort(table, function(a,b) return (a<b) end)
```

#

usage	**#table**
arguments	none
return	The number of elements in a numerically indexed table.

The DSEX Datascript Module

The QlikView Expressor DSEX Datascript Module, introduced in QlikView Expressor 3.10 (Expressor), extends the functions available in the os and io function groupings. These function groups are defined in the Lua modules that underlie Expressor Datascript.

To use the DSEX module, your code must include the statement

```
require "dsex"
```

The following table summarizes the functionality in this module and the Lua defined os function group.

os function group	DSEX extended os function group	Functionality
chdir(d)	chdir(d)	Change the working directory to d (the directory path).
chmod(f,p)	chmod(f,p)	Changes permissions for file f using UNIX permission bits (supplied as a string, e.g., "777").
clock()	clock()	Returns the time used by the CPU (in seconds).
	currentdir()	Returns the path to the current directory.
cwd()	cwd()	Returns the path to the current directory.
date([f[,t]])	date([f[,t]])	Returns the current date and time as a string with format f; default is current date/time with format DD/MM/CCYY HH:MI:SS. Argument t is a value returned by the os.time() function.
difftime(t1,t2)	difftime(t1,t2)	Returns the difference between two values returned by the os.time() function.
	dir(p)	An iterator function for the contents in directory p. Returns a string indexed table that includes indices type, name, & size for each file or subdirectory. **`for entry in os.dir(p) do … end`**
	dirent(f)	Returns a string indexed table that includes indices type and size for the file f.
	environ	Returns a string indexed table where the indices are the names of environment variables and the values are the corresponding environment variable setting.
execute(c)	execute(c)	Executes the command c. Waits until the command completes and returns the exit code from the command.
exit()	exit()	Terminates the program.
getenv(e)	getenv(e)	Returns the value of the environment variable e.
is_dir(p)	is_dir(p)	Returns true if p is a directory.
is_file(f)	is_file(f)	Returns true if f is a directory.
mkdir(p)	mkdir(p)	Creates the directory p.
readdir(p)	readdir(p)	Returns a numerically indexed table where each entry is the name of a file or subdirectory in directory p.
remove(f)	remove(f)	Deletes file f. On error, returns nil and error description.
rename(of,nf)	rename(of,nf)	Renames file of to nf. On error, returns nil and error description.

rmdir(p)	rmdir(p)	Removes directory p. On error, returns nil and error description.
	setenv(e,v)	Sets the value of environment variable e to v. If v is nil, deletes environment variable; if variable doesn't exists, returns nil and error description.
setlocale(s[,c])	setlocale(s[,c])	Sets the local set by string s for category c. Categories: "all", "collate", "ctype", "monetary", "numeric", "time"; default "all".
	sleep(d[,u])	Suspends program execution for duration d in units u. Duration may be decimal; default unit is seconds.
	spawn(p[,[a])	Creates a child process for program p. If specified, a is one of the following: • A numerically indexed table of command line arguments • A string indexed table of environment variables • stdin, stdout, stderror file handles Returns a user object (proc) that implements the following function exitcode = proc:wait() that waits for the child process to complete before completing; exitcode is the value returned by the child process.
time([t])	time([t])	Returns a system dependent number representing a datetime described by the string indexed table t. Table t must have the index entries year, month, day and may have entries for hour, min, sec, isdst(true=daylight savings time).
tmpname()	tmpname()	Returns a random file name (does not create the file).

Note that when using the DSEX function setenv, the operating system's environment variables are not modified. To extract the value of variables set by this function you must use the os.spawn function as illustrated in the following statement, where … is the name of the environment variable.

```
proc=os.spawn("datascript",{"-e","print(os.getenv('…'))"}); proc:wait()
```

The following table summarizes the functionality this module adds to the Lua defined io function group

DSEX extended io function group	Functionality
f:lock(mode,offset,length)	Lock or unlock a file or a portion of a file associated with the file handle f. Mode: "r", read; "w", write; "u", unlock. Offset and length are optional positions in the file.
f:unlock(offset,length)	Unlocks a file or portion of a file associated with the file handle f. Offset and length are optional positions in the file.
io.lock(f,mode,offset,length)	Lock or unlock the file f or a portion of the file f. Mode: "r", read; "w", write; "u", unlock. Offset and length are optional positions in the file.
io.unlock(f,offset,length)	Unlocks a file or portion of the file f. Offset and length are optional positions in the file.

QlikView Expressor Datascript Pattern Matching Syntax

Several of the string functions utilize a pattern to isolate character strings within a larger string. A character pattern could be as simple as a few characters enclosed within quotation marks, such as, “expressor”. Alternatively, a pattern could be a little more cryptic, for example, all alphanumeric characters before the @ character. The QlikView Expressor Datascript Pattern Matching Syntax lets you develop patterns that can identify any character string. In order to work with patterns, you must understand the following concepts.

Character Class

A character class is used o represent a set of characters. The following character combinations are allowed when describing a character class.

- Any keyboard character represents itself.
    - The characters `^$()%.[]*+-?` are “magic” cracters that cannot directly represent themselves. These characters must be escaped with a preceding `%` character.
    - The `%` escape character may be placed before any non-alphanumeric character (e.g., punctuation marks, slashes, or the pipe character) to ensure that no special interpretation is attached to the character.
- A dot (period) represents all characters.
- %a represents all letters.
    - %A represents the complement of %a.
- %c represents all control characters.
    - %C represents the complement of %c.
- %d represents all digits.
    - %D represents the complement of %d.
- %l represents all lower case letters.
    - %L represents the complement of %l.
- %p represents all punctuation characters.
    - %P represents the complement of %p.
- %s represents all space characters.
    - %S represents the complement of %s.
- %u represents all upper case letters.
    - %U represents the complement of %u.
- %w represents all alphanumeric characters.
    - %W represents the complement of %w.
- %x represents all hexadecimal digits.
    - %X represents the complement of %x.
- %z represents the character that represents zero value.
    - %Z represents the complement of %z.

Set

A character class that includes a union of characters indicated by enclosing the characters in square brackets [ ] is referred to as a set. All character combinations can also be used as components in a set.

- You specify a range of characters in a set by separating the end characters with a dash. For example: [1-10]
- The complement of a set is represented by including the ^ character at the beginning of the set. For example: [^1-10]

Pattern Item

A pattern item can be represented by:

- A single character class, which matches any single character in the class.
- A single character class followed by *, which matches zero or more repetitions of characters in the class. These repetition items always match the longest possible sequence.
- A single character class followed by -, which matches zero or more repetitions of characters in the class. These repetition items always match the shortest possible sequence.
- A single character class followed by +, which matches 1 or more repetitions of characters in the class. These repetition items always match the longest possible sequence.
- A single character class followed by ?, which matches zero or 1 occurrence of a character in the class.

Pattern

A pattern is a sequence of pattern items.

- ^ at the beginning of a pattern anchors the match at the beginning of the string.
- $ at the end of a pattern anchors the match at the end of the string.
- A pattern cannot contain embedded zeros; use %z instead.

Captures

A pattern can contain sub-patterns enclosed in parentheses, which describe “captures.” When a match succeeds, the substrings of the analyzed string that match the captures are stored (captured) for future use.

Arithmetic Operators

Everybody is already familiar with the arithmetic operators addition (+), subtraction (-), unary negation (-), multiplication (*), division (/), exponentiation (^), and modulo (%). You use these operators as you would in a programmable calculator or any mathematical expression. In QlikView Expressor Datascript, these operators may be used with string values that can be converted into numbers.

Relational Operators

The relational operators (==, ~=, <, >, <=, >=) also have the same meanings as in other contexts. These operators always return true or false. The equality operator will always return false if the data types of its operands differ. For example, 1 == "1" returns false.

Logical Operators

The logical operators – conjunction (and) and disjunction (or) – do not function exactly as they would in other programming languages.

In QlikView Expressor Datascript, the logical operators do not necessarily return Boolean values, but rather the value of one of their inputs.

- The conjunction operator returns its first input if that input value is, or evaluates to, false or nil, otherwise it returns the value of its second input.
- The disjunction operator returns the value of its first input if that input value is not false or nil, otherwise it returns the value of its second input.
- A Boolean value is returned only if the appropriate input is itself a Boolean value.
- Only the values false and nil are interpreted as false.
- The values zero, one, and the empty string are all interpreted as non-nil, and therefore true, values.
- The conjunction operator has higher precedence than the disjunction operator and both of these operators have lower precedence than the relational and mathematical operators.

Since logical expressions can return values, they may be used in situations in which you would not normally think to use a logical expression. The most unusual example is to use these operators to implement the equivalent of an `if..then..else` block as in the following statement.

```
input_1 and input_2 or input_3
```

Think of input_1 as the conditional statement tested in the `if` clause. What you enter as input_1 can simply be a reference to a record attribute or a typical conditional statement that includes relational operators. Input_2 represents the value returned from the `then` clause. And input_3 is the value returned from the `else` clause. You can't use this syntax when the `then` and `else` clauses contain multiple statements, but it does have some very interesting uses.

For example, what if you want to supply a default value if an attribute is nil? The following statement will do the trick.

```
field_name and field_name or default_value
```

If field_name contains a non-nil value, this statement will return this value, otherwise it will return default_value.

Note that you can also nest the conditional statements. The following statement will return Democrat if the attribute party contains the value Democratic, will return Republican if the attribute party contains the value Republican, and will return Neither a Democrat nor Republican if the attribute party contains any other value. Be careful to use parentheses to control the precedence in an expression.

```
party=="Democratic" and "Democrat" or
        (party=="Republican" and "Republican"
         or "Neither Democrat nor Republican")
```

The relational operator == is properly interpreted as it has higher precedence than either logical operator but the nested statement must be enclosed within parentheses to insure that it is evaluated as a whole before it is used as the second input to the or operator in the outer statement.

The negation (not) operator is more typical and always returns true or false.