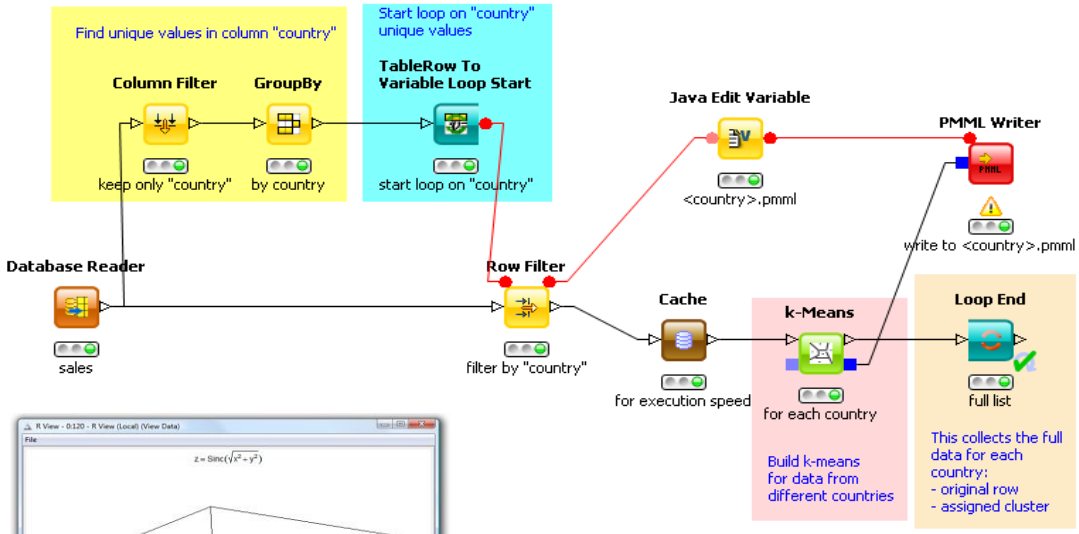
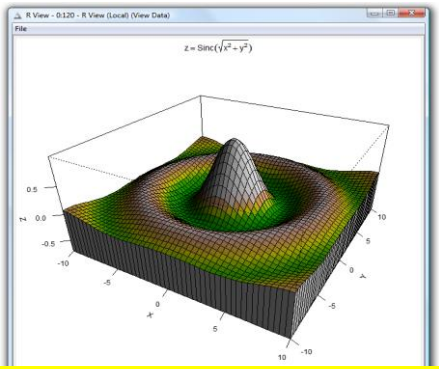
Rosaria Silipo, Michael P. Mazanetz

The KNIME Cookbook

Recipes for the Advanced User





Exercise 3

product	Sum(quantity)	Sum(amount)	
prod_1	74	2590	
prod_2	130	5200	
prod_3	106	8480	
prod_4	1	3	

This book has been updated for KNIME 2.10.

For information regarding permissions and sales, write to:

KNIME Press
Technoparkstr. 1
8005 Zurich
Switzerland

knimepress@knime.com

ISBN: 978-3-9523926-0-7

Table of Contents

Acknowledgements

We would like to thank a number of people for their help and encouragement in writing this book.

In particular, we would like to thank Thomas Gabriel for the advice on how to discover the many possibilities of integrating R into KNIME, Bernd Wiswedel for answering our endless questions about calling external web services from inside a workflow, and Iris Adae for explaining the most advanced features of some of the Time Series nodes.

Special thanks go to Peter Ohl for reviewing the book contents and making sure that they comply with KNIME intended usage and to Heather Fyson for reviewing the book's English written style.

Finally, we would like to thank the whole KNIME Team, and especially Dominik Morent, for their support in publishing and advertising this book.

Chapter 1. Introduction

1.1. Purpose and Structure of this Book

KNIME is a powerful tool for data analytics and data visualization. It provides a complete environment for data analysis which is fairly simple and intuitive to use. This, coupled with the fact that KNIME is open source, has led hundreds of thousands of professionals to use KNIME. In addition, other software vendors develop KNIME extensions in order to integrate their tools into KNIME. KNIME nodes are now available that reach beyond customer relationship management and business intelligence, extending into the field of finance, the life sciences, biotechnology, pharmaceutical and chemical industries. Thus, the archetypal KNIME user is no longer necessarily a data mining expert, although his/her goal is still the same: to understand data and to extract useful information.

This book was written with the intention of building upon the reader's first experience with KNIME. It expands on the topics that were covered in the first KNIME user guide ("The KNIME Beginner´s Luck" [1]) and introduces more advanced functionalities. In the first KNIME user guide [1], we described the basic principles of KNIME and showed how to use it. We demonstrated how to build a basic workflow to model, visualize, and manipulate data, and how to build reports. Here, we complete these descriptions by introducing the reader to more advanced concepts. A summary of the chapters provides you with a short overview of the contents to follow.

Chapter 2 describes the nodes needed to read and write data from and to a database. Reading and writing from and to a database are the basic operations necessary for any, even very simple, data warehousing strategy.

Chapter 3 introduces the DateTime object and the nodes to turn a String column into a DateTime column, to format it, to extract a time difference and so on. The DateTime object provides the basis for working with time series.

A very important concept for the KNIME workflows is the concept of "workflow variables". Workflow variables enable external parameters to be introduced into a workflow. Chapter 4 describes what a workflow variable is, how to create it, and how to edit it inside the workflow, if needed.

KNIME is a new software tool, very easy to use and already empowered with a lot of useful functionalities. One of the strongest points of KNIME, however, is the openness of the platform to other data analysis software. R, for example, is one of the richest and oldest open source data analytics software available and is equipped with a wealth of graphical libraries and pre-programmed statistical functions. Where KNIME might come short then, the many pre-implemented functions of the R libraries can be conveniently taken advantage of. A few KNIME nodes, indeed, allow for the integration of R scripts into the KNIME workflow execution. So, if you are or have been an R expert for long time and have a number of R scripts your fingertips and

ready to use, you can easily recycle them in your new KNIME workflows. Chapter 5 illustrates how to include and run R scripts in KNIME workflows. It describes how to install the R package, how to connect KNIME and R, and which nodes and R code can be combined together to produce plots, data models, or just simple data manipulations.

As of KNIME 2.4, it is also possible to call external web services and to collect the results inside a KNIME workflow. Chapter 6 describes the node that can connect to and run external web services.

Most data operations in KNIME are executed on a data matrix. This means that an operation is executed on all data rows, one after the other. This is a big advantage in terms of speed and programming compactness. However, from time to time, a workflow also needs to run its data through a real loop. Chapter 7 introduces a few nodes that implement loops: from a simple "for" cycle to a more complex loop on a list of values.

Chapter 8 illustrates the use of logical switches to change the workflow path upon compliance with some predefined condition.

Chapter 9 is an extension of chapter 6 in "The KNIME Beginner´s Luck" [1]: it describes a number of advanced features of the KNIME reporting tool. First of all, it explains how to introduce parameters into a report and how workflow variables and report parameters are connected. Later on in the chapter, a few more functions are discussed which can be used to create a more dynamic report.

Chapter 10 concludes this book and gives a few suggestions on how to manage memory in KNIME and how to run workflows in batch mode. These tips are not necessary in order to run workflows, but they can come in handy if a workflow has to be run multiple times or if the size of the workflow is particularly large.

In this introductory chapter, we list the data and the example workflows that have been built for this book and note the additional software required to run some of these example workflows.

1.2. Data and Workflows

In the course of this book we put together a few workflows to show how KNIME works. In each chapter we build one or more workflows and we expect the reader to build a few more in the exercises. The data and workflows used and implemented in this book are available in the "Download Zone". You should receive a link to the "Download Zone" together with this book. In the "Download Zone" you will find a folder for each chapter, containing the chapter's example workflows, and a subfolder called "Exercises", containing the solutions to the exercises in the corresponding chapter. You will also find a folder named "data", which contains the data used to run the workflows.

Structure of the "Download Zone"

Chapter 2

- Database_Operations.zip
- Exercises
  - Exercise1.zip
  - Exercise2.zip
  - Exercise3.zip

Chapter 3

- DateTime_Manipulation.zip
- DateTime_Manipulation_2.zip
- DateTime_Manipulation_3.zip
- Time_series.zip
- Exercises
  - Exercise1.zip
  - Exercise2.zip

Chapter 4

- Workflow_Vars.zip
- Exercises
  - Exercise1.zip
  - Exercise2.zip
  - Exercise3.zip
  - Exercise4.zip

Chapter 5

- R Snippet Example.zip
- Plotting Example 1.zip
- Plotting Example 2.zip
- Plotting Example 3.zip
- R Model Example.zip
- R PMML Node.zip
- Exercises
  - Exercise1.zip
  - Exercise2.zip
  - Exercise3.zip

Chapter 6

- WebServiceNodes.zip
- RESTServiceExample.zip
- Exercises
  - Exercise1.zip

Chapter 10

- Batch Mode.zip
- Batch Mode Databases.zip
- Simple Batch Example.zip
- Exercises
  - Create Database.zip
  - Exercise1.zip

Chapter 7

- Chunk Loop.zip /Chunk Loop 2 Ports.zip
- Counting Loop 1.zip /Counting Loop 2.zip
- Loop on List of Columns.zip
- Loop on List of Values.zip
- Loop with final Condition.zip
- Exercises
  - Exercise1.zip
  - Exercise2.zip
  - Exercise3.zip
  - Exercise4.zip

Chapter 8

- Automated IF Switch.zip
- CASE Switch.zip
- Empty table Replacer.zip
- Java IF & Tables.zip
- Manual IF Switch.zip
- Exercises
  - Exercise1.zip
  - Exercise2.zip

Chapter 9

- New Projects.zip
- Traffic lights.zip
- Exercises
  - Exercise1.zip
  - Exercise2.zip
  - Exercise3.zip

Data

- sales.csv, wrong_sales_file.txt
- sales/sales_*.csv
- cars-85.csv, dates.txt
- slump_test.csv
- Projects.txt, Totals Projects.csv
- images/*.png
- ZIP_CODES.zip, ChEBI_original.csv
- latitude-longitude.csv
- website.txt

This book is not meant as an exhaustive reference for KNIME, although many useful workflows and aspects of KNIME are demonstrated through worked examples. This text is intended to give you the confidence to use the advanced functions in KNIME to manage and mine your own data.

The data files used for the exercises and the example workflows were either generated by the authors or downloaded from the UCI Machine Learning Repository [2], a public data repository (http://archive.ics.uci.edu/ml/datasets). For the data sets belonging to the UCI Repository, the full link is provided below.

Data sets from the UCI Machine Learning Repository [2]:

- Automobile: http://archive.ics.uci.edu/ml/datasets/Automobile
- Slump_test: http://archive.ics.uci.edu/ml/datasets/Concrete+Slump+Test

1.3. Additional Software used in this Book

Two additional applications are required to run some of the examples used throughout this book: a database (we used PostGreSQL) and the R Project software. These tools extend the capabilities of KNIME and are freely available for download.

1.3.1. Database

In Chapter 2 of this book, we illustrate the KNIME approach to database operations. As the chapter progresses, we build an entire workflow to show the usage of database nodes and their potential in a practical example.

For the example to work, though, you have to install one of the many available database software tools. We chose to install the PostGreSQL database software. This choice was not motivated by KNIME considerations, since the KNIME database nodes work equally well for all other database tools, such as MySQL, MS SQL, Oracle, just to cite three among the most commonly used databases. This choice was largely motivated by the fact that PostGreSQL is open source and can be installed by the reader at no additional cost.

More information about PostGreSQL can be found at the database web site http://www.postgresql.org/ . The database software can be downloaded from http://www.postgresql.org/download/. From the binary packages, download the one-click installer for your machine and follow the installation instructions.

After installation of the database tool, start the administration console. If you have installed PostGreSQL, the administration console is called “pgAdminIII” (for Windows users, “All Programs” -> “PostgreSQL x.y” -> “pgAdminIII”).

From the administration console create a:

- User with username “knime_user” and password “knime_user”
- Database named “Book2” accessible by user “knime_user”
- Table named “sales” by importing the “sales.csv” file available in the Download Zone of the book

Note that the “date” field in the newly created “sales” table should have type Date.

If you do not know your way around a database, you can always use the “Database Writer” node to import the “sales.csv” file into the database. We have encountered the “Database Writer” node already in the “KNIME Beginner’s Luck” book [1]. However, for the readers who are new to this node, it is described in chapter 2.

You also need the JDBC database driver for your database to communicate with the KNIME database nodes. If you are using PostGreSQL, its database driver is available for download from http://jdbc.postgresql.org/download.html.

1.3.2. R-Project

R is a free software environment for data manipulation, statistical computing, and graphical visualization.

In Chapter 5 we explore the R KNIME extensions. These extensions make it possible to open R views, build models within R, import data from an R workspace, and run snippets of R code. The KNIME R extension, named “KNIME Interactive R Statistics Integration”, contains all nodes to run R from inside KNIME, but requires a pre-existing installation of the R-project (R binaries) on the local machine or access to an R Server component “Rserve”.

For Windows only, an additional extension, named “KNIME R Statistics Integration (Windows Binaries)”, installs the R binaries under the KNIME installation folder. However, for an R custom installation, you need to install R directly from the R-project web site (http://www.r-project.org/). For Linux and Mac you always need to install R beforehand from the R-project web site (http://www.r-project.org/).

Install R and R nodes

To install the R extension in KNIME:

- Select “Help” –> “Install New Software”
- In the “Available Software” window, select the KNIME update site

Alternatively

- Select “File” -> “Install KNIME Extensions …”

Next:

- Open the “KNIME & Extensions” group
- Select “KNIME Interactive R Statistics Integration”
- Click “Next” and follow installation instructions

This installs the “R” category in the Node Repository containing all nodes to run R from KNIME. R nodes refer to a pre-existing R installation, as defined in the “Preferences” page. To set the path of the pre-existing R installation:

- Select “File” -> “Preferences”
- In the “Preferences” page, select “R”
- Fill option “Path to R Home” with the path of your R installation

The R package can be installed directly from the “Comprehensive R Archive Network (CRAN)” homepage at http://cran.r-project.org. For Windows only, you can avoid all that and install R through a KNIME extension. In the KNIME Extensions window:

- Select “KNIME R Statistics Integration (Windows Binaries)”
- Click “Next” and follow installation instructions

If the R software was installed from the R-project or CRAN web sites, rJava value needs a new setting by running the following command line in the R interface.

```
install.packages("rJava", dependencies = TRUE)
```

Install R Packages Required for the R Workflow Examples

In order to run the example workflows provided in chapter 5 of this book, you need following R packages: “QSARdata”, “corrgram”, “lattice”, “car”, “alr3”, “ggplot2”, “circular”, “reshape”, and “drc”.

If you download the R software from CRAN, the general procedure involves navigating to the “Packages” page by clicking “Packages” in the “Software” section of the R web site (http://cran.r-project.org). This page contains details on how to install new packages and further information and documentation on the available R packages.

However, downloading and installing an R package is best done using the install.packages function in the R interface.

Open your R application and execute the following command (you might be asked to select a mirror for downloading packages):

```
install.packages(c('QSARdata'))
install.packages(c('corrgram', 'lattice', 'car', 'alr3', 'ggplot2', 'circular', 'reshape', 'drc'))
```

Once the command has run, close the R editor.

If you do not know where your R binaries are located, check in the main menu: "File" -> "Preferences" -> "KNIME" -> "R" and then the textbox "Path to R Executables" in the panel on the right. This textbox contains the default R executables path your R nodes are referring to.

Access and Install an R-Server (if needed)

Download and install the latest version of Rserve from RForge, http://www.rforge.net/Rserve.

Follow the instructions in the documentation for your operating system. Once Rserve has been downloaded and installed, the R-Server can be started (see an example in http://www.rforge.net/Rserve/example.html). R Server settings can be set in the "R-scripting" preference page under "File" -> "Preferences" -> "KNIME" -> "R-scripting".

1.3.3. External Software Summary

The table below is a summary of the installations and configurations of external software tools used in this book.

Software type	Name	Download from:
Database	PostGreSQL (or other database software) with: ○ user: "knime_user", "knime_user" ○ Database: "Book2" ○ Table: "sales" imported from file "sales.csv" from the Download Zone	PostGreSQL: http://www.postgresql.org/download/
R	R (local) / Rserve (server)	R-project: http://cran.r-project.org/ RForge: http://www.rforge.net/Rserve/

Chapter 2. Database Operations

2.1. Database Nodes: Multiple SQL Queries vs. All-in-One Approach

In this chapter we begin with the exploration of KNIME´s advanced features by having a look into the database operations. A first glance was already provided in the first book of this series, "KNIME Beginner's Luck" [1, http://www.knime.org/knimepress/beginners-luck]. Here, though, we investigate the multiple possibilities for connecting, reading, writing, and selecting data from and to a database in much greater detail.

First of all, we want to create the workflow group "Chapter2", that will contain all workflows related to this chapter. Then, in this new workflow group, let's create an empty workflow with name "Database_Operations". The goal of this workflow is to show how to connect to a database, retrieve data from a database, and write data into a database.

In the newly created workflow named "Database_Operations", we would like to read the data from the "sales" table in the "Book2" database. The "sales" data set contains the sale records for a fictitious company. The records consist of:

- The name of the product (product)
- The sale country (country)
- The sale date (date)
- The quantity of products sold (quantity)
- The amount of money (amount) connected to the sale
- A flag to indicate whether the purchase was paid by cash or credit card (card)

The sales for four products are recorded: "prod_1", "prod_2", "prod_3", and "prod_4". However, "prod_4" has been discontinued and is not significant for the upcoming analysis. In addition, the last field called "card" contains only sparse values and we would like to exclude it from the final data set.

In general, there are two ways to read data from a database into KNIME.

1. **Using Multiple Nodes**. First we connect to the database; next we build the appropriate SELECT query; and finally we read the data from the database into a KNIME data table. There are many ways to build a SELECT query with KNIME, each one fitting a given level of SQL expertise.

2. **Using the all-in-one node approach**. We connect to the database, build the query, and read the data into KNIME, all in the same node.

The first approach allows us to extract a more targeted set of data, by using a cascade of SQL queries. A more targeted data set usually means a smaller data set and consequently less memory usage within KNIME. The second approach is more compact, but requires some SQL knowledge and does not allow for multiple SQL queries. Whichever data retrieval approach is chosen, all the required nodes are located in the "Database" category.

2.2. Connect to a Database

We start with the first approach. In order to connect to the database, we use a database connector node. There are two types of connector nodes: the "Database Table Connector" node in "Database"/"Read/Write" category and "Database Connector" node in "Database"/"Connector" category. Both nodes simply connect to a specific database and leaves the connection open to retrieve data. While the "Database Table Connector" node allows for some pre-processing on the data by means of a customized SQL statement in its configuration window, the "Database Connector" node just establishes the connection to the database relying on subsequent nodes for a targeted data extraction with a customized SQL statement.

Note. The "Database Table Connector" node and the "Database Connector" node show two different output ports: brown square one and pale red square the other. Different output ports identify different compatible following nodes, with the same type of input ports.

If you are not an SQL expert or if you do not know the database structure, some help can come from the "Database Browser" panel available in some database nodes – like the "Database Table Connector" node – on the left side of the configuration window. After clicking the "Fetch Metadata" button, the "Database Browser" panel shows the structure of the selected database and its tables. Double-clicking a table in the "Database Browser" panel automatically inserts the table name in the SQL Editor on the right. Double-clicking a table field in the "Database Browser" panel automatically inserts the table field name with the right syntax in the SQL Editor.

If a workflow collects data from different database sources, it might be necessary to use different credentials (i.e. username and password). KNIME stores each credentials set (username and password) in a credential identifier. For each workflow, multiple credential identifiers can be defined in the "Workflow Credentials" option of the workflow context menu. For security reasons, credential identifiers are neither visible nor accessible by other workflows. The necessary credential identifiers have to be created for each workflow separately. In addition, passwords are not saved in the KNIME folder. At every new start of KNIME, you are prompted to re-enter the passwords.

Note. The alternative to the Workflow Credentials is to enter the username and password directly in the database node dialog. However, this authentication mode does not encrypt passwords and it is therefore not as secure as the Workflow Credentials mode. For this reason, the username/password authentication mode has been deprecated since the early versions of KNIME and, even if still present in the node dialog for backward compatibility reasons, it is not recommended.

Database Table Connector

The "Database Table Connector" node requires the following settings:

- The database driver. Most commonly used database drivers have been preloaded in KNIME and are available from the "Database Driver" menu. If your database driver is not in the list, check below how to upload a new database driver into KNIME.

- The database URL. The database URL has to be built as follows:
  jdbc: <DB-software-name>://<server-name>:<port nr>/<database-name>

- The Time Zone for date and time values.

- The credentials to access the database:
  - One of the credential identifiers defined in the "Workflow Credentials"

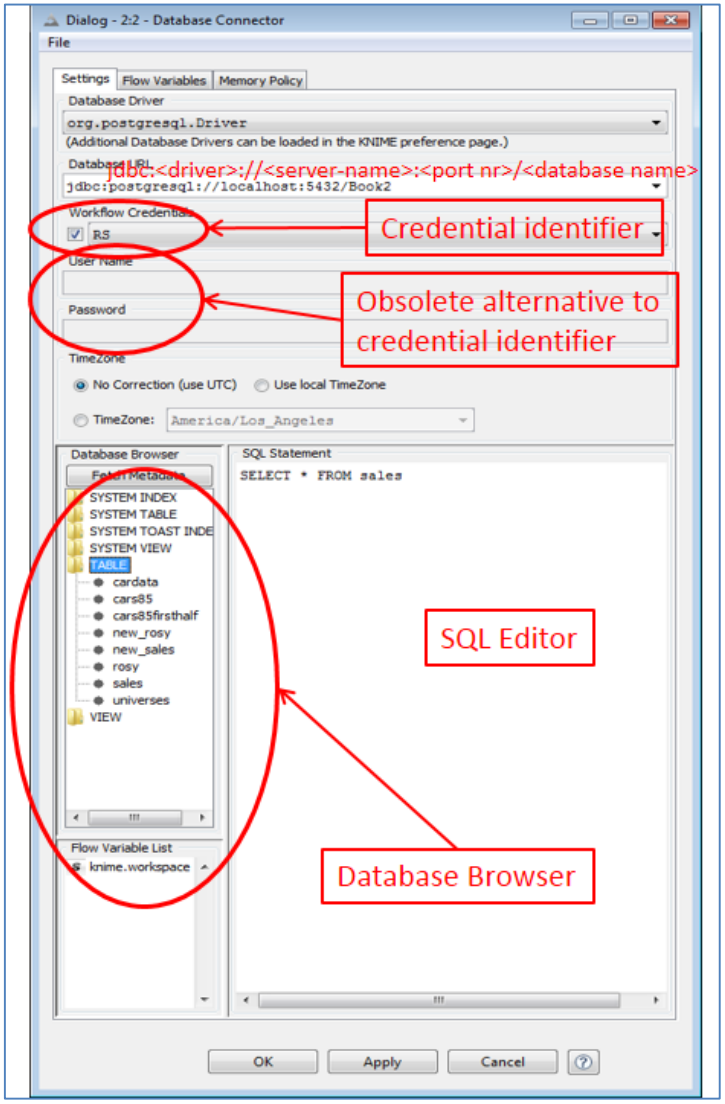
  OR alternatively (but deprecated)
  - username and password

- The SQL SELECT statement to retrieve the data from the database

The SQL statement can be built with the help of the "Database Browser" on the left. The "Database Browser" panel shows the database structure and, if opened, the tables structure as well. Double-clicking a table or a table field in the "Database Browser" panel, inserts the object name with the right syntax in the SQL editor.

The port number in the database URL should be provided by the database vendor.

2.1. Configuration window of the "Database Table Connector" node



Workflow Credentials

In the “KNIME Explorer” panel:

- Right-click the desired workflow
- Select “Workflow Credentials”

In the “Workflow Credentials …” window:

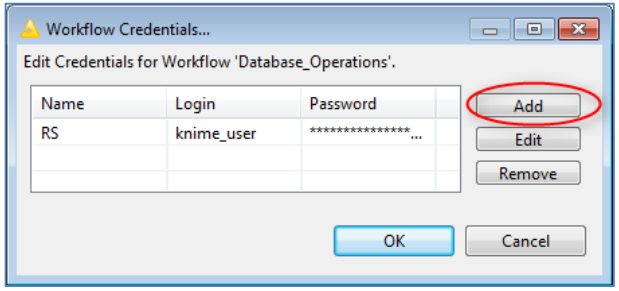
- Click the “Add” button

In the “Add/Edit Credentials” window:

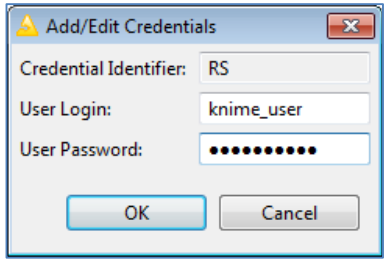
- Define an ID to identify the pair (username, password).
- Enter the username in the “User Login” box
- Enter the password in the “User Password” box
- Click “OK”

Back in the “Workflow Credentials …” window, click “OK”.

2.2. Define multiple database credentials



2.3. „Add/Edit Credentials“ window



Note. The “Workflow Credentials” option in the context menu of each workflow is only active if the workflow is open in the workflow editor.

If you decide to bypass the workflow identifier and go with explicit username and password instead, remember to provide a master key value for password encryption and decryption, to enhance security when accessing the database. The master key can be entered in the “Preferences” page under “File” -> “Preferences” -> “KNIME” -> “Master Key”.

Using the database PostGreSQL described in section 1.3, we configured the workflow “Database_Operations” to access tables in “Book2” database using the “knime_user” user. This translates into one credential identifier with username = “knime_user” and password = “knime_user” in the “Workflow Credentials” option of the given workflow. We named the credential identifier “RS” (Fig. 2.3). We then introduced a “Database Table Connector” node and we configured it using the recently created credential identifier “RS”.

Another way to access a database is to use a connector node. A connector node is a simplified version of the “Database Table Connector” node, since it just establishes the connection to the database without allowing for any SQL statement. Many connector nodes are available in KNIME, from the

general “Database Connector” node to more specific instances of the same node such as “Hive Connector”, “MySQL Connector”, and similar nodes built to address a specific database type.

Database Connector and other dedicated Connector Nodes

The “Database Connector” node just establishes the connection to a database and requires the following values:

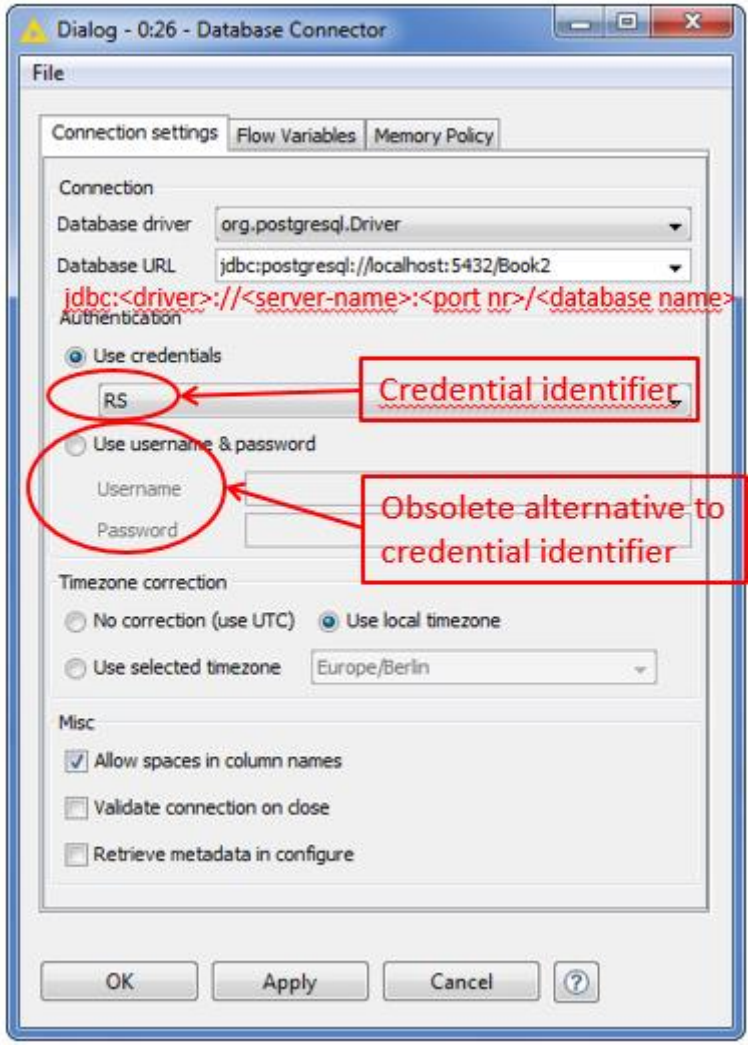
- The database driver. Most commonly used database drivers have been preloaded in KNIME and are available from the “Database Driver” menu. If your database driver is not in the list, check below how to upload a new database driver into KNIME.
- The database URL. The database URL has to be built as follows:
  jdbc: <DB-software-name>://<server-name>:<port nr>/<database-name>

  The port number in the database URL has to be provided by the database vendor.
- The Time Zone for date and time values.
- The credentials to access the database:
  - One of the credential identifiers defined in the “Workflow Credentials”

  OR alternatively (but deprecated)
  - username and password
- A few more useful miscellaneous utility settings.

This node outputs a pure database connection port.

2.4. Configuration window of the “Database Connector” node



2.5. Upload a new JBDC Driver via the Preferences Window

In case the JDBC database driver for your database is not present in the list of pre-loaded JDBC drivers in the “Database Driver” field of any database node dialog, you need to upload it yourself into KNIME via the “Preferences” window.

Database Driver

In the top menu:

- Select “File” -> “Preferences”

In the “Preferences” window

In the left frame:

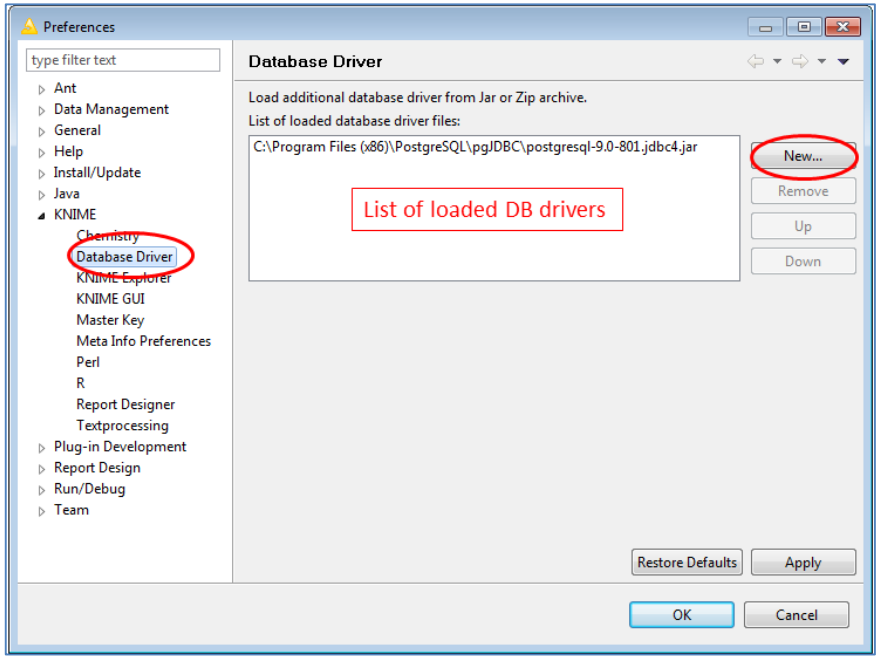
- Open “KNIME”
- Select “Database Driver”

In the right frame:

- Click button “New”
- Load the database driver file
- If the file is accepted, it appears in the list of loaded database drivers in any database node dialog.

- Click “OK”

2.5. Load a database driver into KNIME “Preferences”



Note. The database driver file is usually provided with the database software.

After loading the database driver in the “Preferences” window, the database driver becomes a general KNIME feature and is available for all workflows in all workspaces.

2.6. Implement a SELECT Query

In the "Database Table Connector" node in section 2.2 we used a very simple SQL query: "SELECT * from sales". This query simply downloads the whole data set stored in table "sales". However, what we really want to do is to get rid of the rows that pertain to product "prod_4" and to get rid of the "card" column, before we pull in the data set from the database.

One option is to customize the SQL query in the "Database Table Connector" node, something like:

```
SELECT product, country, date, quantity, amount from sales WHERE product != 'prod_4'
```

If you are not SQL savvy, you can also append two database nodes to customize the SELECT statement, before retrieving the data: the "Database Row Filter" and the "Database Column Filter" nodes.

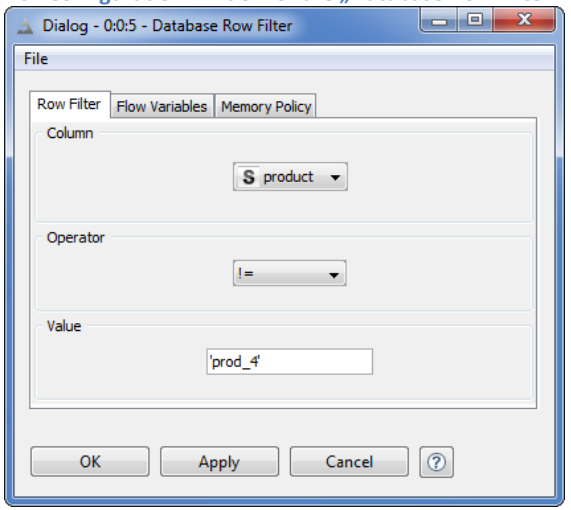
Database Row Filter

The "Database Row Filter" node customizes an SQL query, according to a filtering criterion, to keep only some rows of the data.

In order to define the filtering criterion, the node requires the following settings:

- The table field ("Column") on which to operate
- The operator (=, >, <, !=, etc…)
- The matching pattern ("Value")

2.6. Configuration window of the „Database Row Filter" node



Note. In the configuration window of the "Database Row Filter" node, for most database software tools, the matching pattern value has to be included in single quotation marks ('…') for string matching. Conversely, no software tools require extra delimitations for number matching.

We connected a “Database Row Filter” node to the “Database Connector” node. The “Database Row Filter” node was then set to filter in all rows where the column “product” was different (operator “!=”) from the value “prod_4”, i.e. it was supposed to filter out all rows with value “prod_4” in the column “product”.

In the “Database_Operations” workflow, a “Database Column Filter” node was also introduced to follow the “Database Row Filter” node and to remove column “card” from the data set.

Database Column Filter

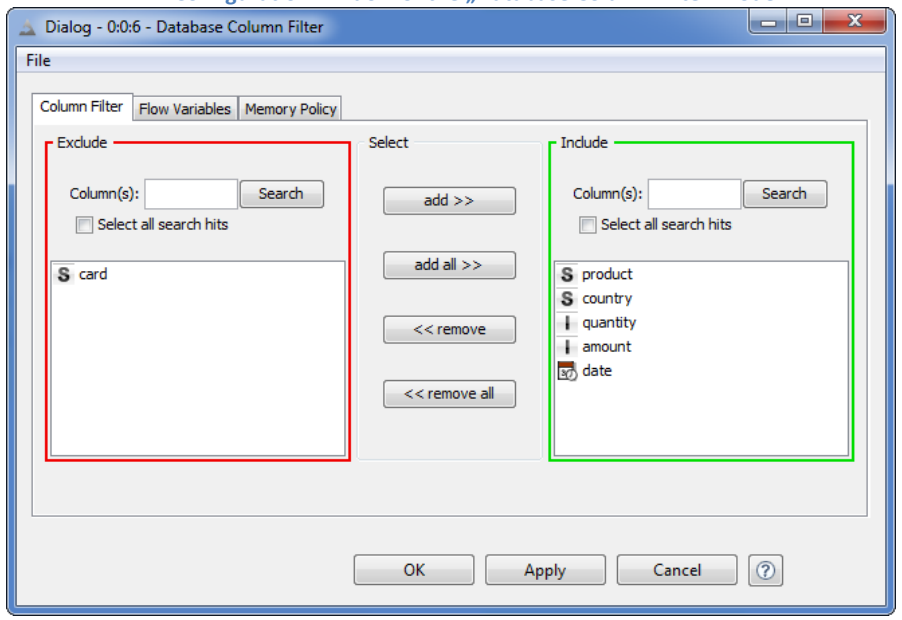
The “Database Column Filter” node customizes an SQL query to exclude or include some of the fields in the original table.

Its configuration window is designed like the configuration window of a “Column Filter” node. That is, it is based on an “Exclude/Include” framework.

- The columns to be kept are listed in the “Include” frame on the right
- The columns to be removed are listed in the “Exclude” frame on the left

To move single columns from the “Include” frame to the “Exclude” frame and vice versa, use the “add” and “remove” buttons. To move all columns to one frame or the other use the “add all” and “remove all” buttons.

2.7. Configuration window of the „Database Column Filter“ node



A “Search” box in each frame allows searching for specific columns, in the event that an excessive number of columns impedes the data column overview.

With an appropriate knowledge of the SQL syntax, it is possible to incorporate the SQL SELECT statement directly into the “Database Table Connector” node. However, in some cases, the goal might be to generate different data sets from the same database table using different SELECT queries. For

example, we might want to keep the “card” column in one data set and exclude it in another data set. If this is the case, the “Database Table Connector” node does not fit our purpose since it cannot execute two different SQL SELECT queries at the same time. For cases like this, we need to use multiple “Database Query” nodes in parallel.

Database Query

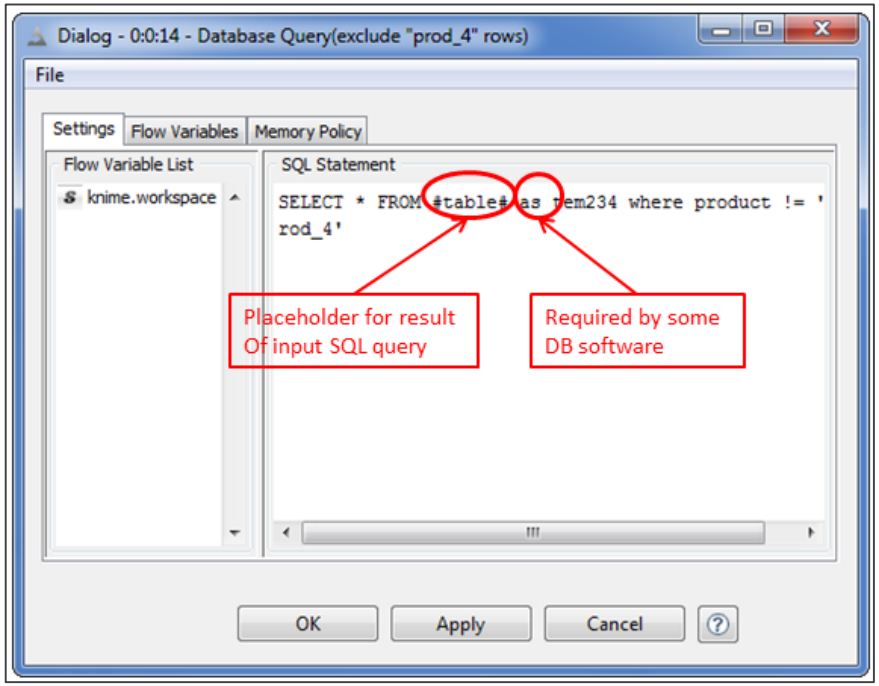
The “Database Query” node has the task of refining the SQL SELECT query at its input port.

The only setting required in its configuration window is the additional SQL SELECT query.

Notes.

- The notation #table# is a placeholder for the result table of the input SQL Query. **Do not remove it!**
- Some database software also requires the statement “`as <new-table-name>`” to work. This is the case, for example, of PostGreSQL.

2.8. Configuration window of the „Database Query“ node



Note. Up to KNIME 2.5 included, not all queries are supported in database nodes. For example, DELETE and INSERT are usually not supported. For these operations you need to use the “Database Delete” and “Database Update” nodes.

In the “Database_Operations” workflow, we introduced two “Database Query” nodes in parallel. One node had the following query, to filter out the “card” column:

```
SELECT product,country,quantity,date, amount FROM #table# as tem234
```

The other “Database Query” node implemented this query, which has to filter out all rows with “prod_4” as product:

```
SELECT * FROM #table# as tem234 where product != 'prod_4'
```

With these two parallel “Database Query” nodes, we get two different data sets from the same “Database Connector” node.

The “Database Row Filter” node, as the “Database Column Filter” node and the “Database Query” node, do not operate directly on the data, they simply customize the SQL query before the data is pulled into KNIME. In fact, all nodes seen so far do not have a data output port (white triangle), but instead a database output port (brown square). This is because they do not output a data table, but just a SELECT query.

Sometimes, for very large database tables, it can be useful to create a much targeted SQL query before pulling in the data. In fact, the download of very large database tables might consume all available memory and slow down the database node execution.

Similarly to the SQL query refining nodes and following a “Database Table Connector” node, there are nodes to implement an SQL query to follow a database connector node (red square port). The “Database SQL Executor”, for example, implements and executes an SQL query on the connected database. Its task is similar to the one of the “Database Query” node, besides the fact that the SQL query is physically executed during the node execution. While the “Database Query” produces an SQL statement that gets appended to previous SQL statements after the nod execution, the “Database SQL Executor” node creates the SQL statement and runs it against the selected database during the node execution.

The “Database Table Selector” node imports and refines previously defined SQL statements and exports them into a database connection port.

The “SQL Inject” node has a similar task to the one of the “Database Table Selector”. The only difference is in the way the new SQL query is defined: the “Database Table Selector” node builds the SQL query in an SQL editor in the configuration window, while the “SQL Inject” node takes the SQL query from an input flow variable of type String.

The inverse path of the “SQL Inject” node is implemented by the “SQL Extract” node. The “SQL Extract” node connects to a database connection and extract the SQL query running on it. The resulting SQL query is output as a flow variable and as a data table.

2.7. Read/Write Data resulting from a SELECT Query

So far we have connected to the database, built and in some case partially executed a SELECT query that fits our purposes. Now we need to download the data from the database table according to the SELECT query. That is, we need a node that reads the SELECT query on the input port and produces the corresponding data set on the output port. We need a “Database Connection Table Reader” node.

Database Connection Table Reader

The “Database Connection Table Reader” node runs the SQL SELECT query at its input port and produces the resulting data table at its output port.

The “Database Connection Table Reader” node does not need any configuration settings. Everything needed is all contained in the input SQL query.

2.8. The "Database Connection Table Reader" node

Note. The “Database Connection Table Reader” node has a brown square as input port and a white triangle as output port; i.e. it takes a database connection with an SQL query and produces a data table.

Sometimes it can be useful to save the data resulting from the SQL SELECT query into another database table. In order to execute the SELECT query and write the resulting data directly into a database table, without ever passing through a KNIME data table, we can use the “Database Connection Table Writer” node.

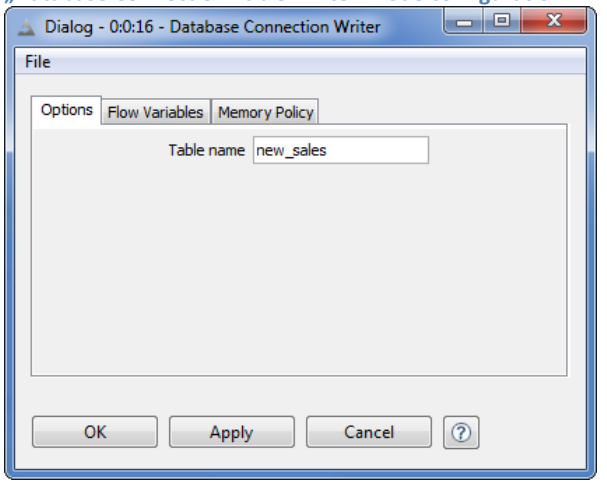
Database Connection Table Writer

The “Database Connection Table Writer” node reads and executes a SELECT query and writes the resulting data into a database table. At the input port (brown square) we find the database connection with the SELECT query. No output port is present, since the data gets written into a database table.

The only setting required in the configuration window is the name of the database table to which the data is written.

If the specified table already exists it will be dropped and recreated (i.e. overwritten).

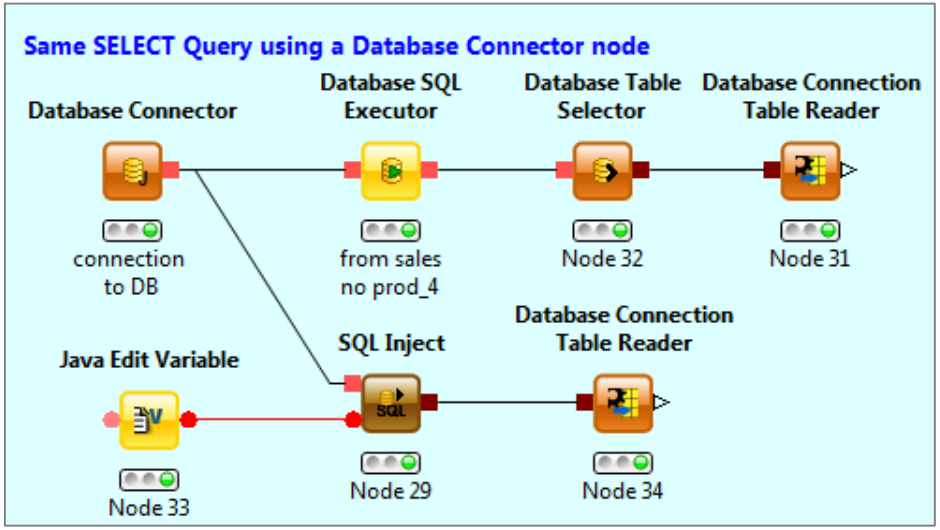
2.9. „Database Connection Table Writer“ node configuration window



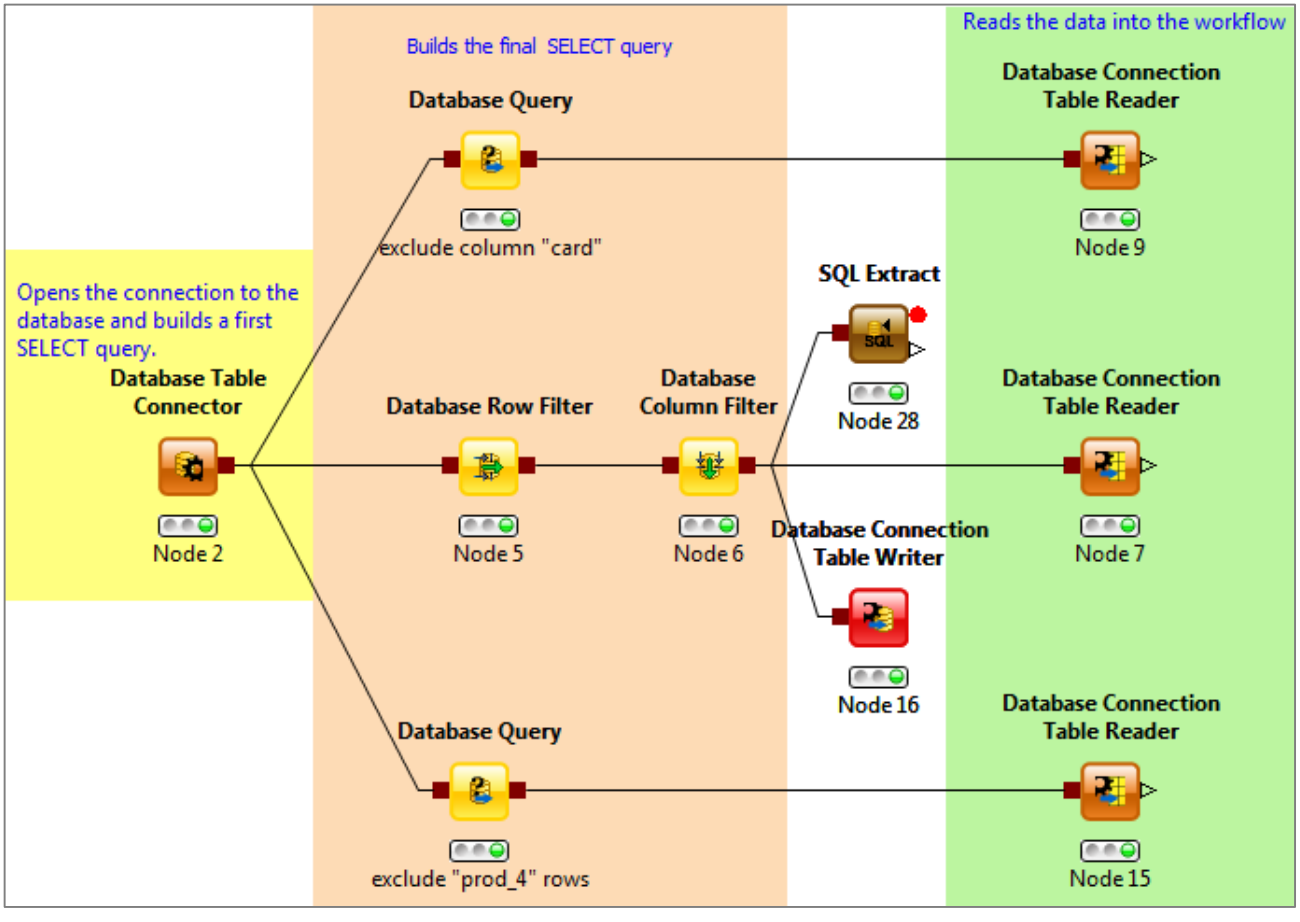
We introduced a few “Database Connection Table Reader” nodes to the “Database_Operations” workflow: one is connected to the “Database Query” node that removes the “card” column; one is connected to the “Database Query” node that excludes rows with “prod_4” as product; one is connected to the sequence with the “Database Row Filter” node and the “Database Column Filter” node; one is connected to a “Database Connector” – “Database SQL Executor” – “Database Table Selector” sequence of nodes; and the last one is exporting data from “Database Connector” – “SQL Inject” node sequence. In this way, we generated as many data sets as many “Database Connection Table Reader” nodes from the same database table named “sales”.

We introduced one “Database Connection Table Writer” node to write the data resulting from the sequence of the “Database Row Filter” and the “Database Column Filter” node into a new database table named “new_sales”. The final workflow, named “Database_Operations”, is shown in figures 2.10 and 2.11.

2.10. The “Database_Operations” workflow: part with “Database Connector” node



2.11. The “Database_Operations” workflow: part “Database Table Connector” node



2.8. All in one Node: Connection, Query, and Reading/Writing

We have seen in section 2.3 that, if we can write little SQL, we do not need the extra nodes to build the SELECT query. We can write the SELECT query directly into the “Database Table Connector” node and then read the data into the workflow by means of a “Database Connection Table Reader” node. Actually, we can reduce the number of database nodes even further by using a “Database Reader” node.

Database Reader

The “Database Reader” node performs all database operations by itself:

- Connects to the database
- Reads and executes the SELECT statement in its configuration window
- Pulls the resulting data from the database into a data table

The configuration window of the “Database Reader” node is similar to the one of the “Database Connector” node and requires the following values:

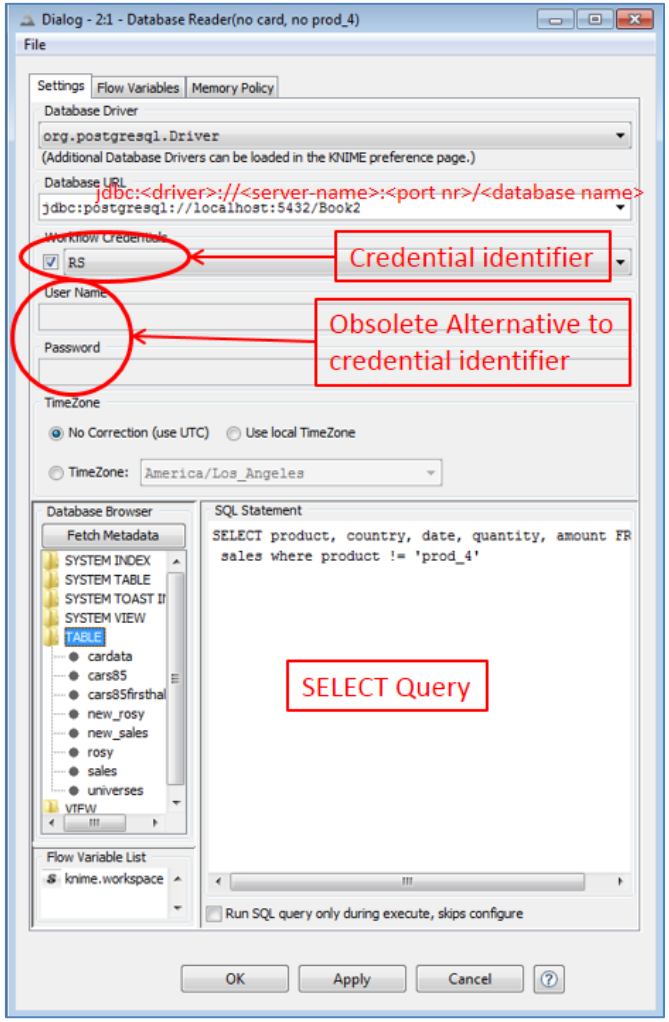
- The database driver. Most commonly used database drivers have been preloaded in KNIME and are available from the “Database Driver” menu. If your database driver is not in the list, check below how to upload a new database driver into KNIME.
- The Database URL. The database URL has to be built as follows:

jdbc: <DB-software-name>://<server-name>:<port nr>/<database-name>

- The Time Zone for date and time values.
- The credentials to access the database:
  - One of the credential identifiers defined in the “Workflow Credentials”

  OR alternatively:
  - Username and password
- The SQL SELECT statement to retrieve the data from the database table

2.12. The configuration window of the “Database Reader” node



With some database software it is possible to use multiple SQL statements in the SQL editor in the configuration window of the database nodes. In this case, the SQL statements have to be separated with a semicolon (“;”).

Note. The disadvantage of using a "Database Reader" node alone consists of the impossibility to generate more than one customized data table from the same database connection.

Similarly to the "Database Reader" node, KNIME offers a "Database Writer" node. The "Database Writer" node writes a data table from the input port into a database table. The "Database Writer" node has of course no output port, since it does not output any data.

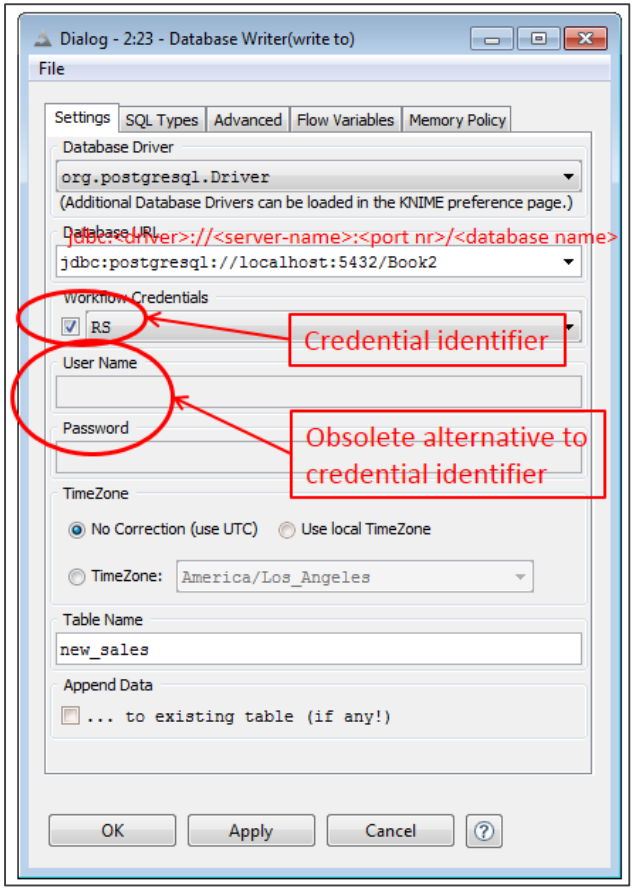
Database Writer: Settings

The "Database Writer" node takes a data table on the input port and writes it into a database table.

The configuration window requires the following values:

- The database driver. The database driver can be chosen from the list of all database drivers that have been loaded in the "Preferences" page.
- The Database URL. The database URL has to be built as follows:
  jdbc: <DB-software-name>://<server-name>:<port nr>/<database-name>
- The credentials to access the database:
  - One of the credential identifiers defined in the workflow credentials
  OR alternatively
  - Username and password
- The Time Zone for date and time values
- The name of the database table to write into
- The append or override flag in case the table already exists in the database

2.13. Configuration window of the „Database Writer" node: the "Settings" tab



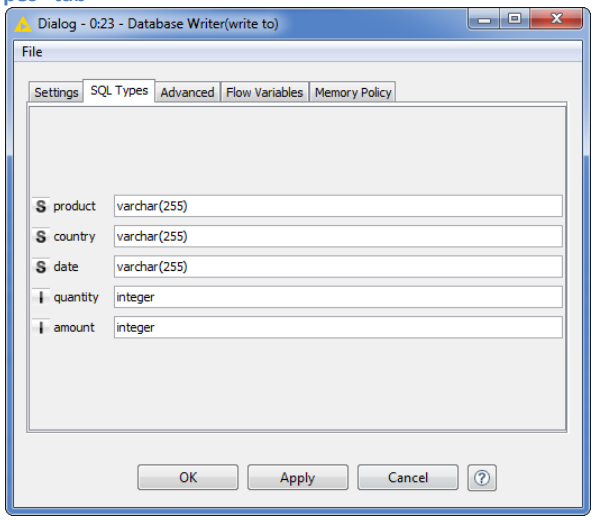
Database Writer: SQL Types

The “Database Writer” node has a second tab in the configuration window: the “SQL Types” tab.

This tab contains the types of the data columns to be used to write into the database.

These types can be changed by typing a new type in the textbox. Of course the new type has to be compatible with the actual data column type.

2.14. Configuration window of the „Database Writer“ node: the “SQL Types” tab



Note. Table names with spaces are not accepted. Table name “new_sales” works, while “new sales” would fail.

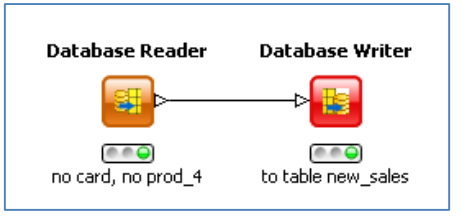
In the “Database_Operations” workflow, under the section that deals with the SQL queries, we introduced a “Database Reader” node using:

- The PostGreSQL jdbc driver
- The “Book2” database located on the same machine (“localhost”)
- The credential identifier “RS” from the “Workflow Credentials”
- The SELECT query excluding the “card” column and all rows with product “prod_4”

This one node produces one of the data tables that were generated by means of a “Database Connector” node, an “SQL Query” node, and a “Database Connection Reader” node all by itself.

In the next step we connected a “Database Writer” node to the “Database Reader” node to write the data table resulting from the “Database Reader” node into a new database table “new_sales”.

2.15. "Database Reader" node + "Database Writer" node.



Two more “all-in-one” nodes are available in the “Database” category: the “Database Update” and the “Database Delete” node. Both nodes connect to a database and perform a specific query (update or delete) on a specific subset of data as defined by the “WHERE” panel in the configuration window. The “WHERE” panel uses the values in the input data columns to match the values in the corresponding database table field, to form the WHERE condition of the SQL query.

2.9. Looping on Database Data

In the “Database Reader” node introduced in the previous section we used the same SQL query shown at the beginning of section 2.3, in order to get only the rows containing product “prod_1”, “prod_2”, and “prod_3” and to remove the “card” column:

```
SELECT product, country, date, quantity, amount FROM sales where product != 'prod_4'
```

Alternatively we could have used the following query:

```
SELECT product, country, date, quantity, amount from sales
WHERE product = 'prod_1' OR product = 'prod_2' OR product = 'prod_3'
```

This is a very commonly used type of SELECT query: a query looping over a number of distinct values. As there are only three values involved in the WHERE condition, this query is still manageable manually. Sometimes, though, the number of values involved in the WHERE condition can be much higher or even unknown.

For example, the condition values can be the unknown result of a previously executed SELECT query, like:

```
SELECT product, country, date, quantity, amount from sales WHERE product IN (SELECT col1 from <another-table>)
```

In some cases, as in the previous example, it can be convenient to use the values of another column as the matching patterns for the WHERE condition in the SELECT query. Using the SELECT query above as an example again, we could create a data table with the values “prod_1”, “prod_2” and “prod_3” in one of its columns and use this column’s values as the matching patterns for the WHERE condition in the SELECT query. To create a data table from inside a workflow we can use the “Table Creator” node.

Table Creator

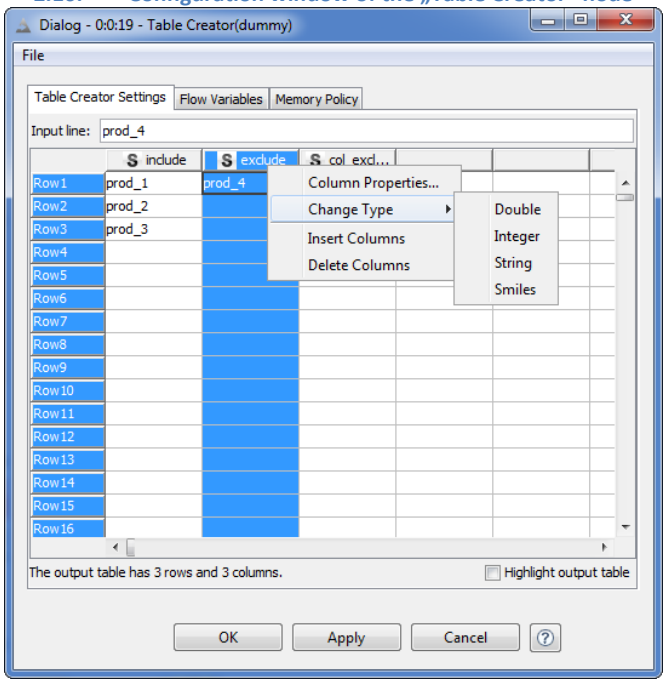
The “Table Creator” node was introduced with KNIME 2.3 and provides a small editor to manually generate data from inside a workflow.

The “Table Creator” node does not belong to the database nodes and it is actually located in the category `"IO" -> "Other"` in the “Node Repository” panel.

The configuration window of the “Table Creator” node contains the data editor with the following properties:

- The cell content is editable
- Right-clicking a column header edits the column’s properties (name, type) and allows to insert or delete columns
- Right-clicking a RowID edits the RowID’s properties and allows to insert or remove rows
- “Copy and paste” from Excel sheets is also enabled

2.16. Configuration window of the „Table Creator“ node



In the workflow “Database_Operations” we introduced a “Table Creator” node named “dummy”. The “Table Creator” node lists “prod_1”, “prod_2”, and “prod_3” in a column named “include”. The goal of this node was to provide a list of matching patterns for the WHERE condition in the SELECT query. In order to search for matching patterns among the values of a column in a data table, we use the “Database Looping” node.

Database Looping

The “Database Looping” node takes a data table at the input port (white triangle) and produces a data table on the output port. Like the “Database Reader” and the “Database Connector” nodes, the “Database Looping” node connects to the database, reads and executes the SELECT statement in its configuration window, and pulls the resulting data from the database into a data table. The difference with respect to the “Database Reader” and the “Database Connector” nodes lies in its SELECT query, which is:

```
SELECT * FROM <table-name> WHERE <column-name> IN ('#PLACE_HOLDER_DO_NOT_EDIT#')
```

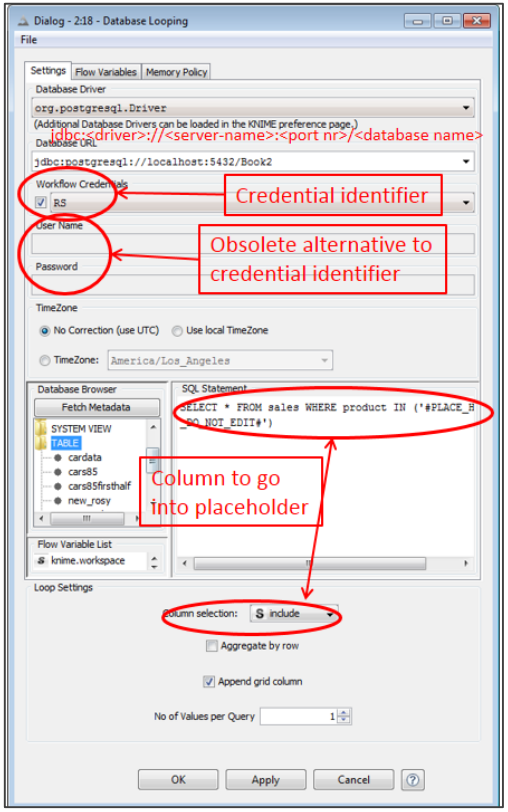
During node execution, `#PLACE_HOLDER_DO_NOT_EDIT#` is substituted with the values in one of the columns of the input data table.

The configuration window requires :

- The database driver, which can be chosen from the list of all database drivers that have been loaded in the “Preferences” page.
- The Database URL, which has to be built as follows:
  jdbc: <DB-software-name>://<server-name>:<port nr>/<DB-name>
- The credentials to access the database:
  - One of the credential identifiers

  OR alternatively
  - username and password to access the database
- The column from the input data table whose distinct values are to replace the “`#PLACE_HOLDER_DO_NOT_EDIT#`” in the SELECT query
- The option “aggregate by row” that aggregates the results for each distinct value of the “`#PLACE_HOLDER_DO_NOT_EDIT#`” into a comma separated list
- The option “append grid column” that appends a column with the values for “`#PLACE_HOLDER_DO_NOT_EDIT#`”
- The option “No values per query” that specifies the number of distinct values from the selected column in `#PLACE_HOLDER_DO_NOT_EDIT#`.

2.17. Configuration window of the „Database Looping“ node

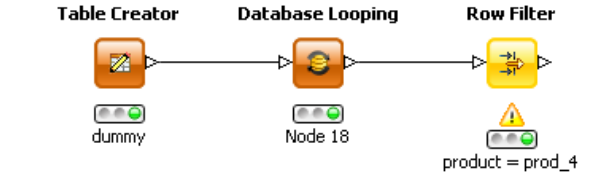


Note. `#PLACE_HOLDER_DO_NOT_EDIT#` cannot be moved around in the SELECT query. The SELECT query of the "Database Looping" node can only be modified in terms of the column selection and the name of the database table from which to extract the data.

The field "No. of Values per Query" limits the number of values per query. This means that each query runs replacing #PLACE_HOLDER# with only a number of values from the selected column as specified in the field "No. of Values per Query". This is useful in case the selected column contains a lot of different values and leads to a huge query statement.

The flag "Append grid column", when enabled, appends a column that includes the resulting values to the input data table.

2.18. Table Creator node + Database Looping node to keep all rows with products in ('prod_1', 'prod_2', 'prod_3')



The resulting sub-workflow, consisting of the "Table Creator" node and the "Database Looping" node, implemented in this section, was placed at the bottom of the "Database_Operations" workflow. We actually also added a "Row Filter" node, filtering in all rows with product "prod_4", to show that none of them has remained in the final data table, hence the warning triangle under this node.

2.10. Exercises

Exercise 1

Create an empty workflow, called "Exercise1", in an "Exercises" workflow group under the existing "Chapter2" workflow group.

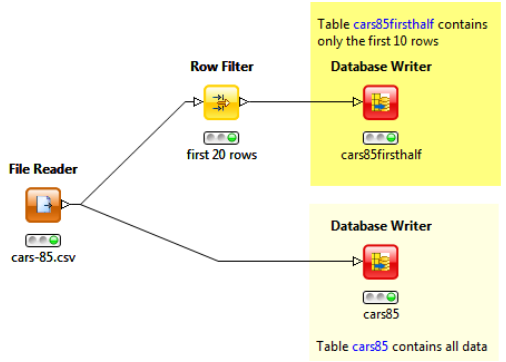
The "Exercise1" workflow should prepare the database tables for the next 2 exercises. It should therefore:

- Read the file "cars-85.csv" (from the "Download Zone");
- Write the data to a database table named "cars85" in the "Book2" database using the "Workflow Credentials" option;
- Write only the first 10 rows of the data into a table called "cars85firsthalf" under the same "Book2" database using the "username" and "password" option.

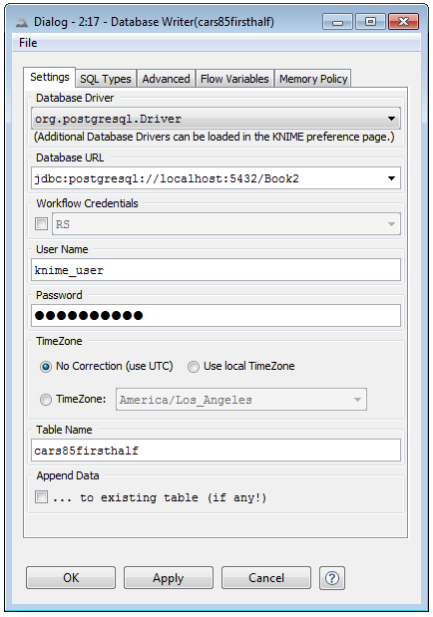
Solution to Exercise 1

The solution workflow to Exercise 1 is shown in figure 2.19. The configuration windows for the "Database Writer" node using the workflow credentials and using the username and password are shown in the following two figures.

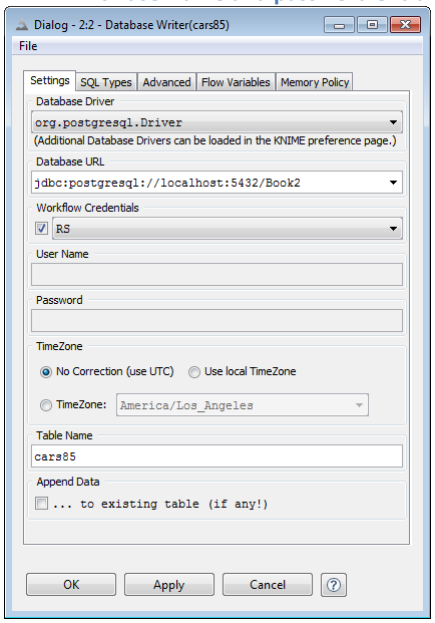
2.19. Exercise 1: The workflow



2.20. Exercise 1: Configuration window of the "Database Writer" node with credential identifier enabled



2.21. Exercise 1: Configuration window of the "Database Writer" node with username and password enabled



Note. The use of username and password instead of a credential identifier is discouraged. In fact the Workflow Credentials are all automatically encrypted, while the username and password are only encrypted if the "Master Key" in the "Preferences" page is activated. In general, usage of "Workflow Credentials" is more secure than using the username and password for each database node.

Exercise 2

In the workflow group named "Chapter2\Exercises" create a workflow called "Exercise2" to perform the following operations:

- Connect to "Book2" database and read from "cars85" table
- Remove the first two columns: "symboling" and "normalized_losses"
- Only keep rows where body_style is "sedan" or "wagon"

Solution to Exercise 2

There are many ways to implement this SELECT query. We propose three of them.

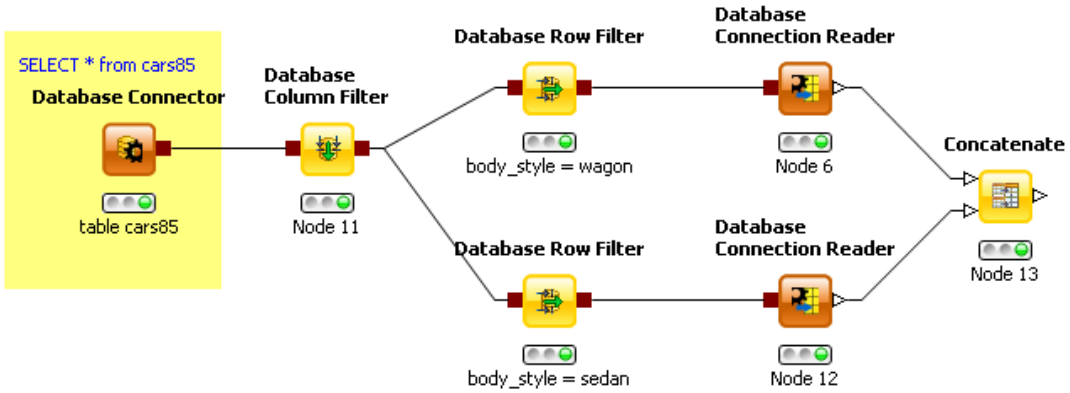
1. We implement all-in-one node with a "Database Reader" node. The required SQL SELECT query then is:

```
SELECT make, fuel_type, aspiration, nr_doors, body_style, drive_wheels, engine_location, wheel_base,
length, width, height, curb_weight, engine_type, cylinders_nr, engine_size, fuel_system, bore, stroke,
compression_ratio, horse_power, peak_rpm, city_mpg, highway_mpg, price
FROM cars85 where body_style = 'wagon' OR body_style = 'sedan'
```

   The SELECT query is a bit tedious to write mainly because of all the columns of the table we want to keep.

2. Let's try it now using a full node-based approach:

2.22. Exercise 2: Workflow where the SELECT query is built by means of database nodes

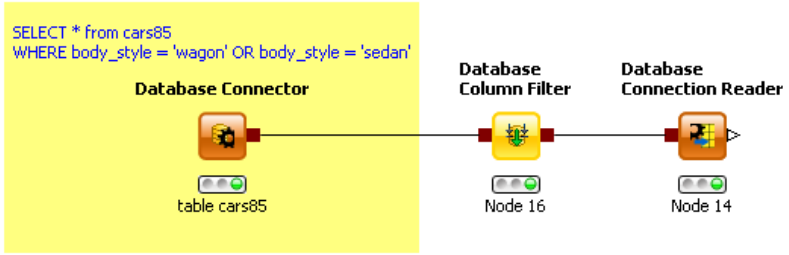


3. Finally, we used a mixed approach. The "WHERE body_style = … " clause was written in the SELECT query in the "Database Connector" node. However, the tedious task of enumerating all the columns we wanted to keep was delegated to a "Database Column Filter" node. Here is the SELECT query:

```
SELECT * FROM cars85 where body_style = 'wagon' OR body_style = 'sedan'
```

Figure 2.23 shows the workflow obtained with this mixed approach.

2.23. Exercise 2: Workflow where the SELECT query is partially built in the "Database Connector" node and also by means of the "Database Column Filter" node



Exercise 3

In the “Chapter2\Exercises” workflow group create a workflow called “Exercise3”.

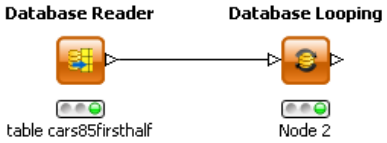
In this exercise you need to extract from table “cars85” all those rows with the same car “make” as in the first 20 rows of the table.

Solution to Exercise 3

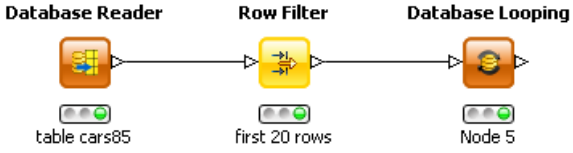
For the solution of this exercise we need a “Database Looping” node to loop across the first 20 rows of the table.

Next, the first 20 rows have to be extracted from the database table, as was done in exercise 1. We can then read the data table that was produced in exercise 1 after changing the number of rows from 10 to 20 or we can just use the first part of the exercise 1 workflow till the “Row Filter” node. In both cases, the final node has to be a “Database Looping” node to loop on all the distinct values of the column “make” in the first 20 rows of the database table.

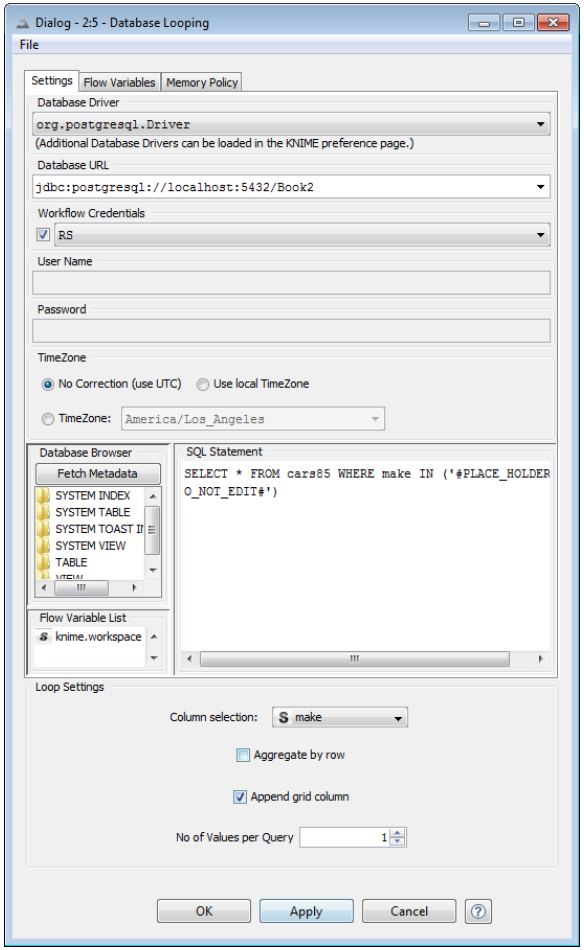
2.24. Exercise 3: Workflow with the “Database Looping” node reading from a table where the first 20 rows of the “cars85” table had been previously saved



2.25. Exercise 3: Workflow with the “Database Looping” node reading the first 20 rows directly from the “cars85” table



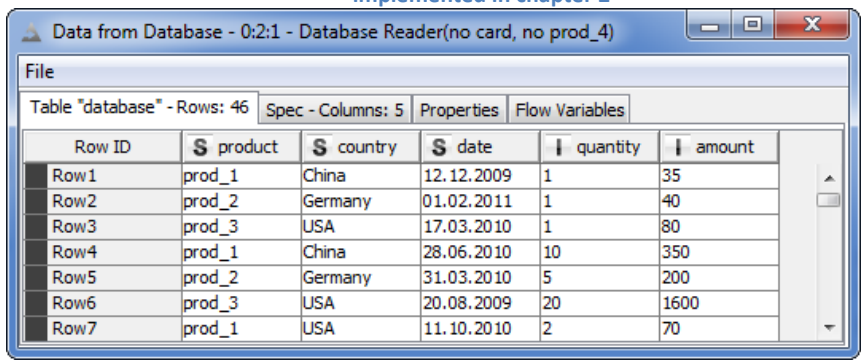
2.26. Exercise 3: Configuration window of the “Database Looping” node



Chapter 3. DateTime Manipulation

3.1. The DateTime Type

3.1. The data table produced by the "Database_Operations" workflow implemented in chapter 2



Row ID	product	country	date	quantity	amount
Row1	prod_1	China	12.12.2009	1	35
Row2	prod_2	Germany	01.02.2011	1	40
Row3	prod_3	USA	17.03.2010	1	80
Row4	prod_1	China	28.06.2010	10	350
Row5	prod_2	Germany	31.03.2010	5	200
Row6	prod_3	USA	20.08.2009	20	1600
Row7	prod_1	USA	11.10.2010	2	70

Let's now have a brief look at the data table resulting from any of the branches of the "Database_Operations" workflow built in chapter 2 (Fig.3.1). We have three columns of String type ("product", "date", and "country") and two columns of Integer type ("quantity" and "amount").

Actually, the data column "date" contains the contract date for each record and should be treated as a date/time variable. KNIME has indeed a dedicated data type for date and time: the DateTime type.

The DateTime type is used to represent dates and times in many different formats.

DateTime formats in KNIME are expressed by means of:

- A number of "d" digits for the day
- A number of "M" digits/characters for the month
- A number of "y" digits for the year
- A number of "h" digits for the hour
- A number of "m" digits for the minutes
- A number of "s" digits for the seconds
- A number of "S" digits for the milliseconds

DateTime format	String representation
dd.MM.yyyy hh.mm.ss.SSS	21.03.2011 15:34:00.000
dd-MM-yyyy	21-03-2011
hh:mm	15:34
MMM/dd/yyyy	Mar/21/2011
ss.SSS	00.000

These dedicated digits are combined in a string to produce a specific string representation of the date and/or time. The table above shows a few examples for 3:34:00pm on the 21st of March 2011.

A full category named "Time Series" offers a wide range of date and time manipulation functionalities.

3.2. How to produce a DateTime Column

How can we generate a DateTime type data column? There are many possibilities to do that. A “Database Reader” node for example automatically reads a data column into a DateTime type if the input SQL Type is a timestamp. Another possibility is to read in a string and then convert it into a DateTime type. This section explores the String to DateTime and DateTime to String conversion nodes.

In order to show the tasks implemented by these and more nodes in the “Time Series” category, let’s create a new workflow, to be named “DateTime_Manipulation”.

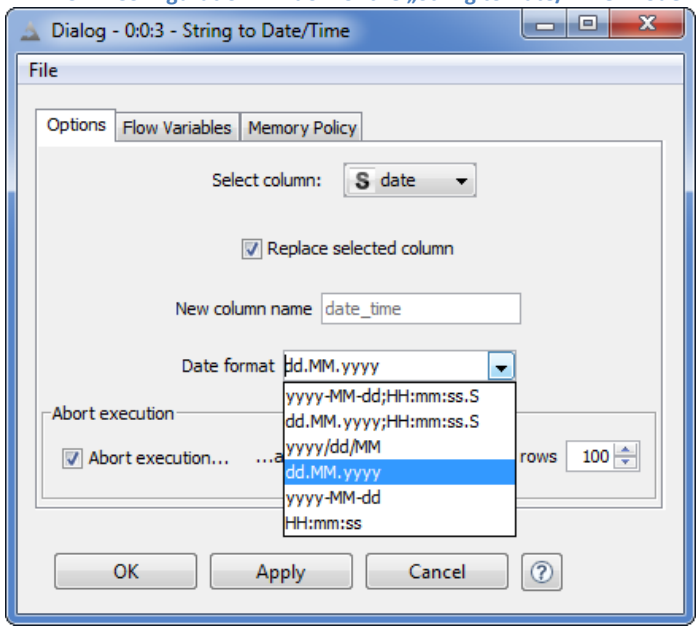
String to Date/Time

The “String to Date/Time” node converts a String cell into a DateTime cell according to a given DateTime format. The configuration window requires :

- The String column containing the dates to be converted to DateTime type
- The flag for the resulting column to:
  - Replace the original String column (where the “Replace selected column” flag is enabled)

  or
  - Be appended to the data table (in this case you need to disable the “Replace selected column” flag and to provide the name for the new column)
- The DateTime format to be used to read the String as a DateTime object
  - You can choose from a number of pre-defined DateTime formats are available in the “Date format” menu
  - Alternatively the “Date format” menu can be edited manually in order to create the appropriate DateTime format, if this is not already available

3.2. Configuration window of the „String to Date/Time“ node



- The maximum number of rows in which a reading error is tolerated. In fact if too many errors occur, node execution fails, implying that the selected DateTime format is not appropriate for the dates in the String cells. If less than the maximum number of allowed errors occurs the execution is successful and missing values are introduced for the erroneous date/times.

In the newly created workflow, let's now use the "Database Reader" node from the previous chapter (Fig. 2.12), connecting to the database "Book2" with the following SQL statement:

```
SELECT product, country, date, quantity, amount FROM sales where product != 'prod_4'
```

As we have seen before, the data column "date" has SQL type varchar(255) and therefore is read in as a String. Before proceeding with more complex manipulations on the DateTime objects, let's convert the "date" column from the String type to the DateTime type by using a "String to Date/Time" node.

Thus, after the "File Reader" node, a "String to Date/Time" node has been introduced to convert the "date" column from the String type to the DateTime type according to the "dd.MM.yyyy" format. In fact the dates contained in the data column "date" are formatted as "dd.MM.yyyy" and should then be read with that format.

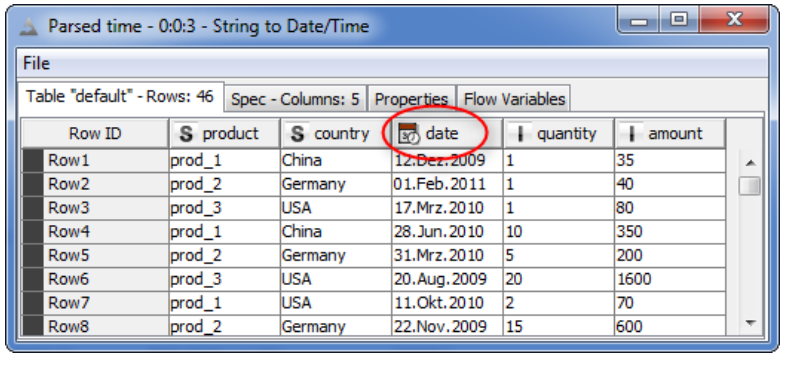
In the configuration window of the "String to Date/Time" node, we also opted to replace the original String column with the new DateTime column.

After executing the node, the resulting data table should be as in figure 3.3, where the data column "date" is now of DateTime type. The little icon showing a clock indicates a DateTime type.

DateTime formats are also available as renderers for columns of DateTime type in the KNIME data table. To change the renderer of a column in a data table:

- Open the data table
- Right-click the column header
- Choose the desired renderer among the available renderers

3.3. Column "date" after the conversion to DateTime with the „String to Date/Time" node

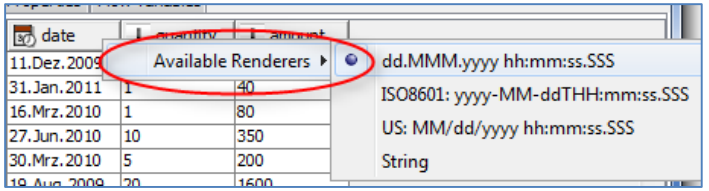


Row ID	product	country	date	quantity	amount
Row1	prod_1	China	12.Dez.2009	1	35
Row2	prod_2	Germany	01.Feb.2011	1	40
Row3	prod_3	USA	17.Mrz.2010	1	80
Row4	prod_1	China	28.Jun.2010	10	350
Row5	prod_2	Germany	31.Mrz.2010	5	200
Row6	prod_3	USA	20.Aug.2009	20	1600
Row7	prod_1	USA	11.Okt.2010	2	70
Row8	prod_2	Germany	22.Nov.2009	15	600

Possible renderer formats are:

- “dd.MMM. yyyy hh:mm:ss:SSS” (default)
- “MM/dd/yyyy hh:mm:ss.SSS” (US)
- “yyyy-MM-ddThh:mm:ss.SSS” (ISO8601)
- “yyyy-MM-dd” (string)

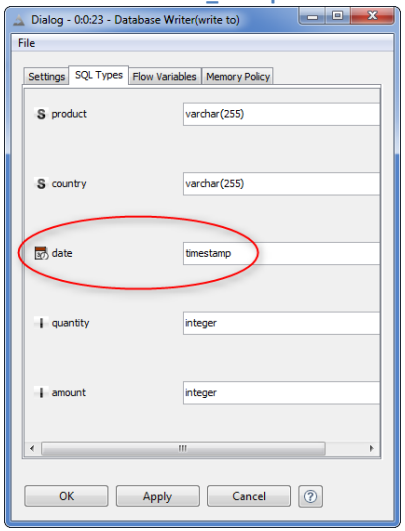
3.4. DateTime Renderers



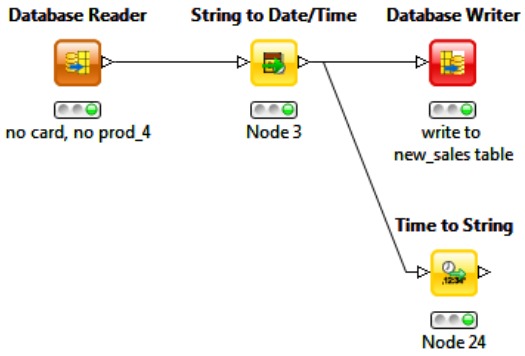
Note. The renderer selection is not saved. If you close and reopen the data table view, the cells of each data column are shown again according to the default renderer’s format.

Finally, we wrote the DateTime type column into a database table. We used a “Database Writer” node to write the resulting data table to a table named “new_sales” in database “Book2” as described in section 2.5. In the “SQL Types” tab of the configuration window of the “Database Writer” node the default data type for the “date” column would be “datetime”. This might not work in PostgreSQL. So we manually set it to “timestamp”. The “date” field then gets written into the database table with “timestamp” SQL type.

3.5. "SQL Types" tab in the configuration window of the "Database Writer" node in the "DateTime_Manipulation" workflow



3.6. The "DateTime_Manipulation" workflow



If we now read the “new_sales” table with a “Database Reader” node, the timestamp SQL type of the field “date” is automatically converted into a DateTime type.

Sometimes it is necessary to move in the opposite direction. That is, it might be necessary to convert a DateTime column into a String column. For example, we might want to write the data table into a text file and have a customized date format rather than the default date format, or we might want to create a report with a customized date format, or perform some string manipulation on the dates, such as adding a prefix or a suffix.

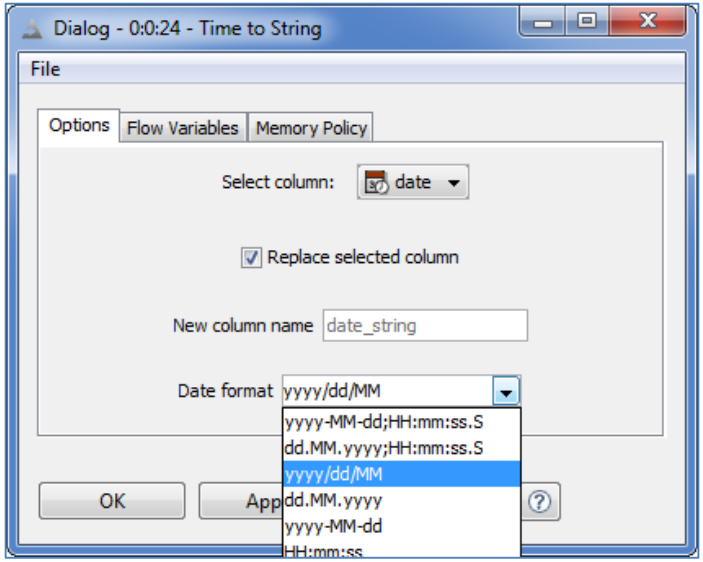
Time to String

The “Time to String” node converts a DateTime cell into a String object according to a given DateTime format. The configuration window requires :

- The DateTime column to be converted into String type
- The flag for the resulting String column to:
  - Replace the original DateTime column (where the “Replace selected column” flag is enabled)

  or
  - Be appended to the data table (in this case you need to disable the “Replace selected column” flag and to provide the name for the new column)
- The DateTime format to build the string pattern
  - A number of pre-defined DateTime formats are available in the “Date format” menu
  - The “Date format” box can be edited manually in order to create the appropriate DateTime format if the one you want is not already available in the menu

3.7. Configuration window of the „ Time to String“ node



In the “DateTime_Manipulation” workflow, we introduced a “Time to String” node to convert the DateTime column, “date”, back to a String type with format “yyyy/MM/dd”. The final “DateTime_Manipulation” workflow is shown in figure 3.6.

Another way to generate dates and/or times in a DateTime data column is offered by the “Time Generator” node. In a second workflow named “DateTime_Manipulation_2, for example, we generated 30 date values (no times) equally spaced between 01.01.2009 and 01.01.2011 with the configuration settings of the “Time Generator” node shown below.

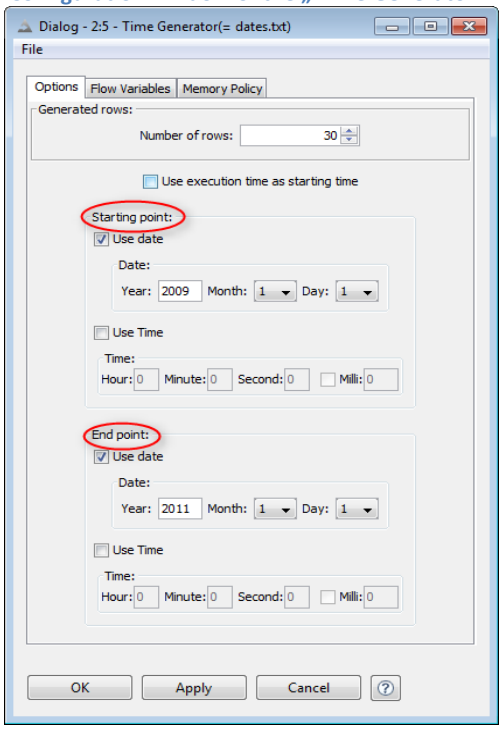
Time Generator

The “Time Generator” node produces a number of DateTime values equally spaced between an initial and a final value. A new column is created with the generated DateTime values.

The configuration window requires :

- The number of DateTime values to generate; that is, the number of rows
- The starting DateTime point, in terms of date and/or time
- The end DateTime point, in terms of date and/or time
- The flag “Use execution time as starting time” sets the start point to the current date and time.

3.8. Configuration window of the „ Time Generator“ node



Note. In Data Warehousing applications, a timestamp is often required to identify the upload time. The flag “Use execution time as starting time”, if enabled, allows to create one record with a timestamp with the current date and time. It is enough to set the end point to a date very far away in time, like in year 2999, and the number of rows to 1.

3.3. Refine DateTime Columns

The “Time Generator” node in the previous section has produced 30 dates without time. In some cases, it might be desirable to extend the DateTime column format to include both date and time.

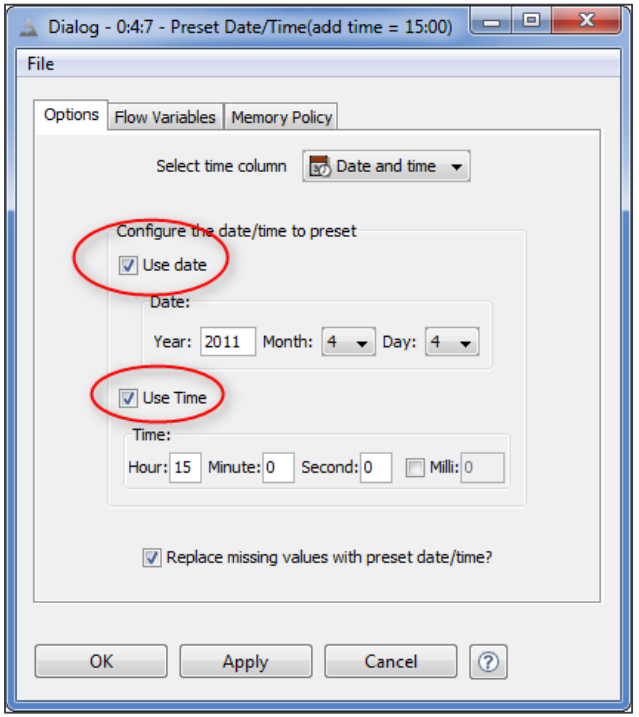
Preset Date/Time

The “Preset Date/Time” node completes the DateTime content of a column with a preset value for time, if time is missing, or of date, if date is missing. If the entire cell content is missing, the “Preset Date/Time” node might fill it with preset values for both date and time.

The configuration window requires :

- The column of DateTime type to be completed
- The flag to enable completion of the cells’ content with the date, where the date is missing
- The preset date value to use for such completion
- The flag to enable completion of the cell content with the time, where the time is missing
- The preset time value to use for such completion
- The flag to enable the replacement of missing values with the preset date and time values

3.9. Configuration window of the „ Preset Date/Time“ node



We appended a “Preset Date/time” node to the “Time Generator” node to complete both generated dates with a “15:00” time.

As the “Preset Date/Time” node widens the DateTime format to include the date and/or the time, the “Mask Date/Time” node moves in the opposite direction: it masks the date and/or the time in the cells of columns of the DateTime type.

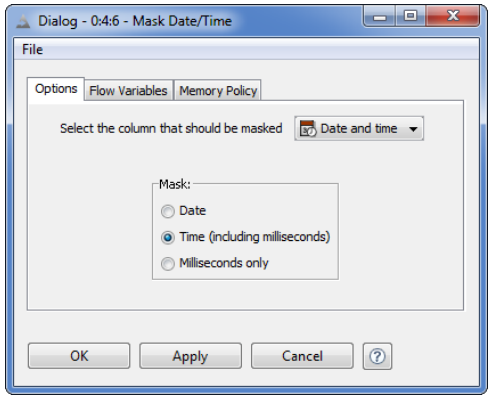
Mask Date/Time

The “Mask Date/Time” node removes the date and/or the time from a DateTime cell content.

The configuration window requires :

- The DateTime column on which to operate
- The object to mask: date, time, or just the milliseconds

3.10. Configuration window of the „ Mask Date/Time“ node



We introduced a “Mask Date/Time” node after the “Preset Date/Time” node to remove the “15:00” time on all dates in column “date”.

Another useful date/time manipulation function consists of adding/removing some fixed or dynamic amount of time from values in a datetime column or a fixed datetime value.

Date/Time Shift

The “Date/Time Shift” node shifts a datetime value of a defined granularity (day, hour, month, etc…) starting from a reference datetime value.

The configuration window requires :

- The shift value type, whether static or dynamic
- The shift value amount, if static
- The shift value granularity (day, hour, month, week, etc…)
- The name of the new output datetime column
- The reference date as the column name, if dynamic; as value in date and time, if static; or as the current time (execution time)
- The format of the output datetime column

3.11. Configuration window of the „ Date/Time Shift” node

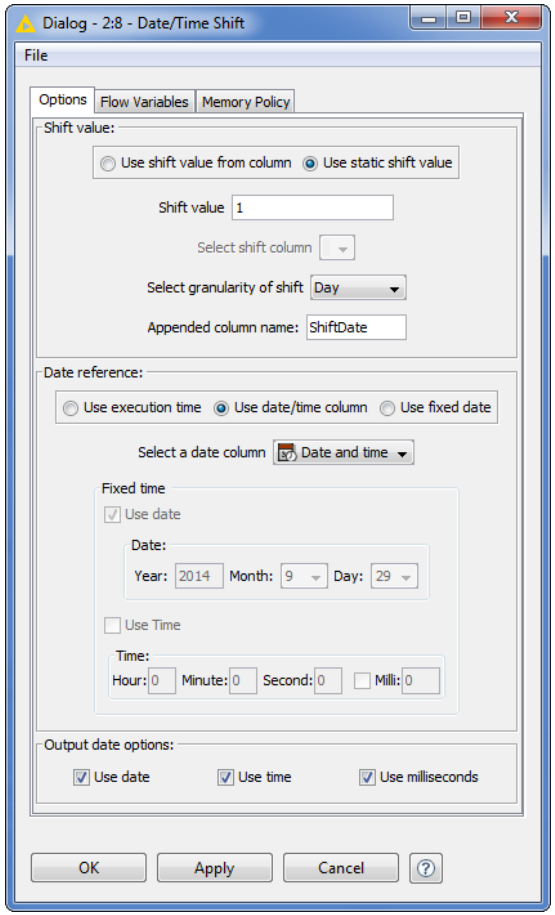
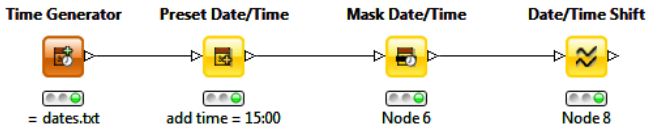


Figure 3.12 shows the “DateTime_Manipulation_2” workflow.

3.12. The "DateTime_Manipulation_2" workflow



3.4. Row Filtering based on Date/Time Criteria

Let's dive deeper now into date/time manipulation. Very often a data analysis algorithm needs to work on the most recent data or on data inside a specified time window. For example, balance sheets usually cover only one year at a time; the results of an experiment can be observed inside a limited time window; fraud analysis runs on a daily basis; and so on. This section shows a number of row filtering operations based on date/time criteria.

In order to show practically how such time based filtering criteria can be implemented we need a data set with a relatively high number of data rows and at least one column of the DateTime type. For this example, we used the "Database Reader" node defined in section 2.5 and built in the "Database_Operations" workflow under the name of "no card, no prod_4". Summarizing shortly, the "Database Reader" node, named "no card, no prod_4", was implemented in a new workflow, "DateTime_Manipulation_3", to read the data from the "new_sales" table in the "Book2" database, created by the "DateTime_Manipulation" workflow, and to filter out the "card" field and the rows where product is "prod_4". The output data table consisted of 5 columns and 46 rows and represented a sales archive of the sold product, the country where it was sold, the quantity of product sold, the amount of money received from the sale, and the sale date. The sale date column, named "date", was extracted from the database already with the "DateTime" type.

The most common data selection is the one based on a time window and it requires setting an explicit initial and final date/time. Only the data rows falling inside this time window are kept, while the remaining data rows are filtered out. The "Extract Time Window" node performs exactly this kind of row filtering based on the explicit definition of a time window.

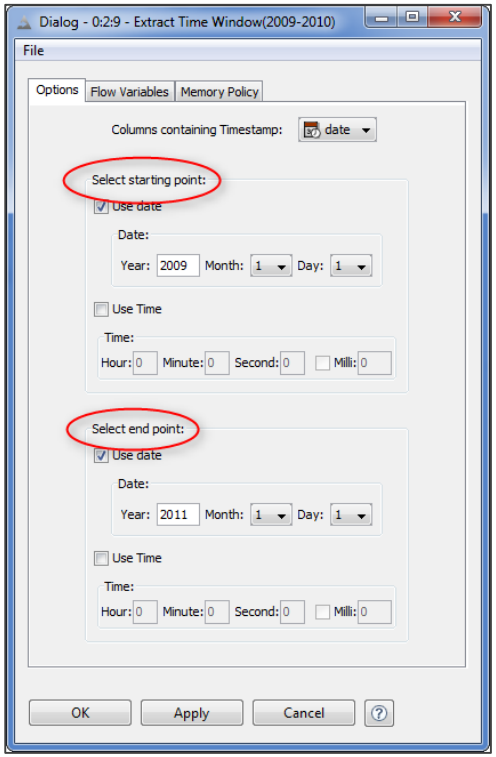
Extract Time Window

The "Extract Time Window" node implements row filtering based on time window criteria. In fact, it keeps all data rows from the input data table inside a pre-defined time window.

The configuration window requires :

- The DateTime column to which the filtering criterion should apply
- The time window, i.e.:
    - The window starting point, in terms of date and/or time
    - The window end point, in terms of date and/or time

3.13. Configuration window of the „Extract Time Window" node



In the new workflow "DateTime_Manipulation_3" after the "Database Reader" node, an "Extract Time Window" node was introduced to select all sales that happened in a pre-defined time range. For the time range we used "2009.Jan.01" and "2011.Jan.01". We set these dates as starting and end points respectively in the configuration window of the "Extract Time Window" node and we obtained a data set with 38 sales data rows at the output port, covering the selected 2-year time span.

Let's suppose now that the year 2010 was a troubled year and that we want to analyze the data of this year in more detail. How can we isolate the data rows with sales during 2010? We could convert the "date" column from the DateTime type to the String type and work on it with the string manipulation nodes offered in the "Data Manipulation"-> "Column" category. There is, of course, a much faster way with the "Date Field Extractor" or the "Time Field Extractor" node.

The “Date/Time Field Extractor” nodes decompose a DateTime object into its components. A date can be decomposed into year, month, day number in month, day of the week, quarter to which the date belongs, and day number in year. A time can be decomposed in hours, minutes, seconds, and milliseconds. Here are some examples:

date	year	Month (no)	Month (text)	Day	Week day (no)	Week day (text)	quarter	No of day in year
26.Jun.2009	2009	6	June	26	6	Friday	2	177
22.Sep.2010	2010	9	September	22	4	Wednesday	3	265

time	hours	minutes	seconds	milliseconds
15:23:10.123	15	23	10	123
04:02:56.987	4	2	56	987

This section only shows the “Date Field Extractor” node. The “Time Field Extractor” node works similarly to the “Date Field Extractor” node only isolating time components.

In the “DateTime_Manipulation_3” workflow, we introduced a “Date Field Extractor” node after the “Extract Time Window” node. The configuration settings were set to extract the year, the month, the day of week, and the day number in the month from each DateTime value in the “date” column. The month and the day of week can be represented numerically, from 1 for January to 12 for December and from 1 for Sunday to 7 for Saturday, or as text strings with the full month and weekday name. We selected a text result for both the month and the day of week component. The data table on the output port thus had 9 columns, 4 more than the original 5 columns. The 4 additional columns are: “Year”, “Month”, Day of month”, and “Day of week”.

Date Field Extractor

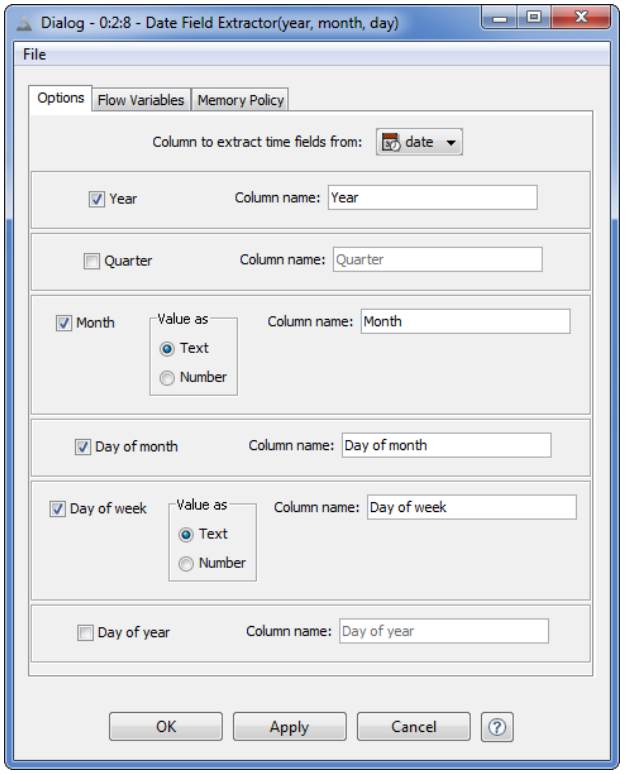
The "Date Field Extractor" node decomposes a date into its components: Year, Month, Number of day in month, Day of week, Quarter in year to which the date belongs, and Number of day in the whole year.

The configuration window requires a flag to enable the output of each date component:

- A flag to show the date's year
- A flag to show the date's year quarter
- A flag to show the date's month as number or as text
- A flag to show the date's day of month
- A flag to show the date's day of week as number or as text
- A flag to show the date's day of year

As many new columns are appended to the output data table as date components are selected in the configuration window.

3.14. Configuration window of the „ Date Field Extractor" node



In order to isolate the 2010 sales, we added a "Row Filter" node after the "Date Field Extractor" node in the "DateTime_Manipulation_3" workflow to keep all rows where Year = 2010. The resulting data table had 9 columns and 25 rows, all referring to sales performed in 2010. Similarly, we could have summed up the sale amounts by month with a "GroupBy" node to investigate which months were more profitable and which months showed a sale reduction across the 2 years. Or we could have counted all sales by weekday to see if more sales were made on Fridays compared to Mondays. The decomposition of date and time opens the door to a number of pattern analysis and exploratory investigations over time.

Let's suppose now that we only want to work on the most recent data, for example on all sales that are not older than one year. We need to calculate the time difference between today and the sale date. All rows with a sale date older than one year should be excluded from the data table. The "Time Difference" node calculates the time difference between two DateTime values. The two DateTime values can be: two DateTime cells on the same row,

one DateTime cell and the current date/time, one DateTime cell and a fixed DateTime value, or finally one DateTime cell and the corresponding DateTime cell in the previous data row. The time difference can be calculated in terms of days, months, years, etc…, even milliseconds.

Time Difference

The "Time Difference" node calculates the time elapsed between two DateTime values. That is:

a. Between a DateTime column and the current date/time ("use execution time")
b. Between a DateTime column and another DateTime column in the same data row ("use second column")
c. Between a DateTime column and a fixed date/time ("use fixed date")
d. Between a DateTime cell and the DateTime cell in the previous row in the same column ("use previous row")

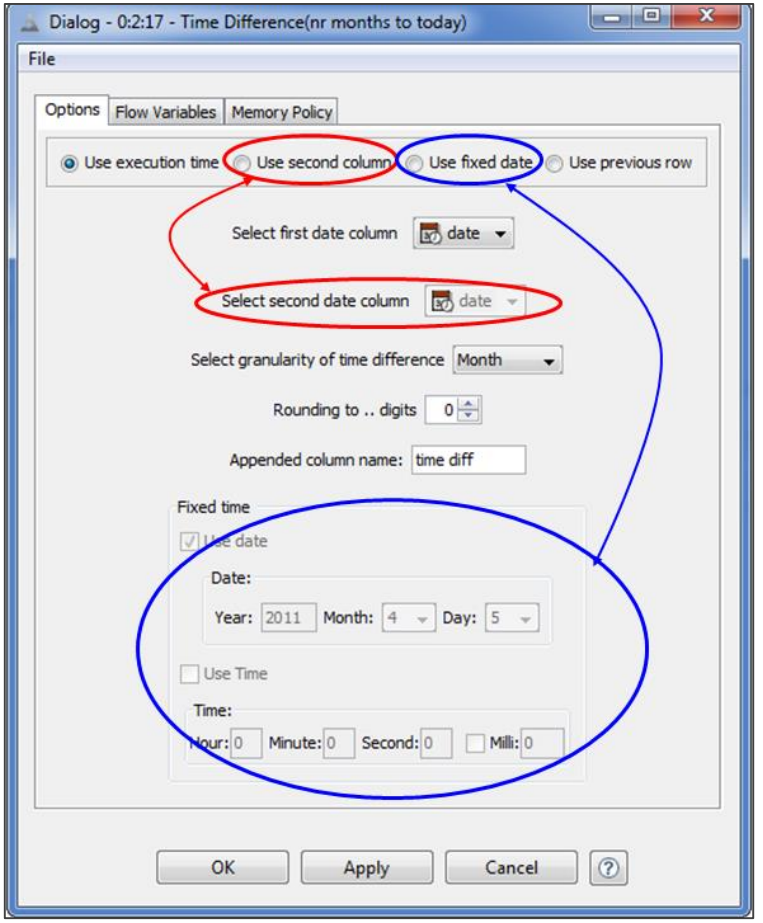
The configuration window requires :

- Working mode: a, b, c, or d
- The column to use for the first DateTime values
- The granularity to calculate the time difference
- The name of the result column
- The rounding factor, i.e. how many fraction digits

Additionally, it requires:

- For case b, the second DateTime column
- For case c, the fixed date/time value

3.15. Configuration window of the „ Time Difference" node



In the "DateTime_Manipulation_3" workflow, we connected a "Time Difference" node to the "Database Reader" node to calculate the time elapsed between today (current date) and the DateTime values in the "date" column, i.e. to calculate how long ago the sale contract was made. We selected

“month” for the granularity of the time difference, i.e. the measure of how old a sale is expressed in number of months. The resulting differences were appended to a column named “time_diff”. Sales older than one year had a “time_diff” value larger than 12 months. We then used a “Row Filter” to filter out all rows where “time_diff” > 12 months.

Another interesting application of the “Time Difference” node is to calculate the time intervals between one sale and the next. This is helpful to see if the sale process improves with time or after some marketing event for example. In this case we need to sort the sale records by sale date with a “Sorter” node. The “Sorter” node can manage DateTime types and can sort them appropriately. After that, we use a “Time Difference” node and we configure it to calculate the time difference between the sale date in the current row and the sale date in the previous row in terms of days. The resulting column “time_diff” contains the number of days between one sale and the previous one. A simple line plot of the “time_diff” column can give us interesting insights into the dynamics of the sale process (see workflow “DateTime_Manipulation_3” in the Download Zone).

3.5. Time Series Average and Aggregation

In the “Time Series” category you also find a node that is more oriented towards time series analysis rather than just DateTime manipulation: this is the “Moving Average” node.

The “Moving Average” node calculates the moving average [3] on the time series stored in a column of the input data table. The moving average operates on a k-sample moving window. The k-sample moving window is placed around the n-th sample of the time series. An average measure is calculated across all values of the moving window and replaces the original value of the nth sample of the time series. A number of slightly different algorithms can be used to produce slightly different moving averages. The differences consist of how the moving window is placed around the n-th sample and how the average value of the moving window is calculated. Two parameters are particularly important for a moving average algorithm:

- The position of sample n inside the k-sample moving window
- The formula to calculate the average value of the moving window

The moving average algorithm is called:

- Backward, when the *n-th* sample is the last one in the moving window
- Center, when the *n-th* sample is in the center of the moving window (in this case size *k* must be an odd number)
- Forward, when the *n-th* sample is at the beginning of the moving window
- Cumulative, when the whole past represents the moving window; in this case is *n=k*
- Recursive, when the new value of the nth sample is calculated on the basis of the *(n-1)-th* sample

If *v(i)* is the value of the original sample *i* inside the moving window, the algorithm to calculate the average value can be one of the following:

Algorithm	Formula	Notes
simple average measure	$avg(n) = \frac{1}{\mathrm{k}} \cdot \sum_{i=0}^{k} \mathrm{v(i)}$	
Gaussian weighted average measure	$avg(n) = \frac{1}{\mathrm{k}} \cdot \sum_{i=0}^{k} \mathrm{w(i)} \cdot \mathrm{v(i)}$	Where w(i) is a Gaussian centered around the nth sample, whose standard deviation is $\frac{k-1}{4}$
harmonic mean	$avg(n) = \frac{n}{\sum_{i=0}^{k-1} \frac{1}{\mathrm{v}\left(\mathrm{n} + \mathrm{i} - \frac{\mathrm{k}-1}{2}\right)}}$	The harmonic mean can only be used for strictly positive *v(i)* values and for a center window.
simple exponential	$avg(n) = EMA(v,n) = \alpha \cdot \mathrm{v(n)} + (1-\alpha) \cdot simple\_exp(\mathrm{n}-1)$ $simple\_\mathrm{exp}(0) = v(0)$	Where: $\alpha = \frac{2}{k+1}$ and *v = v(n)*
double exponential	$avg(n) = 2 \cdot EMA(v,n) - EMA(EMA(v,n),n)$	
triple exponential	$avg(n) = 3EMA(v,n) - 3EMA(EMA(v,n),n) + EMA(EMA(EMA(v,n),n),n)$	
old exponential	$avg(n) = EMA\_backward(v,n) = \alpha \cdot \mathrm{v(n)} + (1-\alpha) \cdot backward\_simple(\mathrm{n}-1)$	Where: $\alpha = \frac{2}{k+1}$ and *backward_simple(n)* is the simple average of the moving window where the *n-th* sample is at the end.

Based on the previous definitions, a backward simple moving average replaces the last sample of the moving window with the simple average; a simple cumulative moving average takes a moving window as big as the whole past of the nth sample and replaces the last sample (*n-th*) of the window with

the simple average; a center Gaussian moving average replaces the center sample of the moving window with the average value calculated across the moving window and weighted by a Gaussian centered around its center sample; and so on. The most commonly used moving average algorithm is the center simple moving average.

Moving Average

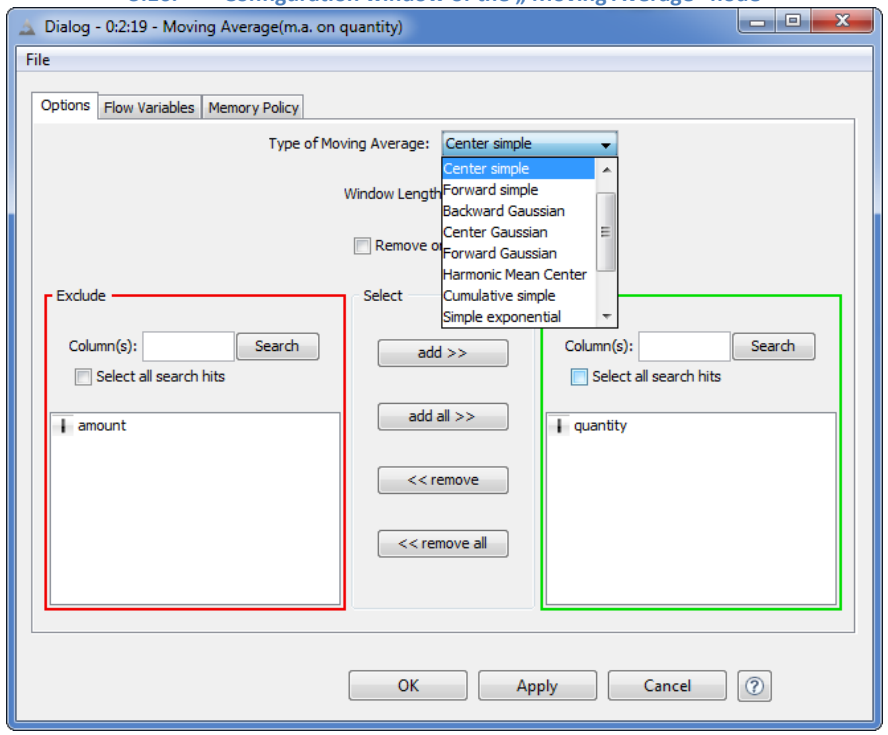
The "Moving Average" node calculates the moving average of one or more columns of the data table. The configuration window requires :

- The moving average algorithm
- The length of the moving window in number of samples
- The flag to enable the removal of the original columns from the output data table
- The input data column(s) to calculate the moving average

The selection of the data column(s), to which the moving average should be applied, is based on an "Exclude/Include" framework.

- ➢ The columns to be used for the calculation are listed in the "Include" frame on the right
- ➢ The columns to be excluded from the calculation are listed in the "Exclude" frame on the left

3.16. Configuration window of the „ Moving Average" node



To move single columns from the "Include" frame to the "Exclude" frame and vice versa, use the "add" and "remove" buttons. To move all columns to one frame or the other use the "add all" and "remove all" buttons.

A "Search" box in each frame allows searching for specific columns, in case an excessive number of columns impedes an easy overview.

Note. If a center moving average is used, the length of the moving window must be an odd number and the first (n-1)/2 values are replaced with missing values.

In the "DateTime_Manipulation_3" workflow we applied a "Moving Average" node to the output data table of the "Database Reader" node. The center simple moving average was applied to the "quantity" column, with a moving window length of 11 samples. A "Line Plot" node placed after the "Moving Average" node showed the smoothing effect of the moving average operation on the "quantity" time series (Fig. 3.18).

The "Moving Aggregation" node extends the "Moving Average" node. Indeed, it calculates a number of statistical measures, including average, on a moving window. In the configuration window you need to select the data column, the statistical measure to apply, the size and type of the moving window, and a few additional preferences about the output data table structure. Many statistical and aggregation measures are available in the "Moving Aggregation" node. They are all described in the tab "Description" of the configuration window.

Moving Aggregation

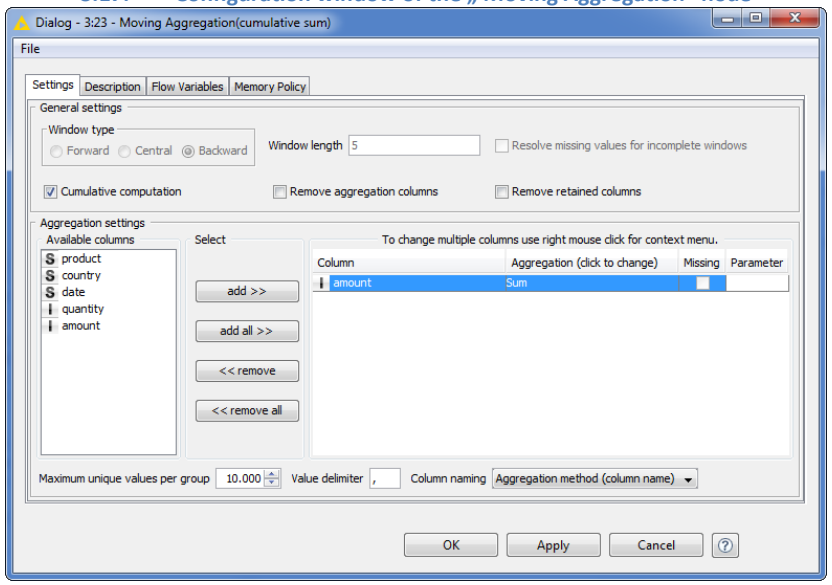
The "Moving Aggregation" node calculates statistical and aggregation measures on a moving window.

The "Settings" tab in the configuration window requires:

- Statistical or aggregation measure to use
- Input data column for the calculation
- Type and size of the moving window
- Checkboxes for the output data table format
- The checkbox for cumulative aggregation

A second tab, named "Description", includes a description of all statistical and aggregation measures available for this node.

3.17. Configuration window of the „ Moving Aggregation" node



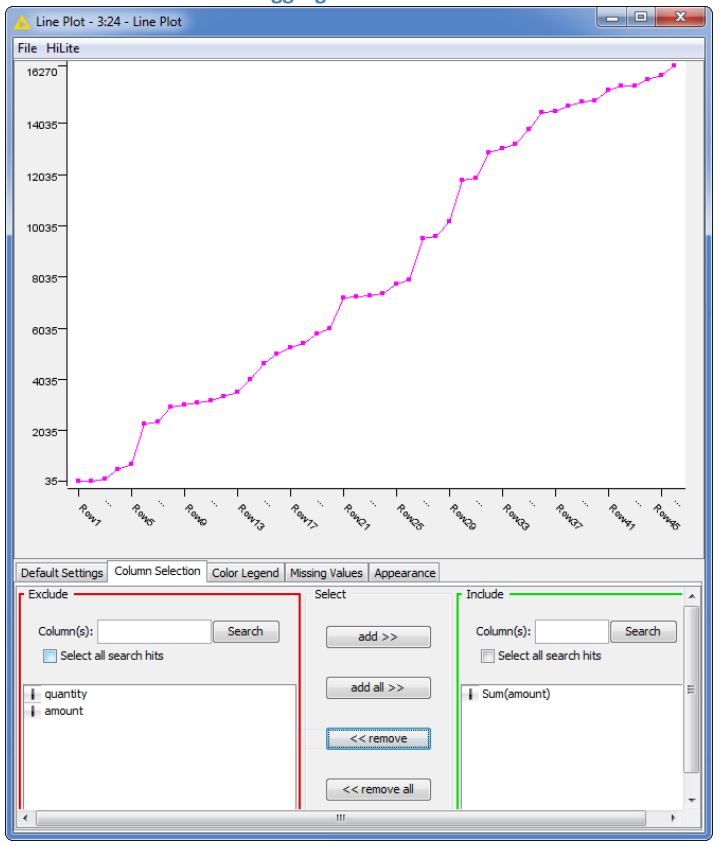
Note. Selecting data column "amount", aggregation measure "mean", window type "Central", and window size 11, the same output time series is generated as by the "Moving Average" node as configured in figure 3.18.

Finally, when checking the “Cumulative computation” checkbox, the “Moving Aggregation” node uses the whole time series as a time window and performs a cumulative calculation. The most common cumulative calculation is the cumulative sum used for example in accounting for year to date measures.

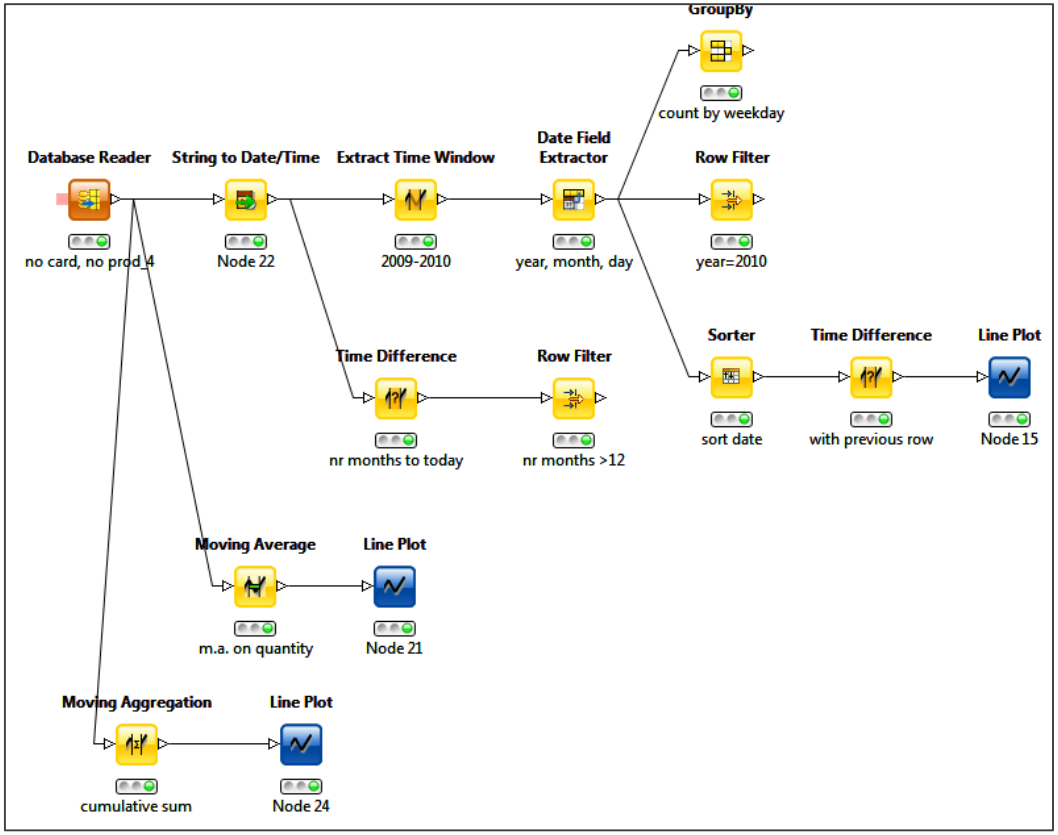
In the “DateTime_Manipulation_3” workflow we added a “Moving Aggregation” node to the output data table of the “Database Reader” node. The cumulative sum was calculated for the “amount” data column, over the whole time series. The resulting time series is displayed in figure 3.19.

3.18. Moving Average effect on a time series

3.19. Cumulative Aggregation of “amount” over time



3.20. The “DateTime_Manipulation_3” workflow



3.6. Time Series Analysis

The “Time Series” category offers also some meta-nodes for time series analysis. Three meta-nodes are available for now: for seasonality correction, to train an auto-prediction model, to apply the auto-prediction model to predict new value(s). All these nodes rely on the “Lag Column” node.

Lag Column

The "Lag Column" node copies a data column x(t) and shifts it n steps down. It can work in two ways:

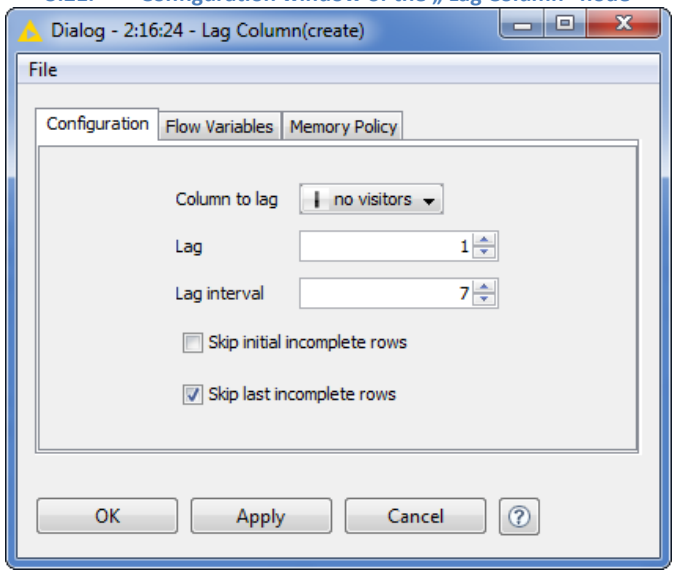
- It can produce x(t), x(t-p)
- It can produce x(t), x(t-1), …, x(t-n)
- It can produce x(t), x(t-p), x(t-p*2), … x(t-p*n)

Where p is the "Lag Interval" and n the "Lag" value.

The data column to shift, the Lag, and the Lag Interval are then the only important configuration settings required.

Two more settings state whether the incomplete rows generated by the shifting process have to be included or removed from the output data set.

3.21. Configuration window of the „ Lag Column" node



Note. If the data column values are sorted in time order, shifting the values up or down means shifting the values in the past (descending time order) or in the future (ascending time order).

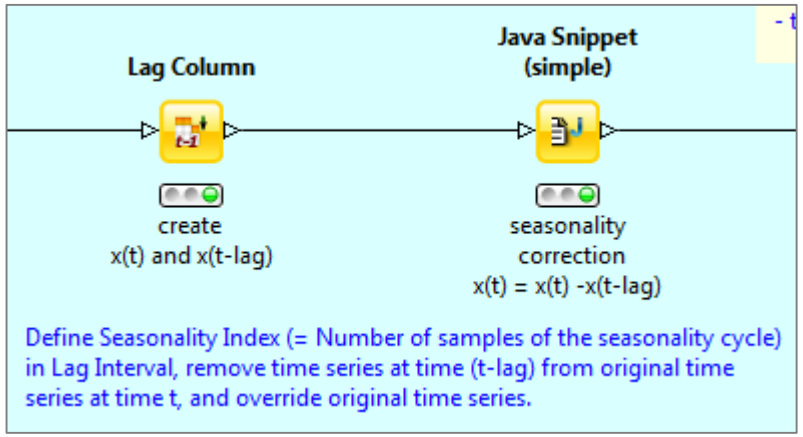
The metanode "Seasonality Correction" defines a seasonality index in a "Lag Column" node, for example Lag Interval=7 for a daily time series with weekly seasonality. The "Lag Column" node then pairs each time series value with its correspondent 7 days earlier. Notice that this temporal alignment is only true if the time series was already sorted by time in ascending order. Following the "Lag Column" node, a "Java Snippet (simple)" node subtracts the past sample (one week earlier in our case) from the current sample, de facto subtracting last week values from current week values and removing weekly seasonality. The output time series is just the series of differences of this week with respect to past week values.

The metanode "Time-Series Auto-Prediction Training" builds an auto-predictive model for time series. First, a "Lag Column" node aligns past and present, that is current value with previous n values (Lag=n), putting in a row x(t-n), x(t-n-1), … x(t). The idea is to use the past (x(t-n), x(t-n-1), …, x(t-1)) to predict the future (x(t)). After the "Lag Column" node, a "Partitioning" node partitions the data sequentially from the top. Sequential partition is important for time series analysis. Since some data mining algorithm can learn the underlying dynamic of a time series, randomly partitioning data would offer bits of future to the learning algorithm, which would never happen in real life applications. Finally, a learner node handling numeric

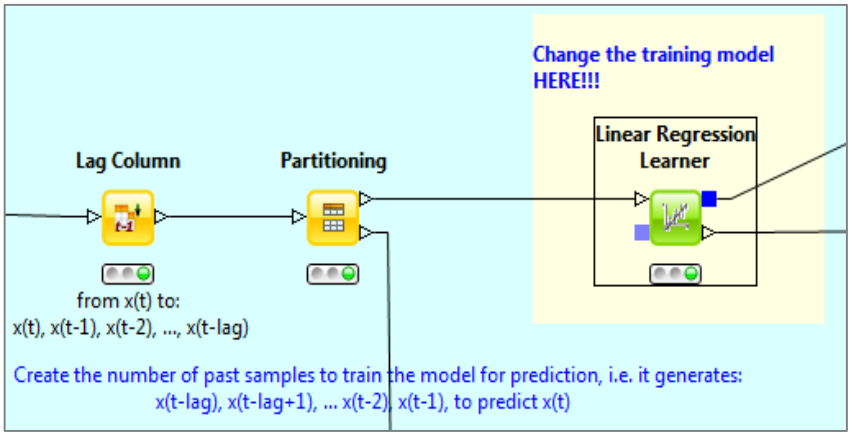
predictions predicts x(t) based on the past values in the same row. The default learner node is the “Linear Regression Learner” node, but it can be changed with any other learner node able to handle numeric predictions.

The “Time-Series Auto-Prediction Predictor” metanode uses the model built by the “Time-Series Auto-Prediction Training” metanode, to predict new values based on the existing ones. It contains the predictor node associated with the learner node in the “Time-Series Auto-Prediction Training” metanode and a “Numeric Scorer” node to qualtify the prediction error.

3.22. “Seasonality Correction” metanode

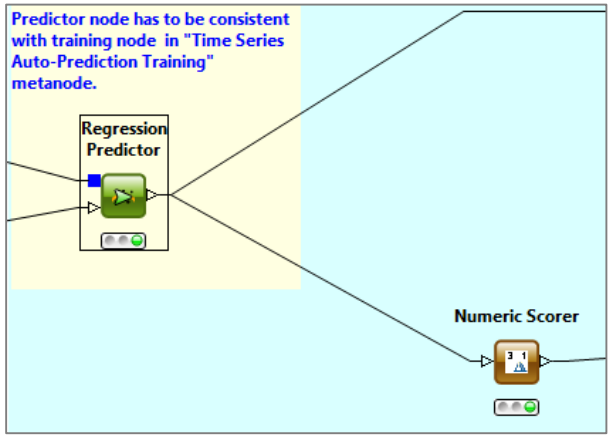


3.23. “Time-Series Auto-Prediction Training” metanode

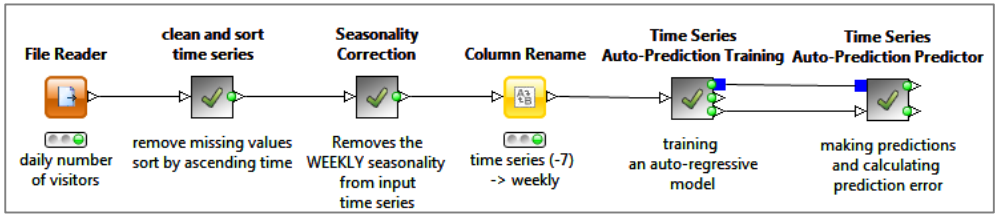


In the workflow named “time_series” we import the time series made of number of daily visitors to a web site over time from 2012 till 2014. After removing missing values and sorting the time series by ascending time, we remove the weekly seasonality (Lag Interval = 7) pattern with a “Seasonality Correction” metanode, we train an auto-regressive model using a “Linear Regression learner” node in a “Time-Series Auto-Prediction Training” metanode, we calculate the predicted values and the numerical prediction error with a “Time-Series Auto-Prediction Predictor” metanode.

3.24. "Time-Series Auto-Prediction Predictor" metanode



3.25. The "time_series" workflow



3.7. Exercises

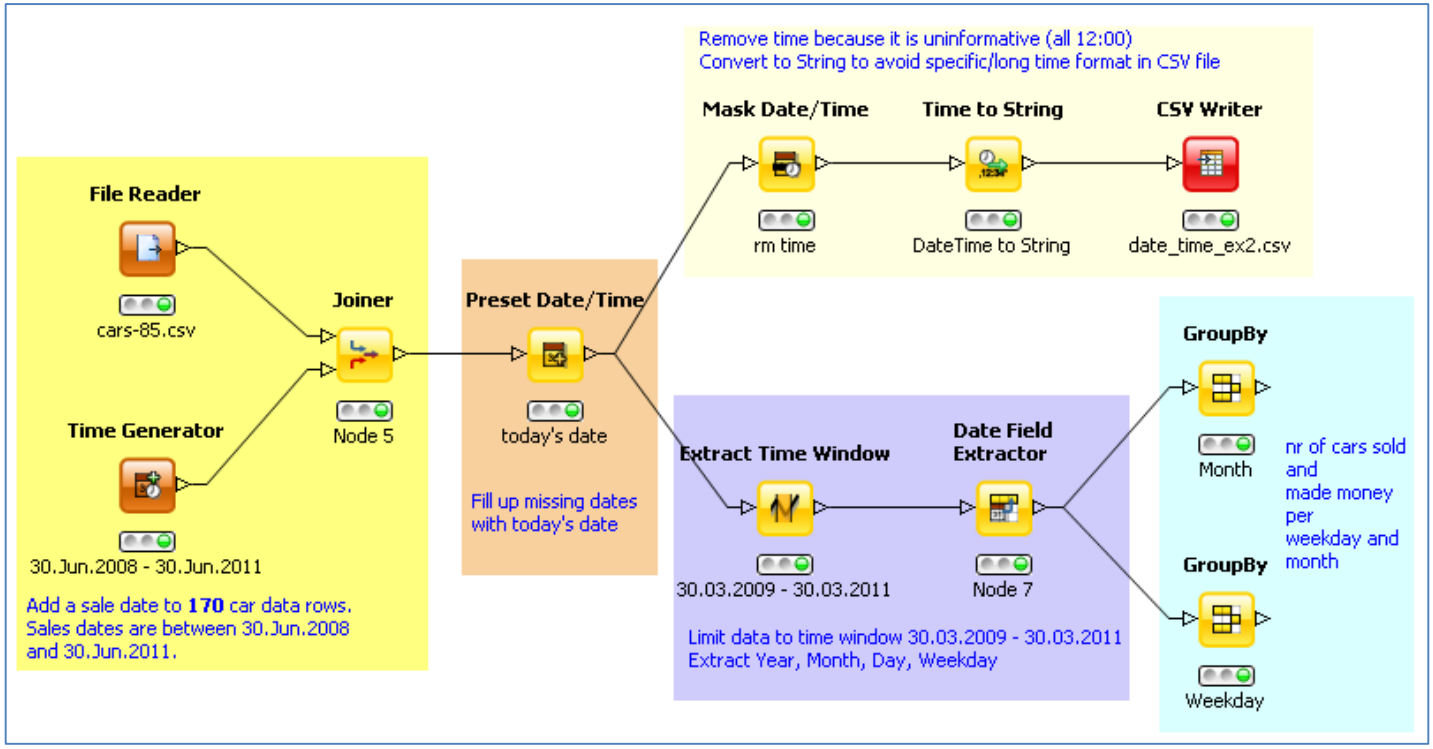
Create a workflow group "Exercises" under the existing "Chapter3" workflow group to host the workflows for the exercises of this chapter.

Exercise 1

- Add a random date between 30.Jun.2008 and 30.Jun.2011 to the first 170 rows of the "cars-85.csv" file;
- In the remaining data rows replace the missing date with today's date (in the workflow solution today's date was "06.Apr.2011");
- Write the new data table to a CSV file;
- If you consider the newly added dates as sale dates and the values in the "price" column as sale amounts, find out which month and which weekday collects the highest number of sales and the highest amount of money.

Solution to Exercise 1

3.26. Exercise 1: The workflow

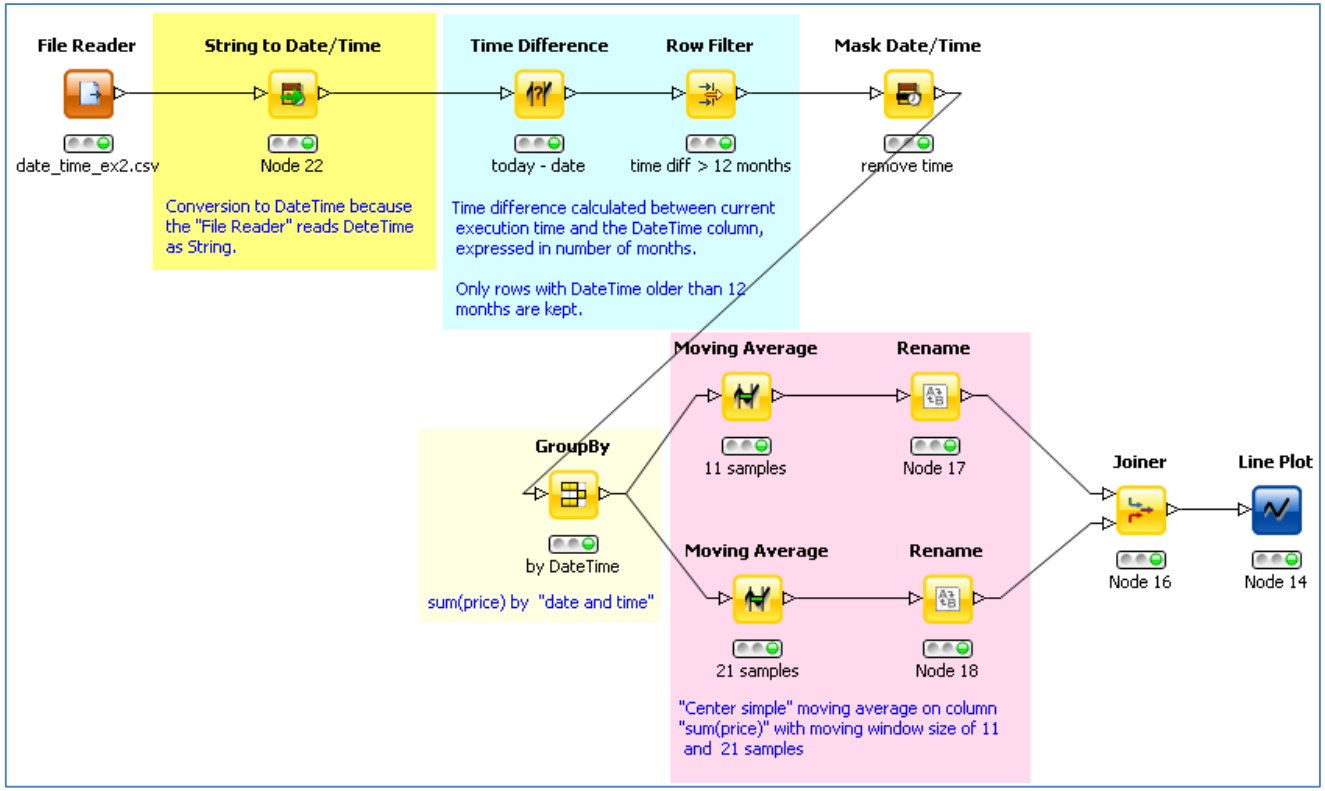


The rows generated by the “Time Generator” node are only 170. As a result, the “Joiner” node is set with a “left outer Join” to keep all rows of the “cars-85.csv” file. The “CSV Writer” node writes a file named “date_time_ex2.csv”.

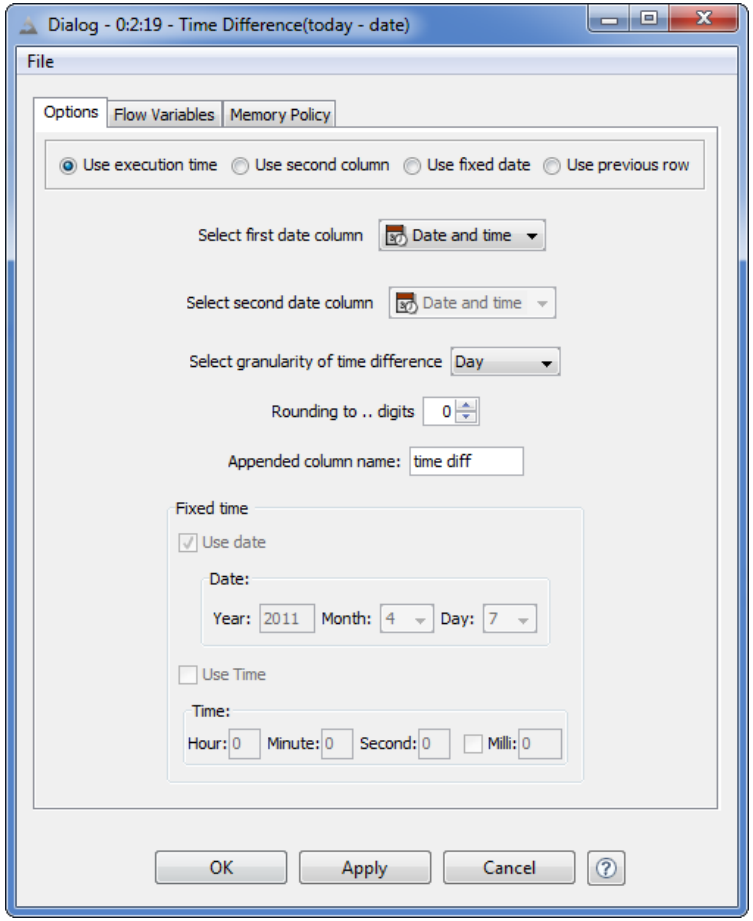
Exercise 2

- Import the “date_time_ex2.csv” file produced in Exercise 1;
- Isolate the data rows with “date and time” older than one year from today (in the workflow solution today’s date was “06.Apr.2011”);
- Calculate the amount of money (“price” column) made in each sale date (“date and time” column);
- Apply a moving average algorithm to the time series defined in the previous exercise and observe the effects of different sizes of the moving window.

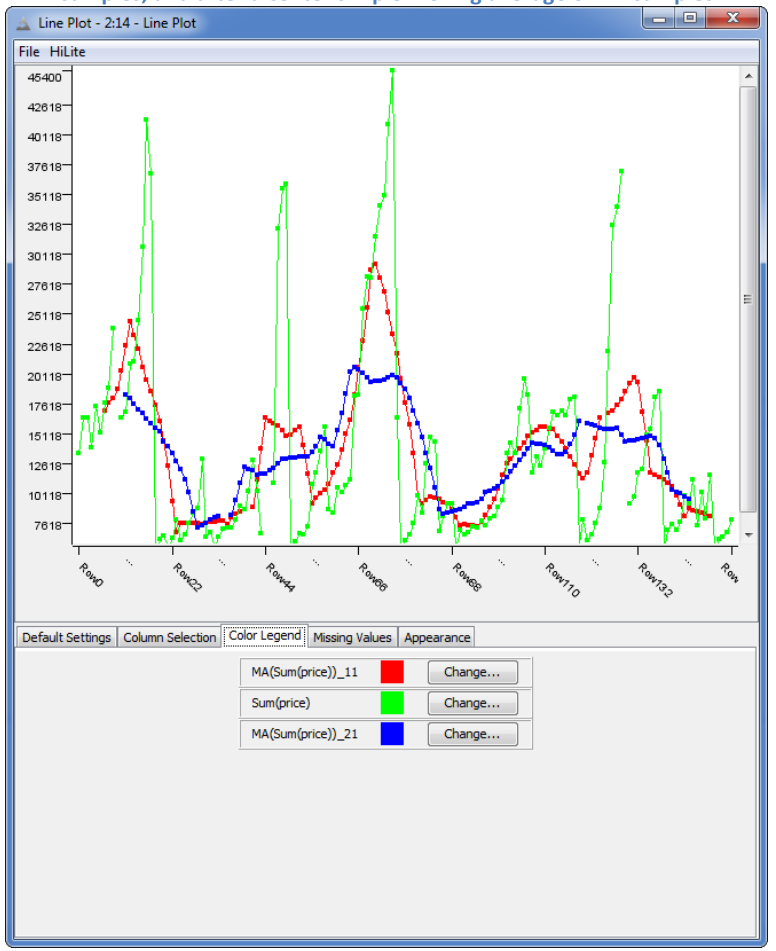
3.27. Exercise 2: The workflow



3.28. Settings for the "Time Difference" node



3.29. Plots of the original time series, after a center simple moving average on 11 samples, and after a center simple moving average on 21 samples



Chapter 4. Workflow Variables

4.1. What is a Workflow Variable?

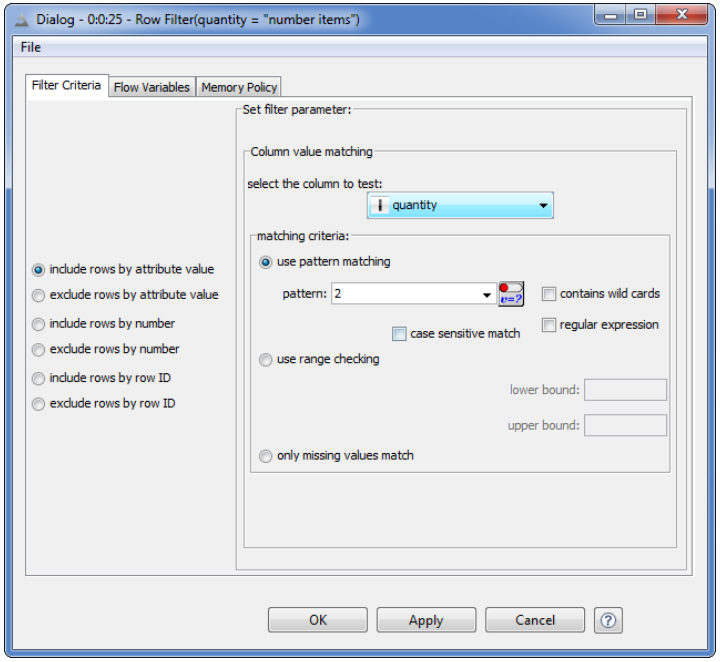
In KNIME it is possible to define external parameters to be used throughout the entire workflow: these parameters are called “Workflow Variables” or shortly “Flow Variables”. Workflow variable values can be updated before or during each workflow run via dedicated nodes and commands; the new values can then be used to parametrically set any node configuration settings.

Let’s create a workflow group called “Chapter4” now to host the workflows implemented in the course of this chapter. Let’s also create an empty workflow named “Workflow_Vars” as its first example workflow. First thing, we read the “sales.csv” file from the Download Zone, then we convert the “date” column from type “String” to type “DateTime”, and finally we apply the “Extract Time Window” node to filter the rows with “date” between “01.Jan.2009” and “01.Jan.2011”. The final data set simulates the same sales archive described in section 2.2.

We now want to find all those sales where a given number of items have been sold. For example, if we look for all sales, where 2 items have been sold, we just use a “Row Filter” node using “quantity = 2” as a filtering criterion (Fig. 4.1).

Let’s suppose now that the number of sold items is not always the same. Sometimes we might want to know which sales sold 3 items, sometimes which sales sold 10 items, sometimes just which sales sold more than *n* items, and so on. In theory, at each run I should open the “Row Filter” node and update the filtering criterion. But this is a very time consuming approach which is not well suited to the workflow running on a KNIME server, for example.

4.1. “Row Filter” node’s settings to find the sales records where 2 items have been sold



We can parameterize the pattern matching the row filtering criterion by using a workflow variable. In this example of sale record selection based on a variable number of sold items, we could define a workflow variable as an integer with name “number items” and default value 2. We could then change the “Row Filter” node to implement the matching pattern of the filtering criterion on the basis of the workflow variable value rather than of a fixed

number. We would like to have a filtering criterion like `quantity = "number items"` rather than `quantity = 2`. At each workflow's execution, the value of the workflow variable can be changed in order to retrieve only those sale records with the specified number of sold items.

There are two ways to update the value of a workflow variable at each workflow run:

- The workflow variable value can be changed for the whole workflow before the run starts;
- The workflow variable value can be changed on the fly during workflow execution by means of dedicated nodes; the new value of the workflow variable subsequently becomes available for all subsequent nodes in the workflow.

The following sections explore the two options. That is how to create and update a workflow variable before each run and make it available for the whole workflow and how to create and update a workflow variable from inside the workflow.

4.2. Creating a Workflow Variable for the whole Workflow

In this section we create a workflow variable for the "Workflow_Vars" workflow. In order to implement the parameterized row filtering described in the previous section, the workflow variable is created of Integer type, with the name "number items", and a default value of 2.

To create a workflow variable, you need to:

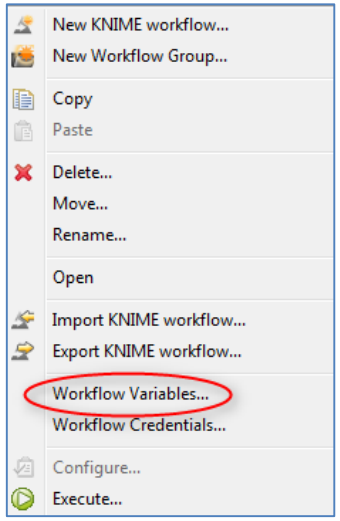
Go to the "Workflow Projects" list:

- Select the workflow to which the workflow variable belongs
- Open the workflow's context menu (right-click)
- Select the "Workflow Variables …" option
- A small window for the workflow variable administration opens. This window contains the list of workflow variables that already exist in this workflow, if any. Each workflow variable is described by three parameters: Name, Type, and Value.

In the "Workflow Variable Administration" window (Fig. 4.3), there is:

- A list of workflow variables
- Three buttons:

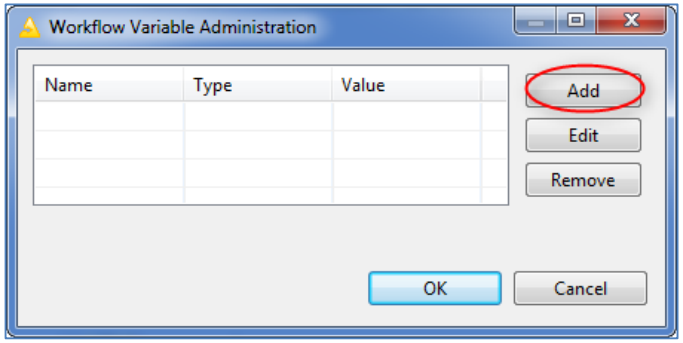
4.2. Option "Workflow Variables" in workflow's context menu



- The “Add” button introduces a new workflow variable
- The “Edit” button allows the three parameters for the selected workflow variable to be modified
- The “Remove” button deletes the selected workflow variable

- Click the “Add” button to create a new workflow variable.
- The “Add/Edit Workflow Variable” window opens (Fig. 4.4).

4.3. “Workflow Variable Administration” window



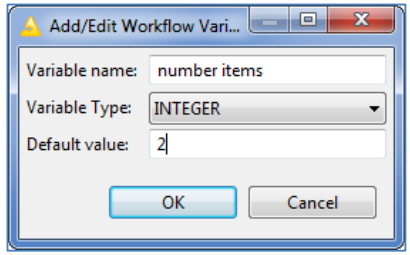
In the “Add/Edit Workflow Variable” window:

- Define the three parameters for the new workflow variable: name, type, and default value.
- Click “OK”.

Back in the “Workflow Variable Administration” window:

- The newly defined workflow variable has been added to the list of workflow variables.
- Click “OK”, to complete creation of the workflow variable. You will be prompted to reset all nodes of the workflow.

4.4. “Add/Edit Workflow Variable” window



Following these steps, we created a workflow variable named “number items”, of Integer type, and with a default value of 2.

Note. This workflow variable can now be accessed by every node, but only throughout the “Workflow_Vars” workflow.

4.3. Workflow Variables as Node Settings

Once a workflow variable is created, we need to configure the nodes of the workflow so that they can use it. In our example workflow named "Workflow_Vars", the "number items" workflow variable has to become the matching pattern in the filtering criterion of the "Row Filter" node.

In the configuration window of some nodes, a "Workflow Variable" button (Fig. 4.5) is displayed on the side of some of the settings. An example of where you can find the "Workflow Variable" button is the "Row Filter" node, where it is located in the "matching criteria" panel for the "use pattern matching" option, to the right of the "pattern" textbox (Fig. 4.1).

The "Workflow Variable" Button

The "Workflow Variable" button allows you to use the value of a workflow variable for a particular node setting.

By clicking the "Workflow Variable" button, the "Variable Settings" window opens and, for this particular setting, asks whether:

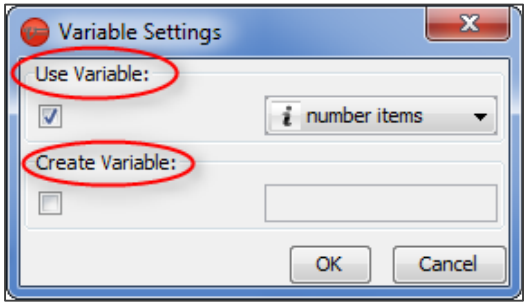
- An already existing workflow variable should be used:
  - Enable the "Use Variable" flag
  - Select one of the exiting flow variables from the enabled menu

  OR
- A new workflow variable should be created:
  - Enable the "Create Variable" flag
  - Provide a name for the new flow variable
  - A new flow variable with that name is then created and made available to all subsequent nodes in the workflow

The value of the selected workflow variable defines the node setting at execution.

4.3. The "Workflow Variable" button

4.4. The "Variable Settings" window



In the workflow “Workflow_Vars”, we selected the existing workflow variable “number items”, created in section 4.2, to act as the pattern matching in a “Row Filter” node. This “Row Filter” node has been named “quantity = "number items" via “pattern matching” criterion”.

You can check which workflow variables are accessible by each node by:

- Opening the Table View option (last option in the context menu)
- Selecting the “Flow Variables” tab

The “Flow Variables” tab contains the full list of the workflow variables, including their current values that are available for that node.

Not all the configuration settings display a “Workflow Variable” button. The “Workflow Variable” button has been implemented for only some settings of some nodes. Thus, a more general way to set the value of a configuration setting with a workflow variable involves the “Flow Variables” tab in the node’s configuration window.

The “Flow Variables” Tab in the Configuration Window

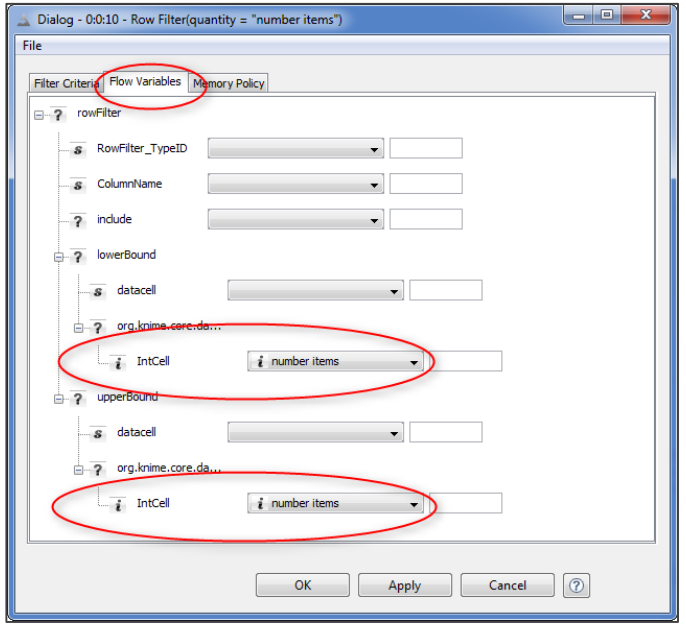
- Open the configuration window of a node
- Select the “Flow Variables” tab

The “Flow Variables” tab allows each of the node configuration settings to be overwritten with the value of a workflow variable.

- Select the desired configuration setting
- Select the workflow variable you want to use from the list of available workflow variables
- Click “OK”

Now, at execution time, the node uses the value of the selected workflow variable for that setting.

4.5. The “Workflow Variable” button



In the workflow “Workflow_Vars”, we performed the same row filtering as before (i.e. column “quantity” = flow variable value “number items”) by using the “use range checking” filter criteria with “lower bound” = “upper bound” = flow variable “number items”. This should implement exactly the same type of row filtering as by using the pattern matching filtering criterion and the “number items” workflow variable as pattern to match.

Since the configuration settings, named “lower bound” and “upper bound”, do not have a “Workflow Variable” button, we need to define their values via the “Flow Variables” tab. In the “Flow Variables” tab in the configuration window of a new “Row Filter” node, we expanded the top category called “rowFilter”, followed by the “lowerBound” category, and finally the “org.knime.core.da…” category. There we set the value of “IntCell” parameter to take the value of the “number items” workflow variable via the combobox on its right. We repeated the same sequence of operations to set the value of parameter “IntCell” in the “upperBound” category (Fig. 4.7). This “Row Filter” node has been named “quantity = "number items" via “range checking” criterion”.

After defining the value of a configuration setting by means of a workflow variable, a warning message, characterized by a yellow triangle, appears at the bottom of the configuration window. The simple goal of this warning message is to warn the user that this particular parameter is controlled by a workflow variable and therefore changing its value manually will be ineffective. In fact, if a parameter value has been set by means of a workflow variable, this workflow variable will always override the current value of the parameter at execution time.

In order to make the manual change of the parameter value effective, the user needs to disable the workflow variable setting for this parameter. This can be obtained by either disabling both options in the “Variable Settings” window (Fig. 4.6) or by selecting the first empty option in the combobox menu in the “Flow Variables” tab of the configuration window (Fig. 4.7).

Note. In some rare cases, it is still possible that some settings of some nodes might not be reachable even with the “Flow Variables” tab, like for example some settings of the configuration window of the “Time Series” nodes.

4.4. Creating a Workflow Variable from inside a Workflow

Let’s change the problem now and suppose that we do not know ahead which rows to keep and that the row filter pattern becomes known inside the workflow. We then need to create a new flow variable inside the workflow as soon as the filtering patterns become known. There are two ways to proceed with this: transform a data value into a flow variable or transform a configuration setting into a flow variable. All nodes dealing with Flow Variables can be found in category “Flow Control” -> “Variables”.

Transform a Data Value into a Flow Variable

Let's imagine that we want to analyze only the sale records of the country with the highest total number of sold items. To find out the total number of sold items for each country, in the "Workflow_Vars" workflow, we connected a "GroupBy" node to the output port of the "Extract Time Window" node. The "GroupBy" node was set to calculate the sum of values in the "quantity" column by country. A "Sorter" node sorted the resulting aggregations by the "sum(quantity)" column in descending order. The country reported in the first row of the final data table is the country with the highest number of sold items. Then a "Row Filter" node should retain all data rows where "country" is the country with the highest number of sold items.

Depending on the selected time window, we do not know ahead which country has the highest number of sold items. Therefore, we cannot define a workflow variable with that country name before running the workflow. On the other hand, we cannot set the "Row Filter" node with a fixed country name as the matching pattern because the country with the highest number of sold items might change for each workflow run. We should be able to find the value of interest, transfer it from the data domain into the workflow variables domain, and then use it to run the "Row Filter" node. The node, that transfers the data table cells into an equal number of workflow variables, is the "TableRow To Variable" node.

The "TableRow To Variable" node is located in the category "Flow Control" -> "Variables". The "Flow Control" category contains a number of sub-categories with nodes that help to handle the data flow in the workflow. In particular, the sub-category "Variables" contains a number of utilities to deal with workflow variables from inside the workflow, such as to transform data cells into variables and vice versa.

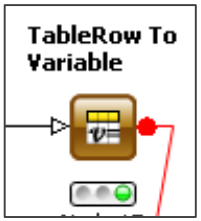
TableRow To Variable

The "TableRow To Variable" node takes a data table on the input port (white triangle), transforms the data cells into workflow variables, and places the resulting set of workflow variables on its output port (red circle).

The "TableRow To Variable" node is located in the category "Flow Control" -> "Variables".

No configuration settings are required for this node.

4.6. The "TableRow To Variable" node



Note. The output port of the "TableRow To Variable" node is a red circle. Red circle ports deal with workflow variables either as input or as output.

Going back to the “Workflow_Vars” workflow, we isolated the “country” cell of the first row in the output data table of the “Sorter” node, by using a “Row Filter” node to keep the first row only and a “Column Filter” node to keep the “country” column only. The resulting data table had only one cell with the name of the country with the highest sold items. This one-cell data table was then fed into a “TableRow To Variable” node and transformed into a workflow variable named “country”.

The workflow variable view of the “TableRow To Variable” node is opened by selecting the last item of its context menu. This view shows the list of workflow variables made available to the subsequent nodes and includes all previously defined workflow variables plus all the workflow variables created by the “TableRow To Variable” node itself.

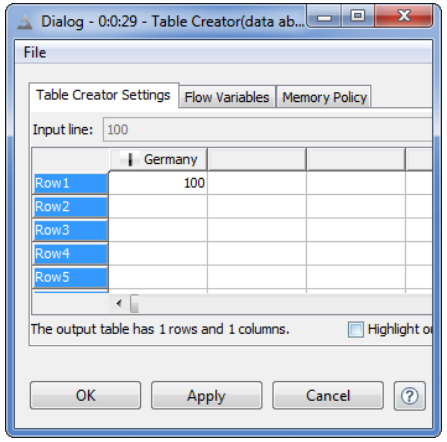
Transform a Configuration Setting into a Flow Variable

Let’s suppose now to have a table with some values about a specific country. We want to synchronize the workflow data coming from the database with the country represented in this table. We simulated the table with values about a specific country with a “Table Creator” node with just one column of value for the desired country. The country name is reported as the column name. We would like then to use exactly that column name as a pattern for a “Row Filter” node. We can actually transfer the value of a configuration setting in any node to the value of a new flow variable.

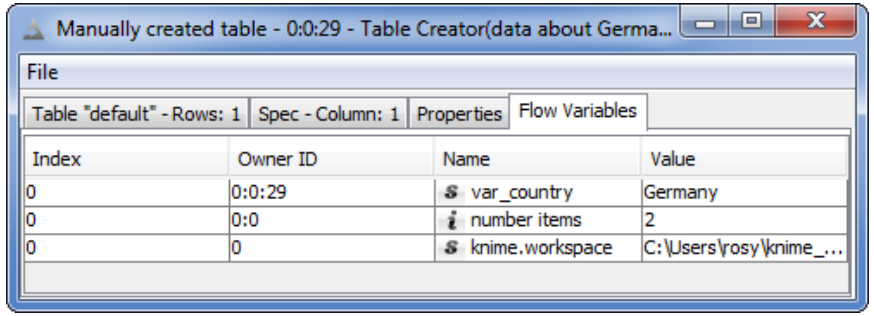
In fact, in the “Flow Variables” tab in each configuration window, close to each setting name there are two boxes. One box we have already seen and it is used to assign the value of an existing flow variable to the configuration setting (Fig. 4.5). The other box, the last one on the right of each row, is a textbox and implements the opposite pass; that is, it creates a new flow variable named according to the string in the texbox and assigns the value of the corresponding configuration setting to this new flow variable.

In the “Workflow_Vars” workflow, in the “Flow Variables” tab of the configuration window of the “Table Creator” node, we assigned the value of the configuration setting “ColumnName” of column # 0 to a new flow variable named “var_country” (Fig. 4.8). If we now check the output data table of the “Table Creator” node and we select the “Flow Variables” tab, we see a new flow variable named “var_country” and with value “Germany”, exactly the value of the only column defined in the “Table Creator” node.

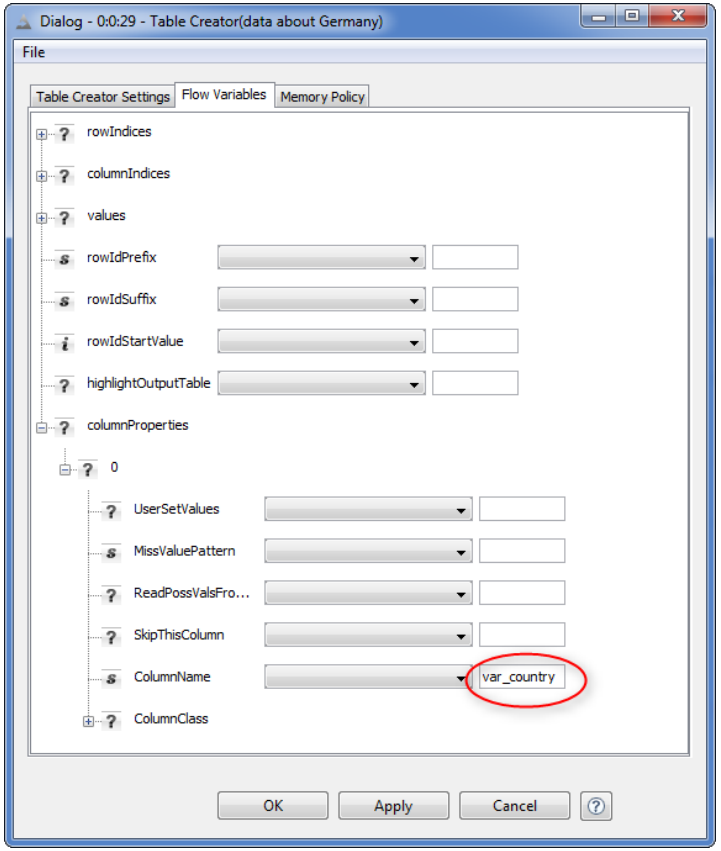
4.7. The "Table Creator" node used in the "Workflow_Vars" workflow



4.8. Output Flow Variables of the "table Creator" node in the "Workflow_Vars" workflow



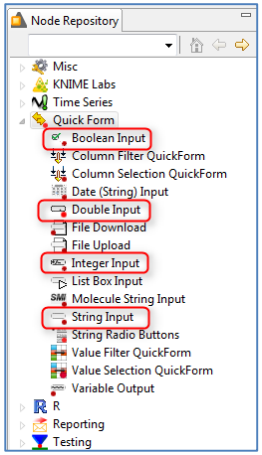
4.9. The "Table Creator" node used in the "Workflow_Vars" workflow



Quickforms to Create Flow Variables

Another way to create a flow variable in the middle of a workflow is to use a Quick Form node. Quick Form nodes are located in the category “Quick Form” and provide interactive forms for a variety of tasks (see section 4.7 below). One of these tasks is the creation and the value assignment of a flow variable. There are many Quick Form nodes that can implement this task, allowing for the creation of any flow variable type and using more or less complex interactive interfaces for its value definition. The simplest Quick Form nodes to generate flow variables are the “Boolean Input” node, the “Double Input” node, the “Integer Input” node, and the “String Input” node.

4.10. The simplest "Quick Form" nodes to generate flow variables



All these nodes just create a workflow variable with a specific name and a specific value. Each node though outputs a flow variable with a specific type. Since they do no processing, they need no input. Therefore, these nodes have no input and just one output port of flow variable type (red circle). They also share the same type of configuration window, requiring: the name and the default value of the flow variable to be created, optionally some description of what the flow variable has been created for, and an orientation label to help the user during the assignment of new values.

Let's have a look again at the example workflow "Workflow_Vars" and particularly at the "Row Filter" node where data rows with "quantity" equal to the value of the flow variable "number items" are selected by using the range checking criterion. A flow variable like "number items", containing the number of sold items for the filtering criterion, can also be created from inside the workflow with a Quick Form node. This flow variable must be of type Integer to match the values in data column "quantity". Therefore, we can use an "Integer Input" node from the "Quick Form" category.

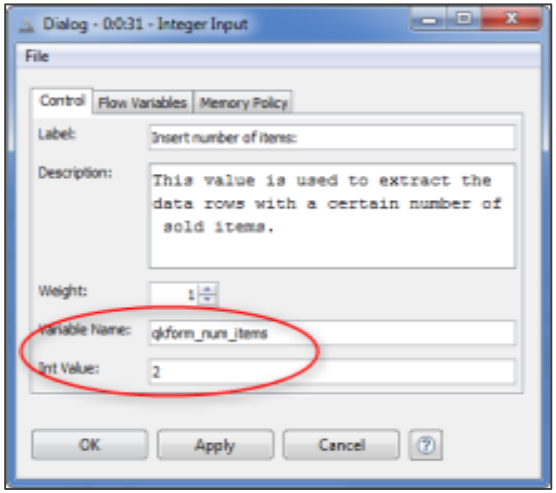
Integer Input

The "Integer Input" node creates a new flow variable of type Integer, assigns a default value to it, adds it to the list of already existing flow variables, and presents it at the output port.

Its configuration window requires:

- The flow variable name
- The flow variable default value
- An optional quick description of the purpose of this flow variable
- A label that will be used as a help for the user when updating the flow variables with new values
- A weight value to be used for the node configuration on the KNIME server

4.11. Configuration window of the "Integer Input" node



The “Integer Input” node then produces a flow variable of type Integer, named “qkform_num_items”, with default value 2.

The “Boolean Input” node, the “Double Input” node, and the “String Input” node are structured the same way as the “Integer Input” node, with the only difference that they produce flow variables of type Boolean, Double, and String respectively. For example, if we now want to write the results of the filtering operation to a csv file and if we want to parameterize the output file path by using a flow variable of type String, we could use a “String Input” node.

4.5. Injecting a Flow Variable into a Workflow Branch

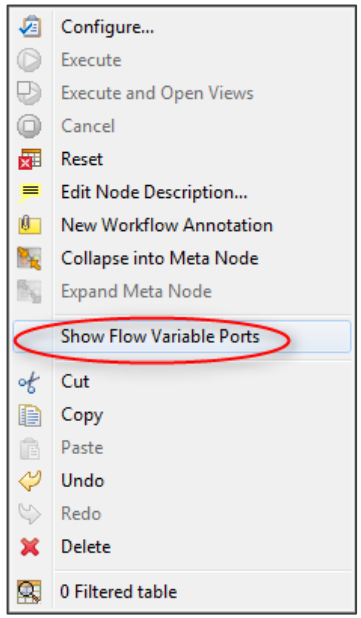
Inject a Flow Variable via the Flow Variable Ports

At this point, in our example workflow, we have three workflow variables: “number items” which is a global flow variable; “country” which contains the name of the country with the highest number of old items; and “var_country” which contains the name of the country contained in the reference table from the “Create Table” node.

Let’s concentrate for now on the variable “country”. We want to keep all data rows related with the country where most items had been sold. This flow variable is at the output of the “TableRow To Variable” node. How do we make a “Row Filter” node aware of the existence of this workflow variable? How do we connect the “TableRow To Variable” node to the “Row Filter” node? We need to insert, or inject, the new updated list of flow variables back into the workflow, in order to make the new flow variables available to the “Row Filter” node and all subsequent nodes.

All KNIME nodes have visible data ports and hidden workflow variable ports. In the context menu of each node, the “Show Flow Variable Ports” option makes the hidden flow variable ports visible (Fig. 4.10). We use these flow variable ports to inject flow variables from one node to the next or from branch to another. Notice that there are two flow variable ports: one port on the left to import flow variables from other nodes and one port on the right to export flow variables to other nodes.

4.12. Option “Show Flow Variable Ports” in the context menu of a node



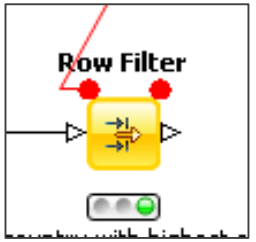
Workflow Variable Injection into the Workflow

When the option “Show Flow Variable Ports” is active, the node shows two red circle ports on the top. The port on the left takes a list of workflow variables as input and the port on the right exposes the list of workflow variables available for this node.

In order to force, i.e. inject, a list of workflow variables into a node:

- Show the workflow variable ports via the context menu,
- Connect the output variable port of the preceding node to the input variable port (i.e. the left one of the two workflow variable ports) of the current node.

4.13. The workflow variable ports of a node



The list of flow variable made available to following nodes can be inspected by opening the output data table (last item in node context menu) and selecting the tab named “Flow Variables”. After the injection of new flow variables, the flow variable list should show include the newly created flow variables as well as the flow variables that existed beforehand.

Note. The flow variables injected into a workflow’s node are only available for this and the connected successor nodes in the workflow. Nodes cannot access workflow variables injected later on in the workflow.

In the “Workflow_Vars” workflow, we first introduced a “Row Filter” node after the “TableRow To Variable” node; next, we displayed its workflow variable ports; finally we connected the flow variable output port of the “TableRow To Variable” node to the input variable port (i.e. the flow variable port on the left) of the “Row Filter” node. As a result, the flow variable named “country” became part of the flow variables available to the “Row Filter” node and to the following nodes in the workflow. Finally, we configured the “Row Filter” node to use the pattern matching filter criterion and the workflow variable “country” as the matching pattern.

However, we might want both flow variables, “country” and “var_country”, to be available to a “Row Filter” node. In this case, the input flow variable port is not enough to inject two flow variables at the same time. KNIME has a node, named “Merge Variables” to merge list of flow variables from different branches of a workflow.

Merge Variables

The “Merge Variables” node only merges flow variables into one stream. This node makes available flow variables defined in branches of the workflow to the following nodes.

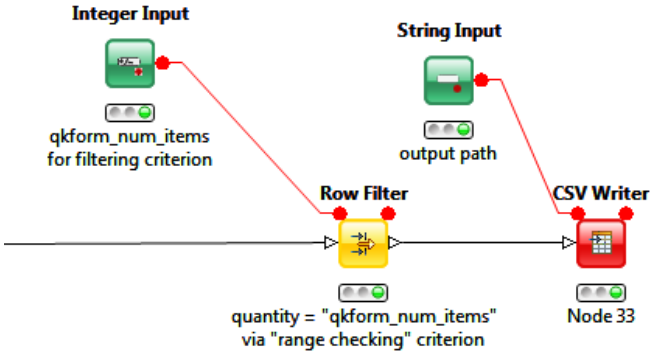
If flow variables with the same name are to be merged, the standard conflict handling is applied: most top inputs have higher priority and define the value of the post-merging variable.

This node can also be used as a barrier point to control the execution order of nodes, i.e. nodes connected to the output port will not start executing until all upstream nodes have been executed.

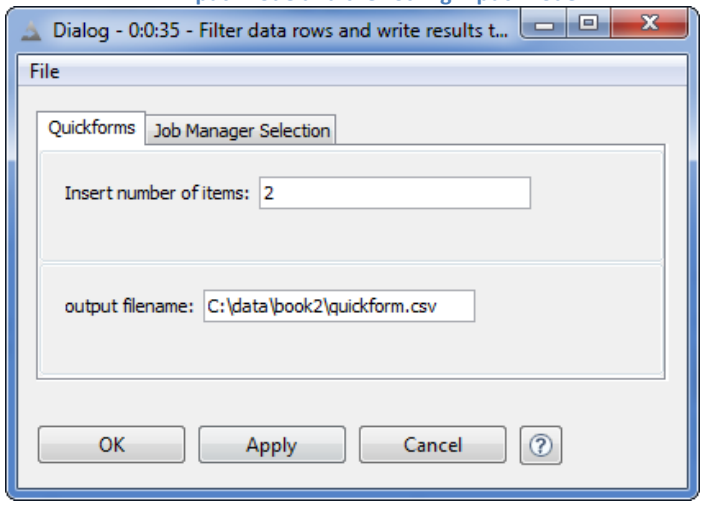
This node needs no configuration settings.

Quickforms, Meta-nodes, and KNIME Server

4.14. "Integer Input" and "String Input" nodes used to define the row filtering criterion and the output file path



4.15. Configuration Window of the meta-node containing the "Integer Input" node and the "String Input" node



In theory we can parameterize everything using Quick Form nodes. Once the workflow though becomes too crowded, we can isolate the part with the flow variable definition by means of Quick Form nodes into a meta-node.

One of the features that make Quick Form nodes so attractive is the transmission of the textboxes for the value definition to the meta-node that contains them, if any. In fact, if a meta-node contains Quick Form nodes, it acquires a configuration window; that is the option “Configure” in the meta-node context menu becomes active. The configuration window is then filled with the textboxes for the flow variables values required by the underlying Quick Form nodes. Collapsing the nodes shown in figure 4.14 into a meta-node, produces a meta-node with a configuration window requiring the value for the output file path and the value for the flow variable “qkform_num_items” (fig. 4.15).

On the KNIME Server Quick Form nodes produce a step-wise execution. If a workflow containing Quick Form nodes is uploaded on the KNIME Server, the workflow is executed step by step till a new Quick Form node has been reached. There the workflow stops and shows a parameter setting window for the Quick Form node’s flow variable value. It might be good design practice to combine all Quick Form nodes in a meta-node at the very beginning of the workflow. In this case only one parameter setting form shows up at the beginning of the execution for the definition of the flow variable values for this particular workflow run. You can even move around with the textbox positions in the parameter setting window by playing with the “weight” parameter in the configuration window of each Quick Form node.

Transform a Flow Variable into a Data Value

Sometimes it is required to save the list of all parameter values under which a workflow is running, in order to know exactly how the resulting data has been obtained. To transform workflow variables into data cells, we can use the “Variable To TableRow” node. The “Variable To TableRow” node mirrors the “TableRow To Variable” node. That is, it transforms a number of selected flow variables into cells of a data table.

Variable To TableRow

The “Variable To TableRow” node transforms selected workflow variables at the input port (red circle) into cells of a data table at the output port (white triangle).

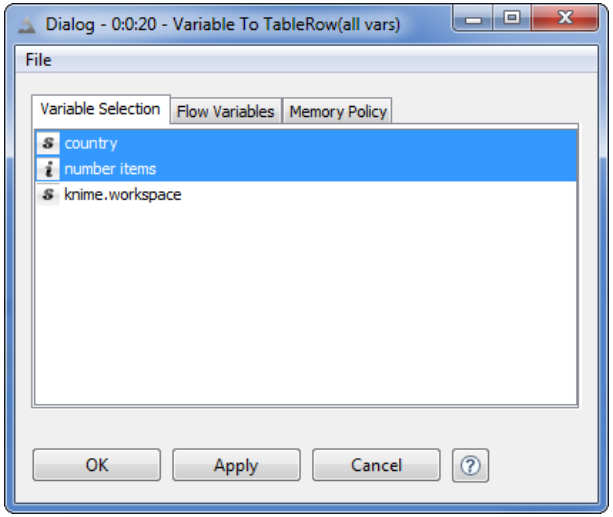
The “Variable To TableRow” node is located in the category “Flow Control” -> “Variables”.

The configuration settings require the selection of the workflow variables to be included in the output data table. Multiple selections (with Ctrl-click and Shift-click) are allowed.

The output data table contains two columns:

- One column contains the name of the flow variable;
- The other column contains the flow variable value at execution time.

4.16. The configuration window of the “Variable To TableRow” node



In our example workflow, “Workflow_Vars”, we connected a “Variable To TableRow” node to the “TableRow To Variable” node, in order to save the values of all flow variables under which the workflow was executed.

4.6. Editing Flow Variables

In the country with the highest number of sales, we now want to restrict the analysis scope to only those data records with a high value in the “amount” data column. This requires an additional “Row Filter” node implementing a filtering criterion “amount” > x, where x is the threshold above which an amount value is declared high.

Let’s suppose that different thresholds are used for different countries. For example, an amount is considered high for a sale in USA if it lies above 500; it is considered high in other countries if it lies above 100. In one case, we set x = 500 and in the other cases x = 100. Therefore, the threshold x cannot be set as the static value of a global workflow variable. Indeed, the workflow variable value needs to be adjusted as the workflow runs. The “Java Edit Variable” nodes have been designed precisely to change the value of a workflow variable from inside the workflow.

The “Java Edit Variable” nodes, like the “Java Snippet” nodes, execute a piece of Java code. Unlike the “Java Snippet” nodes though, they work exclusively on flow variables: they take the list of available flow variables at the input port, process them, and produce one or more additional flow variables at the output port. These new flow variables produced by the “Java Edit Variable” nodes can either override existing flow variables or write into newly created flow variables. The “Java Edit Variable” nodes have red circles as input and output ports; i.e. they take a number of flow variables at the input port and re-present the same widened list of flow variables at the output port.

The “Java Edit Variable” nodes follow the same structure used for the “java Snippet” nodes. They include two nodes: the “Java Edit Variable (simple)” node for less expert Java programmer and the “Java Edit Variable” node for more advanced Java programmers. The “Java Edit Variable” nodes are located in the category “Flow Control” -> “Variables”.

In the “Workflow_Vars” workflow, a piece of Java code has been implemented to create a threshold value “x” as a new flow variable and to change its value according to the value of the existing flow variable “country”. This task was carried on by two “Java Edit Variables” nodes: once with a “Java Edit Variable (simple)” node and once with a “Java Edit Variable” node. Both nodes were introduced after the “TableRow To Variable” node with the following code:

```
int quantity = 100;
String s = new String($${Scountry}$$);
if (s.equals("USA"))
quantity = 500;
return quantity;
```

This code defines a default value for the amount threshold of 100 and changes it to 500 if the country with the highest number of sales is USA.

A second “Row Filter” node was then connected to the first “Row Filter” node, the one that isolates all data rows belonging to the country with the highest number of sales. The second “Row Filter” node goes through all these data rows and filters out those whose amount is smaller than x, where x is sometimes 500 (if country = “USA”) and sometimes 100 (for all the other countries). The threshold x was taken from the workflow variable produced by one of the two “Java Edit Variable” nodes.

Java Edit Variable (simple)

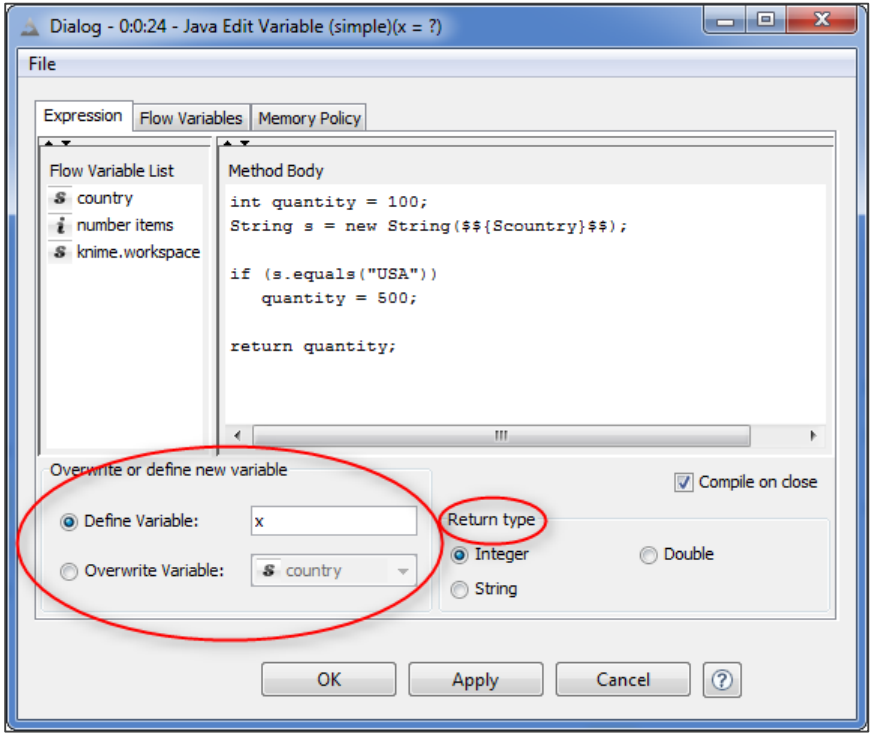
The “Java Edit Variable (simple)” node changes the value of an existing flow variable or creates a new flow variable with a specific value by means of some Java code.

It works similarly to the “Java Snippet (simple)” node, but while the “Java Snippet (simple)” node operates on data tables the “Java Edit Variable (simple)” node operates on flow variables.

The configuration window of the “Java Edit Variable (simple)” node consists of three panels.

- The “Method Body” panel in the center part of the configuration window contains the Java editor to write the required Java code.
- The “Flow Variable List” panel on the left presents the list of available workflow variables: double-clicking a workflow variable in the list inserts it automatically in the Java editor.

4.17. Configuration window of the “Java Edit Variable” node



- The bottom panel defines the type of the returned flow variable on the right and its name on the left. In the left bottom part, there is a textbox for the name either of the existing flow variable or of the new flow variable, depending on whether the option “Overwrite Variable” or “Define Variable” is selected.
- In the “**Global Variable Declaration**” panel global Java variables can be created, to be used recursively in the code.

Note. The type of the returned workflow variable has to be consistent with the return type of the Java code.

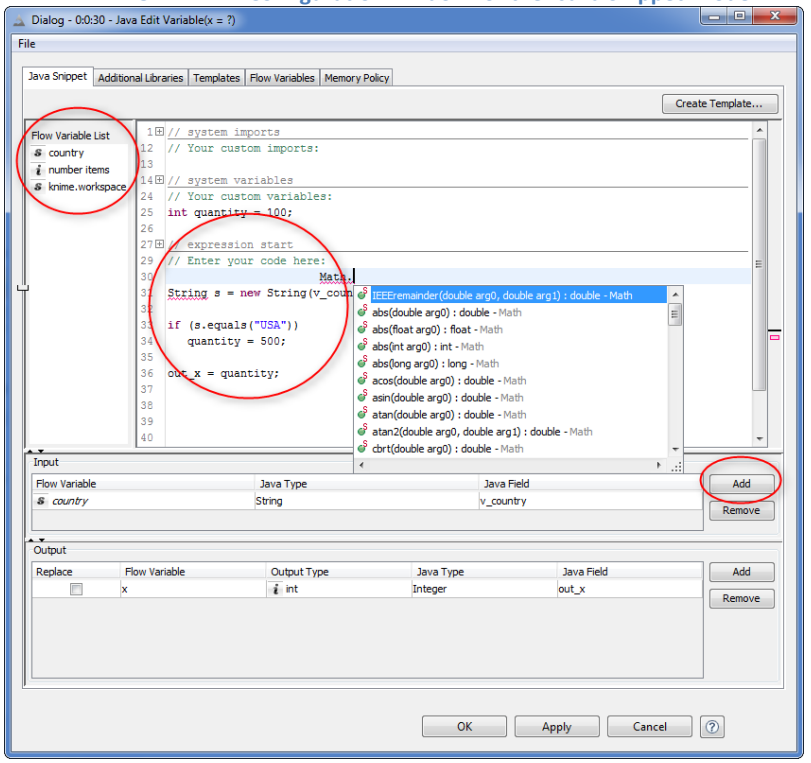
Java Edit Variable

Like the “Java Edit Variable (simple)” node, the “Java Edit Variable” node processes flow variables by means of a piece of Java code and places the result values into new or existing flow variables.
The configuration window of the “Java Edit Variable” node contains:

- The **Java editor.** This is the central part of the configuration window and it is the main difference with the “Java Edit Variable (simple)” node. The editor has sections reserved for: variable declaration, imports, code, and cleaning operations at the end.
    - The **“expression_start”** section contains the code.
    - The **“system variables”** section contains the global variables, those whose value has to be carried on row by row. Variables declared inside the “expression_start” section will reset their value at each row processing.
    - The “**system imports**” section is for the library import declaration.

  Self-completion is also enabled, allowing for an easier search of methods and variables. One or more output variables can be exported in one or more new or existing output data columns.

4.18. Configuration window for the “Java Snippet” node

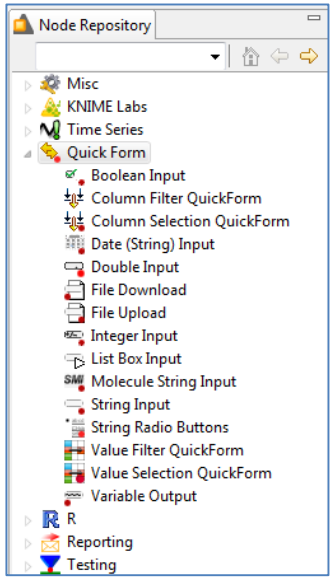


- The **table named “Input”.** This table contains all the available input flow variables. Use the “Add” and “Remove” buttons to add input flow variables to the list of variables to be used in the Java code.
- The **list of flow variables** on the top left-hand side. The flow variables names can be used inside the Java code. After double-clicking the flow variable name, the corresponding Java variable appears in the Java editor and in the list of input variables on the bottom. Java variables carry the type of the original flow variable, i.e.: Double, Integer, String and Arrays. Their type though can be changed (where possible) by changing the field “Java Type” in the table named “Input”.
- The **table named “Output”**. This table contains the output flow variables to be created as new or to be replaced by the new values. To add a new output flow variable, click the “Add” button. Use the “Add” and “Remove” button to add and remove output flow variables. Enable the flag “Replace” if the output flow variable is to override an existing one. The output flow variable type can be “Integer”, “Double”, or “String”. If the flow variable type does not match the type of the Java variable that is being returned, the Java code will not compile. It is also possible to **return arrays** instead of single Java variables, just by enabling the flag “Array”. Remember to assign a value to the returned Java variables in the Java code zone.

Note. Even though the “Java Edit Variable” nodes have an input port for flow variables, this port does not need to be connected. The “Java Edit Variable” nodes also work without input, which makes it an ideal node to start a workflow without reading data.

4.7. Quickforms

4.19. The "Quick Form" category



There is a whole part about flow variable management that deserves a dedicated section: the Quickforms. Quickforms in general implement forms to insert or select values. Among the tasks they fulfill, there is the creation and the subsequent value definition for flow variables, as described in the section “Quickforms to Create Flow Variables” above.

Quickform nodes can create flow variables in many forms and shapes: Boolean, integer, double, and string. However, they can also be used for other tasks where some interaction is required, like file download/upload, definition of a menu for controlled value selection, parametric column filtering and selection, and so on. Basically they provide basic forms to define any flow variable you need and a few more complex interfaces for other tasks. There are so many Quickform nodes to fill a whole category named “Quick Form”.

We are not going to explore here all Quick Form nodes, for space reasons. However, we are going to just show a few examples of very useful Quick Form nodes, just to give you a small taste of how versatile they are and make you curious enough about the remaining ones.

In the “Workflow_Vars” workflow, we would like to select the data rows for a particular country. It would be really helpful if we could choose the country from a list displayed in a menu. This is possible with a “Value Selection Quickform” node.

Also if we want to write the final results to a file by using a “CSV Writer” node for example, it would help to be able to select the destination file from a upload/download file GUI instead of typing in the whole path with all the risk that typing path carries. This is possible by using the “File Upload” node.

Value Selection Quickform

The “Value Selection Quickform” node takes the values from a data column in the input data table and loads them in a menu that is displayed in the configuration window of the node itself, in the meta-node where the node has been possibly included, and in the step-wise execution of the server.

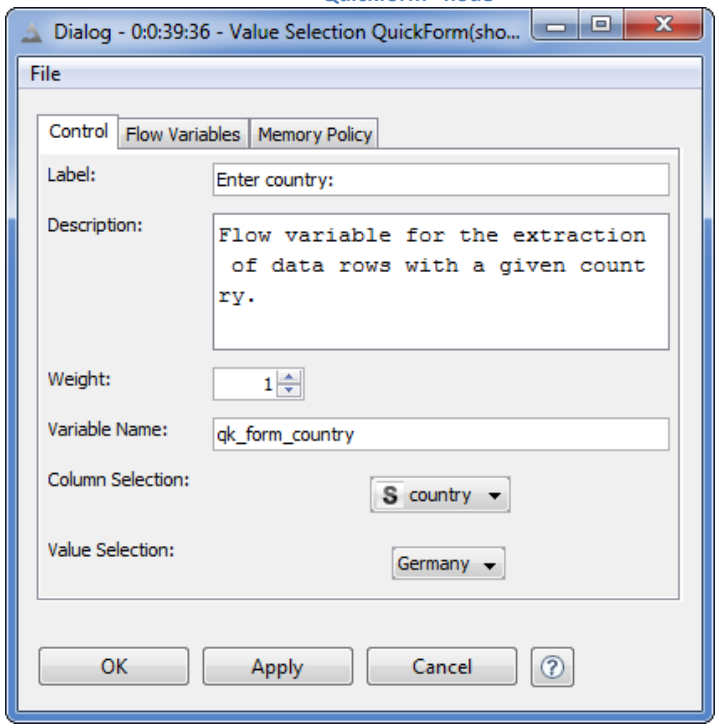
A value then need to be selected from this menu and a flow variable will transmit this value over to the subsequent part of the workflow.

To configure the node we need necessarily:

- The name of the flow variable
- The column name of the input data table to be used to extract all available options for the menu
- Once the column name is given, all possible options are uploaded on the “Value Selection” menu. The value to assign to the flow variable has to be selected from this menu

Optionally, it would help to have a description of what this flow variable has been created for, and a label that explains what is supposed to be selected.

4.20. The configuration window of the “Value Selection Quickform” node



Setting “weight” decides the position of the textbox in the GUI form of the server execution.

File Upload

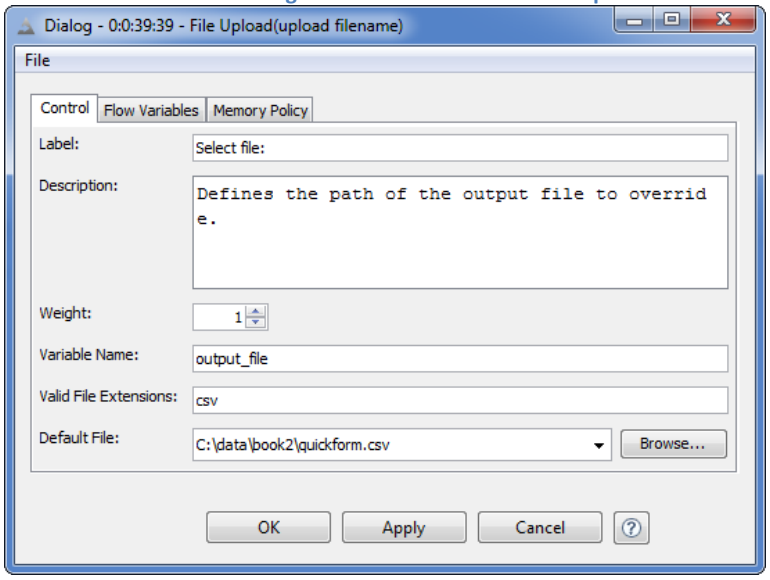
The “File Upload” node explores the folder of the default file path and loads all found files into a menu.

The menu is then triggered by the “Browse” button in the configuration window of the node itself, in the meta-node where the node has been possibly included, and in the step-wise execution of the server.

A file then need to be selected from this menu and a flow variable will transmit this value over to the subsequent part of the workflow.

Usually this node is connected to a reader node, like a “File Reader” node or to writer node, like a “CSV Writer” node, to define the file to be read or overridden.

4.21. The configuration window of the “File Upload” node



To configure the node we need necessarily:

- The name of the flow variable
- The default file path (it must be a valid path)

Optionally, the following information might help in displaying a more informative GUI form:

- The file extension to limit the number of files uploaded in the menu
- A description of what this flow variable has been created for
- A label that explains what is supposed to be selected
- Setting “weight” decides the position of the textbox in the GUI form of the server execution.

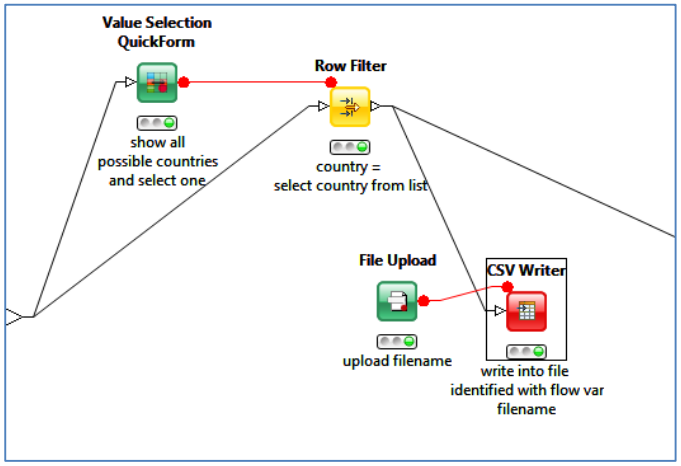
In our example, we used the “Value Selection Quickform” node to create a flow variable named “qk_form_country”. The data column “country” was uploaded to create the menu of possible values and a value for “qk_form_country” was then selected among all possible values displayed in the menu.

A “Row Filter” using this flow variable to select data rows for the specified country was then connected to the “Value Selection Quickform” via its flow variable ports.

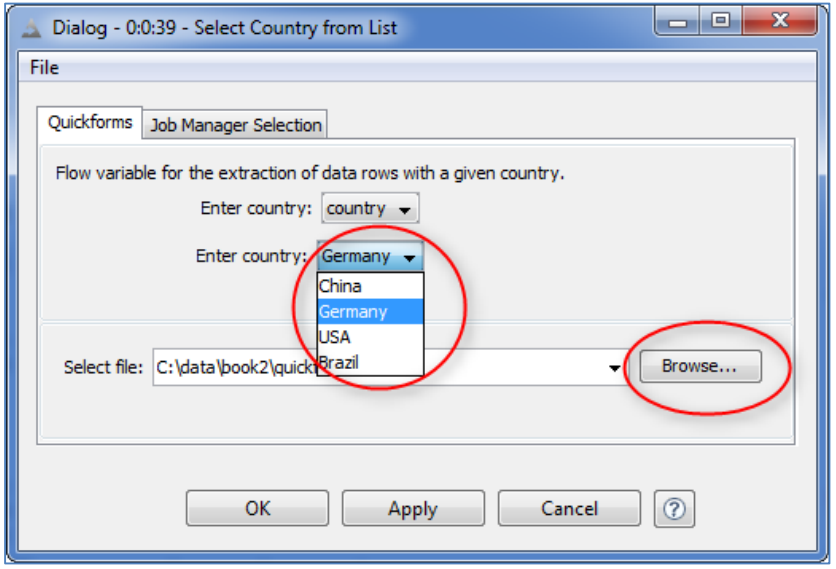
We then used a “File Upload” node to select a file among all files in a pre-defined folder. That is, we used the “File Upload” node to create a new flow variable “output_file” to contain the output file path. In the configuration window of the “File Upload” node we set a default file from folder “C:\data\book2”. The node then explored that folder and loaded all file names in the menu matching the extension “csv” that we provided. Now, when we want to assign a new value to the “output_file” flow variable, we click the “Browse” button and the list of all “csv” files contained in folder “C:\data\book2” is loaded. Selecting any of these files changes the content of the “output_file” flow variable. A “CSV Writer” node was connected to the “File Upload” node to override the file corresponding to the filename selected in the “Upload File” node.

The whole set of nodes was finally collapsed into a meta-node. Figure 4.22 shows the content of the meta-node with the two Quick Form nodes and figure 4.23 shows the configuration window of the same meta-node.

4.22. Meta-node content with the two Quick Form nodes

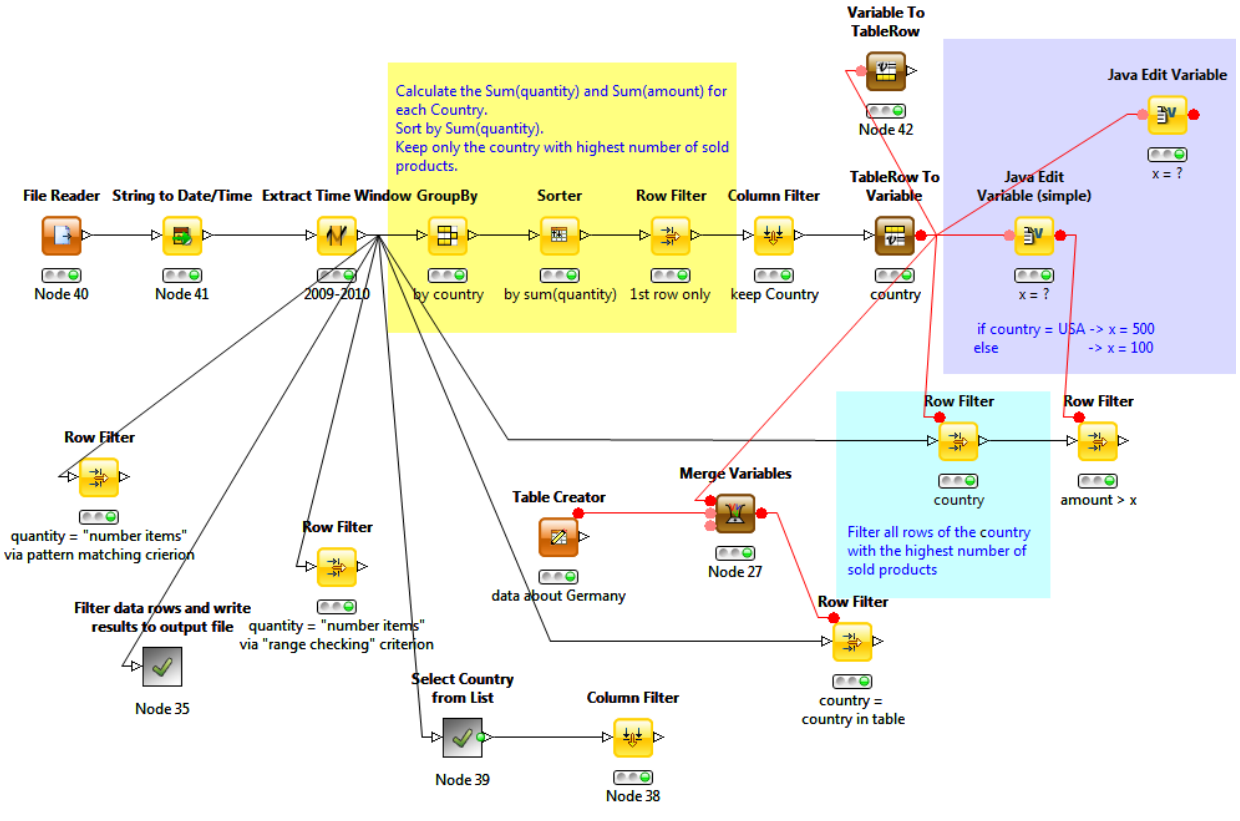


4.23. Meta-node configuration window



The final workflow is shown below.

4.24. Workflow “Workflow_Vars”



Above we have described two important Quickform nodes (“Value Selection Quickform” and “File Upload”). However, the “Quick Form” category contains many other useful nodes. For example, the “String Radio Buttons” node allows selecting one of two values in a UI mask with two radio buttons. The “Value Filter Quickform” node during the workflow execution presents a list of values in a menu and allows for the selection of one of them. The “ListBox Input” node allows for multiple selections from a list of values. The “Column Selection Quickform” node lets the user select some data columns and transmits the names of the data columns to the following nodes by means of a flow variable.

Finally, a number of Quick Form nodes are not dedicated to inject external values into the workflow, but rather to export the current data in a format compatible with the visualization on the Web Portal of the KNIME Server. The “TextArea Output” node, the “Image Output” node, and the “File Download” node all export data in a specified format to the Web Portal for visualization.

4.8. The “drop” Folder

How many times has it happen to you, that you import a new KNIME workflow into your workspace pretty easily and fast and then you spend a reasonable amount of time importing the data that the File Reader node is supposed to read? Well, this section is about an interesting trick to auto-configure a “File Reader” node, or any other reader node, by means of an automatically defined flow variable.

As you know, to each workflow and to each node corresponds a folder structure located in:

<WORKSPACE_FOLDER>/<WORKFLOW_GROUP>/<WORKFLOW>/<NODE>.

If you have a “File Reader” node in your workflow, then you will have a folder like:

<WORKSPACE_FOLDER>/<WORKFLOW_GROUP>/<WORKFLOW>/File Reader (#N)

where N is the progressive number assigned to your File Reader node.

We create now a new “File Reader” node in the “Workflow_Vars” workflow, configure it and execute it to read the intended input file. The File Reader path in the folder system, for example, is:

```
…\knime_2.x.z\book2_workspace\Chapter4\Workflow_Vars\File Reader (#40)
```

If we open this folder we see that it is populated by two xml files: node.xml and settings.xml. Let’s add now a folder named “drop” and let’s populate this folder with the text file read by the File Reader node. This triggers the creation of a flow variable containing the file path at the next node reset. Go back into the workflow editor of KNIME and reset the File Reader node.

Reopen the configuration window of the File Reader node, select the "Flow Variables" tab, open the setting named “DataURL”, and select the new flow variable "knime.node(drop)”. This flow variable contains the path to the file in the local “drop” folder. At this point the File Reader node is configured to automatically read the file in the drop folder. In fact, if you go back to the tab "Settings", you will see the message "The DataURL parameter is controlled by a variable".

Now reset the workflow, if you have executed it, and export it with the option "Exclude data from export" disabled. To export a KNIME workflow, right-click the workflow in the Workflow Projects panel and select "Export KNIME workflow".

If you reimport the workflow that you have just exported, you should get the same workflow, with the “drop” folder in the reader node's folder, with the input file in the “drop” folder, and with the reader node automatically configured to read this input file via the flow variable "knime.node(drop)”.

This trick is extremely useful to export data and workflows together without losing the configuration settings and it works in a similar way for all the other reader nodes as well.

4.9. Exercises

Exercise 1

Read the “sales” table from the “Book2” database, remove the “card” column and the rows with product = “prod_4”, and keep only the records with sale date in 2009 and 2010.

Select sale records for a specific month and year, for example, but not limited to, March 2009 and March 2010, and January 2009 and January 2010.

Produce a table of the sale records with:

- Column headers = products
- Row IDs = countries
- Cell content = sum(quantity) or sum(amount)

Countries \ products	Prod_1	Prod_2	Prod_3
Country 1	Sum(quantity/amount)	Sum(quantity/amount)	Sum(quantity/amount)
…	…	…	…
Country n	Sum(quantity/amount)	Sum(quantity/amount)	Sum(quantity/amount)

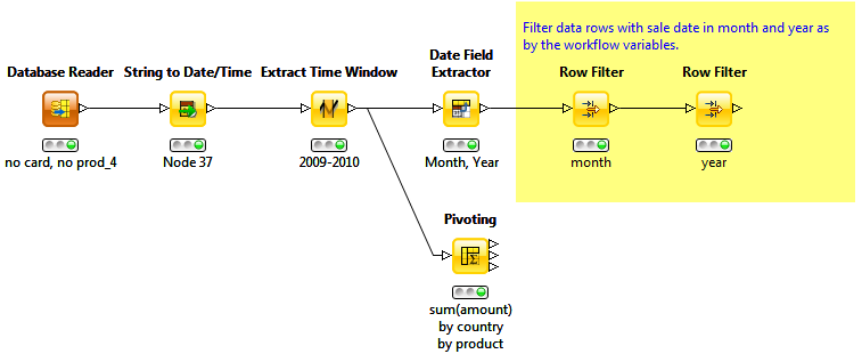
Switch between sum(quantity) and sum(amount) by using the workflow variables.

Solution to Exercise 1

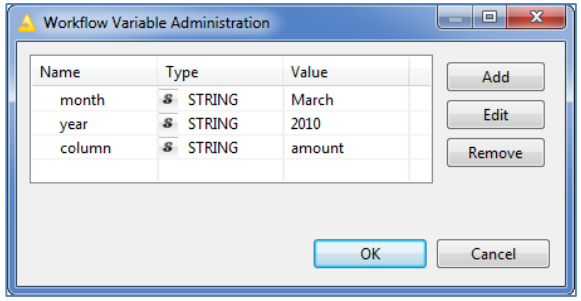
To implement an arbitrary selection of data records based on month and year, we applied a “Date Field Extractor” node and defined two workflow variables: one for the year and for the month. Two “Row Filter” nodes were then extracting the data rows with that particular month and that particular year as indicated by the two workflow variables’ values.

To implement the table above, we used a “Pivoting” node and we passed the name of the “aggregation column” with a workflow variable. The workflow variable, named “column”, could alternatively be “quantity” or “amount”. The “aggregation column” was set via the “Flow Variables” tab in the configuration window of the “Pivoting” node.

4.25. Exercise 1: Workflow



4.26. Exercise 1: Workflow Variables



Exercise 2

Using the file called “cars-85.csv” build a data set with:

- The cars from the car manufacturer that is **most** represented in the original data
- The cars from the car manufacturer that is **least** represented in the original data

Put your name and today’s date on every row of the new data set, to identify author and time of creation.

Solution to Exercise 2

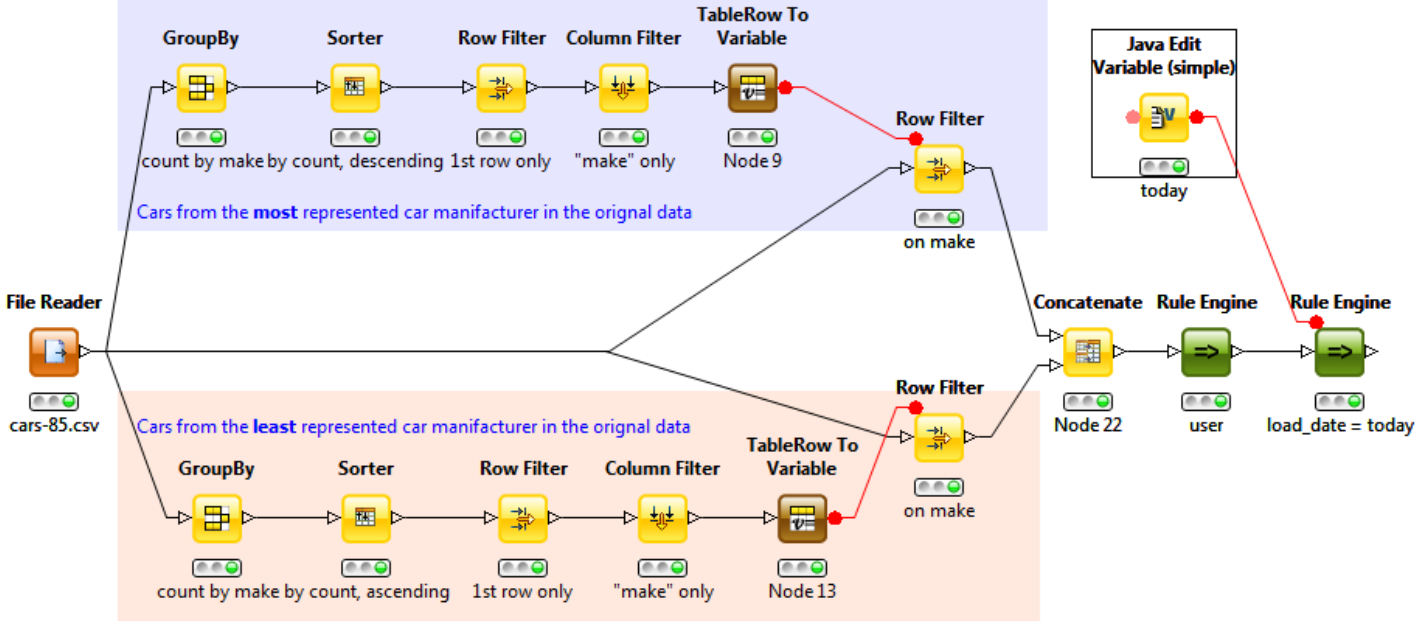
To identify the most/least represented car manufacturer in the data, we counted the data rows for each value in the “make” column with a “GroupBy” node; we sorted the aggregated rows by the count value in descending/ascending order; we kept only the first row and only the “make” column; we transformed this single value in the “make” column into a workflow variable named “make”; and finally we used a “Row Filter” to keep only the rows with that value in the “make” column.

We then defined a workflow variable, named “user”, for the whole workflow to hold the username.

After concatenation of the two data sets, a “Rule Engine” node writes the username from the workflow variable “user” in a column “user”.

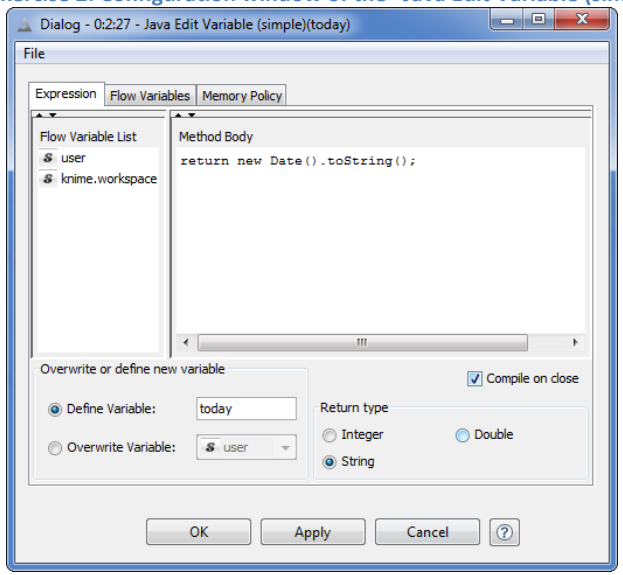
We used a “Java Edit Variable” node to create today’s date and to append it to the data set in a column called “load_date” by means of a second “Rule Engine” node.

4.27. Exercise 2: Workflow

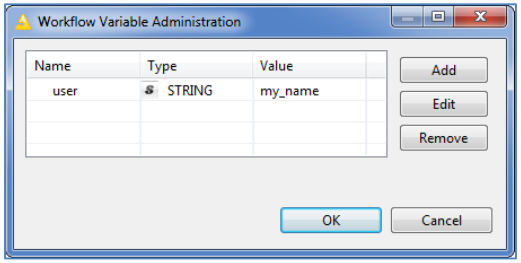


Note. A “Java Snippet” node could also write today’s date in the “load_date” column. However, if you have more than one data set to timestamp, it might be more convenient to use only one “Java Editor Variable” node rather than many “Java Snippet” nodes.

4.28. Exercise 2: Configuration window of the “Java Edit Variable (simple)” node



4.29. Exercise 2: Workflow Variables



Exercise 3

Define a maximum engine size, “max”. Then from the “cars-85.csv” file build two data sets respectively with:

- All cars with “engine_size” between “max” and “max”/2
- All cars with “engine_size” between “max”/2 and “max”/4

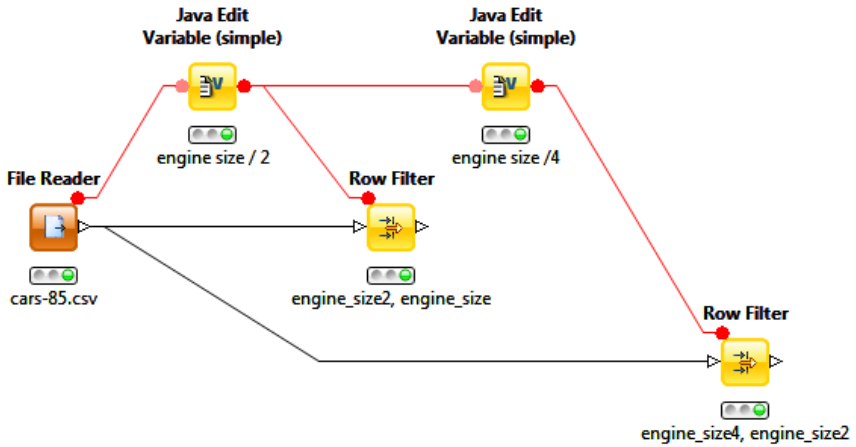
The workflow must run with different values of the maximum engine size.

Solution to Exercise 3

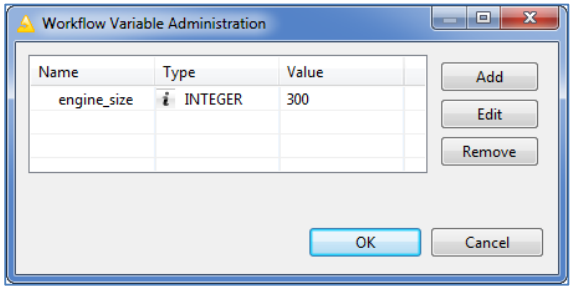
Define “max” as an integer workflow variable. Build “max/2” and “max/4” with two “Java Edit Variable” nodes.

Configure two “Row Filter” nodes with the appropriate workflow variables to build the required data tables.

4.30. Exercise 3: Workflow



4.31. Exercise 3: Workflow Variables



Exercise 4

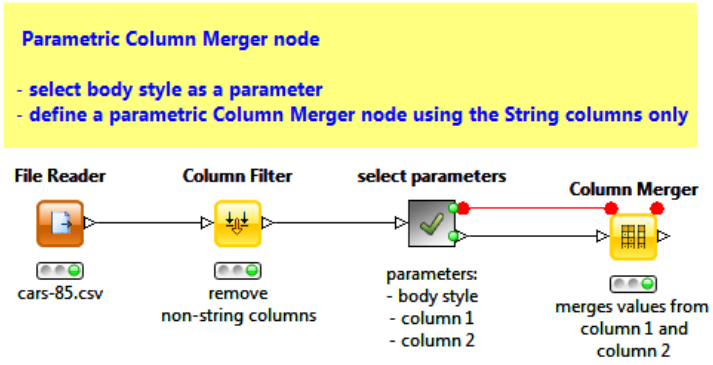
Using the "cars-85.csv" file, for a specific body style of the car (use a Quickform and a Row Filter node) build a parametric "Column Merger" node; that is a "Column Merger" node whose first and second columns are Quick Form parameters. Limit the choice of columns to only String columns.

Solution to Exercise 4

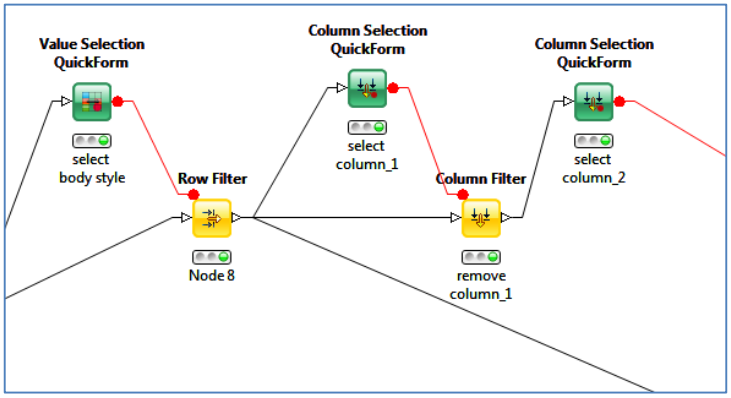
First, we read the data and we extract all String columns with a "Column Filter" node. Then we define all the Quickform parameters:

- With a "Value Selection Quickform" node applied to the "body style" column, we select car records with a particular body style;
- Using a "Column Selection Quickform" node we select the first column to feed the "Column Merger" node with;
- Using a second "Column Selection Quickform" node, applied to all existing column minus the column selected at the previous step, we select the first column to feed the "Column Merger" node with;
- We configure a "Column Merger" node using the variables defined for the first and second column.

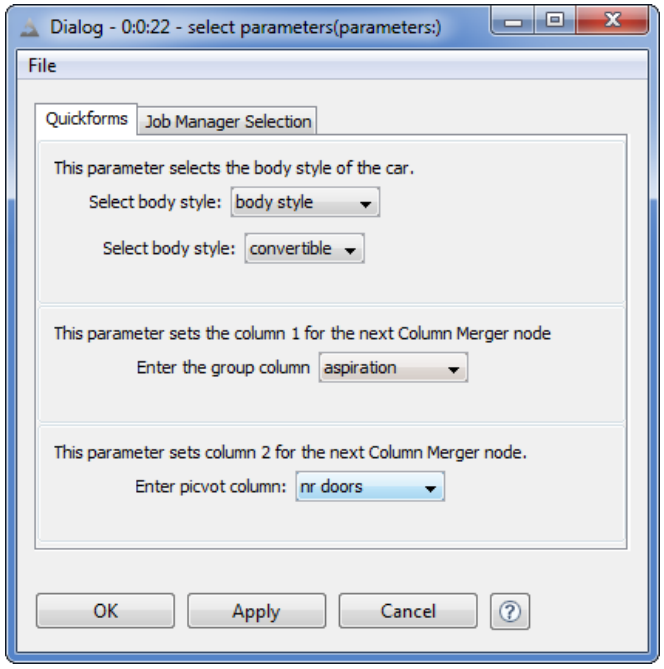
4.32. Exercise 4: the Workflow



4.33. Exercise 4: the Meta-node Sub-workflow



4.34. Exercise 4: the Meta-node Configuration Window



Chapter 5. Calling R from KNIME

5.1. Introduction

R is a free, open source package for graphics and statistical programming. The R program has been around for a number of years now and has developed a large number of statistical and graphical libraries for the most various usages. A large community of R developers maintains around 3000 existing add-ons and produces new additions every year.

R is also very well documented. The online archive for R help documents and tutorials is extensive. However, the first time you use R it may seem a little daunting as there is a steep learning curve to use the software. Writing the correct syntax or navigating through the documentation is not always that straightforward.

KNIME dedicates a few nodes to call R from within the KNIME framework, which allows taking advantage of the KNIME's ease of use and of the large amount of R graphical and statistical libraries at the same time. This chapter guides you in setting up KNIME to connect with R and in using the R nodes to call and run R functions. The usage of the KNIME R nodes and of the most common R graphical and statistical functions is described through a number of recipes. Each recipe contains the solution to a particular problem and includes a detailed description of the R functions and the KNIME workflows used to produce the end result. There are a great many functions and methods within R. Function descriptions within the examples are not exhaustive, and you are therefore urged to read the help pages to discover the full potential of each function.

This chapter is an introduction to the R integration into KNIME, it is not a tutorial on R, nor is this an introduction to statistical programming. The recipes have been written to be as straightforward as possible, yet a basic understanding of statistics is assumed. Nonetheless, the worked examples are thought to enable the generic KNIME user to produce R plots, use R statistical functions, and run simple R scripts from within KNIME.

There is a wealth of information available on all aspects of R mainly from the R community (for example: http://cran.r-project.org/manuals.html). An introductive guide to R also comes together with the R installation package [4].

To run the examples provided in this chapter you need to have R installed on your machine or be able to connect to an Rserve. For instructions on how to download and install the R package or Rserve, see section 1.3.2 of this book.

5.2. The R Nodes Extensions

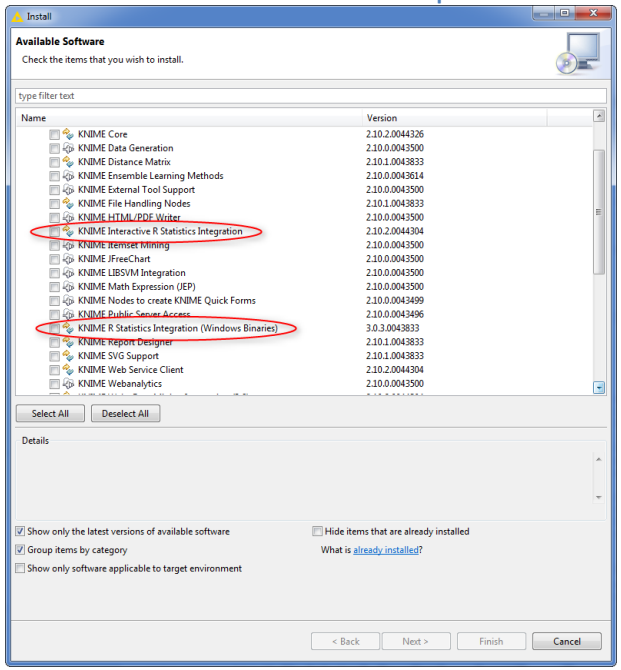
If you have R already installed and ready to use, you can now download and install the KNIME Extensions for the R nodes. The R nodes are available as a plug-in from the KNIME update site.

From the top menu:

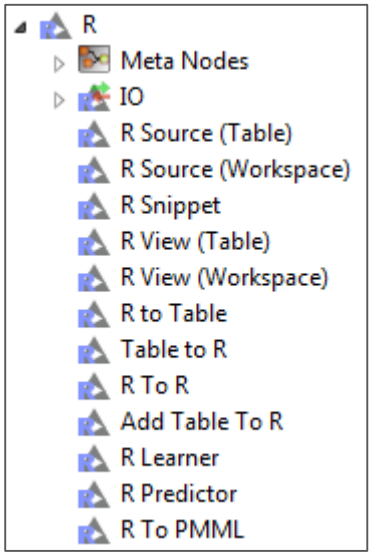
- Select “File” -> “Install KNIME Extensions…”
- In the group “KNIME & Extensions”, select the extension package called “KNIME Interactive R Statistics Integration” (Fig. 5.1)
- For Windows only, you can also install the “KNIME R Statistics Integration (Windows Binaries)” extension which installs the latest R package
- Click “Next” and follow the installation instructions.

If the installation was successful, a category named “R” can be found in the “Node Repository” panel (Fig. 5.2).

5.1. R Nodes Extensions in the KNIME Update window.



5.2. The available R Nodes in KNIME



Note. The "KNIME R Statistics Integration (Windows Binaries)" package is only needed if you do not have R installed on your Windows computer. It will install the R software on Windows (see section 1.3.2).

Installing the R extension into KNIME makes a number of R nodes available in the "Node Repository" panel: to read data ("R Source"), to draw graphics ("R View"), to train and apply models ("R Learner", "R Predictor"), to import data from KNIME into R and vice versa, and to write free bits of code ("R Snippet"). Some nodes - the "(Table)" ones - work with data in the KNIME workflow; that is exporting data from KNIME into R and importing back into KNIME the results of the R script. Some other nodes – the "(Workspace)" ones – work only on data in the R workspace. These two types of nodes have different ports: white triangles for KNIME data tables and gray squares for access to the R workspace. Other R nodes are dedicated to importing and exporting data from one environment to the other.

In a KNIME workflow that includes "(Table)" R nodes, the data transfer between KNIME and the local installation of R is implemented via CSV files. Data from the last KNIME node is written into a CSV file; an R script is generated and executed in R; the results are saved into a CSV file or a PNG image; and finally the output CSV file is read back into KNIME or alternatively the PNG file is opened as a KNIME view.

The first node that is examined in this chapter is the "R Snippet" node. Mathematical functions, statistical functions, and an ad hoc snippet of code can be implemented within the "R Snippet" node to operate on the input KNIME data table. The results of such operations are then output as another KNIME data table.

The "R View" node implements commands from the R graphical libraries. It outputs a graphics view and a PNG image.

The "R Learner" node allows for the creation and training of data models in R. Like all learner nodes in KNIME, the "R Learner" node takes data at the input port and outputs a model at the output port. The model can then be passed to the "R Predictor" node that applies it to another set of data.

Models built in R are not straightforwardly compatible with KNIME predictor nodes. To reuse R models with KNIME predictor nodes, we need to pass through a standard model description: the PMML format. There is a special node to convert R models into a Predictive Model Markup Language (PMML) object: "R To PMML" node. This node allows models created in R to be understood by KNIME nodes.

Two nodes are dedicated to reading and writing R models. The "R Model Reader" node reads an R model from a file and the "R Model Writer" node writes an R model to a file in a compressed zip format.

Data can be read using one of KNIME reader nodes and then transferred to one of the R(Table) nodes. However, nodes can also be read directly using R. Using R, and especially its "foreign" package, to read data files extends the file formats supported by KNIME reading nodes virtually to all existing formats. Indeed the R "foreign" library can read specific formatted data , such as SPSS, SAS, Matlab, and more. The "R Source" node has been designed

as a pure reading node with no input ports, where you can import data from the R libraries, read data from files, and import file formats from other R libraries.

A number of nodes are devoted to the data transfer from R to KNIME and from KNIME to R, including “R to Table”, “Table to R”, and “Add Table to R”.

In this chapter, we will work mainly with “(Table)” nodes, assuming that “(Workspace)” nodes work similarly with just a different input and output space.

5.3. Connect R and KNIME

In order for KNIME to run R commands on a local installation of R, it needs to know where the R executable are sitting on the machine. The default path to the R executable (the R binary file) is configured from the KNIME Preferences page (“File” -> “Preferences” -> “KNIME” -> “R”). There is no need to change anything if R was installed via the “KNIME R Statistics Integration (Windows Binaries)” extension for Windows only.

Set the R Binary (R.exe) location in the “Preferences” page

In the top menu:

- Select “File” -> “Preferences”

In the “Preferences” window

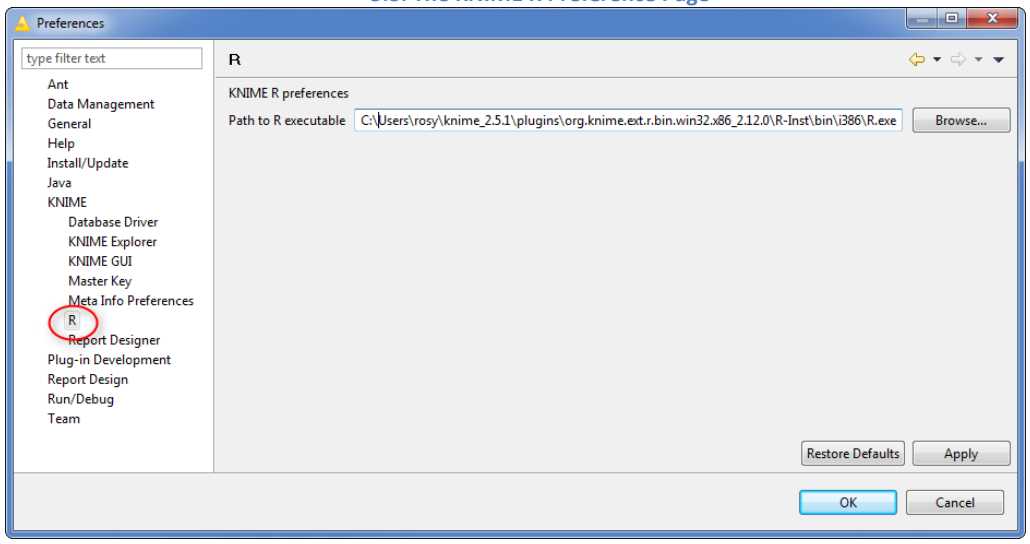
In the left frame:

- Open group “KNIME”
- Select “R”

In the right frame:

- Select the R.exe file location using the “Browse” button
- Click “OK”

5.3. The KNIME R Preference Page



5.4. The R Script Editor in the Configuration Window of the R Nodes

If you create an instance of any "R" node in a workflow and you open its configuration window, you will notice two tabs: "R Snippet" and "Templates". These two tabs are present in the configuration windows of all R nodes. A few R nodes might present a few more specific tabs.

All R nodes require some R code. In order to write the R code, an R editor is provided in the "R Snippet" tab in the configuration window of each node.

The "R Snippet" Tab

Within the "R Snippet" tab, we find the **"R Script" editor** in the center of the configuration window. This is where the R script is edited.

If this is a "(Table)" node, the KNIME data table coming from the input port is identified as `knime.in`. The output data to be imported back into KNIME through the node output port is identified as `knime.out`. Similarly, if the node outputs a model, the model should be identified as `knime.model`.

Columns within the `knime.in` data frame can then be accessed by using one of the following expressions:

- `knime.in$"<name-of-column>"` returns the column named `<name-of-column>` (notice the "")
- `knime.in[["<name-of-column>"]]` returns the column named `<name-of-column>` (notice the "")
- `knime.in[[n]]` returns the n^{th} column in the `R` data frame
- `knime.in[c(1,2)]` returns the columns at the positions described in `c(1,2)`, where `c(1,2)` = [1,2]
- `knime.in[1:4]` returns all columns between the 1st and the 4th column

For further information on how to use R syntax to extract columns from a data frame, we advise reading the introductory R manual [4].

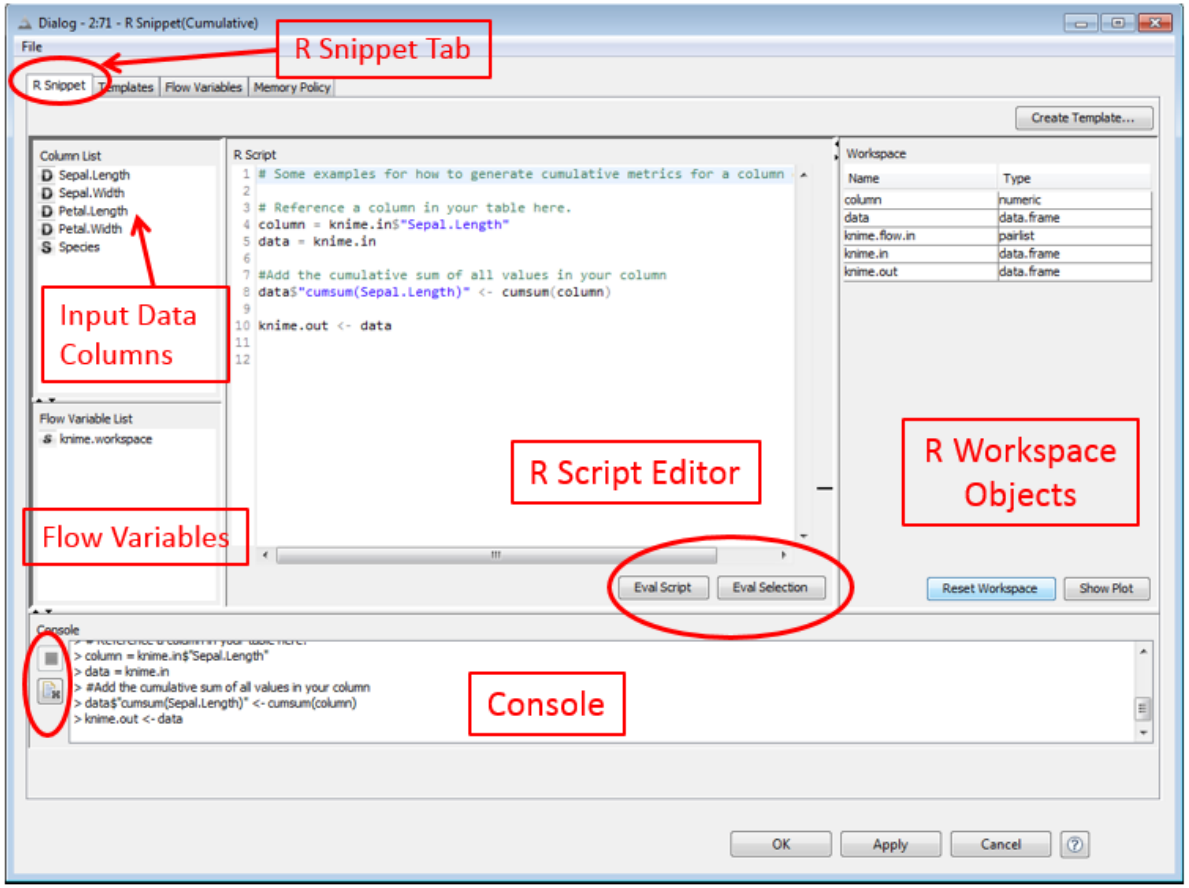
In the configuration window of the R nodes, you can also see, on the left side, the **"Column List"** and **"Flow Variable List"** panels. These panels contain the list of all available data columns and flow variables respectively. Double-clicking any of them automatically inserts it in the R script editor with the correct syntax.

At the bottom of the R script editor frame, there are two buttons**: "Eval Selection**" and **"Eval Script"**. These two buttons evaluate the R script as either a selected segment of the code or the entire script respectively. These two buttons are very useful for debugging: to find out errors in the script syntax as well as in the script logic.

The evaluation output message is sent to the **"Console"** panel and is the same as the information sent by R to the R Std Output. Here possible code errors show up when evaluating the R script with the two buttons described above. These two buttons and the "Console" panel provide the interactive

coding and debugging functionalities for the R nodes. The “Console” panel also has two buttons on the left. The top one is a stop button to terminate the currently running script evaluation command; while the lower button clears the “Console” panel of all messages.

5.4. R nodes: the “R Snippet” Tab in the Configuration Window



The area named “**Workspace**” on the right shows all R workspace data frame and variables by their name and type; both the initially read variables `knime.flow.in` and `knime.in` as well as all the other variables generated during the script evaluation. For example, `knime.in` is read as an R data.frame object and `knime.flow.in` as an R pairlist object. In order to visualize the content of the listed R variables into the “Console”, just double-click the variable name or type. Double clicking a variable in the “Workspace” area has the effect of running the `print()` R function for that R variable, with the result being sent to the “Console” panel. This is particularly useful to test and debug the R script on the fly before executing the node.

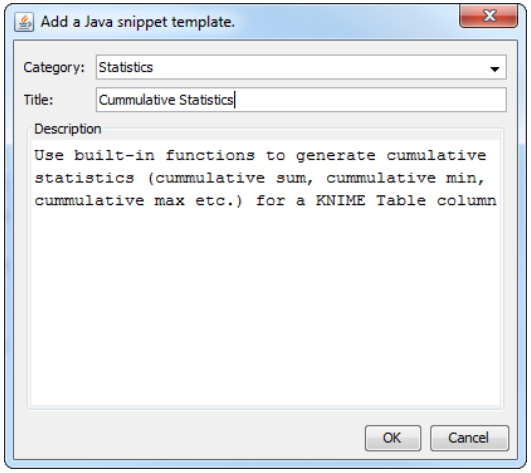
If a plot is produced by the R script, this can be previewed by clicking the **"Show Plot"** button.

In order to reset the R workspace, click the "**Reset Workspace**" button.

Once a script is written and evaluated without errors, it can be saved as a template for future use. Clicking the **"Create Template…**" button in the upper left corner pulls up the "Add a Java Snippet Template" window. This window, borrowed from the "Java Snippet" nodes, allows the user to store this R script with some additional information, such as a title, a category, and a description.

Finally, the "OK" button saves the R script in the node's configuration settings. The "Execute" option in the node context menu sends this R script to the R software to be executed.

5.5. The "Add a Java Snippet template" window used here to store the current R script as a template



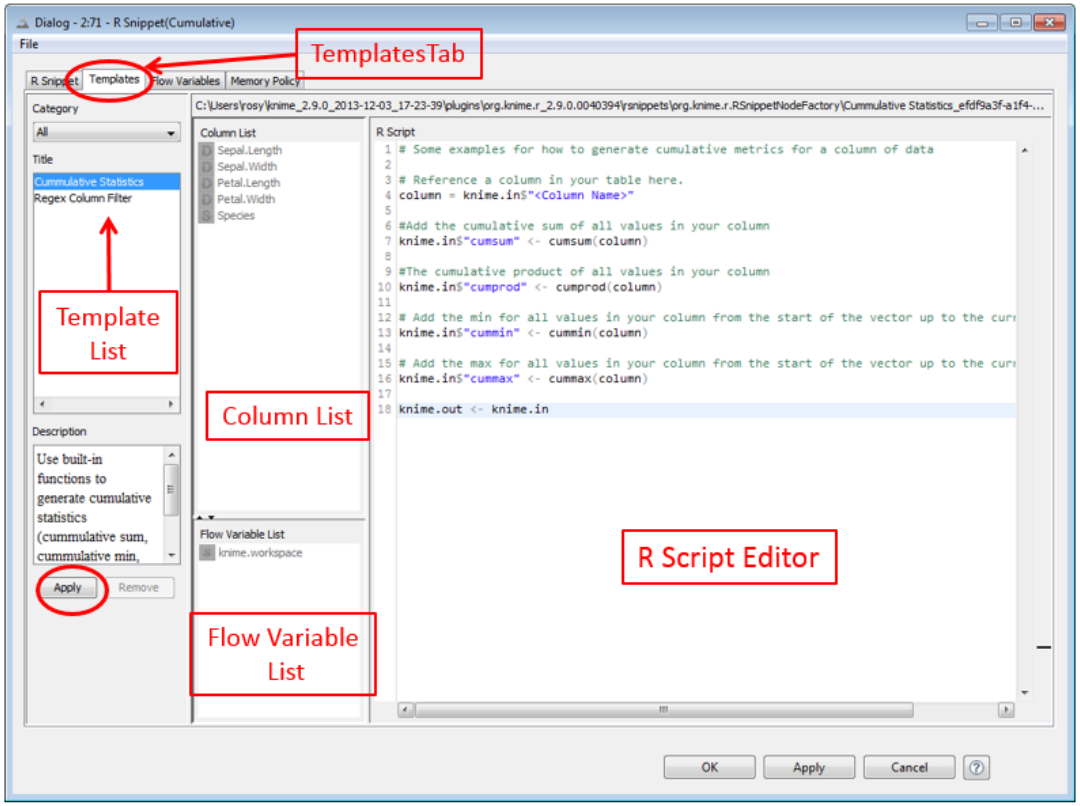
The "Templates" Tab

Moving now to the "Templates" tab, we find a "Title" panel on the left, listing all available template R scripts. R templates are listed by title and can be filtered by "Category" (menu above the "Title" panel). Selecting an R template from the list, shows its description in the panel named "Description" below "Title" and its code in the "R Script" panel in the center.

"Column List", "Flow Variable List", and "R Script" panels are for information only and are not interactive. The R script cannot be edited; it can only be copied into the "R Snippet" tab by clicking the "**Apply**" button under the "Description" panel. Only there, in the "R Snippet" tab, the script can be customized for this particular node.

Selecting an R template from the list and clicking the "**Remove**" button removes this R template from the list. R nodes come with a few preloaded templates. For these preloaded templates the "Remove" button is grayed out and inoperative.

5.6. R nodes: the "Templates" tab



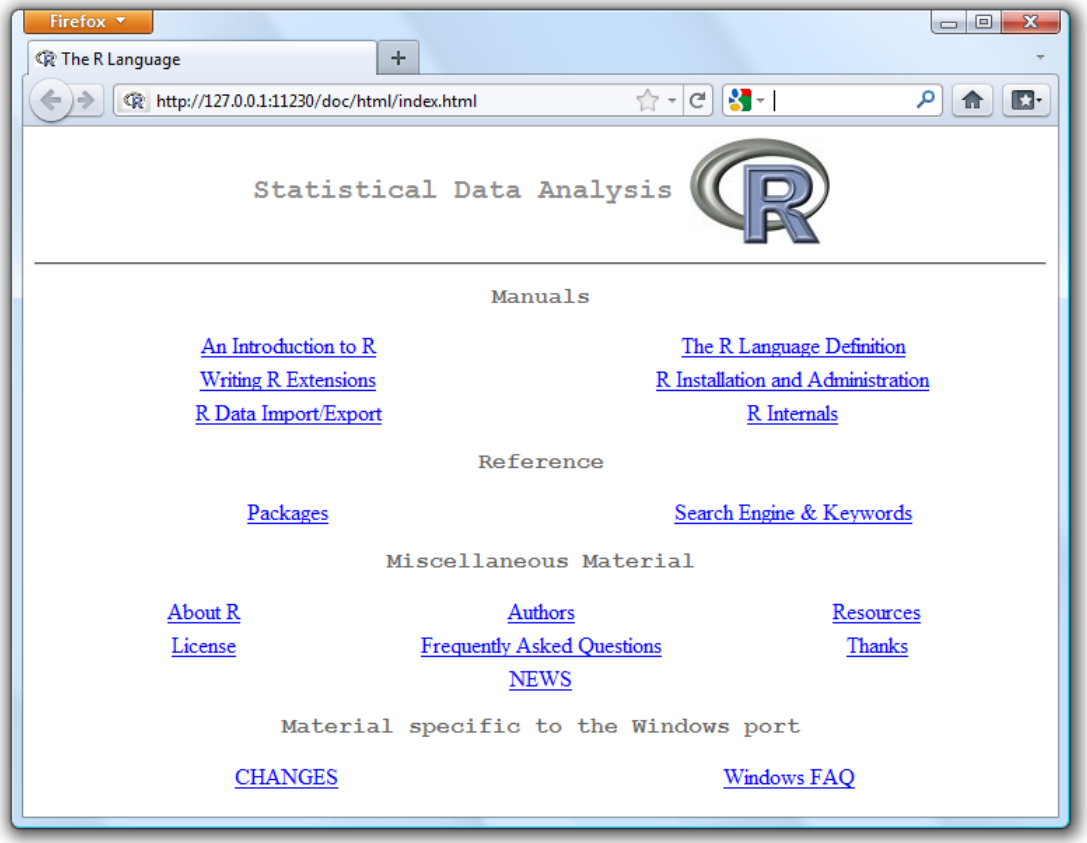
User defined R templates are saved as XML files in folders within the `<KNIME installation dir>\workspace\.metadata\knime\rsnippets\` directory. All R nodes of the same type, like for example all "R Snippet" nodes, share the same list of R templates. Indeed, templates for each R node type are saved in separate folders, such as `rsnippets`, `rmodels`, `rviews`, or `rreaderworkspace`.

The Context Menu of the R Nodes

The context menu of any R node includes options common to all other nodes, like Configure, Execute, etc… The last option, as usual, visualizes the results, image or data. However, two more options are available in the context menu of these nodes: “View R Std Output” and View R Error Output”. These two options show the output text of the R script on the standard output and on the standard error screen respectively.

5.5. Getting Help on R

5.7. The R Help Documentation



You are now ready to write and run an R script, either to build a graphical plot or a model or simply to integrate some R functions. If you are not already an expert R user, you can access to the manual pages directly from inside R. Start the R program and use the R command "`help.start`" which will show the documentation's table of contents in the form of a web browser. You do not need to start R by itself; you can also use an "R Snippet" node to invoke the help pages.

The web site for the table of contents of the R Help then opens. From here you can find links to all the installed documentation. For example, the *Packages* link takes you to the list of all available packages.

There are a number of valuable websites that can be accessed from R. The "`RSiteSearch`" function can also be used within an R node to find information on a topic collated from a number of different sources:

```
RSiteSearch("key phrase")
```

Help online on single commands is not available.

5.6. Creating and Executing R Scripts

The most straightforward R Extension node is the "R Snippet" node. The "R Snippet" node allows for execution of R commands on a local R installation. Data is read from the data input port and the result of the R code is returned at the output port.

As we already described in section 5.2 the data transfer between KNIME and R is implemented via CSV files. That is: the input data is written to a CSV file; the data is imported into R and referenced by the variable named "knime.in" in the R code; the R code is executed; and the final content of the variable named "knime.out" is saved into a CSV file again and imported back into KNIME.

Note. Since the output data of the "R Snippet" node consists of the final value of the "knime.out" data frame, we need to make sure that the end results of the R script are stored into the variable "knime.out". In fact, after the R script execution, the "knime.out" values are copied into the output port of the "R Snippet" node.

The best way to see the usefulness of the R snippet node is by way of example. Here we extract some data from the R data set library, run some statistics on the data, and return the results.

Within R itself there are roughly 100 available data sets. In order to see which data sets are supplied with the installed version of R, type the following commands into the R editor of the "R Snippet" node:

```
knime.out<-data()$results[,3:4]
knime.out
```

The output from these commands is a list of the items in the `data()` package.

For our example we use the "MeltingPoint" data which is available from the user contributed package "`QSARdata`" (see section 1.3.2, for how to install the "QSARdata" package in your R.exe application). This data set contains a large number of chemicals and their melting points, i.e. the temperature for the transition from solid to liquid state. By using an "R Snippet" node we can have a look at the content of this data.

R Snippet

The "**R Script**" panel in the center hosts the R code that will be executed.

The "**Column List**" pane is at the top left. This is the list of the available input data columns.

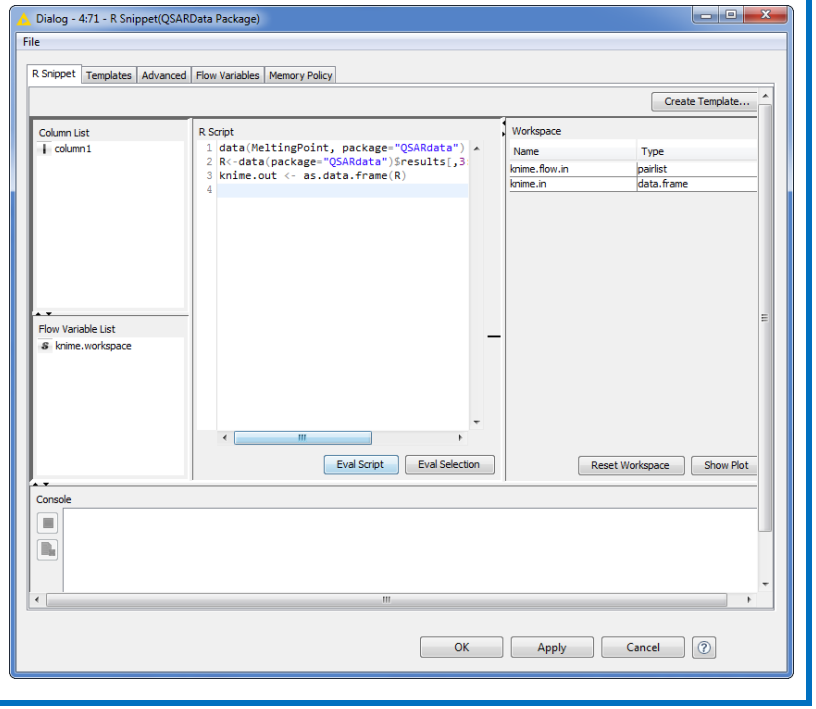
The "**Flow Variable List**" pane is at the bottom left. This is the list of the workflow variables available to this node.

Double-clicking a data column or a workflow variable inserts it into the "R Script" editor with the correct reference syntax.

Input data are available in the R script through the variable `knime.in`. At the end of the R snippet code, always remember to pass back the resulting data to the variable `knime.out` to be copied at the node output port.

Also, remember the debugging possibilities available through the "Eval …" buttons, the "Workspace" monitoring panel, and the possible error messages on display in the "Console".

5.8. The configuration window of the "R Snippet" node



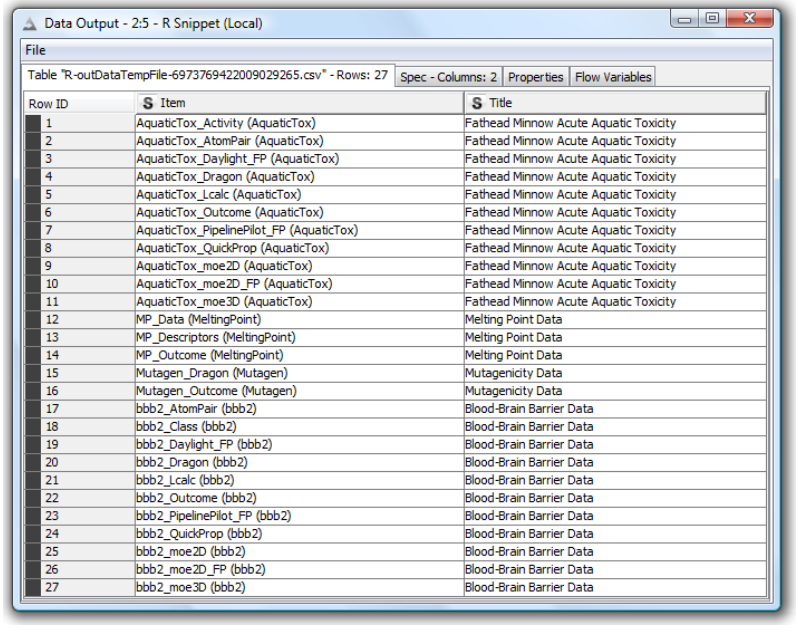
First of all, we created a new workflow group named "Chapter5", to host all workflows for this chapter. Then, we created a new workflow named "R Snippet Example". We populated the workflow with a first "Table Creator" empty node. We didn't actually need this node, because we wanted to use

an R internal data set. However, the “R Snippet” node cannot be enabled if the input port is not connected to a previous node. Thus, the empty “Table Creator” node to enable the “R Snippet” node.

We wrote the following code in the “R Script” frame in the configuration window:

```
data(MeltingPoint, package=”QSARData”)
knime.out<-data(package="QSARdata")$results[,3:4]
knime.out <- as.data.frame(R)
```

5.9. Output data of the “R Snippet” node after importing the “QSARdata” package details



Row ID	Item	Title
1	AquaticTox_Activity (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
2	AquaticTox_AtomPair (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
3	AquaticTox_Daylight_FP (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
4	AquaticTox_Dragon (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
5	AquaticTox_Lcalc (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
6	AquaticTox_Outcome (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
7	AquaticTox_PipelinePilot_FP (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
8	AquaticTox_QuickProp (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
9	AquaticTox_moe2D (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
10	AquaticTox_moe2D_FP (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
11	AquaticTox_moe3D (AquaticTox)	Fathead Minnow Acute Aquatic Toxicity
12	MP_Data (MeltingPoint)	Melting Point Data
13	MP_Descriptors (MeltingPoint)	Melting Point Data
14	MP_Outcome (MeltingPoint)	Melting Point Data
15	Mutagen_Dragon (Mutagen)	Mutagenicity Data
16	Mutagen_Outcome (Mutagen)	Mutagenicity Data
17	bbb2_AtomPair (bbb2)	Blood-Brain Barrier Data
18	bbb2_Class (bbb2)	Blood-Brain Barrier Data
19	bbb2_Daylight_FP (bbb2)	Blood-Brain Barrier Data
20	bbb2_Dragon (bbb2)	Blood-Brain Barrier Data
21	bbb2_Lcalc (bbb2)	Blood-Brain Barrier Data
22	bbb2_Outcome (bbb2)	Blood-Brain Barrier Data
23	bbb2_PipelinePilot_FP (bbb2)	Blood-Brain Barrier Data
24	bbb2_QuickProp (bbb2)	Blood-Brain Barrier Data
25	bbb2_moe2D (bbb2)	Blood-Brain Barrier Data
26	bbb2_moe2D_FP (bbb2)	Blood-Brain Barrier Data
27	bbb2_moe3D (bbb2)	Blood-Brain Barrier Data

Let’s analyze this piece of R code.

`data(MeltingPoint, package="QSARdata")`	This line extracts the “MeltingPoint” data set from the package “QSARdata”.
`R<-data(package="QSARdata")$results[,3:4]`	This line extracts the 3rd and 4th column of the “QSARdata” package description and re-assigns them back to the “knime.out” variable.
`knime.out <- as.data.frame(R)`	This line exports the result as a data frame. R nodes only accept data frames as output data.

Note. There are no "(Table)" and "(Workspace)" versions for the "R Snippet" node. The "R Snippet" node works between KNIME data tables as input and output. The equivalent of the "R Snippet" node working directly on the R workspace is the "R To R" node.

After execution, the output data table of the "R Snippet" node looks like in figure 5.9. The first tab is the CSV file containing the data from the `knime.out` object. Notice that the output data table only contains the "QSARdata" package details that were re-assigned to the `knime.out` data frame: that is only column 3 and 4 of the full description data frame.

The "MeltingPoint" data set is composed of three separate data sets: "MP_Data", "MP_Descriptors", and "MP_Outcome". The "MP_Outcome" contains a list of experimentally determined melting points from 4401 chemical compounds. Associated with each chemical compound are 202 chemical descriptors ("MP_Descriptors") which include information such as weight, volume, and surface of each of the compounds. "MP_Data" splits the data into training and testing data.

In the workflow called "R Snippet Example", we introduced three "R Snippet" nodes with the following R codes to extract the three Meling Point data sets directly from R:

Code 1

```
data(MeltingPoint, package="QSARdata")
knime.out<-as.data.frame(MP_Data)
```

Code 2

```
data(MeltingPoint, package="QSARdata")
knime.out<-MP_Descriptors
```

Code 3

```
data(MeltingPoint, package="QSARdata")
knime.out<-as.data.frame(MP_Outcome)
```

After execution, the three "R Snippet" nodes showed at the output port the three different pieces of information for each row/chemical compund.

The next step was to combine all the data columns from the three "R Snippet" nodes together. This can also be accomplished in R, but in this instance we decided to do it in KNIME using the "Joiner" node. We subsequently joined all those columns together by their unique RowID. After the join, the

4401 chemicals were split into two sets, 4126 chemicals for model training and 275 compounds as a final validation set, according to the information contained in the “MP_Data” data set.

Instead of that awkward construction using an empty “Table Creator” just to enable the “R Snippet” node, we could have used an “R Source” node. The “R Source” node requires no input and because of that it is very suitable to load and process R resident data.

R Source (Table)

The configuration window of the “R Source (Table)” node is exactly the same as the configuration window of the “R Snippet” node, with:

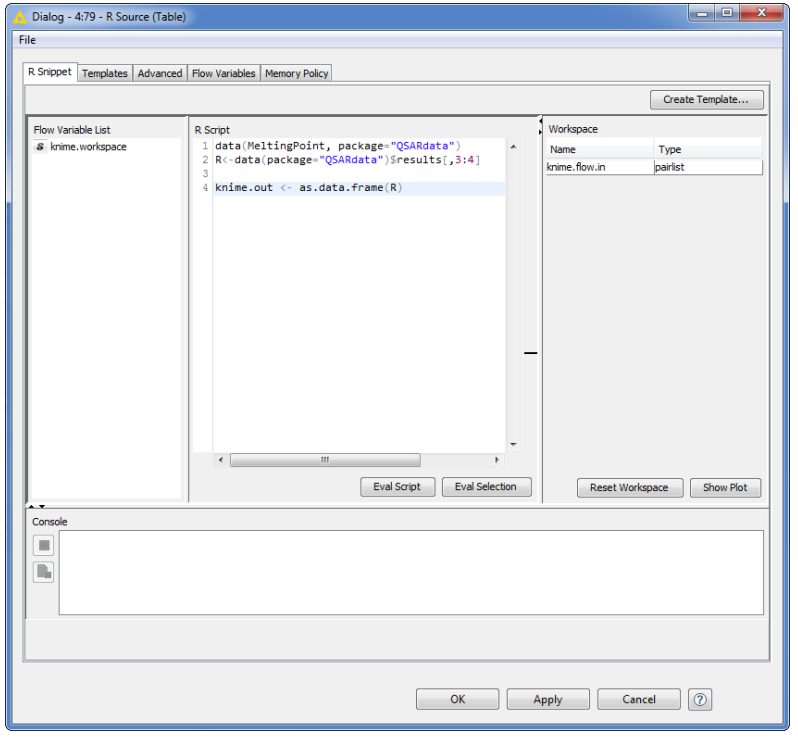
- The “**R Script**” panel in the center hosts the R code.
- The “**Column List**” pane is at the top left. This is the list of the available input data columns.
- The “**Flow Variable List**” pane is at the bottom left. This is the list of the workflow variables available to this node.

Double-clicking a data column or a workflow variable inserts it into the “R Script” editor with the correct reference syntax.

Input data are available in the R script through the variable `knime.in`. At the end of the R snippet code, always remember to pass back the resulting data to the variable `knime.out` to be copied at the node output port.

The same debugging options as for the “R Snippet” node are available through the “Eval …” buttons, the “Workspace” monitoring panel, and the possible error messages on display in the “Console”.

5.10. The configuration window of the “R Source (Table)” node



Usage of the R statistical functions via an "R Snippet" node

Now that we have the data, we can examine a number of statistical parameters. We start with the simple measures of minimum value, maximum value, mean, median, standard deviation, variance and correlation, focusing on just a few of the descriptors for the sake of brevity. A few simple R functions can calculate the minimum, maximum, and median of a data vector. Given two vectors, correlations can also be determined.

Note. It's important to remember that missing data is not handled well by these functions and care should be taken to avoid having so-called not available (NA) data in the vectors. In addition, these functions require data vectors as arguments. However, when R reads the input CSV file prepared by KNIME, it transforms the data into data frames and not vectors. Therefore such vector functions might not always work as expected if data is coming from a KNIME workflow.

Some simple statistical functions are demonstrated in the R code below:

```
for(n in c(1:15)) {
   min <- apply(R[n],2,min)
   max <- apply(R[n],2,max)
   median <- apply(R[n],2,median)

   if (n == 1){
        lista<-t(min)
        listb<-t(max)
        listd<-t(median)
        rownames(lista)="min"
        rownames(listb)="max"
        rownames(listc)="median"
        rows<-rbind(lista,listb,listc)
        list<-rows
   }
   else {
        tempa<-t(min)
        tempb<-t(max)
        tempc<-t(median)
        temp<-rbind(tempa,tempb,tempc)
        list<-cbind(list,temp)
   }
}
knime.out<- as.data.frame(list)
```

We have made use of a "for" loop to handle data processing. In this way each column in the R data frame is handled one at a time, looping over the 15 columns of interest.

We used the "if-else" statement to control the addition of columns to create the output table.

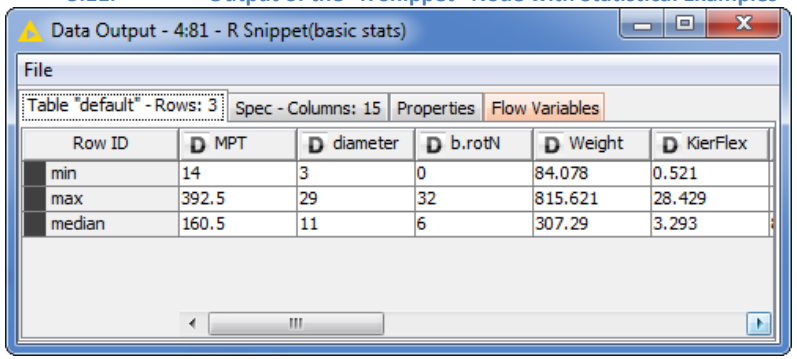
A portion of the output from this "R Snippet" node is shown in figure 5.11.

The correlation between two columns can be determined using the cor() function. When used on a data frame, we get a correlation matrix. Therefore the R code:

```
knime.out<- as.data.frame(cor(knime.in))
```

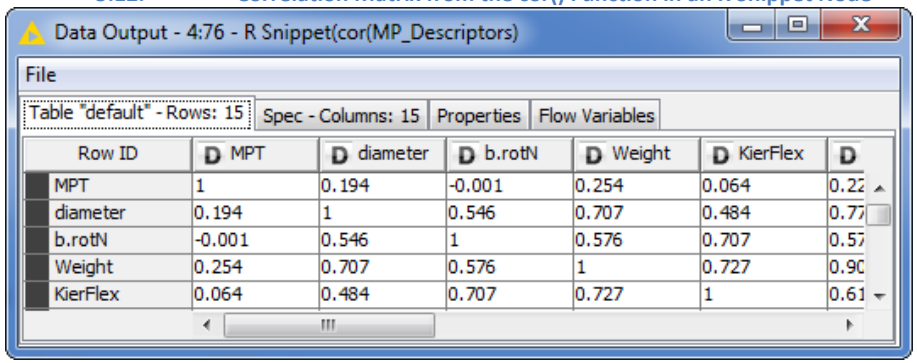
returns a correlation matrix, a portion of which is shown in figure 5.12.

5.11. Output of the "R Snippet" Node with Statistical Examples



Row ID	MPT	diameter	b.rotN	Weight	KierFlex
min	14	3	0	84.078	0.521
max	392.5	29	32	815.621	28.429
median	160.5	11	6	307.29	3.293

5.12. Correlation Matrix from the cor() Function in an R Snippet Node



Row ID	MPT	diameter	b.rotN	Weight	KierFlex	D
MPT	1	0.194	-0.001	0.254	0.064	0.22
diameter	0.194	1	0.546	0.707	0.484	0.77
b.rotN	-0.001	0.546	1	0.576	0.707	0.57
Weight	0.254	0.707	0.576	1	0.727	0.90
KierFlex	0.064	0.484	0.707	0.727	1	0.61

To determine whether these correlations are statistically significant it is important to use the `cor.test()` function. This function returns the *p*-value, the result of the corresponding *t* distribution, the degrees of freedom, and the confidence interval of the correlation. If the variables are normally distributed, as is the default, then the Pearson method is used. For data that is not bivariate and normally distributed, the Kendall's *tau* or Spearman's *rho* statistic can be used to estimate a rank-order correlation. The value of greatest interest is the *p*-value generated in the calculation. Generally speaking, at the 95% confidence interval, a $p \leq 0.05$ indicates that the correlation observed is likely to be significant, and a $p > 0.05$ is unlikely to be significant.

Information can be extracted from the `cor.test()` function as the following R code demonstrates.

```
a<-t(knime.in[2])
b<-t(knime.in[15])
cor<-cor.test(a,b)

parameters<-c("t","df","p-value","cor")
knime.out<-
c(cor$statistic,cor$parameter,cor$p.value,cor$estimate)
knime.out<- as.data.frame(cbind(knime.in,parameters))
```

5.13. Results of the cor.test() Function in the R Snippet Node

Data Output - 4:75 - R Snippet(cor.test(MP_...

File

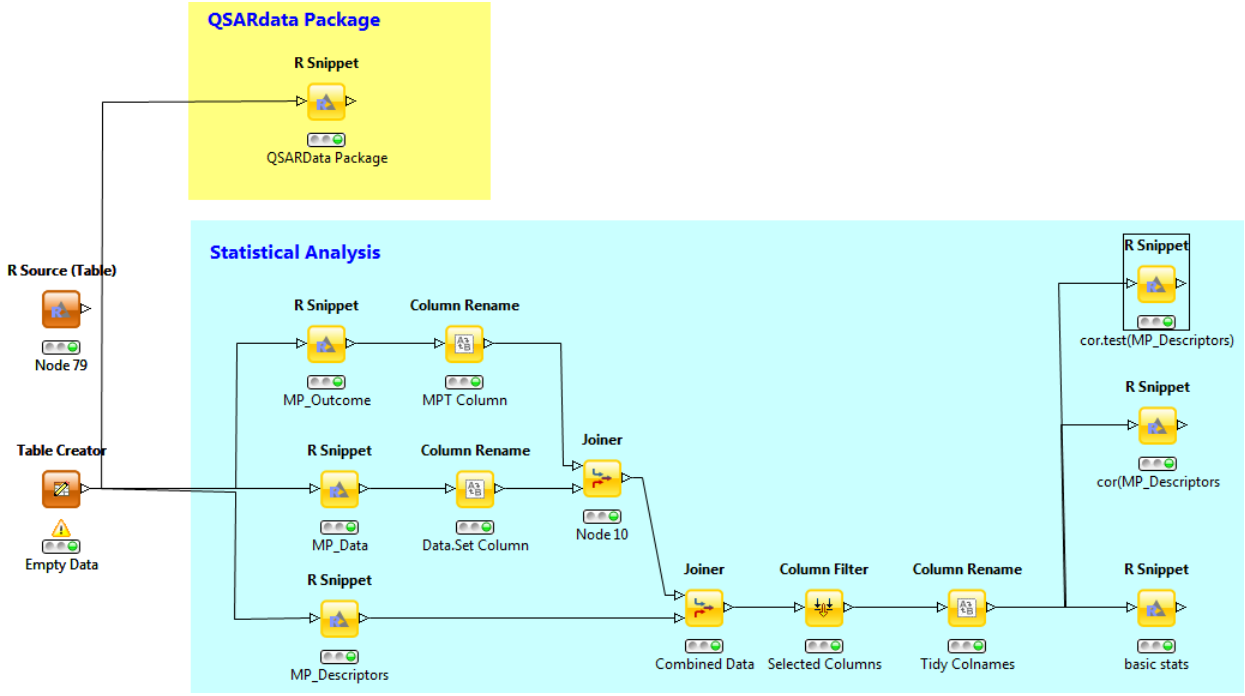
Spec - Columns: 2 | Properties | Flow Variables

Table "default" - Rows: 4

Row ID	R	parameters
t	77.520859075...	t
df	4350	df
	0	p-value
cor	0.7616396125	cor

The various attributes of the cor.test() function contain all the relevant statistical results, including the *t* distribution, the degrees of freedom (length(x)-2, assuming that the samples follow independent normal distributions) and the *p*-value.

5.14. The final workflow "R Snippet Example"



5.7. Using the "R View" Node to plot Graphs

Another one of the many advantages of integrating R within KNIME is the possibility to use the large and well established R graphical libraries. In fact, R offers a very large number of functions for many kinds of different plots. This section demonstrates the plotting capabilities of the R libraries. The two KNIME nodes designed to exploit R graphics are the "R View (Table)" and "R View (Workspace)". Both nodes work similarly.

R View Node

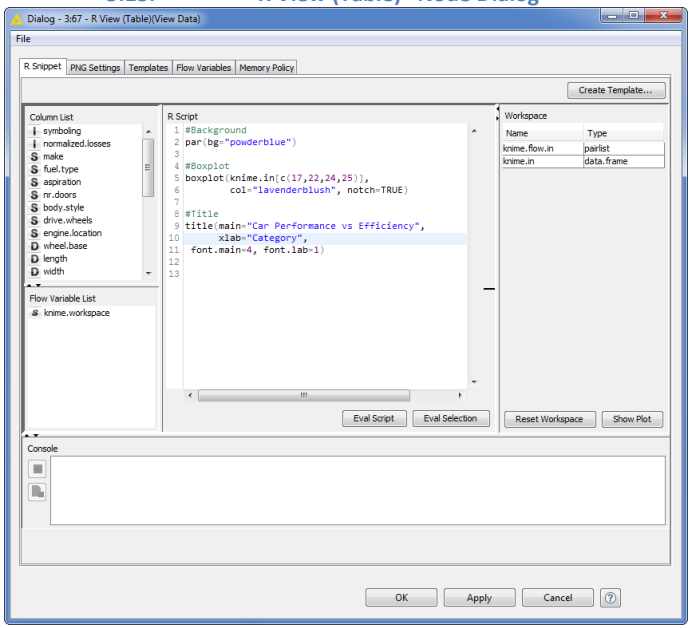
The "R View" node configuration window contains three important tabs: "R Snippet" and "Templates", like all other R nodes, and "PNG Settings", only available for this node.

The "**R Snippet**" tab contains all necessary lists (data columns, flow variables), the R script editor, and the debugging buttons and infos necessary to create, edit, and evaluate an R script. It also contains the "Create Template" button to store the current script as a template for future usage.

The "**Templates**" tab then takes the user to the R script storage space, where all previously saved templates are available.

The "R View" node produces an image and a view. The graphical settings are controlled in the "**PNG Settings**" tab. Here the image height, width, resolution, point size, and background colour can be adjusted.

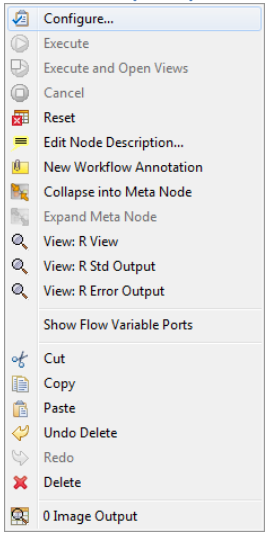
5.15. "R View (Table)" Node Dialog



Note. R does not accept variable names with special characters or spaces. We need to rename such column headers before we use them in an R script. Also, if the column name starts with a number, the letter 'X' must be pre-pended to the column name, to have for example `knime.in$"X1Column1"`.

`knime.in` is the data table imported from KNIME into the R script. `knime.out` is not necessary, since the "R View" node has no output data.

5.16. "R View (Local)" Node Context Menu



To visualize the plot, once the node is configured, in the context menu you can:

- Select "Execute and Open Views". This command opens the plot image immediately after execution.

or

- First, select "Execute"; then, select "View: R View". The KNIME view displays the R view.

The command "View: R Error Output" shows the R execution errors, if anyThe command "View: R Std Output" displays the standard output generated by the R script, including variable assignments and the R code itself.

Remember the "Show Plot" button in the "R Snippet" tab? The "R View" node is the perfect candidate to use it, when checking whether the plot is as expected.

Finally, the view output is a graphics file in PNG format which is generated in R, passed into KNIME, displayed in the node's view, and available at the node's output port.

What follows is a detailed look at some of the R plotting functions, with descriptions of the R code used within the command editor to create user specific views. We use the "cars-85.csv" file available from the Download Zone of this book. The resulting view generated by each plot is also shown. For each new R plot example, new pieces of R code are described in detail. Advancing through the examples, the description of the R code becomes briefer.

Generic X-Y Plot

This function allows for the generation of a scatter plot from pairs of numerical observations. However, if a single column is selected the result is the plot of the Row ID against a numerical column type, which, for nominal data, results in an aggregated histogram. If the data frame contains more than two columns, this plot function returns multiple scatter plots. Data can be plotted by specifying the columns, for example the 1st and the 2nd column:

```
plot(knime.in[1:2])
```

or equivalently by first assigning a data frame to contain the columns of interest:

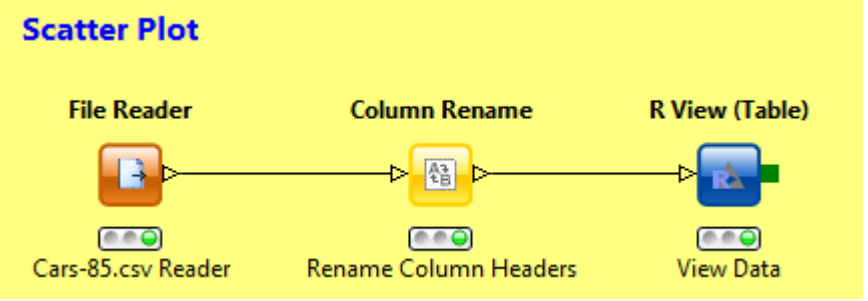
```
knime.in<-knime.in[c(1,2)]
plot(knime.in)
```

In the “Chapter5” workflow group we created a new workflow named “Plotting Examples 1”. We used this workflow to show how to produce an R “Generic X-Y plot” in a KNIME workflow. Using the “cars-85.csv” data file, we graphed the engine size and engine horse power in a scatter plot and colored the data by the city fuel consumption.

First, the “cars-85.csv” data file was read using a “File Reader” node. Then the column headers were renamed using the “Column Rename” node by replacing the non-character strings with full stops “.”. The “Column Rename” node was then connected to an “R View (Table)” node. The “R View” node was subsequently configured and the context menu command, “Execute and Open View”, was run to see the plot result.

The final workflow generating the example of an R “generic X-Y plot” is shown in the figure on the right.

5.17. Workflow "Plotting Examples 1"



The following piece of R code was written into the R Snippet panel of the “R View” node.

```
#Define Colour Scheme

len=length(knime.in$"city.mpg")

split=25

mpg=knime.in$"city.mpg"[1:len]

colours <- mpg
colours[mpg>=split] <- "red"
colours[!(mpg>=split)] <- "blue"

#Plot Graph and Label Axes and Title

plot(
```

The “#” character means that all the text in that line is a comment and is ignored by the R compiler.

The `length()` function obtains the length of the column `city.mpg` and assigns the value to the variable “`len`”.

The value 25 is assigned to the “`split`” variable. “`split`” is used later to split the data into cars which have a fuel consumption greater than or equal to 25, or not.

This line creates a vector named “`mpg`”. “`mpg`” contains the data from the “`city.mpg`” column.

A “`colours`” vector is created from the “`mpg`” values. The vector values are then changed to strings equal to either “`red`” or “`blue`”. This will be the color scale for the graph.

The next four lines draw the plot.

The `plot()` function takes a number of arguments, which can be split over lines to make the code easier to read. The x and y coordinates are of course mandatory arguments. However, there are a number of optional arguments that can also be used; we have only used a few of them. For more information on plot() options, check the R help.

```
      knime.in$"engine.size",
      knime.in$"horse.power",

      main=sprintf("Fuel Economy", len),

      xlab="Car Engine Size (cu in)",

      ylab="Horse Power (HP)",

      col=colours,


      pch=20,


      cex=2
)


#Legend
points(c(70,70),
       c(195,210),
       col=c("red","blue"),
       pch=20)

text(100,210, "City MPG <= 25")
text(100,195, "City MPG > 25 ")
```

The first two arguments are `knime.in$"engine.size"` and `knime.in$"horse.power"`. They represent the x and y coordinates respectively of the points displayed in the plot.

The argument `main` sets the title for the plot.

`xlab` sets the title for the x axis.

`ylab` sets the title for the y axis.

The `col` parameter assigns the colors for the points taking our "`colours`" vector as input.

The parameter `pch` indicates the plotting 'character', hence the symbol to use for each point. This can either be a single character or an integer code to indicate a graphics symbol; a value of `pch=20` is the bullet symbol.

The parameter `cex` is a numerical vector giving the amount by which plotted symbols should be scaled relatively to the default setting.

Finally, a legend is added to the plot using the `points()` and the `text()` functions.

Executing and viewing the graphical output of the "R View" node result in the graphical display of the R plot function. Actions can be performed on this window by clicking the "File" menu item on the top left corner of the window. The "File" menu item allows three operations to be performed:

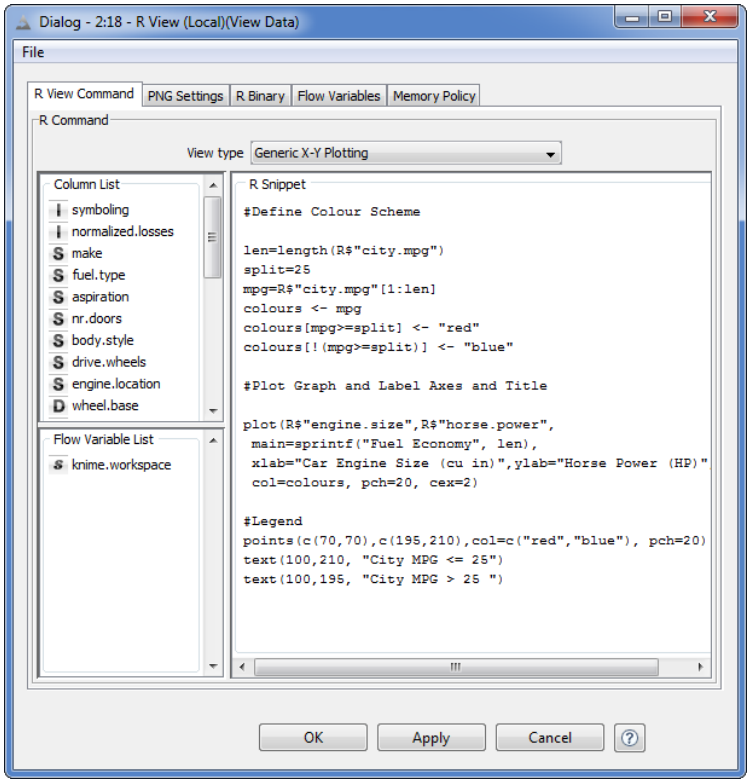
- "Export as" gives the option to save the plot as an image file in the PNG or the SVG format
- "Always on top" makes the window appear in front of other opened windows
- "Close" closes the window.

As we said before, the plot can be viewed again by right-clicking the “R View” node and selecting the option “View: R View”. It is possible that this option is grayed-out: this means that the R code failed to run successfully. In this case, check the “View: R Error Output” to identify which line caused an error in the R code.

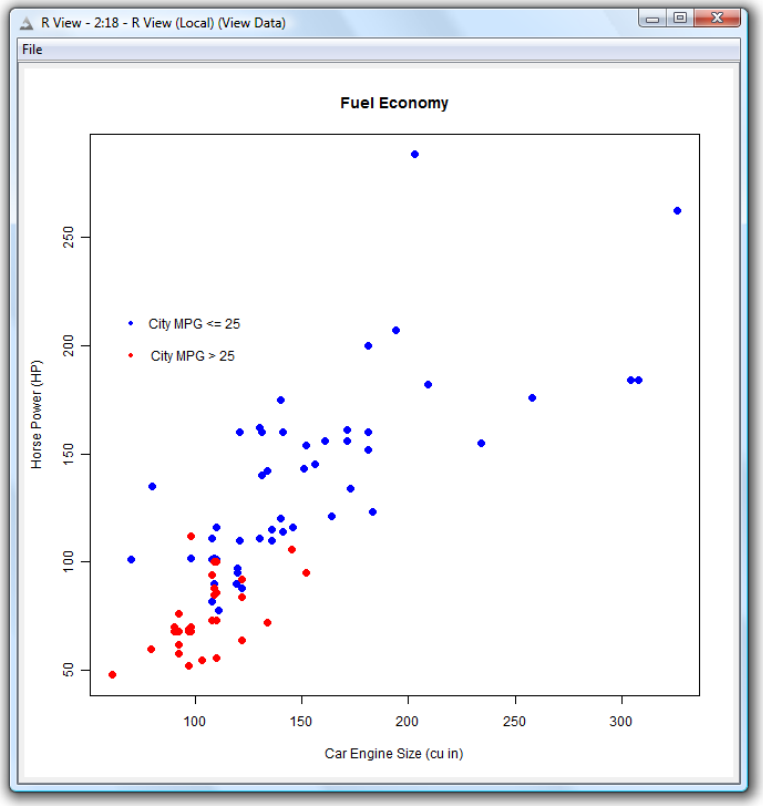
Note. You should **NOT** use R commands like *png()* or *bmp()* to produce the PNG output of the plot. In fact, functions like these open a new graphic device to render the plot. As a result, the graphic device used by KNIME in the “R View” node is not used anymore and the node’s view shows an empty image.

The configuration and the view window of the “R View” node are shown in the following figures.

5.18. Configuration window of the “R View” node for a “Generic X-Y” Plot



5.19. View of the “R View” node implementing a “Generic X-Y Plot”



Box Plot

Similar to the generic X-Y plot, a number of other plots can be implemented in the "R View" node. Let's see here how to implement a box-plot. The main difference with the previous plot consists in the usage of the function boxplot() to produce a box-and-whisker plot of the input vector of numerical values.

We built a new workflow in the "Plotting Examples 1" workflow, in order to show how to create a box plot view. The new workflow is similar to the one that implemented the X-Y generic plot (Fig. 5.18). Again it reads in the "cars-85.csv" data file, renames the column headers, and plots the engine size, engine horse power, and the city and highway fuel consumption but this time with a boxplot() function.

The following piece of R code was written into the "R Snippet" panel of the "R View" node.

```
#Background

par(bg="powderblue")


#Boxplot
boxplot(

      knime.in[c(17,22,24,25)],
      col="lavenderblush",
      notch=TRUE)


#Title
title(main="Car Performance vs Efficiency",
      xlab="Category",
      font.main=4,
      font.lab=1)
```

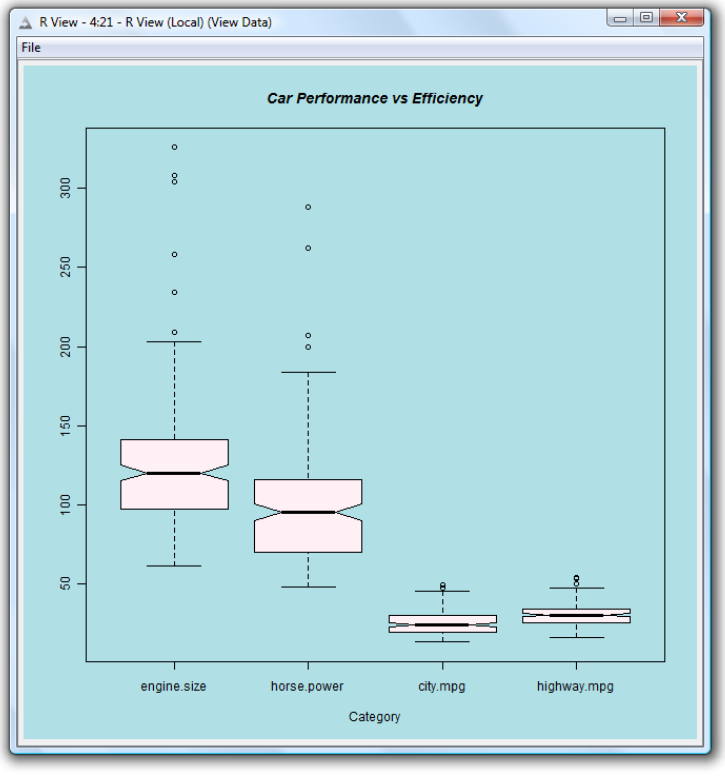
The background of the plot is first drawn using the parameter function `par` and setting the background parameter `bg` to one of the many color options within R. Use the `colours()` function to discover more interesting colors.

We want to implement a box plot.
The data for the box plot can either be a numeric vector or a single list containing such vectors. For this box plot we use specific columns from the data table (engine size, engine horse and the city and highway fuel consumption). Even using the columns names, *e.g.* `knime.in$"engine.size"`, instead of the column indices still results in the column names being lost in the plot and instead the column numbers being displayed.

A title is added using the `title()` function.

The `font.main` parameter takes an integer value to specify which font type to use for text: 1 for plain text (default), 2 for bold face, 3 for italic, and 4 for bold italic.

5.20. View of the box plot example



Executing and running the “R View” node with the R script above results in the graphical display of a box plot function (workflow “Plotting Examples 1”). This is a typical box plot and includes the following features:

- A notch with a horizontal bold line indicates the median
- The box above the median is the top third quartile and the box below is the first quartile
- The whiskers above and below the box display the range of the data set without the outliers
- The small circles indicate the outliers, which are defined as any value that is greater than 1.5 times the inter-quartile range (Q3-Q1) away from the edge of the box.

Bar Plot

To create a bar chart we used the `barplot()` function. This simple function assumes that the data contains the bar heights in a vector. More often though the data table contains a vector of numerical data and requires a parallel factor to group the data, like a group average or any other data aggregation.

We decided to work on the "cars-85.csv" data file again and to examine the average prices of the different makes of car. There are many ways of calculating the average prices, including within the R Snippet. We used a "GroupBy" node with group column = "make", aggregation column = "price", and aggregation method = "Mean". The output column containing the average prices was then called "Mean(price)", which would prompt a warning in the "R View" node about the column renaming from "Mean(price)" to "Mean.price.". To avoid the warning message, we used a "Rename" node to do exactly this. We limited the plotting to the first 6 rows for the clarity of the plot picture. Finally, we placed an "R View" node to run the `barplot()` function in R. Here is the R code used in the "R Snippet" of the "R View" node.

```
#Set Graphical Parameters
par(las=1,col.axis="royalblue")
```

The graphical parameters are set using the `par` function. The `las` parameter is used to align the text along the axes.

```
#Set colouring Scheme
len=length(knime.in$"make")

col <- colours()

mybarcol <- "gray20"
```

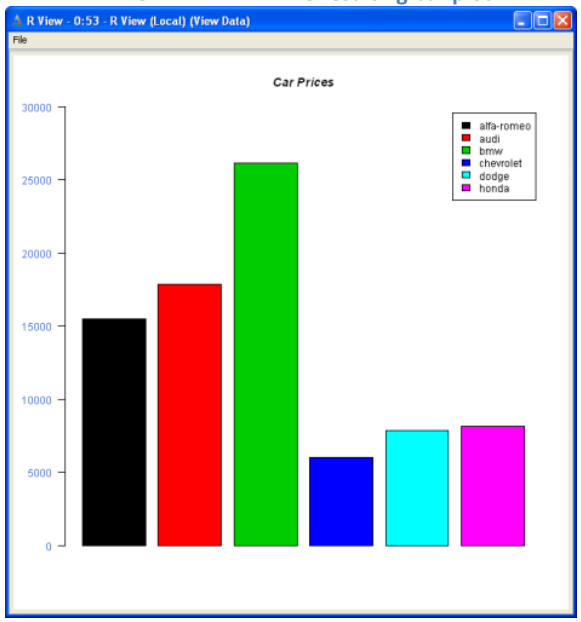
As a means to differentiate the bars generated by the plot a color gradient is established using a vector of colors from the R palette. The default color settings are `gray` if the `height` is a vector, and a gamma-corrected gray palette if `height` is a matrix.

```
#Bar Plot
barplot(
      knime.in$"average.price",
      col = c(1:len),

      legend = knime.in$"make",
      ylim= c(0,30000),
      main = "Car Prices",
      font.main = 4,
)
```

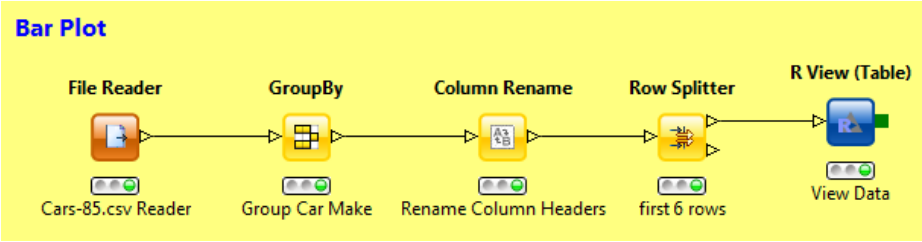
This is the `barplot()` command.

We have created a legend, because it is easier to read than having the bar individually labeled.
The `ylim` parameter sets the limit of the y axis.

5.21. The resulting bar plot



5.22. The workflow used to build the bar plot (in "Plotting Examples 1")



Histograms

The command `hist(x)` draws a simple histogram of a data column, where `x` is the numerical data column and bar widths are determined automatically. The bar widths, or breaks, can be defined in either one of four methods:

- a vector of breakpoints,
- a single number of suggested bars,
- the name of a particular binning algorithm,
- another function to compute bins.

As with the second method, the last two are also treated as suggested bin widths, and R adapts the histogram according to the suggestion. By default, both the heights of bars and the areas are proportional to the number of data points falling within the bin, given that the breaks are equally spaced. In the case where space between breaks is non-equidistant, this results in a plot of unit area. Here the area of the bars is a fraction of the data within each bar.

Using our favorite “cars-85.csv” data file and performing the usual column renaming step (see plotting workflows in the sections above) we then plot a histogram of the car `price` column. Here is the R code:

```
#Background
par(bg="cornsilk")

#Plot Histogram

r <- hist(
      knime.in$"price",
      br = c(5000*0:10),
      col='slateblue1',
      main="Car Prices",
      font.main=2,
      xlab = "Price"
      )
text(
      r$mids, r$counts,
      r$counts, adj=c(.5,
      -.5), col='blue1')
lines(r, lty = 1, border
= "red")

#Add a Box
box()
```

We have assigned the `r` variable to be the object of class `histogram`. This object has a number of components which can then be called to add information about the histogram to the plot. To see the contents of `r`, use the `edit()` function: `edit(r)`

We have chosen to define the breaks using a vector, `br = c(5000*0:10)`. This returns 10 equally spaced bars of a length of 5000.

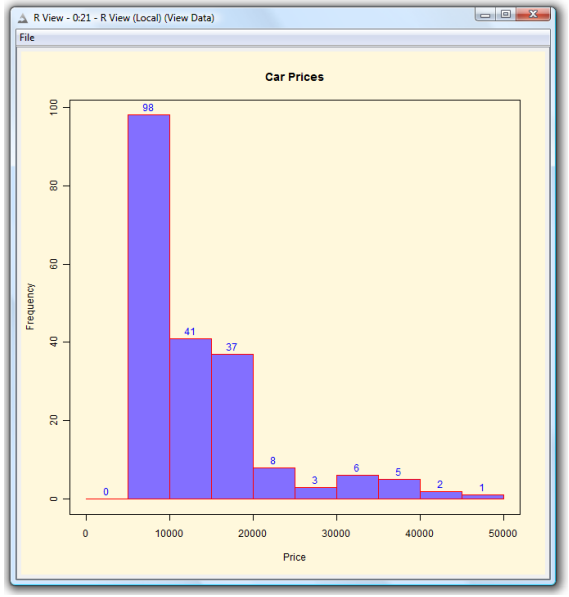
The `text()` function makes use of the `r` object components called `mids`, which returns the midpoints for the bar, and `counts`, the number within the bar. Using this function we can add text showing the size of bar, positioned in the middle of the bar or slightly above it.

We use the `breaks` component of `r` together with the `counts` component to define a line vector in order to highlight the bars of the histogram.

Finally, to make the histogram more attractive we’ve added a box around the plot.

The workflow “Plotting Examples 2” contains the workflow implemented to plot the histogram.

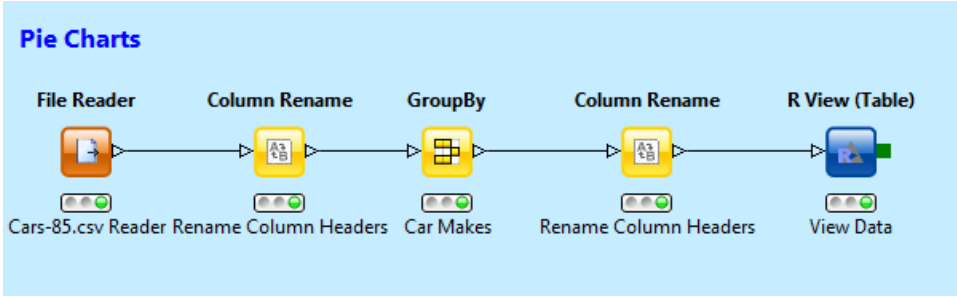
5.23. Histogram of Car Prices



Pie Charts

This type of plot is created using the `pie()` function. Although pie charts are not considered the best method for data visualization they are often used in business and in the media to represent data. In this example we plot the average of the engine sizes for each of the car makes in the data set as a fraction. First we read in the "cars-85.csv" data file and perform the usual column renaming. Then the "`GroupBy`" node calculates the mean of the engine sizes for each car make. So that we don't have any R warnings, we rename the "`Mean(engine.size)`" column to "`Mean.engine.size`".

5.24. Workflow „Plotting Examples 2" to build the Pie Chart



In this example, we used more R functionalities to create the data required for the pie chart.

```
#Calculate Fractions
a<-sum(knime.in$"Mean.engine.size")
cars<-knime.in$"Mean.engine.size"/a

#Pie Chart
size<-length(knime.in$"make")
names(cars)<-knime.in$"make"
pie(
      cars,

      col = rainbow(size),
      radius = 0.9
      )

#Title
title(main="Comparison of Engine Sizes",
      font.main = 2)
```

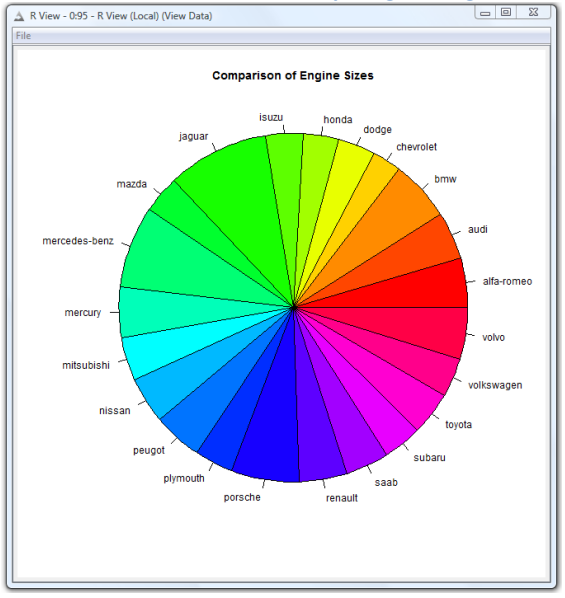
These first two lines find the total for the `Mean.engine.size` column and determine the relationship of the engine size of one car make to the others.

The `size()` function gives the number of rows in the data set.
Here we assign names to our object using the `names()` function.

The `pie()` function takes a vector of non-negative numerical quantities as its first argument which are displayed as the areas of pie slices in the chart.

We chose to color the pie chart using the `rainbow()` function, assigning the number of colors to use based on the number of rows, i.e. different makes of car, of the data set.

5.25. Pie Chart Comparing Car Engine Sizes



Scatter Plot Matrices

The function `pairs()` produces a matrix of scatter plots. When plotting in a matrix, panels within the matrix can be assigned a different plotting function. This is a very useful way to represent a lot of data in a clear fashion. In this example, we plot a number of scatter plots comparing various car parameters from the "cars-85.csv" data set. Along the main diagonal are the car parameters which we are interested in comparing. We display the R^2 coefficient for the plotted data points in the panels above the main diagonal, the upper panels. In the lower panels we graph the points and add a best fit regression line to the plot. Here is the R code to produce this desired scatter plot matrix.

The upper and lower panels take the `function(x,y,...)` function, which is used to plot the contents of each panel of the display.

```
#Function for Upper Panel
panel.r <- function(
      x, y,
      digits=2,
      prefix="", ...
      )
{
      usr <- par("usr")
      par(usr = c(0, 1, 0, 1))
      r <- summary(fit<-lm(y~x))$r.squared
      txt <- format(r, digits=digits)[1]
      text(0.5, 0.5, txt, cex = 2)
}
#Function for Lower Panel
panel.fit <- function(x,y)
{
      points(x, y)
      fit<-lm(y~x)
      abline(fit, col="red")

}
#Plot Scatter Matrix
pairs(
      knime.in[10:14],
      lower.panel=panel.fit,
      upper.panel=panel.r
      )
```

The upper function calculates the R^2 for each plot and assigns this to the variable `r`. The value is returned and printed as text

The `digits=2` argument specifies the desired number of significant figures to print.

The coordinates of the text are assigned here by defining a panel size with `par(usr = c(0, 1, 0, 1))`

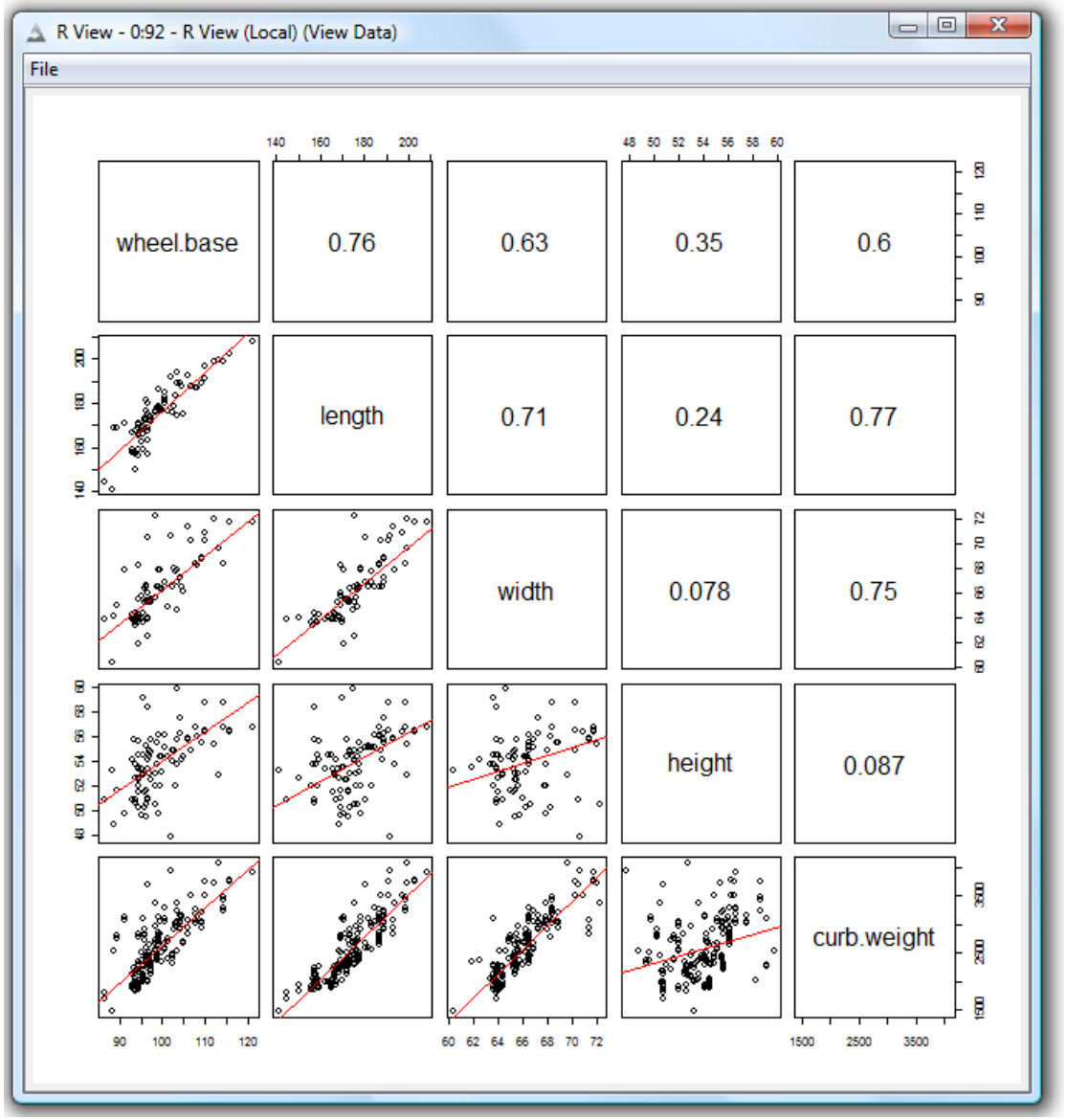
Here we position the text and we set the text size using `cex` (character (or symbol) expansion).

The lower function plots the scatter plot for each matrix panel using the `points()` function.

A line of best fit is added to each graph using the `abline()` function. As its argument it takes the y-intercept and the gradient of the line as calculated by the `lm()` function which is used to fit a linear model with the form response ~ terms, in this case `y ~ x`.

The final plot is obtained by calling the `pairs()` function and calling the columns over which we want to plot first, followed by the two functions, `panel.fit` and `panel.r`.

5.26. The Scatter Matrix from the “R View” node



Functions Plots and Polygon Drawing

As well as the multitude of plots available for data analysis and visualization, R also has mathematical functions available for plotting. This can be achieved by using the `curve()`, the `plot()`, or the `polygon()` function. This section explores the usage of the `curve()`, `plot()`, and `polygon()` function and of a number of mathematical functions that can be used to improve the plots.

```
op <- par(mfrow=c(2,2))
#Plot 1

curve(
      sin(x),
      -2, 2,
      lwd=2, lty=3, ylab="y"
      )
curve(
      x^2-1, add = TRUE,
      col = "violet",
      lwd=2, lty=3
      )
#Plot 2
theta=seq(-20,20,0.002)
r<-2*sin(4*theta)
x=r*cos(theta)
y=r*sin(theta)
plot(x,y,type="l",col="gold",lwd=2)
r2<-sin(theta/1.25)
x2=r2*cos(theta)
y2=r2*sin(theta)
lines(x2,y2,type="l",col="black",lwd=2)

#Plot 3
x4<-seq(-10,10,length=400)
y4<-dnorm(x4,m=0,s=1)
par(mar=c(6,4,2,1))
plot(
      x4, y4, xlim=c(-4,4),
      type="n",
      lab=quote(
```

First, we tell R to plot the graphs in a 2 by 2 array using the `mfrow()` function. We want to produce 4 plots, one for each cell of the 2 by 2 matrix.

"Plot 1" is a line graph using the `curve()` function to plot the function of `sin(x)`.

The next arguments are `from` and `to` (-2,2) which is the range of the plot.
The line width is set at `lwd=2`, and the line type as dotted using the `lty=3` parameter.

We also plot a simple function of x, in this instance `x^2-1`, the symbol ^ stands for raised to the power of. We can plot multiple graphs together by using the `add=TRUE` command.

"Plot 2" is a bit of fun to demonstrate that the plot() function does not need to be a graph of points; in fact elaborate functions can also be drawn easily. Here we used two polar coordinate function sets.

The first polar coordinate is drawn as a line plot using the `plot()` function `type="l"`.

The second polar coordinate is drawn using the function `lines()`.

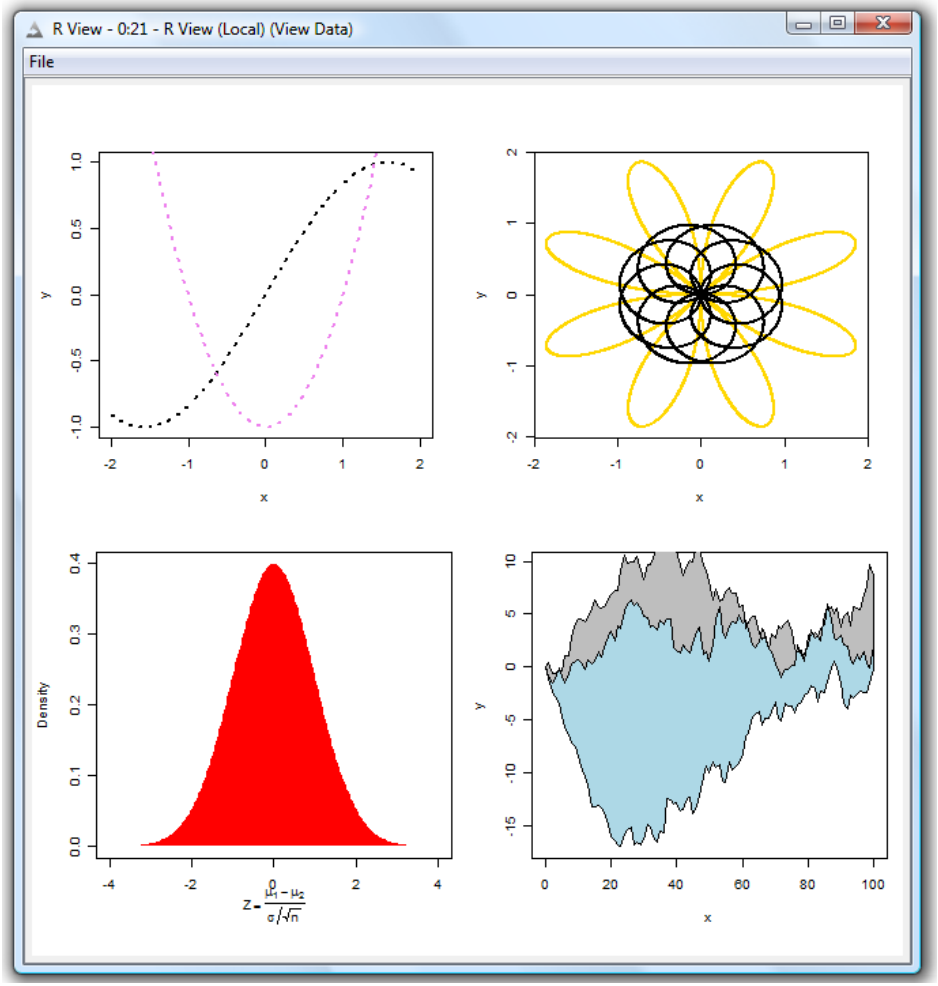
"Plot 3" and "Plot 4" demonstrate how plots can be filled with polygons.

```
            Z==frac(mu[1]-mu[2],
            sigma/sqrt(n))
            ),
      ylab="Density"
      )

polygon(
      c(1.96,1.96,x4[0:400],10),

      c(0,dnorm(-5,m=0),y4[0:400],0),
      col="red", lty=0
)
#Plot 4
n<-100
set.seed(45000)
x3a<-c(0,cumsum(rnorm(n)))
y3a<-c(0,cumsum(rnorm(n)))
y4a<-c(0,cumsum(rnorm(n)))
x5a<-c(0:n, n:0)
y5a<-c(x3a, rev(y3a))
y6a<-c(x3a, rev(y4a))

plot(
      x5a, y5a, type="n",
      xlab="x", ylab="y"
      )

polygon(
      x5a, y5a, col="lightblue"
      )
lines(x5a,y6a,type="n")
polygon(x5a, y6a, col="grey")
```

The `polygon()` function draws a polygon which has its vertices given by vectors x and y. The coordinates of these vectors can be passed in a similar way to the plotting structure, as a list with x and y components.
Here we make use of the normal distribution density function `dnorm()` which takes a vector of quantiles, a value for the mean and the standard deviation, in this case the value of `y4` takes the values of `x4`, `m=0`, and `s=1` respectively.

The KNIME workflow to create this plot is “Plotting Examples 3”.

5.27. Demonstration of Draw Functions Plots



Contours Plots

The plotting function `contour()` creates a contoured plot or it adds contour lines to an existing plot. This function can either be written as `contour(x,y,z,...)` or `contour(z,...)` . The first version gives the locations (`x`,`y`) in ascending order of grid lines at which the values in `z` are measured. The second version plots the values in matrix `z`. By default the plot creates equally spaced values from 0 to 1.

A contour plot would not be complete without a description of the size of the contours or a graphical representation of the gradients. Labels on the contours are defined using the `method` parameter. The default method for positioning the labels on contours is `flattest` (`method = "flattest"`) which adds labels to the flattest section of the contour, embedded in the contour line and with no overlapping labels. Two other methods are `simple`, which adds labels at the edge of the plot overlaying the contour line, and `edge`, which adds labels at the edge of the plot embedded in the contour line with no labels overlapping. These last two methods may not draw a label on every contour line.

The number of contour lines added is determined by the `nlevels` parameter.

```
#Parameters
par(bg = "white",mar=rep(2,4))


#Function

y <- x <- seq(knime.in$"x", 10, length = 50)

rotsinc <- function(x,y){


    sinc <- function(w){
          u <- sin(w)/w;
        u[is.na(u)] <- 1;
        u;
    }

    sinc( sqrt(x^2+y^2) )

}
sinc.exp <- expression(
      z == sinc(sqrt(x^2 + y^2))
      )


z <- outer(x, y, rotsinc)


z0 <- min(z) - 0.5
z <- rbind(z0, cbind(z0, z, z0), z0)

#Plot Contours
```

In this example we use a function to create the data to plot in order to demonstrate some of the mathematical functionality of R.
First we have defined the vectors `x` and `y`, making them both equal to a sequence of numbers, in this case -10 to 10 over intervals of 20/length-1.
In this line we define the first function, `rotsinc`, which requires two arguments `(x,y)` and returns one value as:

`sinc( sqrt(x^2+y^2)`.

The `sinc()` function takes one argument: w. First it calculates `sin(w)/w`, and assigns this value to `u`; then in the event that the result is NaN, not a number, returns the value 1.

The return value is evaluated as the square root of the sum of the squares of `x` and `y` and processed by the `sinc()` function.

This is an expression, in the literal sense, and is used later on as the title for the plot.

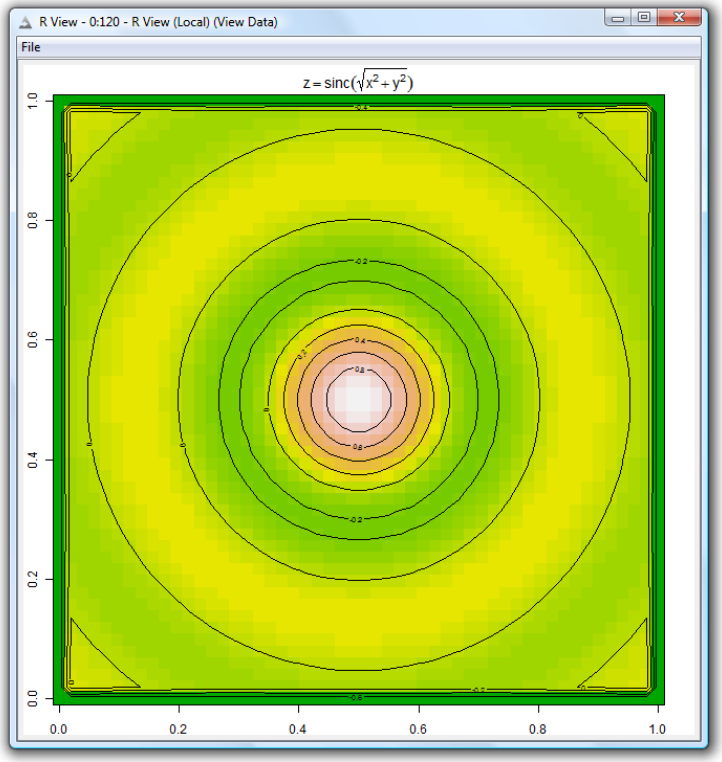
This function produces the outer product of the arrays `X` and `Y`. X and Y are built as an array `A` with dimensions `c(dim(X), dim(Y))` where an element (`A[c(arrayindex.x, arrayindex.y)]`) in the array `A` is equal to a function operating on (`X[arrayindex.x], Y[arrayindex.y], ...`). The function in this case is `rotsinc`.
These lines create the variable `z0` and a new value for `z` and are there to create a boundary around the plot which help to accentuate the gradients.

```
image(
      z,col= terrain.colors(50),
      axes = TRUE
      )
contour(
      z,add=TRUE,
      nlevels=10,
      col="black"
      )

#Labels
title(main = sinc.exp)
box()
```

Then to plot the contours we first create an image and colour it using the `terrain.colors()` function.

Then we add the contours with the `contour()` function.

5.28. Contour Plot of the Sinc Function (workflow "Plotting Examples 3")



Perspective Plot

The contour plot produces a 2 dimensional image, but sometimes a 3 dimensional image has a greater impact and the results are easier to interpret. To generate images which have a perspective, use the `persp()` function: the end result is a perspective plot of surfaces over the x–y plane. The plots are produced by first transforming the coordinates to the interval [0,1]. The surface is then viewed by looking at the origin from a direction defined by two attributes: `theta` and `phi`. When these parameters are both set to zero the viewing direction is directly down the negative y axis. In spherical coordinated terms, changes in `theta` vary the `azimuth` , while changing `phi` varies the `colatitude`.

The `persp()` function interprets the `z`-matrix as a table of $f(x[i], y[i])$ values. In this fashion the x-axis corresponds to the row numbers and the y-axis corresponds to column numbers. With the standard rotation angles, with column 1 at the bottom, the top left corner of the matrix is displayed at the left hand side, closest to the user.

```
#Parameters
par(bg = "white",mar=rep(2,4))

#Function
y <- x <- seq(knime.in$"x", 10, length = 50)

rotsinc <- function(x,y){
    sinc <- function(w){
          u <- sin(w)/w;
        u[is.na(u)] <- 1;
        u;
    }
    sinc( sqrt(x^2+y^2) )
}

sinc.exp <- expression(z == Sinc(sqrt(x^2 + y^2)))
z <- outer(x, y, rotsinc)
data <- z

z0 <- min(z) - 0.5
z <- rbind(z0, cbind(z0, z, z0), z0)

x <- c(min(x) - 1e-10, x, max(x) + 1e-10)
y <- c(min(y) - 1e-10, y, max(y) + 1e-10)

#Color Fill
fill <- matrix("blue", nr = nrow(z)-1, nc = ncol(z)-1)
```

Here we adapted the R code written for the contour example above, but we colored the surface using a gradient defined by the population of the z- axis. We used the same function to define `sinc()`.

The colors used to fill the surface are the main difference.

First we fill in all the elements of the z-matrix with default colors: blue and gray.

```
fill[ , i2 <- c(1,ncol(fill))] <- "grey"
fill[i1 <- c(1,nrow(fill)) , ] <- "grey"
fcol <- fill
zi <- data[ -1,-1] + data[ -1,-50] + data[-50,-1] +
data[-50,-50]
```

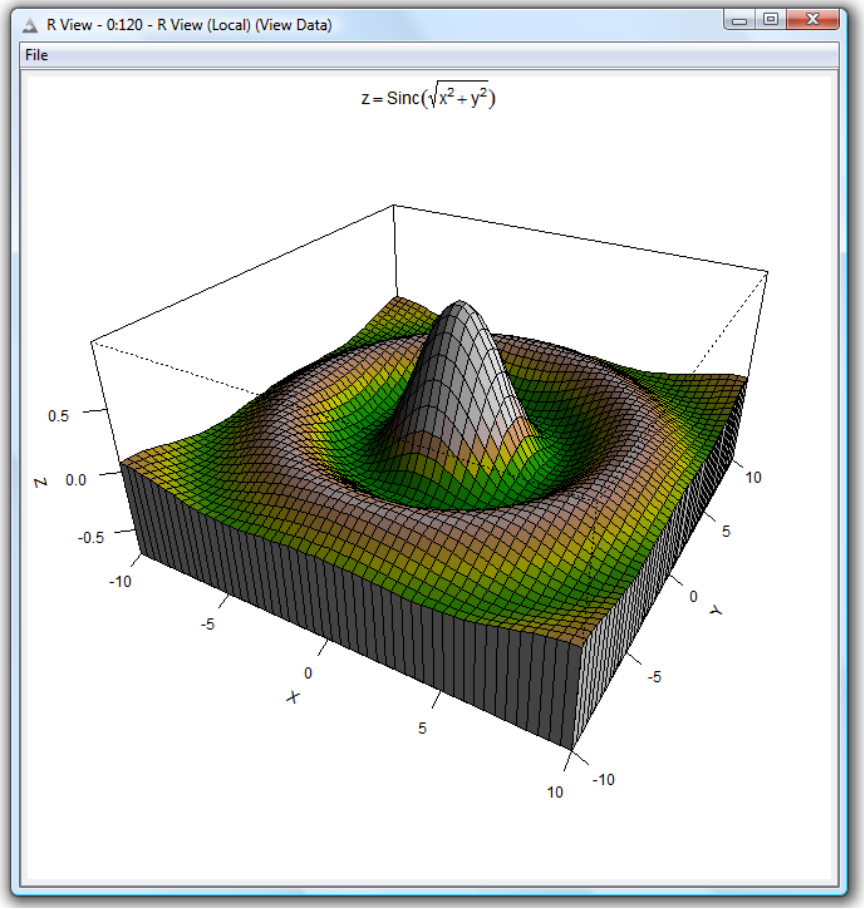
This is the line of code to quantile the data to fill it with colors.
The `terrain.colors(x)` function creates a topographical color schemes suitable for displaying ordered data, where x is the number of desired colors, in our case 20.
The boundaries for each color are defined by using break points. The `cut()` function produces a vector of intervals which are closed on the right and open on the left except for the lowest interval, for example:
`20 Levels: [-0.851,-0.782] (-0.782,-0.614] (-0.614,-0.397] (-0.397,-0.351] ... (0.98,3.94]`
The `cut(x)` function takes a numeric vector and divides the range of `x` into a number of intervals, coding the `x` values according to which interval they fall.
In this case the intervals are defined by the `quantile()` function. The `quantile()` function produces sample quantiles corresponding to the given probabilities that are stored in a numeric vector with values in [0,1]. The smallest observation corresponds to a probability of 0 and the largest to a probability of 1.

```
fcol[-i1,-i2] <- terrain.colors(20)[
      cut(
            zi,
            quantile(
                  zi,
                  seq(0,1, len = 21)
            ),
      include.lowest = TRUE)
]
```

```
#Plot Perspective
persp(
      x, y, z,
      theta = 30, phi = 30,
      expand = 0.5,
      col = fcol, ltheta = 120,
      shade = 0.75, ticktype = "detailed",
      xlab = "X", ylab = "Y", zlab = "Z"
      )

#Labels
title(main = sinc.exp)
```

The perspective is then adjusted by changing the `theta` and `phi` values. The z scaling is adjusted by the `expansion` parameter in `persp()`.
In this example we reduced the z scaling so that the resulting plot does not look deformed.

5.29. Perspective Plot of the Sinc Function



5.8. Statistical Models Using the "R Learner", "R Predictor", and R IO Nodes

There are two nodes available to help build and use statistical models using R: the "R Learner" node and the "R Predictor" node. Models built using the "R Learner" node are returned to the output port and can be used by the "R Predictor" node to predict unseen data. Both the "R Learner" and the "R Predictor" nodes are available in the "Local" version of R. No "R Learner" and "R Predictor" nodes are available in the "Remote" version.

R Learner

The "R Learner" node receives data at the input port and returns the model built on these data at the output port.

The "**R Script**" panel in the center hosts the R code that will be executed via the R engine.

The "**Column List**" in the top left corner contains the list of available input data columns.

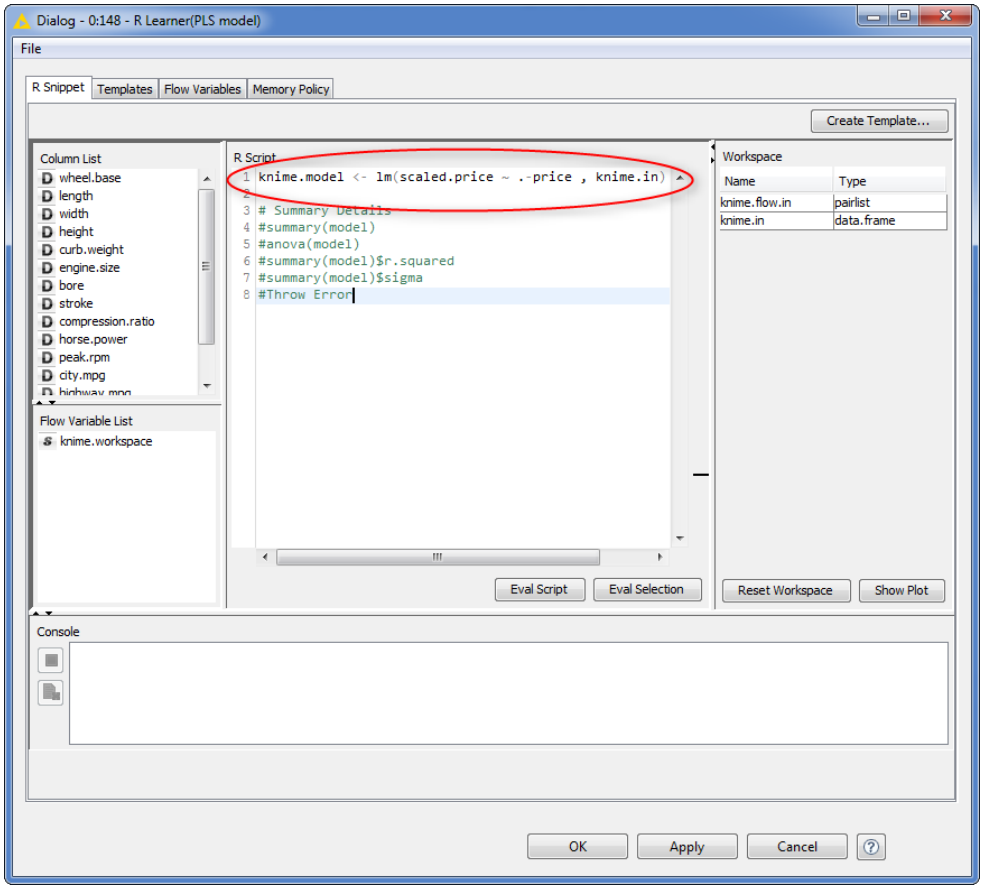
The "**Flow Variable List**" in the bottom left corner contains the list of the flow variables available for this node.

Double-clicking a data column or a flow variable automatically inserts it into the R Script editor with the correct reference syntax.

Also with regards to the "R Learner" node, data is passed via CSV files: the input data is written to a CSV file, imported into R via the variable named `knime.in`.

The final model must be returned by the R script within a variable named `knime.model` and it is passed to KNIME via a CSV file as well.

5.30. The configuration window of the "R Learner" node



There is no "(Workspace)" version of this node. To train a model only in the R workspace you need to use the "R To R" node.

The difference between an "R Learner" node and an "R Snippet" node is in the output port, which produces a model for the "R Learner" node and data for the "R Snippet" node.

R Predictor

The "R Predictor" node has two input ports: one for the model used to predict data; the other for the input data. The predicted data is returned at the data output port.

The "**R Script**" panel in the center hosts the R code that will be executed via the R engine.

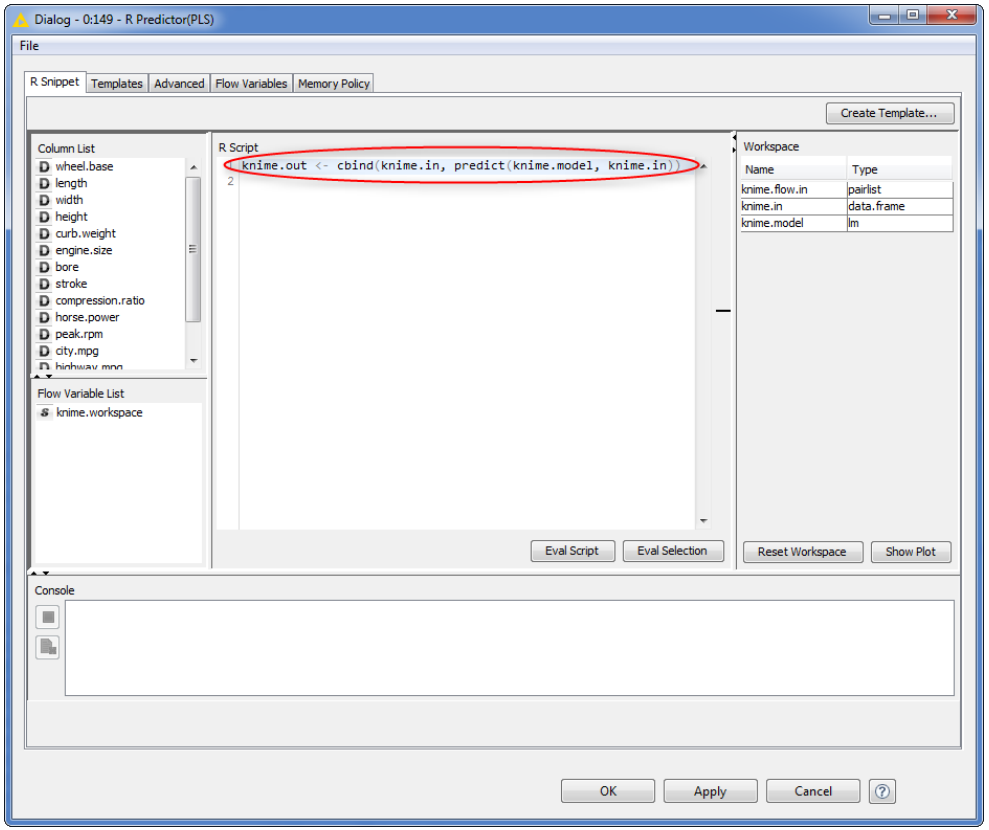
The "**Column List**" in the top left corner contains the list of available input data columns.

The "**Flow Variable List**" in the bottom left corner contains the list of the flow variables available for this node.

Double-clicking a data column or a flow variable automatically adds it into the R Script editor with the correct reference syntax.

As for all the other R nodes, data transmission happens via CSV files. The input data is written to a CSV file and imported into R via the variable named `knime.in`. The input model is imported into R via the variable `knime.model`.

5.31. The configuration window of the "R Predictor" node



The final data must be returned via the `knime.out` variable.

The default R prediction command is: `knime.out<-cbind(knime.in, predict(knime.model, knime.in));`

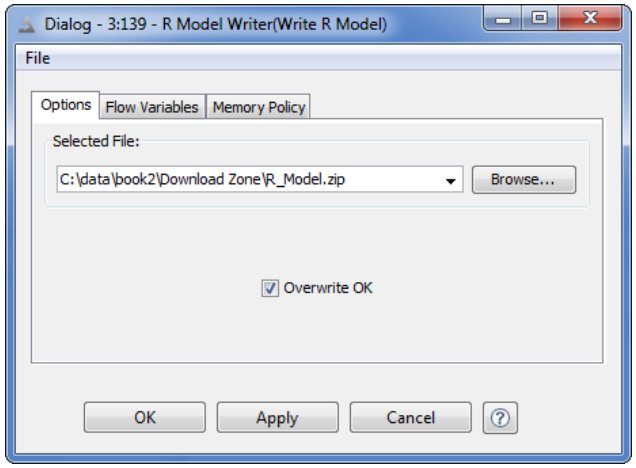
R Model Writer

The “R Model Writer” node takes an R model as input and writes it to a file.

The configuration window then requires:

- The path of the output file
- The overwriting flag

5.32. The configuration window of the “R Model Writer” node

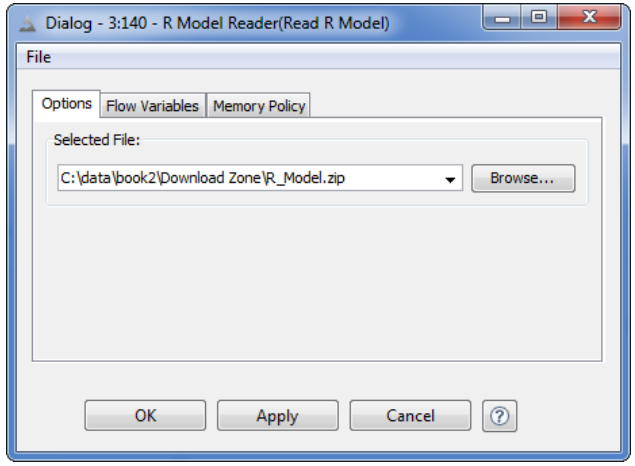


R Model Reader

The “R Model Reader” node reads an R model from a zip file (for example written by an “R Model Writer” node) and makes it available at the output port.

Its configuration window requires only the path of the file containing the R model.

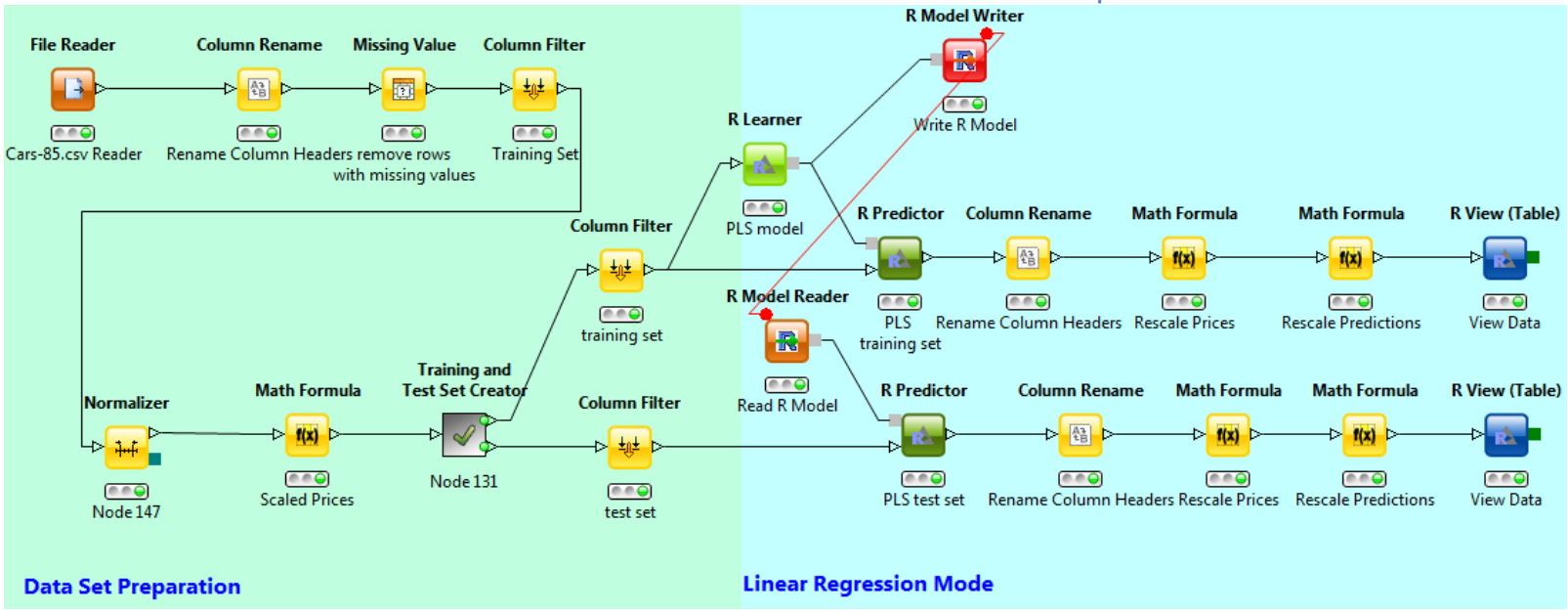
5.33. The configuration window of the “R Model Reader” node



Using the familiar “cars-85.csv” data set, we built an R model for this section in the “R Model Example” workflow. First, we split the data to form the training data set and the test data set by means of the KNIME K-Means algorithm. Clustering using K-Means allows representative training and test sets to be created and distributed by an 80:20 split of the original data set. We then used a multiple linear regression to build a model to predict the car prices, using “price” as the response variable. The predictor variables for this calculation were all other car attributes. Once we created the model, we exported it for future use using the “R Model Writer” node. Similarly we can use the “R Model Reader” node to read in a model, once it has been created.

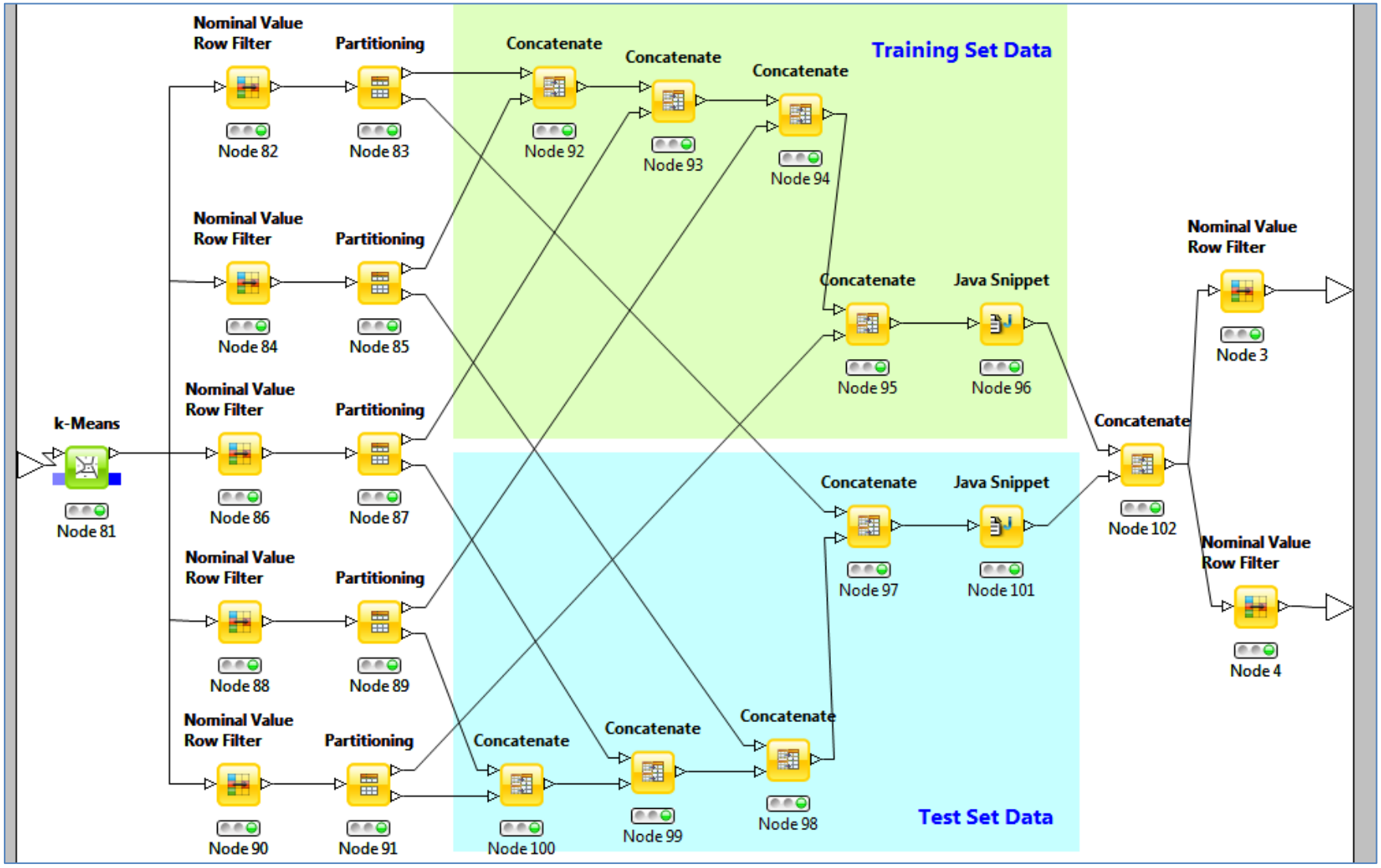
The next two figures show the resulting workflow, ”R Model Example”, with the meta-node to create the training set and the test set based on the K-Means algorithm, the “R Learner” node implementing the multiple regression on the variable “price”, the “R Predictor” node to predict car prices on the training and test data, an “R Model Writer” to write the resulting model, an “R Model Reader” node to read in the model and test it on the test data, and two “R View” nodes to visualize the results.

5.34. “R Learner” and “R Predictor” nodes in the “R Model Example” workflow



Note. The “R Model Writer” node and the “R Model Reader” node are connected via a flow variable connection, even though no flow variables are used in the “R Model Reader” node. This is just to ensure the dependency of the “R Model Reader” node from the “R Model Writer” node. With such connection the “R Model Reader” node will execute only once the “R Model Writer” node has finished executing.

5.35. Meta node “Training and Test Set Creator”: Data Set Splitting Using K-Means Clustering



Linear Regression in R

The linear regression that we used in R is an ordinary least squares based linear regression, to produce a linear model of the predictor variables (u,v,w *etc*) to find the response variable (y):

$$y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 w_i + \epsilon_i$$

where β_0 , β_1 , β_2 and β_3 are the regression coefficients and ϵ_i is the error term. The R function to perform the linear regression is one we have met previously in Scatter Plot Matrices, known as the `lm()` function. The argument in the function is a model formula of the type `y ~ x`. The formula syntax places the response variable on the left separated from the predictor variable on the right by a tilde character. Multiple predictor variables can be added using the plus sign.

```
lm(y ~ u + v + w)
```

The `lm()` function estimates the regression coefficients, β_0 and β_1, β_2, and β_3 , and reports them respectively as the intercept and the coefficients of the predictor variables, u, v and w. This same function was used in the Scatter Plot Matrices example to generate a simple linear regression model. Let us now estimate a model from the "cars-85.csv" training data set.

```
model<-lm(scaled.price ~ .-price , knime.in)
knime.model<-model
```

Here we have used the full stop (.) to specify that we want to use all the columns less than the response variable (`price`). The output of `lm()` is the model object named "`model`". Once run, the important statistical information about the model including the R^2, the *F* statistic, the residuals, the confidence intervals for the coefficients and a variety of other pieces of data can be extracted. Most of this statistical information can be found using the function `summary()`, as applied below:

```
# Summary Details
summary(model)
anova(model)
summary(model)$r.squared
summary(model)$sigma
Throw Error
```

As the R integration in KNIME is not offering a console, we use a trick to fetch the textual output: We throw an error and look at the output in the error output view. This is done by writing some non-R code ("`Throw Error`"), and check the results in the "View: R Error Output" node. Here is the output from the `summary(model)`:

```
> model<-lm(scaled.price ~ .-price , knime.in)
> knime.model<-model
>
> # Summary Details
> summary(model)

Call:
lm(formula = scaled.price ~ . - price, data = knime.in)

Residuals:
     Min       1Q   Median       3Q      Max
-1.26821 -0.23529 -0.04692  0.20096  1.44877

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)         0.6568     0.5361   1.225 0.223104
wheel.base          1.3744     0.5848   2.350 0.020535 *
length             -0.4846     0.6542  -0.741 0.460331
width               0.8799     0.5862   1.501 0.136192
height             -0.1262     0.2985  -0.423 0.673235
curb.weight         2.1434     0.8850   2.422 0.017059 *
engine.size         2.9089     0.8093   3.594 0.000486 ***
bore               -0.9358     0.3292  -2.842 0.005332 **
stroke             -0.9235     0.3522  -2.622 0.009963 **
compression.ratio   0.2485     0.2800   0.887 0.376783
horse.power         0.7446     0.5833   1.276 0.204496
peak.rpm            0.4046     0.3033   1.334 0.184886
city.mpg            0.4612     1.2533   0.368 0.713601
highway.mpg        -0.6888     1.1236  -0.613 0.541131
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4723 on 111 degrees of freedom
Multiple R-squared: 0.8519,    Adjusted R-squared: 0.8346
F-statistic: 49.11 on 13 and 111 DF,  p-value: < 2.2e-16
```

And the following is the summary of the `anova(model)`:

```
> anova(model)
Analysis of Variance Table

Response: scaled.price
                   Df Sum Sq Mean Sq  F value    Pr(>F)
wheel.base          1 86.536  86.536 387.9102 < 2.2e-16 ***
length              1 13.106  13.106  58.7507 7.288e-12 ***
width               1 15.386  15.386  68.9712 2.705e-13 ***
height              1  2.738   2.738  12.2741 0.0006631 ***
curb.weight         1 17.886  17.886  80.1779 9.067e-15 ***
engine.size         1  2.078   2.078   9.3160 0.0028428 **
bore                1  1.383   1.383   6.1995 0.0142607 *
stroke              1  1.491   1.491   6.6858 0.0110137 *
compression.ratio   1  0.098   0.098   0.4378 0.5095571
horse.power         1  1.167   1.167   5.2292 0.0241076 *
peak.rpm            1  0.467   0.467   2.0947 0.1506287
city.mpg            1  0.016   0.016   0.0709 0.7905757
highway.mpg         1  0.084   0.084   0.3758 0.5411309
Residuals         111 24.762   0.223
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The results can also be sent to the data output; however the significance stars will be lost. The R^2 value has to be extracted from the summary results using `$r.squared`, the same for residual standard error (σ) using $sigma.

```
> summary(model)$r.squared
[1] 0.8519
> summary(model)$sigma
[1] 0.472316
```

5.9. R To PMML Node and other Conversion Nodes

Predictive Model Markup Language (PMML) is a standard for statistical and data mining models. A model written in PMML allows for it to be developed in one particular type of system and utilized in another system. In KNIME, a model created as an R object can be converted into a corresponding PMML object which can then be used in conjunction with all KNIME predictors supporting PMML. The R object is first loaded into a new R workspace and then converted into PMML (using the PMML library in R). The `toString()` method in R is used to generate a character stream describing the PMML object which is written to the PMML output port.

Let us demonstrate the use of PMML using a decision tree generated by recursive partitioning of the data set. The decision tree that is built is used to predict the classification of members of a data set. Recursive partitioning is a fundamental tool in data mining and was first introduced in Chapter 4 of the "KNIME Beginner's Luck" book. This method allows for the structure of a set of data to be efficiently interrogated through easy to visualize decision rules for predicting a factors (classification tree) or continuous (regression tree) outcomes.

Let us look at how to build a decision tree model using R and the "R To PMML" node.

The following example R script learns a decision tree based on the "cars-85.csv" file and exports this as a PMML and as an R model which is understood by the "R Predictor" node.

```
# load the library for learning a tree model
library(rpart);

# load the pmml export library
library(pmml);

# use class column as predicted column
dtree<-rpart(prediction ~ ., R)

# export to PMML
r_pmml <- pmml(dtree, app.name="KNIME")

# write the PMML model to an export file
```

To generate the regression tree used in this example we have used the `rpart()` package.

Generally exporting the PMML model from R is not recommended (see the note below). To circumvent version issues use the app.name which tells the reader to import the PMML irrespective of the version.

```
write(toString(r_pmml), file="C:/data/R.pmml")
```

The R object can be converted to PMML directly. The file must be saved to a writable location.

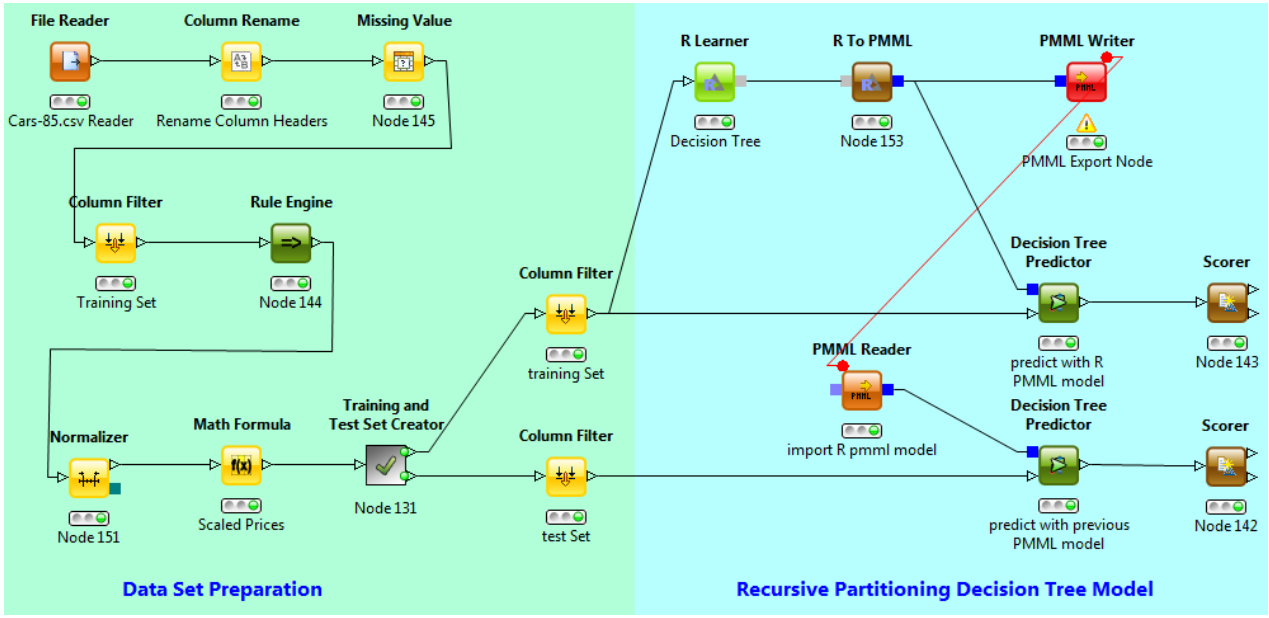
```
# provide the native R model at the output port
knime.model<-dtree
```

Here the R model is converted to PMML directly in the R code. Alternatively, the R object can be passed to the model output port and then converted into PMML using the "R To PMML" node. Then all that remains to do is to write out a file using the "PMML Writer" node. Once the PMML file is written, the "PMML Reader" node can be used to import the model and connect it to the native KNIME "Decision Tree Predictor" node. In this way you can export, import, and convert R models to PMML, while at the same time using R models with the "R Predictor" node or any other Predictor node with the PMML model as input.

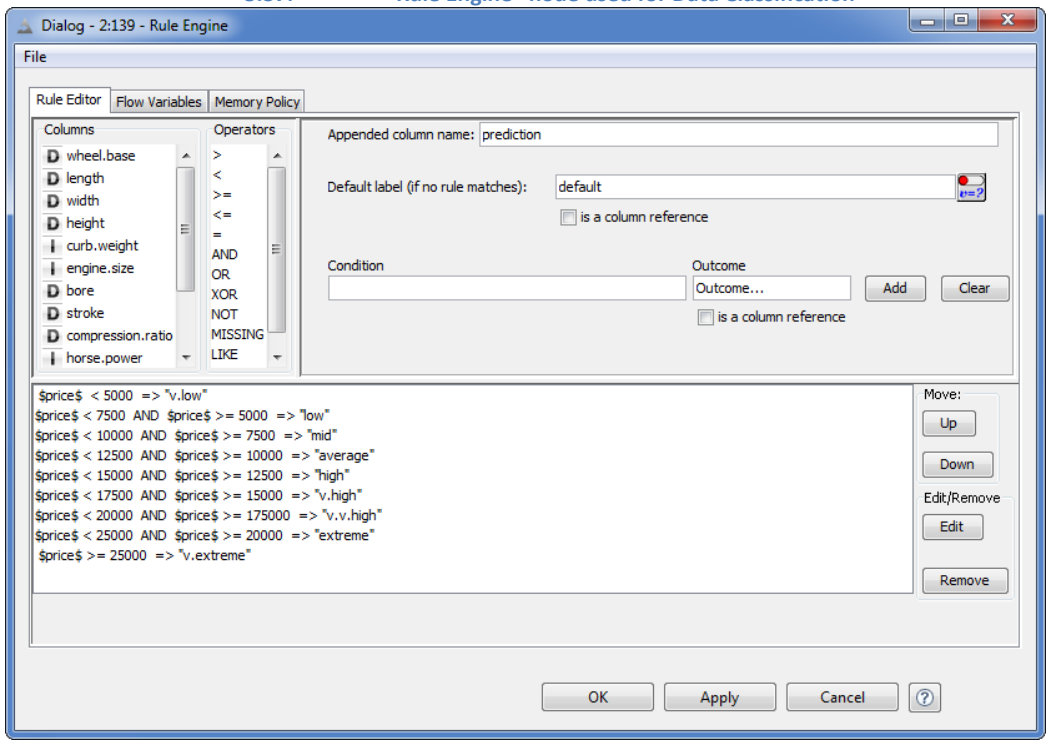
Note. PMML models written using the "PMML Writer" node may contain less information than the PMML exported explicitly using the R code.

The workflow for this example is shown in the figure below. This is very similar to the previous workflow used in the last section. Here though we are adding a factor (categorical) data column using the "Rule Engine" node.

5.36. Workflow "R PMML Node"



5.37. “Rule Engine” node used for Data Classification



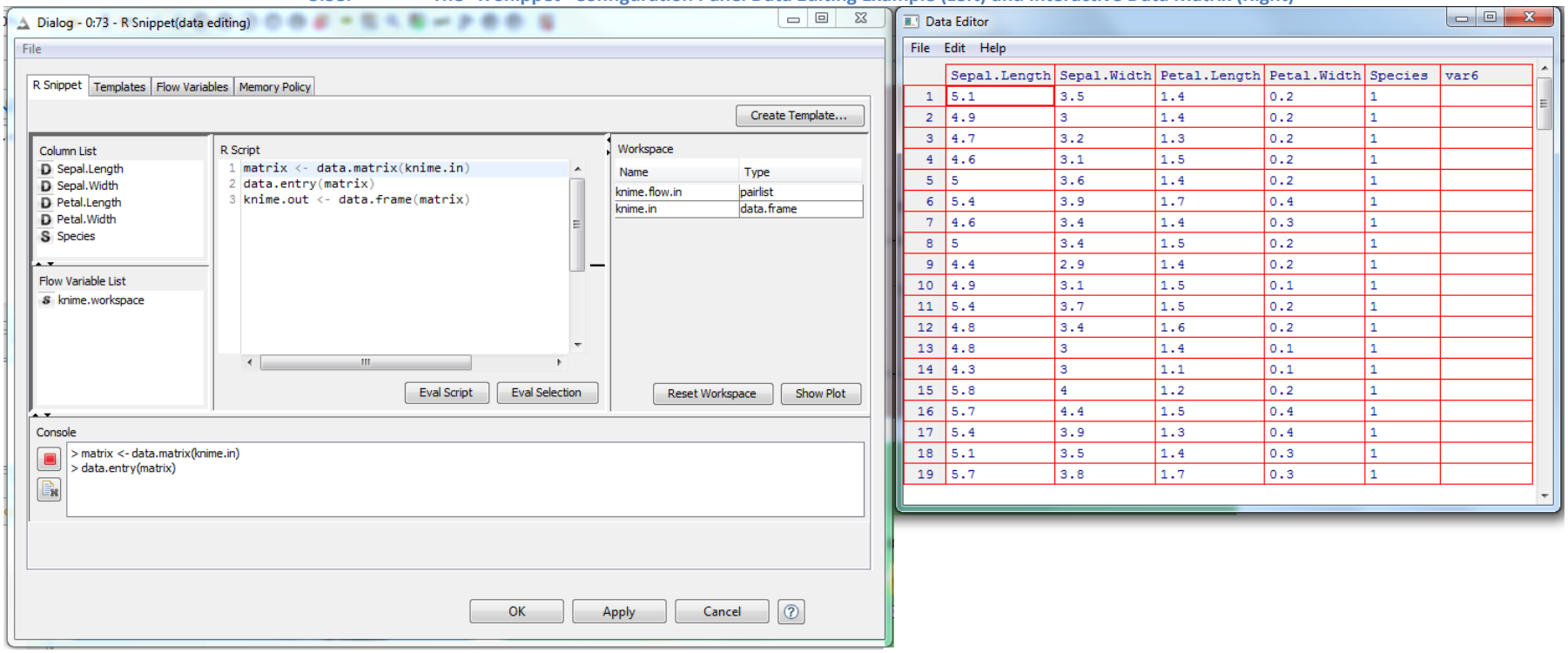
A few more nodes are available to convert data between KNIME and R, like “R to Table”, “R+Table to R”, “Table R-View”, and “Table to R”.

5.10. Interactive Data Editing using an R Script

The next example demonstrates how to do simple interactive data editing in KNIME using an R script.

The knime.in data frame is recast as a matrix on which the function `data.entry()` is called and finally the data produced by `data.entry()` is assigned to the output data frame. When this script is either executed or evaluated, KNIME creates a pop-up window for data editing. Editing the values in the pop up window and closing the window, changes the values of the R variable named “data”. To make sure of that, after changing the “data” values, closing the window, and finishing the script execution, you can double-click the R variable “data” in the “Workspace” panel and see the new values in the “Console”.

5.38. The "R Snippet" Configuration Panel Data Editing Example (Left) and Interactive Data Matrix (Right)



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	var6
1	5.1	3.5	1.4	0.2	1	
2	4.9	3	1.4	0.2	1	
3	4.7	3.2	1.3	0.2	1	
4	4.6	3.1	1.5	0.2	1	
5	5	3.6	1.4	0.2	1	
6	5.4	3.9	1.7	0.4	1	
7	4.6	3.4	1.4	0.3	1	
8	5	3.4	1.5	0.2	1	
9	4.4	2.9	1.4	0.2	1	
10	4.9	3.1	1.5	0.1	1	
11	5.4	3.7	1.5	0.2	1	
12	4.8	3.4	1.6	0.2	1	
13	4.8	3	1.4	0.1	1	
14	4.3	3	1.1	0.1	1	
15	5.8	4	1.2	0.2	1	
16	5.7	4.4	1.5	0.4	1	
17	5.4	3.9	1.3	0.4	1	
18	5.1	3.5	1.4	0.3	1	
19	5.7	3.8	1.7	0.3	1	

5.11. Exercises

Exercise 1

Read in the "cars-85.csv" data file and then produce a bar chart comparing the average city and highway fuel consumption for the 4 most expensive car constructors. Detail the average car prices in the graph.

Solution to Exercise 1

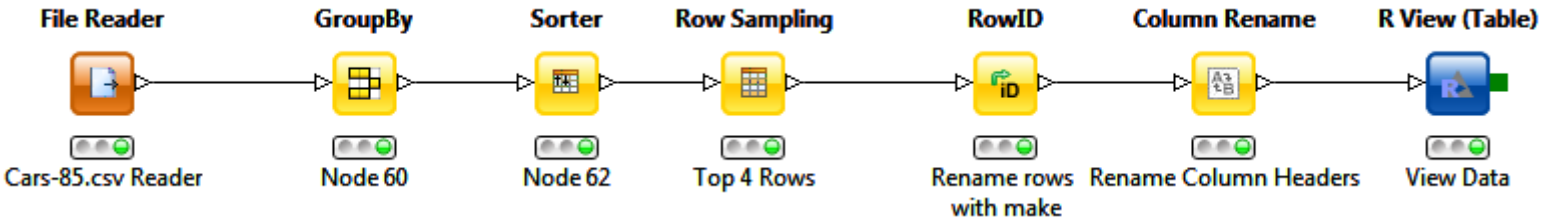
The information required can be obtained by a "GroupBy" node with group column = "make", aggregation columns = "city mpg", "highway mpg" and "price" and aggregation method = "Mean".

To extract just the top 4 most expensive car constructor, we first sort the price column using the “Sorter” node, and then use “Row Sampling” to select the absolute top 4.

In order to get the correct bar plot we re-label the row IDs with the car makes using the “RowID” node. In this way when we transpose the data table, the new column headers are the car constructors. This produces a bar plot with the different makes of cars along the x-axis showing a comparison of the mileage per gallon.

A “Column Rename” node is used to prepare column headers before being read into R to avoid any warning messages. An “R View” node is then placed at the end to build the bar plot in R and to visualize it in KNIME.

5.39. Exercise 1: Workflow

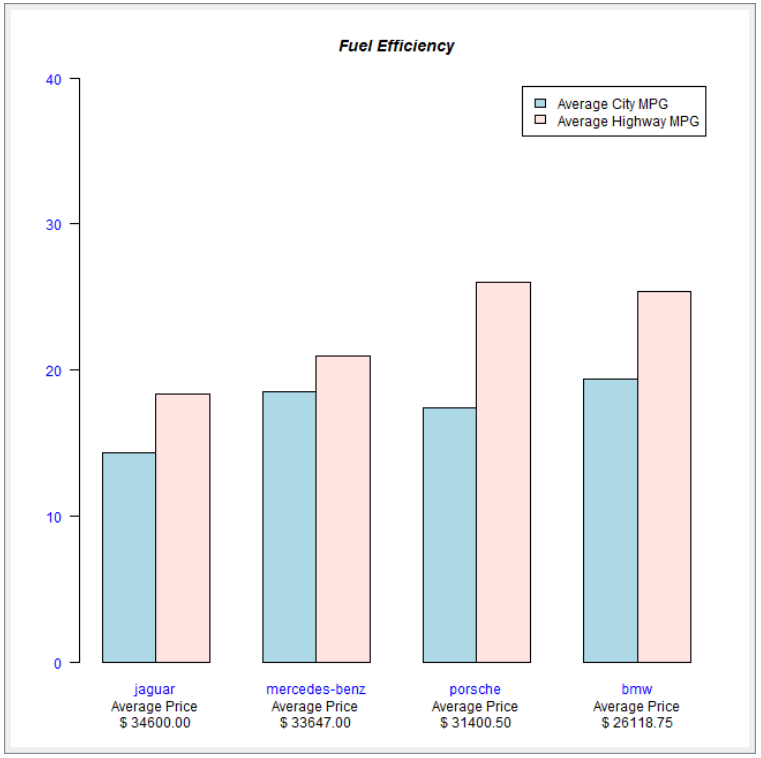


An example of how a bar plot could be produced is detailed in the R snippet below. To add additional textual information to the bar plot use the `mtext()` function.

```
# Set Graphical Parameters
par(las=1,col.axis="blue", sub)
prices<-t(format(R[3]))

# Set colouring Scheme
col <- colours()

# Bar Plot
R <- t(knime.in[1:2])
graph <- barplot(R, beside = TRUE, col = c("lightblue", "mistyrose"),
        legend =  c("Average City MPG","Average Highway MPG"), ylim= c(0,40),
```

```
        main = "Fuel Efficiency", font.main = 4)

# Labels
mtext(side = 1, at = colMeans(graph), line = 2, text = paste("Average Price"), col = "black")
mtext(side = 1, at = colMeans(graph), line = 3, text = paste("$", prices), col = "black")
```

5.40. Exercise 1: Bar Plot



Exercise 2

Read in the “slump_test.data” data file from the Download Zone directory and generate a linear regression model for the output variable, “28-day Compressive Strength (Mpa)”. Then run some diagnostics on the model and identify any outliers.

Solution to Exercise 2

The data is read using the "File Reader" node and column headers are amended to processing with R using the "Rename" node. The accompanying file "slump_test.names" details the data attributes and output variables. First, we define which columns we wish to include in the model by writing: `R<-knime.in[c(1:7,10)]`.

We are interested in making a linear regression model of the attributes to predict the output variable "28-day Compressive Strength (Mpa)", the ideal function to use is the `lm()` function. We perform the linear regression with the response variable given by "28-day Compressive Strength (Mpa)" and the predictor variables defined on the right of the tilde character. The model object is named "`model`".

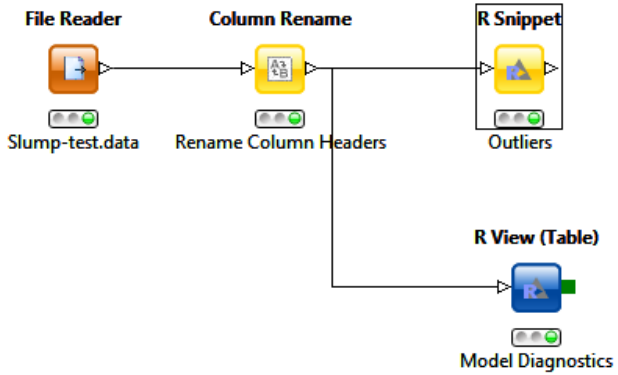
In order to produce the diagnostic plots of the model object use the `plot()` function. Four diagnostic plots are produced. To see them all we have to set the number of rows and columns to hold our plots using the `par()` function. The R snippet is detailed below:

```
usr <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
R<-knime.in[c(1:7,10)]
model<-lm(CompressiveStrength.28.day.Mpa ~ .-CompressiveStrength.28.day.Mpa,data=R)
plot(model)
par(usr)
```

From the graphs it would appear that observation 49 is an outlier. To confirm this suspicion there is an `outlier.test()` function available to identify the most outlying observation from the model. This function returns the Bonferroni *p*-values for Studentised residuals for both linear models using the t-test and generalized linear models based on a normal distribution test. This function is part of the car package that has to be called with `library(car)`. Using the `outlier.test()` function we find that indeed there is an outlier and it is observation 49.

```
library(car)
R<-knime.in[c(1:7,10)]
model<-lm(CompressiveStrength.28.day.Mpa ~ .-CompressiveStrength.28.day.Mpa,data=R)
edit(outlier.test(model))
edit(summary(model))
```

5.41. Exercise 2 : Workflow



Exercise 3

The "slump_test.data" data set is composed of 103 data points of which 25 are new data points collated several years after the initial data set was formed. Evaluate the latest 25 data points in terms of the "Flow" output variable using an R Decision Tree model to ascertain whether this data can be predicted with the historical 78 data points. Use sample quartile binning of the Flow column to create categories for the Decision Tree model.

Solution to Exercise 3

The same data is read and the columns renamed as per Exercise 2.

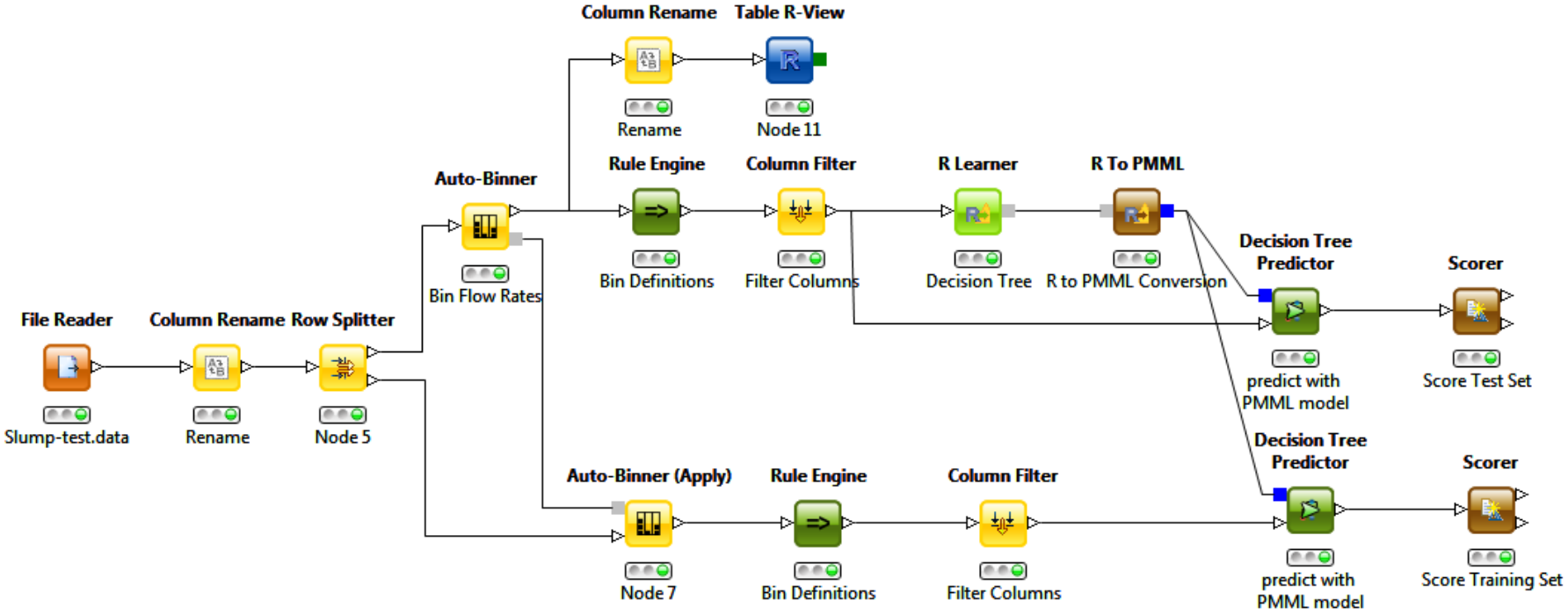
Using the "Row Splitter" node the training and test sets can be generated. As required, we take the top 78 data points as our training set. Using the "Auto-Binner" we sample the Flow column to create four bins.

A good way to visualize the bins is to plot them using the "R View" node with:

```
plot(knime.in$"FLOW.cm",knime.in$"FLOW.cm.Binned").
```

We can rename the bin categories so that they are more meaningful. Here we use the "Rule Engine" to classify our bins according to their distributions.

5.42. Exercise 3: Workflow



5.43. Exercise 3: Rule Engine Parameters

```
$FLOW.cm [Binned]$ = "Bin 1"  => "Very Slow"
$FLOW.cm [Binned]$= "Bin 2"  => "Slow"
$FLOW.cm [Binned]$ = "Bin 3"  => "Medium"
$FLOW.cm [Binned]$ = "Bin 4"  => "Fast"
```

The new binning class is appended as the “Flow.Speed” column to the original data table. The R Decision Tree model is created similarly to that described earlier in Section 5.9. (“R to PMML Node”). The “R Learner” node uses the class column, in this case the “Flow.Speed” column, to train the model.

```
# load the pmml export library
library(pmml);
```

```
# load the library for learning a tree model
library(rpart);

data = knime.in # use knime.in here and fix class column
model <- rpart(data$"Flow.Speed" ~ ., data[1:(ncol(data) - 1)])

# export to PMML
r_pmml <- pmml(model)

# write the PMML model to an export file
#write(toString(r_pmml), file="C:/R.pmml")

# provide the native R model at the outport
knime.model<-model
```

The generated model is then converted into a PMML format. We can subsequently use this model to predict both the original data set and the more recent data set of 25 points.

Using the “Scorer” node we can examine how well the training set of 78 original data points predicts the new test set.

Chapter 6. Web and REST Services

6.1. Web Services and WSDL files

We are all familiar with the World Wide Web, the Web: a huge data pool whose resources can be accessed through Universal Resource Locators (URLs). Whilst manually browsing the web is one means to crawl through web data, application to application communication is becoming more widespread. These methods of communication between applications over a network are referred to as Web Services. Web Services have been standardized as a means of interoperating between different operating systems running different software programs in different programming languages.

The software system to support interoperable machine-to-machine interactions is based on message exchange between applications. In order to comply with the message structure, web service software programs use an interface described in a language called a Web Service Description Language, or WSDL. WSDL is an Extensible Markup Language (XML) format for describing network services as a set of endpoints, operating on messages containing either document- or procedure-styled data. The mechanics involved in message exchange are then documented in a Web Service Description (WSD): a machine-readable specification of the web service's interface. A WSD represents the means of interacting with a web service by defining the message formats and the data types to be used between the client and a web service software program [5].

Nowadays a number of web services are available, such as financial estimations, weather reports, chemistry related data, and many other types of web services. This means that in many data analysis fields, tools are already directly available in the form of web services. Often it is enough to connect to such web services and send the appropriate message request with the appropriate input data to get back the required information. Following the knowledge recycling principle, meaning that it is better to use a tool available on the web rather than to re-implement it ourselves, we would like to be able to connect to and run a web service inside any of our workflows.

One of the most useful KNIME extension packages, the “KNIME Web Service Client”, covers this need for connecting to and running external web services. The “KNIME Web Service Client” package contains only one node: the “Generic Web Service Client” node. In this chapter we show how to access data from external web services using the “Generic Web Service Client” node.

First of all, you need to download and install the “KNIME Web Service Client” extension package.

As to install any KNIME Extension Package, go to the Top Menu:

- Select “Help”

- Select “Install New Software”
- From the “Available Software” window, in the box called “Work with:”:
    - Select the appropriate KNIME update site, like http://www.knime.org/update/2.x for the KNIME 2.x.y versions
    - Select the package called “KNIME Web Service Client”
    - Click “Next”
    - Follow the installation instructions

After the package has been successfully installed, you should find a node named “Generic Webservice Client” in the “Misc” category in the “Node Repository” panel.

In order to show the potentialities of this “Generic Webservice Client” node, we created an empty workflow in the “Chapter 6” workflow group and we named it “WebServiceNodes”.

6.2. How to connect to and run an external Web Service from inside a Workflow

To demonstrate the usage of web services in a KNIME workflow, we decided to use a weather forecast web service as an example. There are a number of free web services available on the web, which return information about the current and forecasted weather. The “WebServiceNodes” example workflow connects to the free web service “CDYNE_Weather” (http://wiki.cdyne.com/index.php/CDYNE_Weather). This SOAP (Simple Object Access Protocol) Web Service provides up to date weather information in the United States and is sourced from the National Oceanic and Atmospheric Administration's (NOAA) National Weather service.

The “CDYNE_Weather” web service makes three weather report operations available. The “WebServiceNodes” workflow concentrates only on two of these three operations: “getCityWeatherByZIP” and “getCityForecastByZIP”. Both of them require the zip code of the city you are interested in as the only input parameter. The zip code has to be supplied to the web service operations as a String. As a response the two web service operations generate a number of details about the local current and forecasted weather respectively for the supplied zip code. The URL for the web service WDSL location is http://wsf.cdyne.com/WeatherWS/Weather.asmx?wsdl. The web service has two available ports, either of which could be used. Our “WebServiceNodes” workflow connects to the “Weathersoap12” port.

Generic Webservice Client: WebService Description

The "Generic Webservice Client" node invokes a generic document-style service through an open source service framework to parse the WSD, create the requests, and read the responses of the web service.

The "Generic Webservice Client" node is located in the "Misc" category.

Its configuration window has two tabs: "Webservice Description" and "Advanced".

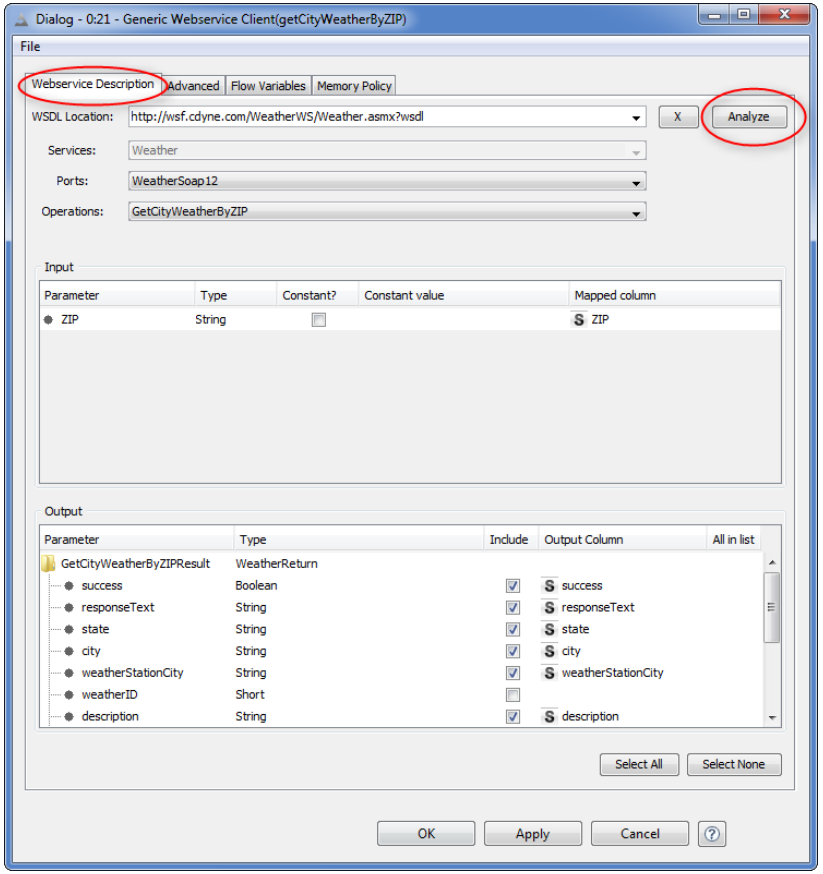
The **"Webservice Description"** tab collects all basic information to connect to and run the web service.

- First of all, it needs the URL WSDL location
- The "Analyze" button on the right fetches and parses the wsdl file and automatically fills the web service details into the remaining fields of the "Webservice Description" tab.
- The "X" button resets all fields of the "Webservice Description" tab.

Once the "Analyze" button has run, the node needs to select:

- The web service to use among all web services available (often there is only one service available in the WSDL file, in which case the box is grayed out indicating that it is disabled).
- The port to use among all ports available.
- The operation to run among all operations available.

6.1. Configuration window of the "Generic Webservice Client" node: Tab "Webservice Description"



The "**Input**" panel contains the required input parameter and its type. You can associate either a constant value or a data column of the input data table with the input parameter. If you use a constant value, the "Constant?" checkbox needs to be enabled and the value provided in the "Constant value" field.

The bottom panel, titled "**Output**", shows the output tree of the selected operation from the web service. Here you need to select the output parameters and the possible new column names to include in the output data table.

The “Constant?” field contains a check box, which when ticked indicates that a constant value is provided for this input parameter. The corresponding constant value must then be entered in the text box of the “Constant value” field. If no constant value is required, the value for the input parameter comes from the “Mapped column” of the input data table. Assigning an input data column to the “Mapped column” field feeds the web service input parameter with the values from the selected column.

Note. The type of the selected data column in the “Input” pane must correspond to the data type required for the web service parameter.

In the “Output” pane, the “Parameter” field details the available output values and the “Type” field shows the output data type. For every desired output parameter tick the check box in the “Include” column. The default name of the output column can be changed just by clicking it and typing a new name. For each selected output parameter a new column is appended to the output table. The “All in List” column shows check boxes only if the output parameter of the web service represents a collection. Checking this box returns all the list elements.

The “WebServiceNodes” workflow has been designed to read in a data table of zip codes as input for the web service, run the web service query using the “Generic Webservice Client” node (sending multiple row queries in parallel), and then display the results in a table. First of all, we created a list of 170 zip codes in the US (the “ZIP_CODES.csv” file from the Download Zone). The file was read in using a “File Reader” node named “ZIP Codes”. The zip codes were automatically read in as Integer. We then converted the zip codes from Integer into String to make it compatible with the type of the corresponding WSDL input parameter of the web service.

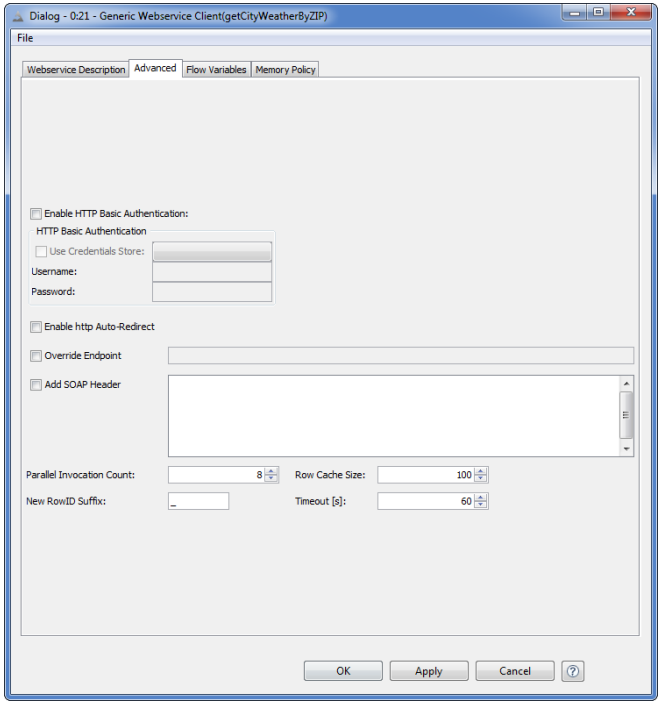
At this point the workflow had the right data to run an operation from the “CDYNE_Weather” web service. We then introduced two “Generic Webservice Client” nodes: one to retrieve the current weather for the given zip code (operation “GetCityWeatherByZIP”) and the other to retrieve the forecast for the next day(s) for the same zip code (operation “GetCityForecastByZIP”). We set the configuration window of the “Generic Webservice Client” node for the operation “GetCityWeatherBYZIP” as shown in figure 6.1. That is:

- URL: http://wsf.cdyne.com/WeatherWS/Weather.asmx?wsdl
- Port: WeatherSoap12
- Operation: GetCityWeatherByZIP
- Input parameter “ZIP” mapped to the input data column named “ZIP”
- Keeping all output parameters besides “weatherID”

We did not configure any settings in the “Advanced” tab. Indeed, often the call to the web service does not require any further information than what is entered in the “Webservice Description” tab. However, there may be instances where more advanced features need setting in the “Advanced” tab.

Generic Webservice Client: Advanced Tab

6.2. Configuration window of the "Generic Webservice Client" node: Tab "Advanced"



The "**Advanced**" tab contains a number of options to enable more of the web service features.

If the service requires basic http authentication, enable the "Enable HTTP Basic Authentication" flag and enter the login credentials. This can be done either by selecting credentials globally defined on the workflow or by entering user name and password in the corresponding fields of this "Advanced" tab.

The "Enable http Auto-Redirect" checkbox enables the underlying http connection to automatically follow any redirects given by the server.

The "Override Endpoint", if checked, overrides the web service endpoint given in the wsdl file using the value entered in the text field.

There is also a checkbox to allow for valid XML to be inserted into the SOAP header of an outgoing SOAP request. The checkbox is appropriately called "Add SOAP Header".

The "Parallel Invocation Count" defines the maximum number of parallel asynchronous invocations (one invocation is equivalent to processing a single input row).
A high number of parallel asynchronous invocations increases the parallel job count at the service endpoint; this decreases the overall execution time but could lead to an overload situation or a lock-out of the client. A low value increases the overall execution time. The default setting is 8.

The "Row Cache Size" selects the number of rows to be kept in memory at most. The default is 100 rows. If, for any reason, the processing of one row takes significantly longer than the processing time of other rows, the web service invocation will stall, keeping at most this number of rows in memory.

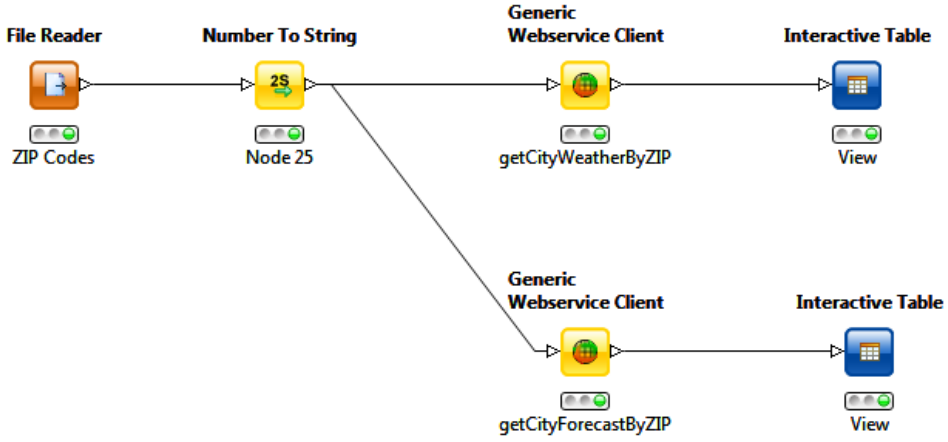
If the web service output generates list of lists, then a new Row ID is created. These new IDs will have the same base name as the original Row ID followed by a suffix (as entered into the corresponding text field), the default being '_', and then an increasing counter number.

The "Timeout [s]" option allows to change the time (in seconds) the client waits for an answer from the web service server. If you have a slow connection to the server and get a lot of time out errors (without the server being actually unresponsive) you should increase this number.

Note. The “Generic Webservice Client” node does not yet support optional input parameters, authentication, and RESTful services. Nodes to connect to REST services are available from the trusted community extensions http://tech.knime.org/book/krest-rest-nodes-for-knime-trusted-extension

The final “WebServiceNodes” workflow is shown in figure 6.3. During execution of the “Generic Webservice Client” node, data is sent to the WSDL URL and the response data is returned. The results are then passed to an “Interactive Table” node for easy visualization.

6.3. Workflow “WebServiceNodes”: checking the weather by ZIP code using “Generic Webservice Client” nodes



6.3. REST Services Nodes (KREST Extension)

A community Extension of KNIME, named KREST, provides REST services utility nodes (http://tech.knime.org/book/krest-rest-nodes-for-knime-trusted-extension). This extension can be installed via "File" -> "Install KNIME Extensions...", then "KNIME Community Contributions - Other" -> "REST Client". The extension contains two types of nodes: *submitters* and *helpers*. Submitters communicate with the RESTful service, while helpers convert REST resources representations into KNIME data tables and vice versa.

To give an example of how a REST service can be accessed and information retrieved with the KREST nodes, we refer to the workflow named “RESTServiceExample” in the Download Zone. The workflow reads one point geographical coordinates from a file, retrieves elevation from the Google API REST service, and adds it to the original data.

After reading the latitude and longitude coordinates, the REST request is built using a “String Manipulation” node as:

```
join("http://maps.googleapis.com/maps/api/elevation/json\?locations=",string($lat$),",",string($long$),"\&sensor=false")
```

where lat and $long$ are the point latitude and longitude.

Then, the "GET Resource" node from the KREST extension submits the request to the REST server. The "GET Resources" node has two tabs in its configuration window:

- "Base Settings" contains all REST service settings, including the URL source (in our case the data column generated by the "String Manipulation" node);
- "Header Settings" allows for customization of the REST request header.

GET Resource

The "**Base Settings**" tab contains a number of settings for the REST service.

First of all the REST service URL, the request. This can be set manually or automatically from the value of an input data column.

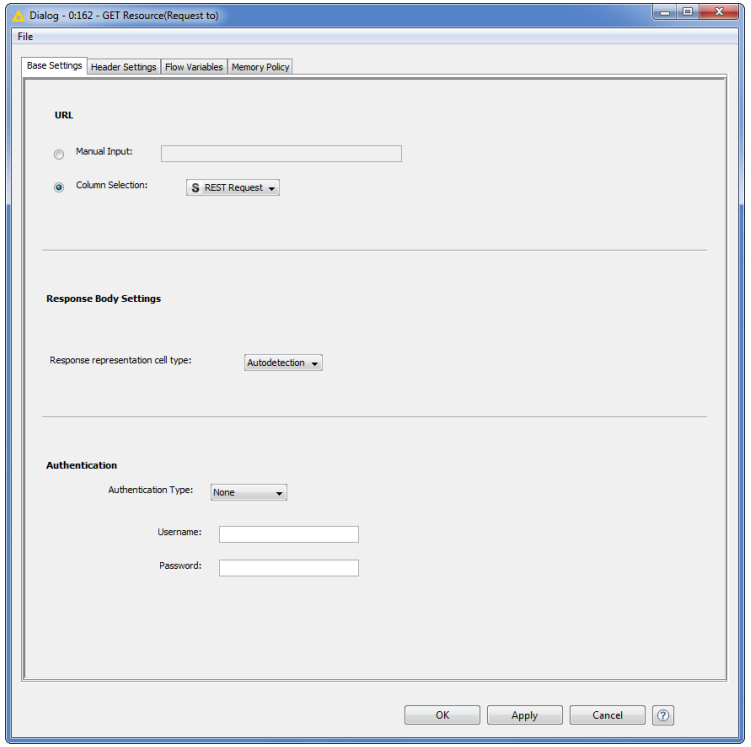
Then we need the response format: XML or String. One of the possible options is the format auto-detection, which works well in most cases.

And finally we need the authentication mode and the credentials, if credentials are required to access the service.

The "**Header Settings**" tab allows the definition of a number of side settings for the REST service, such as encoding, language, cookies, etc …

The "GET Resource" node has two ports: one for the response header and one for the response body.

6.4. "GET Resource" node configuration: "Base Settings" tab



After getting the response from the REST service, we need to interpret it and extract the elevation value. The node to interpret REST responses is the "Read REST Representation" node.

Read REST Representation

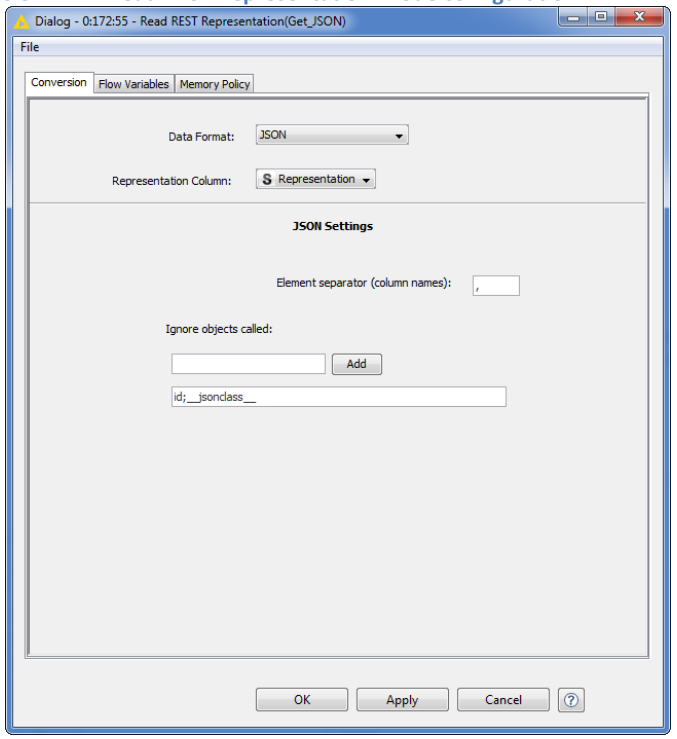
The "Read REST Representation" node converts a REST response into a KNIME data table.

In order to do that, it needs to know:

- the input data column with the REST response;
- the REST representation format (JSON, XML, ATOM, or CSV). A format auto-detect option is available here as well. If you select the *Autodetect* option, the data format will be parsed from the header table (optional top input port).
- A number of format specific options

The result at the output port is a collection of String Cells, containing the REST response, and a “Status” data column, containing the REST response status.

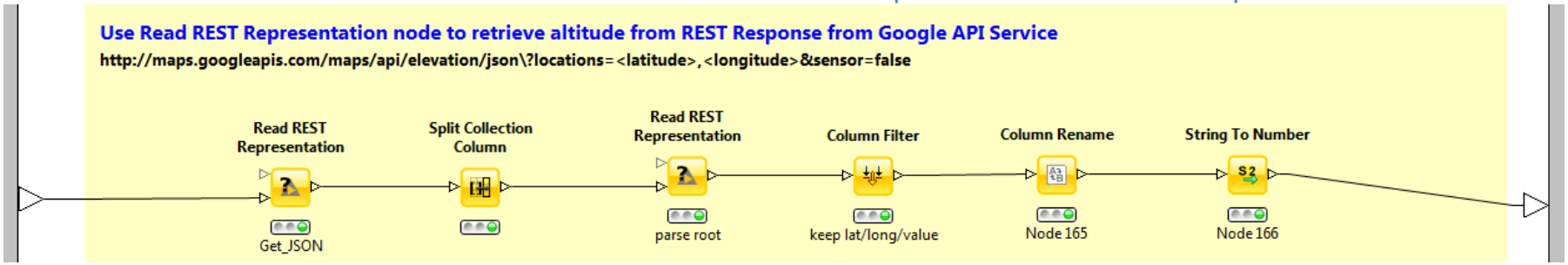
6.5. “Read REST Representation” node configuration



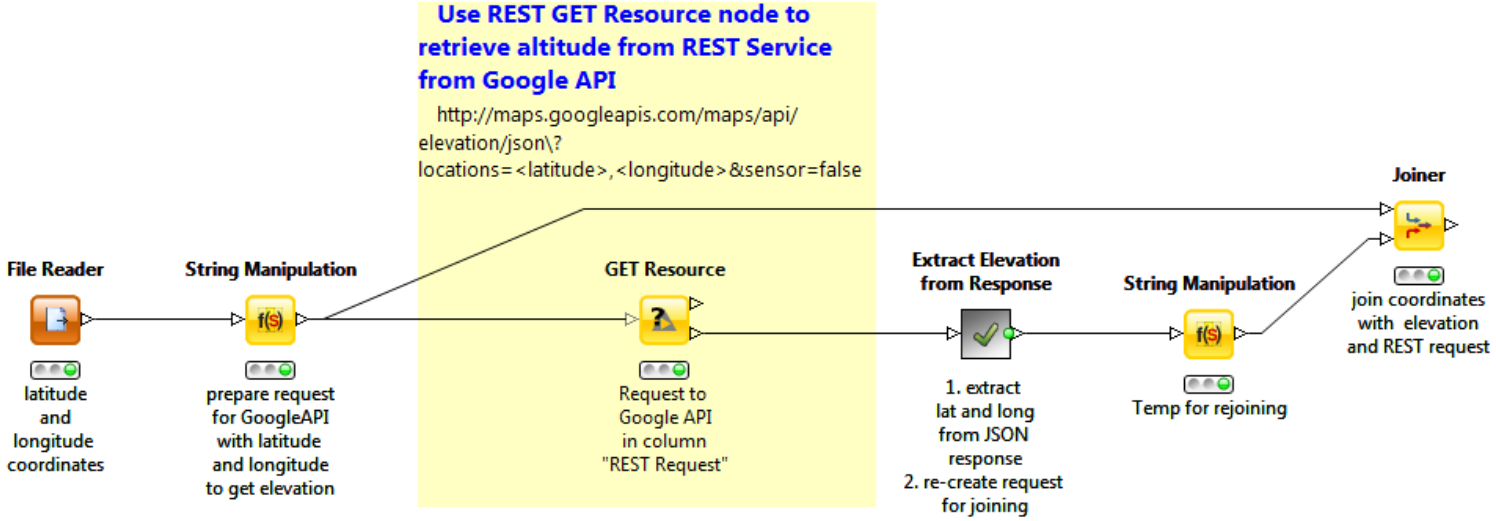
In our example workflow, a first "Read REST Representation" node separates the array of REST results from the response status. Then a second "Read REST Representation" node separates all results in the array: latitude, longitude, elevation, and resolution, all as String. A final "String To Number" node transforms the result Strings into numbers. The sequence of nodes for the REST response interpretation is in the metanode named “Extract Elevation from Response” and shown in figure 6.6.

The final workflow is shown in figure 6.7.

6.6. Content of Metanode "Extract Elevation from Response" in Workflow "RESTServiceExample"



6.7. Content of Metanode "Extract Elevation from Response" in Workflow "RESTServiceExample"



6.4. Exercises

Exercise 1

In this exercise we are going to query a chemical database, specifically "ChEBI" (the Chemical Entities of Biological Interest) using the ChEBI Web Services. This resource is a freely available dictionary of small chemical compounds. ChEBI provides SOAP access to its database. The web service is part of a

suite of chemical related web services and is made available by the European Bioinformatics Institute (EBI). The ChEBI WSDL uses the document-style binding; so it is suitable for the "Generic Webservices node".

We have provided a list of compound ID numbers in the file "ChEBI_original.csv" available in the Download Zone. Using this data, and given the WDSL location as: http://www.ebi.ac.uk/webservices/chebi/2.0/webservice?wsdl, use the "getLiteEntity" operation to retrieve all the response data in the output list.

Solution to Exercise 1

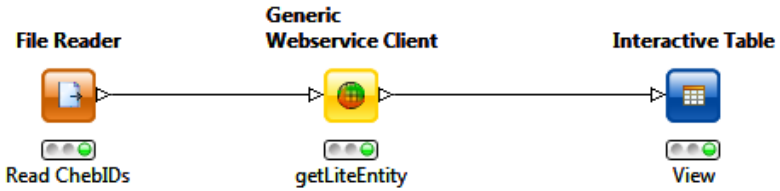
The final workflow is shown in figure 6.4.

Click the "Analyze" button in the node configuration dialog of the "Generic Webservice Client", in order to populate the fields. Select the "getLiteEntity" operation. Once this is done, a number of default input parameters become available. Further information on these parameters can be obtained from the ChEBI web service: http://www.ebi.ac.uk/chebi/webServices.do#SOAP%20Clients.

The most important input parameter is the one named "search" of type String, which we mapped to the "chebiID" column of the input data table. The "maximumResults" parameter requires an integer and represents the number of results returned to a maximum of 5000. Keeping the other parameters as default and selecting "All in List" to return all the elements in the output tree, we are ready to execute the node.

The final result should be a table containing ChEBI ID numbers, compound names and some other database fields. Note that some compounds are no longer available and will not be in the final table.

6.8. Exercise 1: The workflow



Chapter 7. Loops

7.1. What is a Loop

In the “Node Repository” there is a full category called “Flow Control” for logical and control operations on the data flow. The “Flow Control” category contains three sub-categories:

- “Loop Support” contains those nodes that allow the implementation of recursive operations; that is of loops;
- “Switches” contains the nodes that can switch the data processing flow one way or another;
- “Variables” contains all those nodes that create, delete, and update the workflow variables (Chapter 4).

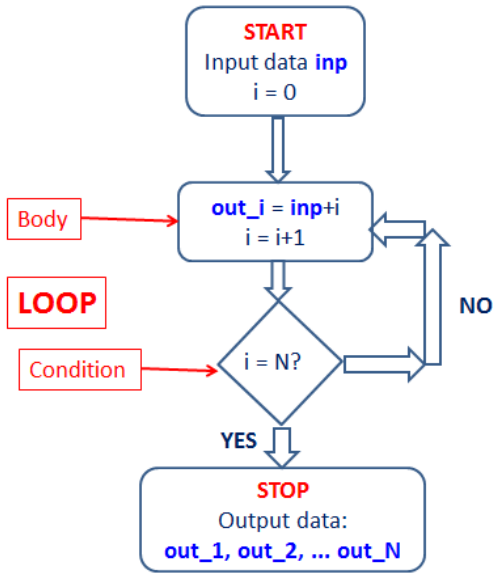
In this section we will work with the “Loops” sub-category.

A loop is the recursive execution of one or more operations. That is, a loop keeps re-executing the same group of operations (the body of the loop) till some condition is met that ends the loop.

A loop starts with some input data; processes the input data a number of times; each time assigns the result to an output variable and piles it up into a data table; and finally stops when the maximum number of times is reached or some other condition is met. A loop then has a start point, an end point to verify that the stopping condition is met, and a body where the data processing happens. Each execution time of the loop body is called a loop iteration.

An example, depicted in figure 7.1, consists of adding one unit to an integer variable N times. The integer input variable is named “inp”; the iteration counter is named “i”; at the end of each iteration i, the processed data are stored in an output variable named “out_i” where i=1,2, …N; all iteration processed data are collected together till the loop stops. The output data of the loop consists then of a number of integers “out_1”, “out_2”, “out_3”, ... till “out_N”. In practice, if inp = 5 and N = 10, the output data will be a data table with the following values: 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14.

7.1. Loop Example

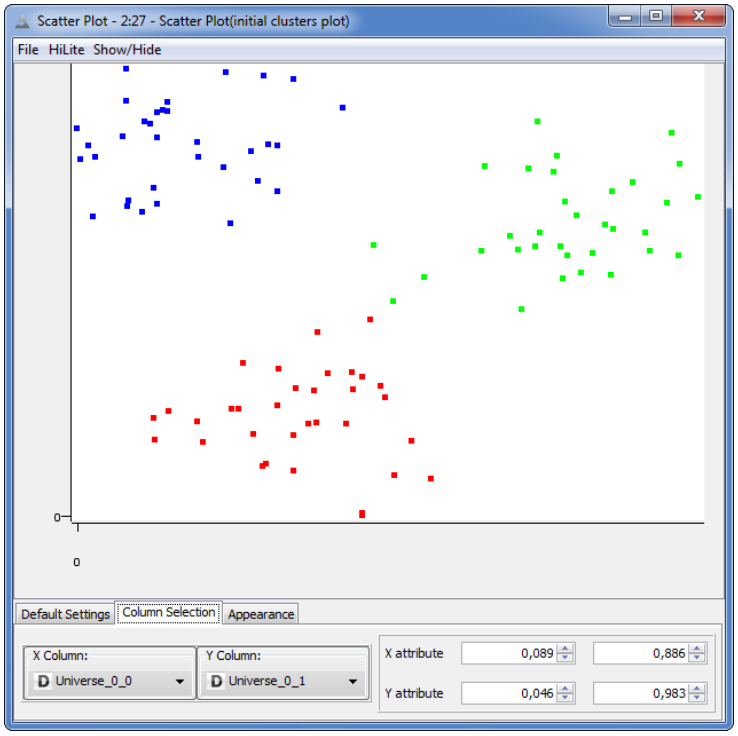


In KNIME we have a number of nodes that can be used to implement loops: either loops on a pre-defined number of iterations or loops that stop when some condition is met. Loops in KNIME are always built with at least two nodes:

- The node to start the loop
- The node to end the loop
- A few additional but not necessary nodes to build the body of the loop

Loops in KNIME are the equivalent to “for” and “do-while” cycles in most programming languages.

7.2. Data set with 3 clusters across two variables



Note. Not all loops are in the “Flow Control/Loop Support” category. You will find a few specialized loops here and there in other categories, like “Ensemble Learning”, “Meta”, “KNIME Labs”->”Optimization”, and more.

Note. Most loops in KNIME cannot alter the input data for the next iteration. The only loop, at the moment, that can do that is the "Delegating" loop in the "Ensemble Learning"->"Utility Nodes" category.

7.2. Loop with a pre-defined number of iterations (the "for" loop)

The easiest loop to implement is the loop with a pre-defined number of iterations. That is, the loop where a group of operations is repeated a fixed number of times.

In this section we build an example workflow on a data set with 3 clusters distributed across two variables as shown in figure 7.2.

Let's now add a copy of these three clusters shifted to the right on the x-axis of one unit. Let's repeat this operation 4 times to obtain 4 copies of the original three clusters, each copy shifted slightly more to the right than the previous copy.

In order to build the required 4 copies of the original data set, we introduce a loop with 4 iterations. Each iteration:

- takes the original data set,
- moves it one unit to the right,
- concatenates the data processed during this iteration with the data processed so far.

In a new workflow group named "Chapter 7", we created a new empty workflow called "Counting Loop 1".

First of all, we would need to read in the data set as described in figure 7.2. Since we did not have such a data set available, we created it by using the "Data Generator" node.

Data Generator

The “Data Generator” node generates artificial random data normalized in [0,1], grouped in equal size clusters, and with a fixed number of attributes.

Data points for each cluster are generated following a Gaussian distribution with a given standard deviation and adding a certain fraction of random noise to the data.

The “Data Generator” node is located in the `“IO” -> “Other”` category.

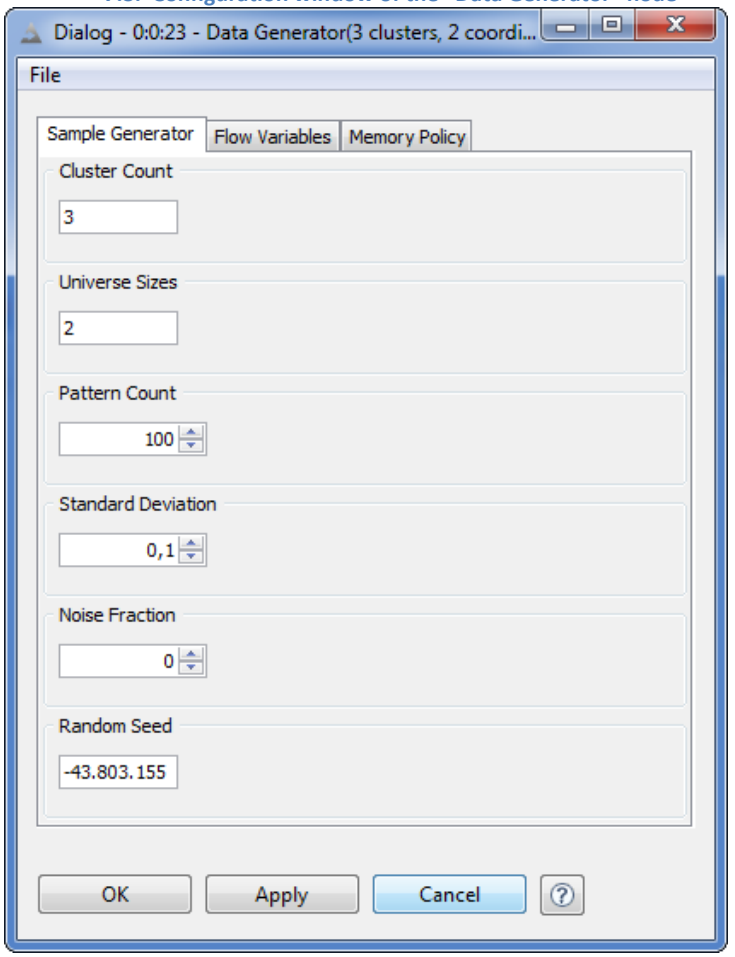
In order to perform its task, the “Data Generator” node needs to be configured with the following settings:

- The number of clusters to be created
- The number of space coordinates (i.e. Universe Sizes)
- The number of total data rows (i.e. patterns). An equal number of patterns (= number of data rows/number of clusters) is subsequently generated for each cluster.
- The standard deviation to apply to each cluster. Notice that it is the same standard deviation value for all clusters.
- The noise fraction to apply to the generated data
- A random seed to make this random generation of patterns reproducible

The “Data Generator” node offers two output ports:

- One port with the generated data rows, each with its own cluster ID
- One port with the cluster centers on the space coordinates

7.3. Configuration window of the “Data Generator” node

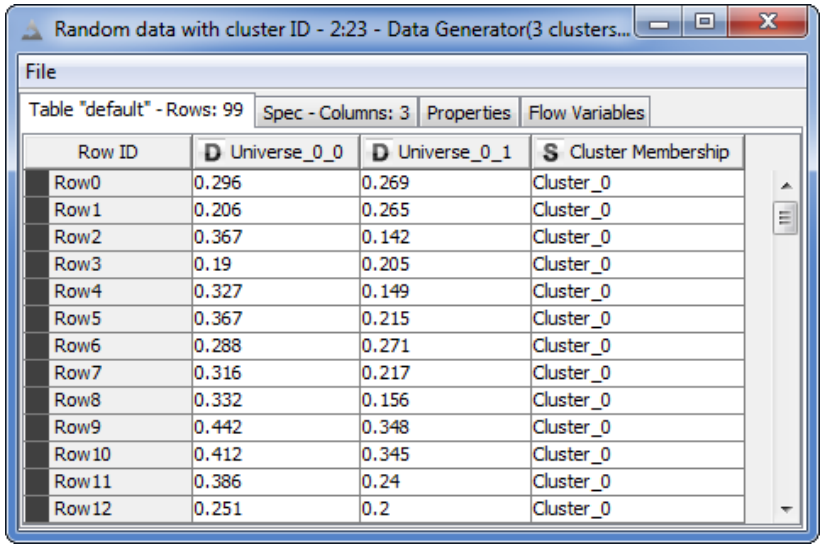


Note. Data can be generated in more universes, each universe with a different number of clusters and a different number of coordinates.

In case of more than one universe, the cluster counts and the universe sizes are inserted into the corresponding text boxes as a sequence of comma separated numbers. For example, 2 universes with 3 and 4 clusters and 2 and 3 coordinates respectively are described by “3,4” in “Cluster Count” and “2,3” in the “Universe Sizes” text boxes.

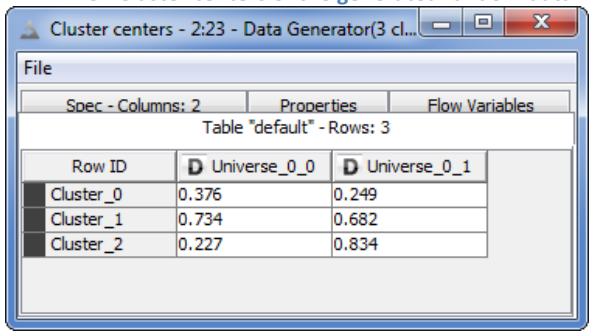
We then started the new workflow “Counting Loop 1” with a “Data Generator” node with 100 data rows equally distributed across 3 clusters, on a 2 attribute space, with a standard deviation 0.1 for each cluster, and no noise added. The node was named “3 clusters, 2 coordinates”. The generated random data and their cluster centers are reported in the figures below.

7.4. Generated random data



Row ID	Universe_0_0	Universe_0_1	Cluster Membership
Row0	0.296	0.269	Cluster_0
Row1	0.206	0.265	Cluster_0
Row2	0.367	0.142	Cluster_0
Row3	0.19	0.205	Cluster_0
Row4	0.327	0.149	Cluster_0
Row5	0.367	0.215	Cluster_0
Row6	0.288	0.271	Cluster_0
Row7	0.316	0.217	Cluster_0
Row8	0.332	0.156	Cluster_0
Row9	0.442	0.348	Cluster_0
Row10	0.412	0.345	Cluster_0
Row11	0.386	0.24	Cluster_0
Row12	0.251	0.2	Cluster_0

7.5. Cluster centers of the generated random data



Row ID	Universe_0_0	Universe_0_1
Cluster_0	0.376	0.249
Cluster_1	0.734	0.682
Cluster_2	0.227	0.834

In order to visualize the data and their clusters, we added a “Color Manager” node and a “Scatter Plot” node. The data visualization performed with the “Scatter Plot” node on a two-coordinate space, “Universe_0_0” and Universe_0_1”, is shown in figure 7.2.

At this point, we wanted to generate 4 copies of the original data set and shift each copy one more unit to the right. That is, the first data set should be identical to the original data set inside [0,1] x [0,1]; the second data set should be identical to the first data set but inside [0,1] x [1,2], and so on. We then needed a loop cycle to copy the data to the different regions 4 times. In particular, since we already knew how many times the copy had to be made, we decided to use a loop with a pre-defined number of iterations, i.e. 4.

In order to move this data set one unit to the right 4 times, we could implement a loop and add the iteration number to the data set x-values at each iteration. Thus:

- **iteration 0:** x = x+**0** -> the data set is the exact copy of the original data set
- **iteration 1:** x = x+**1** -> the data set is moved one unit to the right
- **iteration 2:** x = x+**2** -> the data set is moved two units to the right
- **iteration 3:** x = x+**3** -> the data set is moved three units to the right

A loop with a pre-defined number of iterations in KNIME starts with a “Counting Loop Start” node. In the “Flow Control” -> “Loop Support” category there are a number of nodes to end a loop. A few of them can be used to end a loop started by a “Counting Loop Start” node; that is a loop with a pre-defined number of iterations. The most generic loop ending node is the one named “Loop End”.

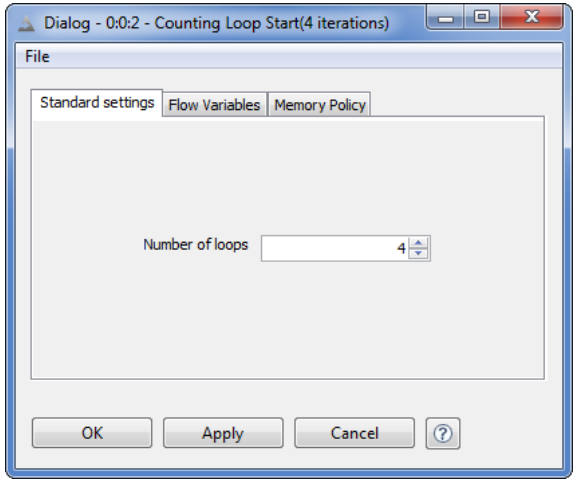
Counting Loop Start

The “Counting Loop Start” node starts a loop with a pre-defined number of iterations.

That is, it starts a cycle where the operations, between the “Counting Loop Start” node and the end loop node, are repeated a pre-defined number of times.

The only setting required for such a loop start node is this number of pre-defined iterations (“Number of loops”).

7.6. Configuration window of the “Counting Loop Start” node



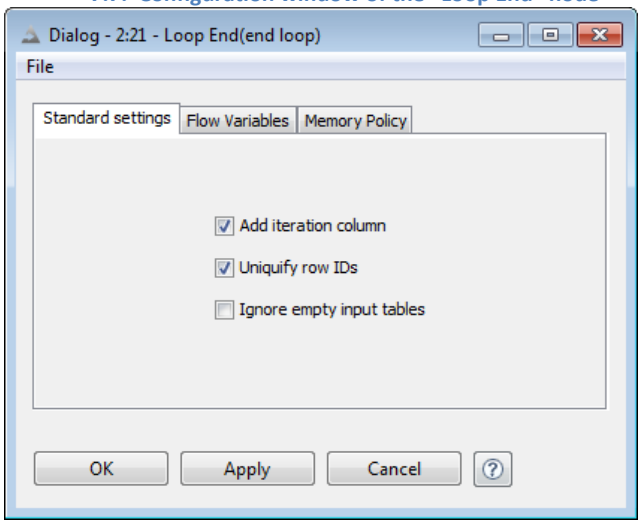
Loop End

The “Loop End” node closes a loop started by a loop start node and **concatenates** the resulting data tables from all loop iterations.

This is a generic node used to close a loop and collect results. Therefore it requires only minimal configuration:

- The flag specifying whether to append the iteration number at the end of each data table. The iteration number identifies the iteration during which a given data table has been created. Iteration numbers start with 0.
- The flag specifying whether to force the RowIDs of the resulting data table to be unique.
- The flag to ignore possible empty input tables

7.7. Configuration window of the “Loop End” node



Note. The “Loop End” node **concatenates** data tables. That is, it concatenates the results coming from the loop body one iteration after the other. However, there are other loop end nodes that append the resulting data tables as additional columns, one iteration after the other.

Till now, we have started and closed a loop with a pre-defined number of iterations but without any operations in between. At each iteration, the loop simply goes through all data rows of the original data set and collects them into the “Loop End” node, one row after the other. It repeats this operation 4 times and each time concatenates the resulting data rows to the already collected result data tables.

In addition, the “Counting Loop Start” node, while starting a loop, creates two new workflow variables: one contains the current iteration number, named “currentIteration”, and one contains the number of total iterations, named “maxIterations”.

We still need to introduce a loop body that moves each copy of the original data table one step further to the right on the x-axis. To do that, we can exploit the “currentIteration” flow variable created by the “Counting Loop Start” node and use a “Java Snippet (simple)” node with the following code line:

```
return $Universe_0_0$ + $${IcurrentIteration}$$;
```

where `$Universe_0_0$` is the attribute on the x-axis and `$${IcurrentIteration}$$` is the workflow variable containing the current iteration number.

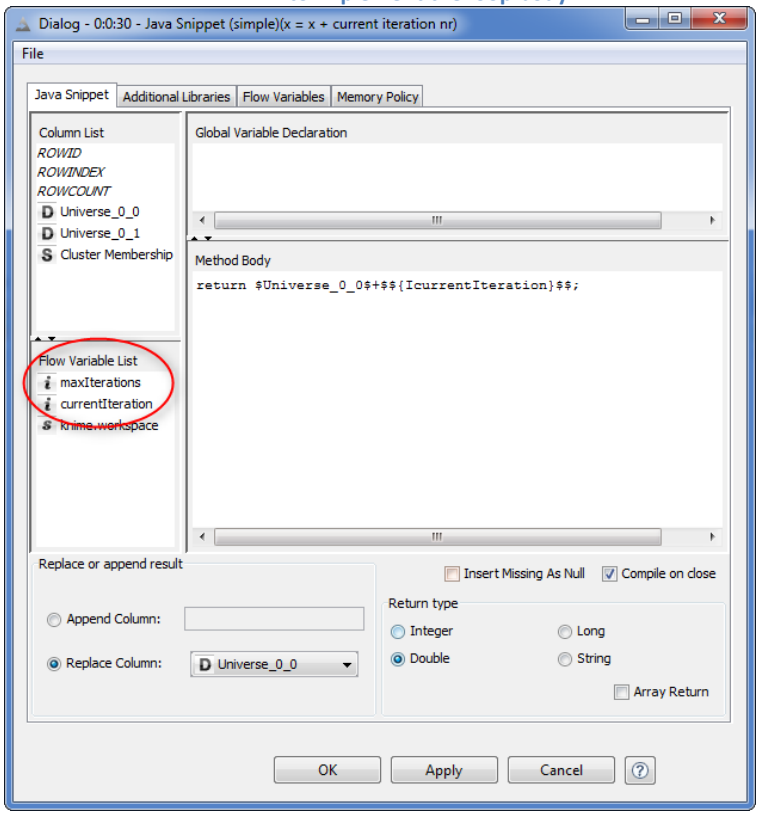
Figure 7.8 shows the configuration window of the “Java Snippet (simple)” node used to implement the loop body.

In the “Flow Variable List” panel on the bottom left, you can see the two new workflow variables introduced by the “Counting Loop Start” node to describe the loop execution at each iteration.

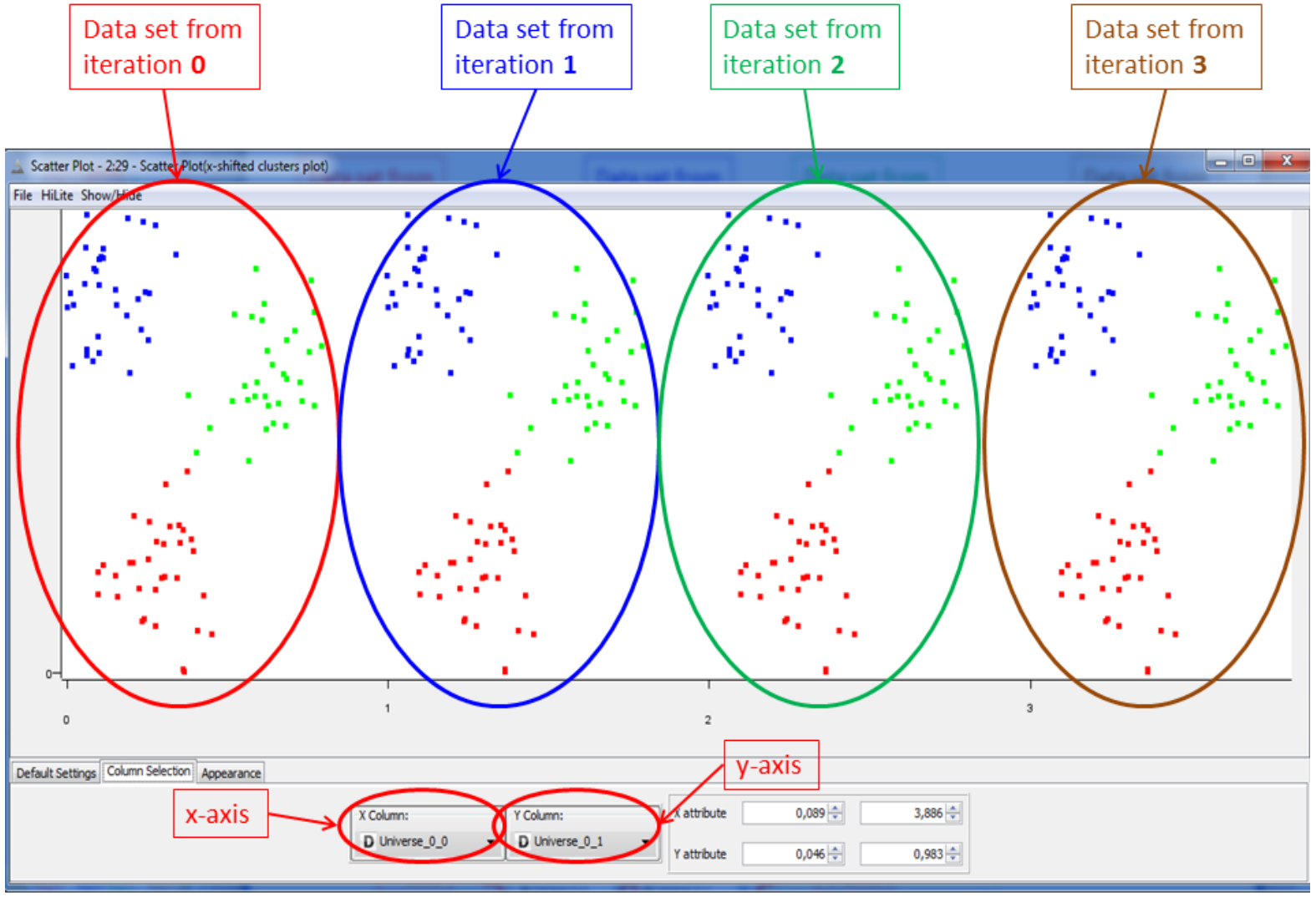
The program line used in the “Java Snippet (simple)” node moves each copy of the data to the right on the x-axis for as many units as its iteration number. The final data set should then consist of 4 identical vertical stripes of data equally spaced in [0, 4] on the x-axis.

We used a “Scatter Plot” node again to end the workflow and to verify the data placement in the final data table (Fig. 7.9).

7.8. Configuration window of the "Java Snippet (simple)" node used to implement the loop body

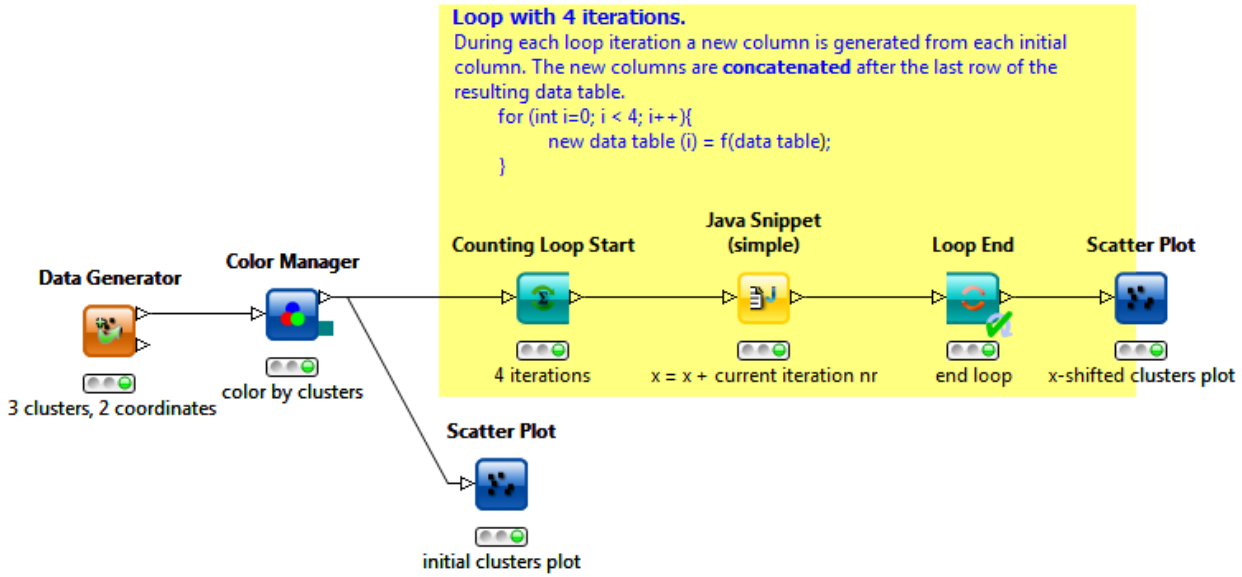


7.9. Visualization of the data set resulting from the loop implemented by the “Counting Loop 1” workflow



Finally, figure 7.10 shows the full “Counting Loop 1” workflow.

7.10. The "Counting Loop 1" workflow



7.3. Additional Commands for Loop Execution

The result in figure 7.9 can only be seen after the workflow execution, but loops have a number of different execution options.

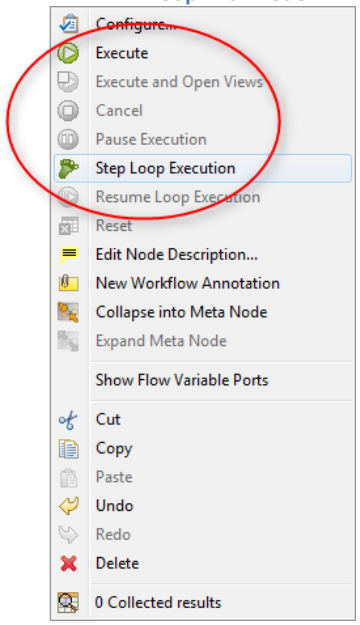
Running the “**Execute**” command from the context menu of the loop start node only reads the input data into the loop. In order to run the whole loop with all required iterations, you need to run the “Execute” command for the loop end node.

Similarly, running the “**Cancel**” command from the context menu of the loop end node stops the loop execution at the current iteration.

To reset the loop, it is enough to select the “**Reset**” option in the context menu of any node in the loop.

Not a lot is new in the execution, cancellation, and reset of the loop nodes with respect to other nodes, besides the fact that the execution and the execution cancellation of the loop should be performed on the end node, while the resetting of the whole loop can be run from any node in the loop.

7.11. Context menu of the "Loop End" node.



However, the loop end nodes have a number of additional execution options. Besides the traditional commands "Execute", "Reset", and "Cancel", in the context menu of the loop end nodes (Fig. 7.11) we can also find:

- The "**Pause Execution**" command, which pauses the loop execution at the end of the current iteration and allows investigation of the intermediate results of the nodes in the loop body.
- The "**Step Loop Execution**" command, which executes the next loop iteration and pauses at the end, allowing visualization of the intermediate results of the loop body nodes.
- The "**Resume Loop Execution**" command, which restarts the loop execution that was previously paused with the "Pause Execution" or the "Step Loop Execution" command.

These three additional commands, which are only available for loop end nodes, make loop execution very flexible and easy to debug.

Note. It might be necessary, in some cases, to execute the loop nodes manually for the first time to be able to configure them properly.

7.4. Appending Columns to the Output Data Table

In section 7.2 we have seen that the "Loop End" node concatenates the resulting data tables together after each loop iteration. A part of the output table of the "Loop End" node of the "Counting Loop 1" workflow is shown in figure 7.12. The last column of the data table, named "Iteration", is created by the "Loop End" node (if so specified in the configuration window), and shows the iteration number during which each data row was created. In the figure we can see that the data rows resulting from iteration 2 follow the data rows resulting from iteration 1. That shows that the loop results are concatenated at the end of each loop iteration.

However, sometimes we might like to append the data resulting from each iteration as new columns to the previous results. That is, at each iteration a new data table is generated and appended to the current output data table.

7.12. Output Table of the "Loop End" node in the "Counting Loop 1" workflow

Collected results - 2:21 - Loop End(end loop)

File

Table "default" - Rows: 396 | Spec - Columns: 4 | Properties | Flow Variables

Row ID	Univers...	Univers...	Cluster ...	Iteration
Row90#0	0.243	0.829	Cluster_2	0
Row91#0	0.314	0.81	Cluster_2	0
Row92#0	0.152	0.983	Cluster_2	0
Row93#0	0.192	0.701	Cluster_2	0
Row94#0	0.155	0.707	Cluster_2	0
Row95#0	0.154	0.697	Cluster_2	0
Row96#0	0.367	0.962	Cluster_2	0
Row97#0	0.184	0.869	Cluster_2	0
Row98#0	0.245	0.798	Cluster_2	0
Row0#1	1.296	0.269	Cluster_0	1
Row1#1	1.206	0.265	Cluster_0	1
Row2#1	1.367	0.142	Cluster_0	1
Row3#1	1.19	0.205	Cluster_0	1
Row4#1	1.327	0.149	Cluster_0	1
Row5#1	1.367	0.215	Cluster_0	1
Row6#1	1.288	0.271	Cluster_0	1
Row7#1	1.316	0.217	Cluster_0	1
Row8#1	1.332	0.156	Cluster_0	1

Figure 7.13 shows an output data table where, for example, column "Cluster Membership (Iter #1)" contains the data generated for column "Cluster Membership" at the end of loop iteration number 1, and column "Universe_0_0 (Iter #2)" contains the data generated as column "Universe_0_0" at the end of loop iteration number 2, and so on.

In order to demonstrate how such a loop output data table can be implemented in KNIME, we slightly modified the workflow used in the previous section "Counting Loop 1" and renamed it "Counting Loop 2". The "Counting Loop 2" workflow is in general identical to the "Counting Loop 1" workflow except for the choice of the loop end node. Here, instead of using the generic "Loop End" node, we used the "Loop End (Column Append)" node.

7.13. Output Table of a loop end node where the data resulting from each iteration is appended as new columns

Collected results - 0:31 - Loop End (Column Append)(loop end)

File

Table "default" - Rows: 99 | Spec - Columns: 12 | Properties | Flow Variables

Row ID	Universe_0_0	Universe_0_1	Cluster Membership	Universe_0_0 (Iter #1)	Universe_0_1 (Iter #1)	Cluster ...	Universe_0_0 (Iter #2)	
Row0	0.296	0.269	Cluster_0	1.296	0.269	Cluster_0	2.296	0
Row1	0.206	0.265	Cluster_0	1.206	0.265	Cluster_0	2.206	0
Row2	0.367	0.142	Cluster_0	1.367	0.142	Cluster_0	2.367	0
Row3	0.19	0.205	Cluster_0	1.19	0.205	Cluster_0	2.19	0
Row4	0.327	0.149	Cluster_0	1.327	0.149	Cluster_0	2.327	0
Row5	0.367	0.215	Cluster_0	1.367	0.215	Cluster_0	2.367	0
Row6	0.288	0.271	Cluster_0	1.288	0.271	Cluster_0	2.288	0
Row7	0.316	0.217	Cluster_0	1.316	0.217	Cluster_0	2.316	0
Row8	0.332	0.156	Cluster_0	1.332	0.156	Cluster_0	2.332	0
Row9	0.442	0.348	Cluster_0	1.442	0.348	Cluster_0	2.442	0
Row10	0.412	0.345	Cluster_0	1.412	0.345	Cluster_0	2.412	0
Row11	0.386	0.24	Cluster_0	1.386	0.24	Cluster_0	2.386	0
Row12	0.251	0.2	Cluster_0	1.251	0.2	Cluster_0	2.251	0
Row13	0.544	0.123	Cluster_0	1.544	0.123	Cluster_0	2.544	0
Row14	0.444	0.31	Cluster_0	1.444	0.31	Cluster_0	2.444	0
Row15	0.396	0.24	Cluster_0	1.396	0.24	Cluster_0	2.396	0
Row16	0.399	0.432	Cluster_0	1.399	0.432	Cluster_0	2.399	0

Loop End (Column Append)

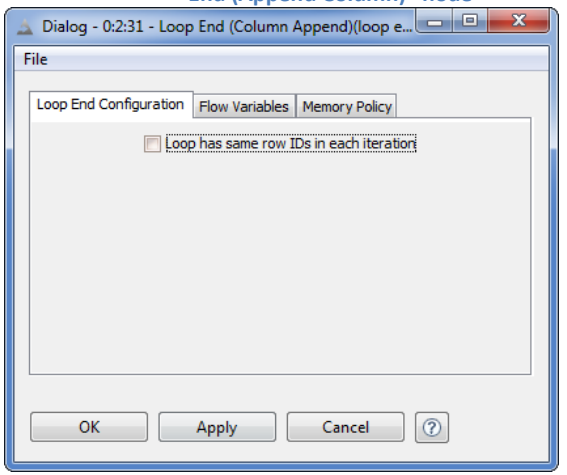
The “Loop End (Column Append)” node marks the end of a loop and collects the resulting data tables from all loop iterations.

At each iteration, the output data table is appended as a set of new columns to the output data table as collected so far.

This node requires minimal configuration:

- a flag to mark the fact that the RowIDs are the same for each iteration. This flag speeds up the joining process of the new columns to the table of collected results.

7.14. Configuration window of the “Loop End (Append Column)” node

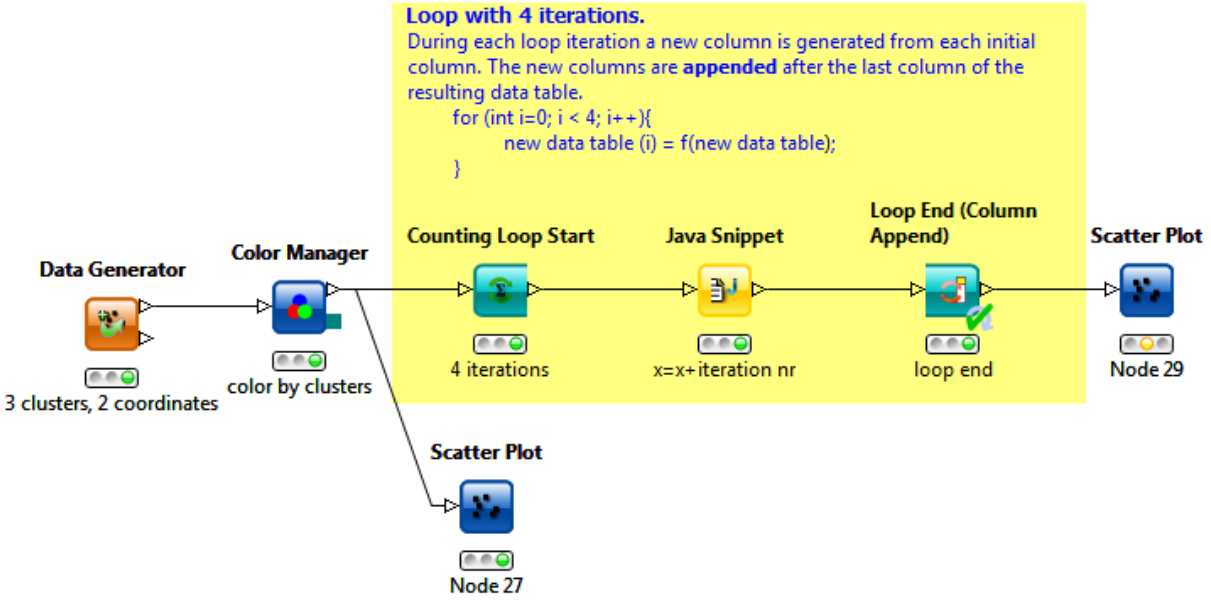


The “Loop End (Column Append)” node is used to close a loop, if the loop body produces new columns and the new columns are supposed to be appended to the final output data table.

By using the “Loop End (Column Append)” node to close the loop in the “Counting Loop 2” workflow, the data table in figure 7.13 is generated. Here an x-shifted copy of the original data set is generated at each iteration and then appended as a new set of columns to the current output data table. The “Counting Loop 2” workflow is shown in the figure below.

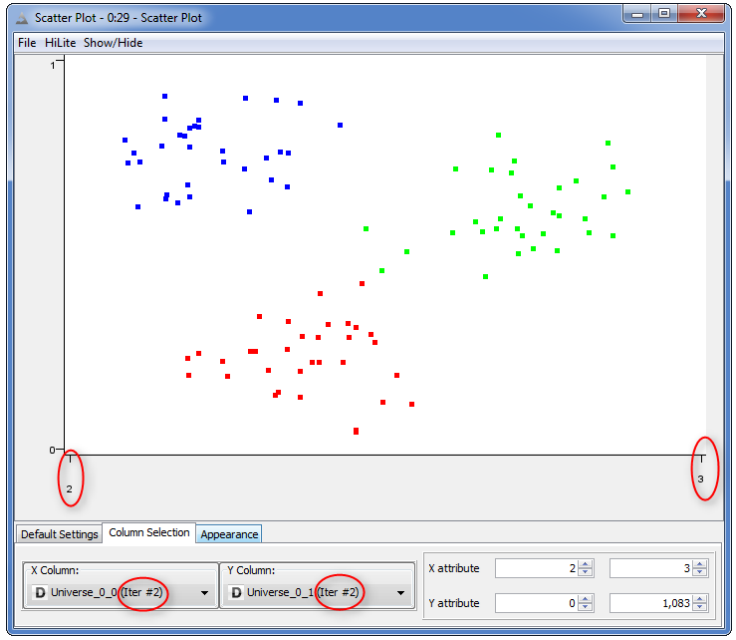
Note. The “Loop End (Column Append)” node appends all columns of the output table, not only the processed ones. Therefore the input columns, if not removed with a “Column Filter” node, appear multiple times unchanged (as many times as many iterations) in the result table.

7.15. Workflow "Counting Loop 2"



Notice that the View produced by the “Scatter Plot” node at the end of the “Counting Loop 2” workflow looks very different from figure 7.9. The plot can only show two columns at a time as x-axis and y-axis; that is, it can only show the scatter plot of the data coming from one iteration like (“Universe_0_0 (iter #n), “Universe_0_1 (iter #n)”). Thus the scatter plot of each iteration looks exactly the same as figure 7.2, except for the x-range. Indeed, assuming that the “Universe_0_0 (iter #n)” variable is being used as x-axis, the range of the x-axis will change depending on the iteration number. The scatter plot for the data generated during iteration # 2 is shown in the figure below.

7.16. Scatter Plot of the Data Table generated by Loop Iteration # 2



7.5. Keep Looping till a Condition is Verified ("do-while" loop)

In the previous section we have seen how to repeatedly execute a group of operations on the input data for a pre-defined number of times. However, there are other cases where we do not know upfront how many iterations are needed. We only know that the loop has to stop when a certain condition is met. For example, the loop described in figure 7.1 can be implemented as iterating the `out=inp+1` operation 10 times or as iterating the `out=inp+1` operation till out >=10.

In this section, we are going to show how to implement the second solution: iterating till a condition is met. In order to implement a "do-while" cycle, i.e. a loop that iterates till a condition is met, we use the "Generic Loop Start" node to start the loop and the "Variable Condition Loop End" to end the loop and collect the results.

Generic Loop Start

The “Generic Loop Start” node is located in the “Flow Control” -> “Loop Support” category.

The “Generic Loop Start” node starts a generic loop without any previous assumptions and, because of that, it needs no configuration settings.

The “Generic Loop Start” node starts a generic loop without any assumptions. The “Variable Condition Loop End” implements the loop condition that is checked at the end of each iteration. If the loop condition is verified the “Variable Condition Loop End” ends the loop and makes the collected results available at the output port.

Variable Condition Loop End

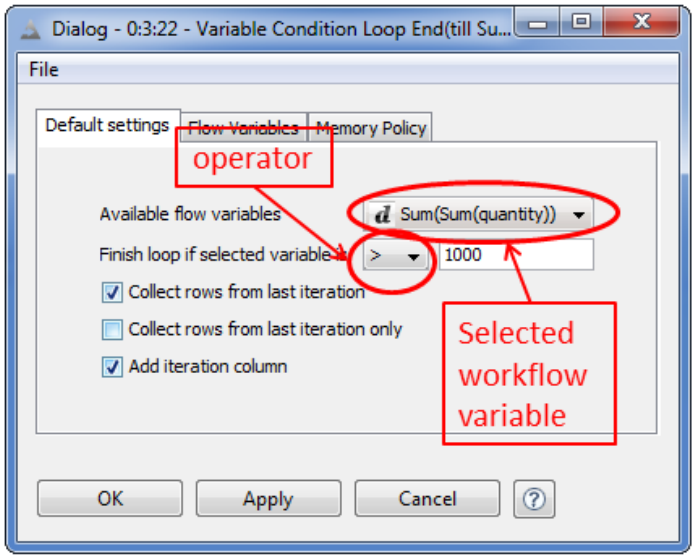
The “Variable Condition Loop End” implements a condition to terminate a generic loop. At the end of each loop iteration the condition is evaluated and, if it is true, the loop is terminated and the collected results are made available at the output port.

The terminating condition can only be implemented on a flow variable. The configuration window then requires:

- the flow variable on which the condition is evaluated
- the condition (i.e. comparison operator + value)
- a few flags to exclude or only include the last iteration results and/or to append the iteration column

A second output port shows the progressive values for each loop iteration of the flow variable used to implement the loop condition.

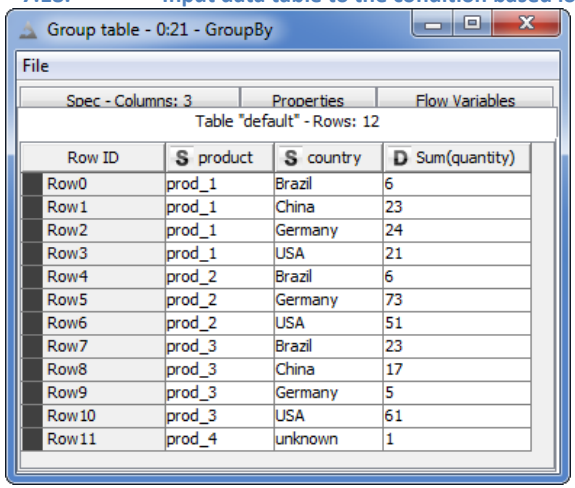
7.17. Configuration window of the “Variable Condition Loop End” node



Note. The loop terminating condition can only be implemented on one of the flow variables available to the "Variable Condition Loop End" node. This means that the flow variable for the loop condition has to be defined before the loop is started and updated at the end of each iteration.

In order to show how a such a loop can be implemented, we created a new workflow under the "Chapter7" workflow group and we named it "Loop with Final Condition".

7.18. Input data table to the condition based loop



Row ID	S product	S country	D Sum(quantity)
Row0	prod_1	Brazil	6
Row1	prod_1	China	23
Row2	prod_1	Germany	24
Row3	prod_1	USA	21
Row4	prod_2	Brazil	6
Row5	prod_2	Germany	73
Row6	prod_2	USA	51
Row7	prod_3	Brazil	23
Row8	prod_3	China	17
Row9	prod_3	Germany	5
Row10	prod_3	USA	61
Row11	prod_4	unknown	1

A "Database Reader" node was introduced to read the "sales" table from the "Book2" database, as already described in section 2.5. Then a "GroupBy" node was set to calculate the sum(quantity) by country and product type, producing the data table in figure 7.18.

We then built a condition based loop to calculate the sum of values in the "Sum(quantity)" columns. The original data table is copied and appended at the output port as many times as necessary till the sum("Sum(quantity)") reaches 1000.

The loop was started with a "Generic Loop Start" node. Since no operations needed to be performed on the input data, the loop has no body and the loop end node follows the loop start node directly. As loop end node we used the "Variable Condition Loop End", because the loop has to stop when a given condition is met.

The loop condition must look something like: $\sum_{i=0}^{current\ iteration} sum(quantity) > 1000$. Thus, at the end of each iteration we have to calculate $\sum_{i=0}^{current\ iteration} sum(quantity)$ and feed it into a new workflow variable.

A second "GroupBy" node is then used to calculate the sum of values in the "Sum(quantity)" column at the end of each iteration. No group settings is necessary for this "GroupBy" node, because the total sum is calculated. The "GroupBy" node should actually work on the intermediate results, i.e. the results at the end of each iteration, at the output port of the "Variable Condition Loop End" node. Unfortunately loop end nodes do not show the intermediate loop results at their output port. This second "GroupBy" node uses the same input data table as the "Generic Loop Start" node.

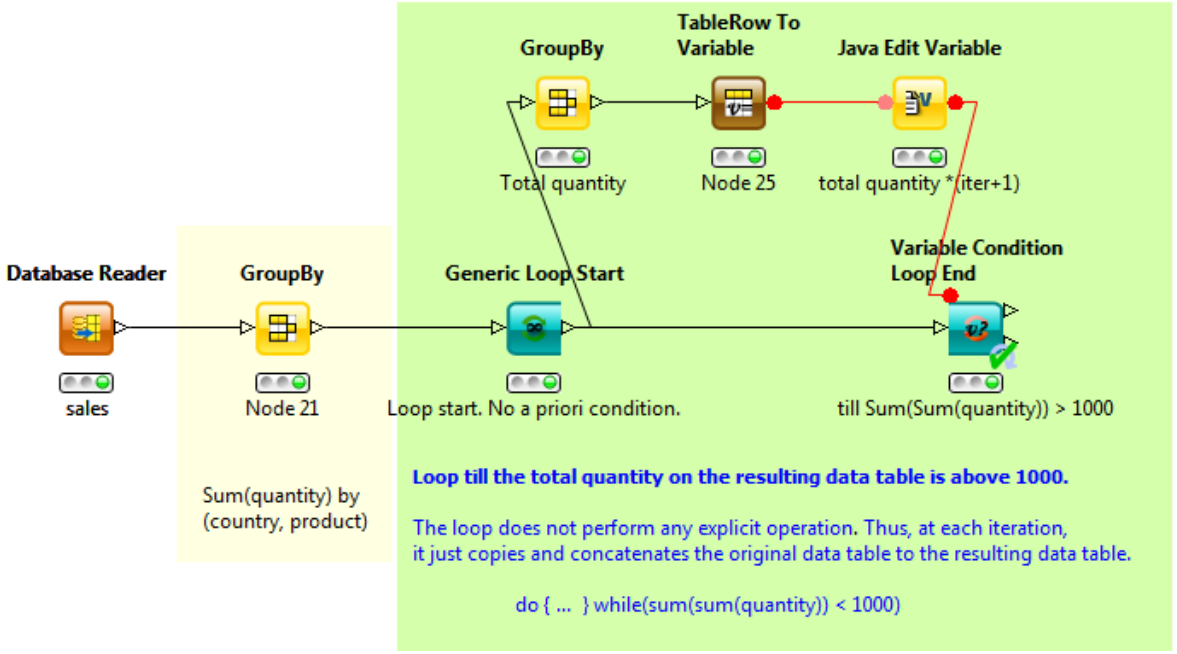
In order to implement the loop condition, we multiplied the total sum - Sum(Sum(quantity)), which is the same at each iteration - by the current iteration number, since the original data table is not altered inside the loop. This is the final solution:

- A "GroupBy" node calculates the total sum of values in the "sum(quantity)" column in the input data table

- The total sum is transformed into a workflow variable named “Sum(Sum(quantity))” by means of a “TableRow To Variable” node
- A “Java Edit Variable” node multiplies the total sum in the workflow variable “Sum(Sum(quantity))” by the current iteration number
- The final workflow variable is then used to implement the loop condition in the “Variable Condition Loop End” (Fig. 7.17).

Note. The part of the workflow updating the flow variable for the loop condition has to be included in the loop.

7.19. Workflow "Loop with final Condition"



Note. Like other loop end nodes, the “Variable Condition Loop End” node introduces a few new flow variables related to the loop execution, like “currentIteration” which contains the current iteration number.

7.6. Loop on a List of Values

Let's now use the same data as in the previous section: the data from the "sales" table from the "Book2" database. Next, let's imagine clustering the original data by country. That is, we want to build and save a cluster model from the data of each one of the countries represented in the original data set. After building a list with all country names, we would like to loop on this list, build a model for each row, and save it to a file.

In a new workflow named "Loop on List of Values", we started by reading the data from the "sales" table in the "Book2" database with the "Database Reader" node named "sales" as explained in section 2.5. After that, we built the list of countries, to loop on such a list, and to build a k-means model at each iteration, i.e. for each country.

In order to build the list of unique values from the "country", we used a "Column Filter" node to keep only the "country" column and a "GroupBy" node with the "country" group column and no aggregation columns to make the list of unique countries. The "GroupBy" node was named "by country". For each country, we still needed:

- To isolate the data rows referring to only one country;
- To run the k-means algorithm on the filtered data;
- To save the cluster model into a PMML file.

This procedure was supposed to be repeated for all countries present in the original data. We therefore needed to build a loop to isolate the data rows for a given country at a time, force this data through the same clustering procedure, and iterate across a list of country values.

In the "Flow Control" -> "Loop Support" category, the node "TableRow To Variable Loop Start" starts exactly the kind of loop that iterates across a list of values.

In the new "Loop on List of Values" workflow, we placed a "TableRow To Variable Loop Start" node after the "GroupBy" node named "by country". The output data table of the "by country" node contains just one column named "country" with all country values (Fig. 7.20). This is the input data table of the "TableRow To Variable Loop Start" node.

TableRow To Variable Loop Start

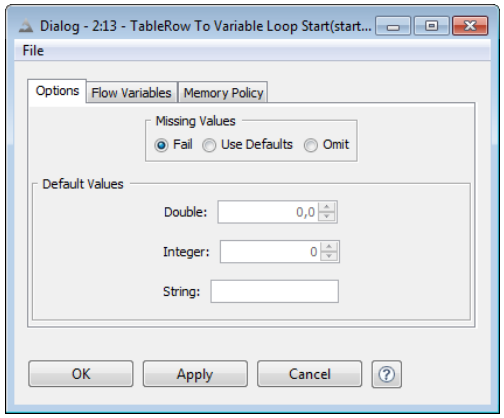
The “TableRow To Variable Loop Start” starts a loop.

This node takes a data table at the input port and, row by row, transforms the column values into workflow variables with the same name as the column name. At each iteration, the workflow variables update their value with the next value from the corresponding data column.

The “TableRow To Variable Loop Start” node does not require any configuration settings, besides the selection of the strategy for missing value handling.

The loop performs as many iterations as the number of rows in the input table at the loop start node.

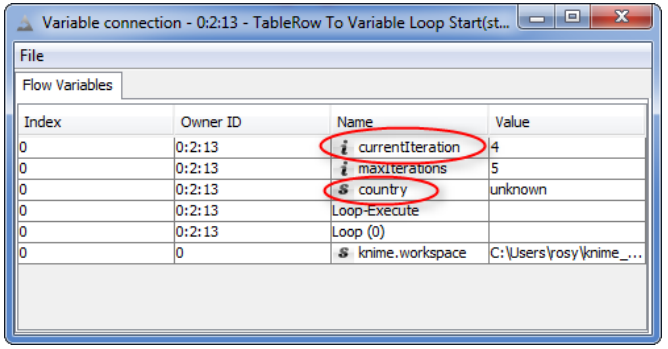
7.20. Configuration window of the “TableRow To Variable Loop Start” node



If we execute the loop start node and look at the output port view (right-click the context menu of the “TableRow To Variable Loop Start” node and select the option “Variable Connection”), we can see that there is a new workflow variable named “country” with value “Brazil” and that the current iteration is numbered 0. “Brazil” was indeed the first value in the “country” column in the input data table.

We closed the loop with a generic “Loop End” node. If we now run “Step Loop Execution” from the context menu of the “Loop End” node, at the second iteration we see that the workflow variable “country” takes the next value in the list: “China”. And so on (Fig. 7.21.).

7.21. Output port view from the context menu of the "TableRow To Variable Loop Start" node



We still need to build the body of the loop. The goal was to train a k-Means algorithm on the data of the selected country and to write the final cluster model into a PMML file, at each loop iteration. After the "TableRow To Variable Loop Start" node and before the "Loop End" node, we therefore introduced:

- A "Row Filter" node to select only those data rows referring to the current value of the workflow variable "country";
- A "k-Means" node to build 3 clusters on the filtered data by using the columns "amount" and "";
- A "Java Edit Variable (simple)" node to create the output path for the PMML file to save the cluster model ;
- A "PMML Writer" node to write the cluster model into a PMML file.

The execution of a loop on a list of values can become quite slow, if the data set to loop on is very large and the list of values is quite long. One way to make the workflow execution faster is to cache some of the data. For example, in the "Loop on List of Values" workflow, we can cache the data after the "Row Filter" node, making the execution of the rest of the loop faster. In KNIME there is a node used only to cache data: the "Cache" node.

Cache

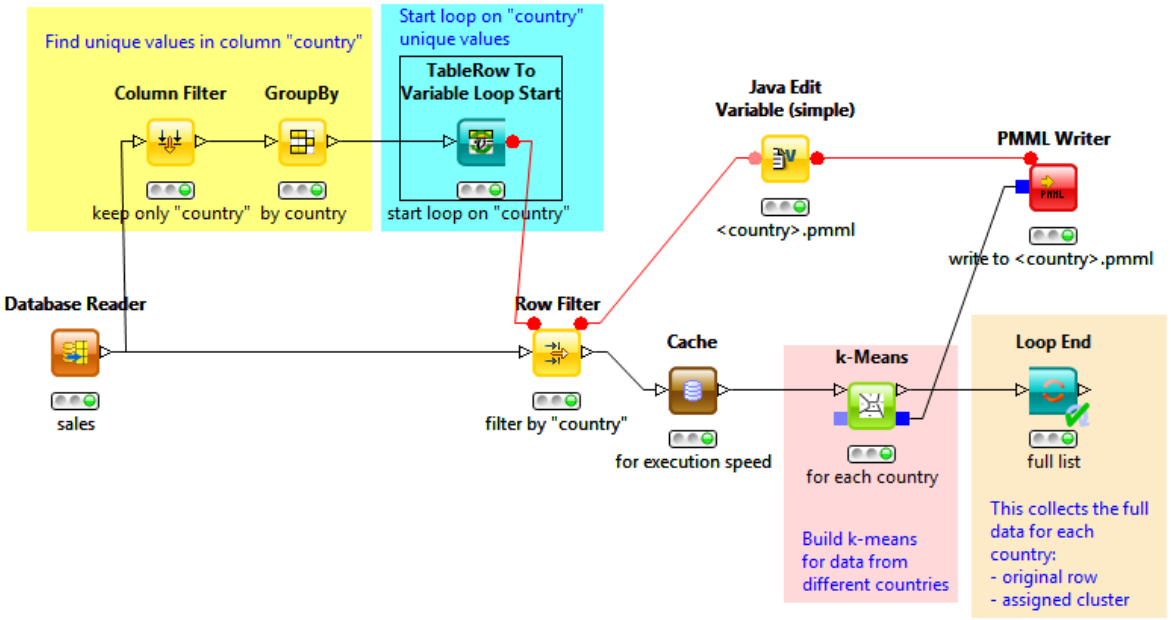
The "Cache" node caches all input data rows onto disk. This node only writes the data that is actually contained in the input table.

This is particularly useful when the preceding node performs a column transformation. In fact, many column transformation nodes only hide data, but the real data set is still all read in. For example, a "Filter Column" node hides most of the columns to the output. Since the "Cache" node only caches the data that is actually contained in the input table, using a "Cache" node after a "Column Filter" node might make the workflow execution faster, especially with loops when a data table is iterated many times.

The "Cache" node does not require any configuration settings.

The final version of the "Loop on List of Values" workflow, including the "Cache" node after the "Row Filter" node named "filter by country", is shown in figure 7.22.

7.22. Workflow "Loop on List of Values"



7.7. Loop on a List of Columns

Let's go back to the "Counting Loop 2" workflow used in section 7.4 to produce identical data sets shifted towards the right on the x-axis and appended as new columns to the resulting data set. Let's suppose now that we want to create two data sets, but shifted differently on the x-axis, and that we also want to append the new columns to the resulting data set. We could, of course, modify the "Counting Loop 2" workflow to iterate only twice and, depending on the iteration number, apply a custom x-shift to the data set.

We could also use a loop that iterates on a list of selected columns. This kind of loop iterates on a list of columns and runs the group of operations specified in the body loop for each column.

In KNIME a loop that iterates on a list of columns starts with a "Column List Loop Start" node and ends with a "Loop End (Column Append)" node.

Column List Loop Start

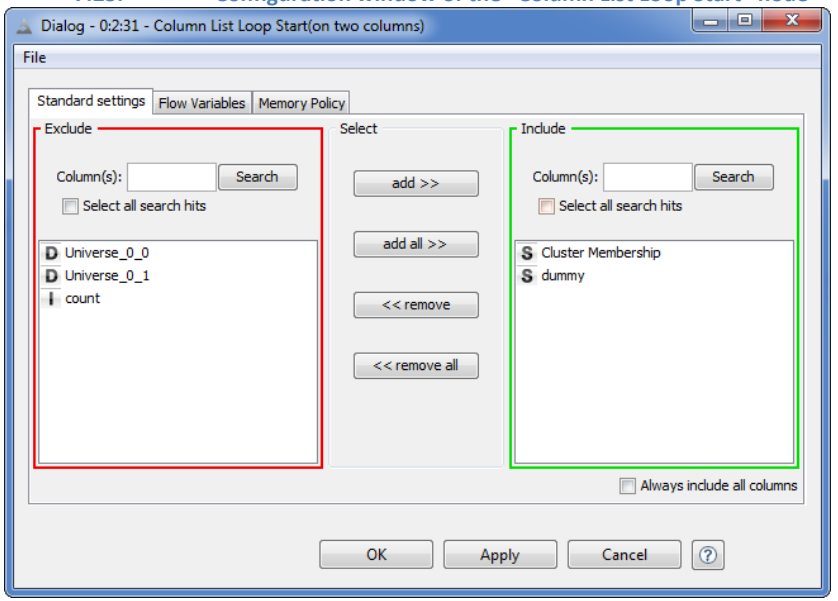
The “Column List Loop Start” node iterates over a list of columns in the input table.

The columns on which to iterate are selected in the configuration window by means of an “Exclude/Include” framework.

- The columns on which to iterate are listed in the “Include” frame on the right
- The columns to be excluded from the loop are listed in the “Exclude” frame on the left

To move single columns from the “Include” frame to the “Exclude” frame and vice versa, use the “add” and “remove” buttons. To move all columns to one frame or the other use the “add all” and “remove all” buttons.

7.23. Configuration window of the “Column List Loop Start” node



A “Search” box in each frame allows searching for specific columns, in case an excessive number of columns impedes an easy overview of the data.

The “Column List Loop Start” node separates the columns in the input data table into iteration columns and non-iteration columns. At each iteration, the non-iteration columns are processed and appended to the resulting data table together with the current iteration column, while all other iteration columns are excluded from the results.

In order to demonstrate how to implement a loop that iterates over a list of columns, we created a new workflow named “Loop on List of Columns”. This workflow generates a data set with 3 clusters in a 2-coordinate space and assigns a color to each data row depending on its cluster, as described in section 7.2. A “Scatter Plot” node allows the visualization of the clusters in the original data set.

We then inserted a loop on a list of columns to produce two data sets identical to the original data set, but differently x-shifted. The loop on a list of columns was implemented by means of a “Column List Loop Start” node, to start the loop, and of a “Loop End (Column Append)” node, to end the loop. The “Loop End (Column Append)” node is needed to append the resulting data columns to the final data table.

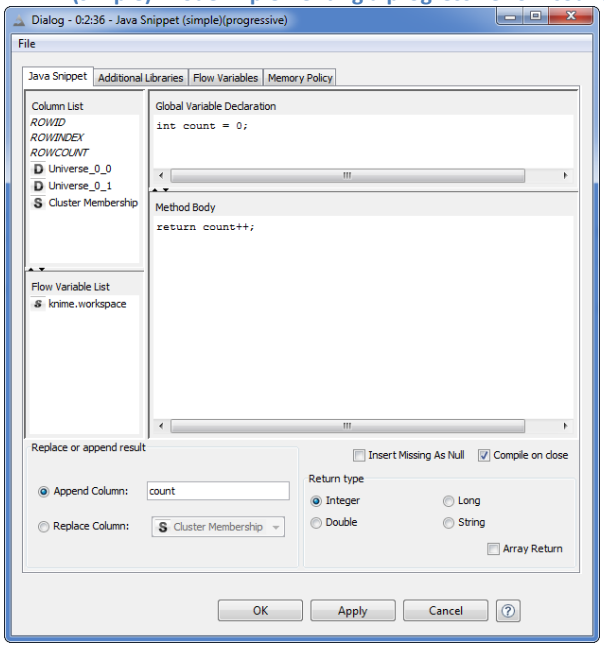
Since we need two final data sets each one with a customized shift on the x-axis, the loop had to iterate on two iteration columns. In addition, these two iteration columns could not be the x and y coordinates of the data set, since only the current iteration column can be part of the resulting data set.

The input data table had only 3 data columns: “Universe_0_0” (the x-coordinate), “Universe_0_1” (the y-coordinate), and “Cluster Membership”. The loop could not iterate on either “Universe_0_0” or on “Universe_0_1”, because they are being processed in the loop body. Only “Cluster Membership” was available as iteration column. We then created a second iteration column named “dummy” by means of a “Rule Engine” node, simply to make the loop possible.

We also introduced a progressive row counter, named “count”, with a “Java Snippet (simple)” node, whose configuration is reported here on the right.

The input data table at this point had 5 data columns: “Universe_0_0” (the x-coordinate), “Universe_0_1” (the y-coordinate), “Cluster Membership”, “dummy”, and “count”. We set “Cluster Membership” and “dummy” as iteration columns in the “Column List Loop Start” node (Fig. 7.23).

7.24. Configuration window of the "Java Snippet (simple)" node implementing a progressive row counter



The loop body was implemented by using a “Java Snippet (simple)” node again with the following code:

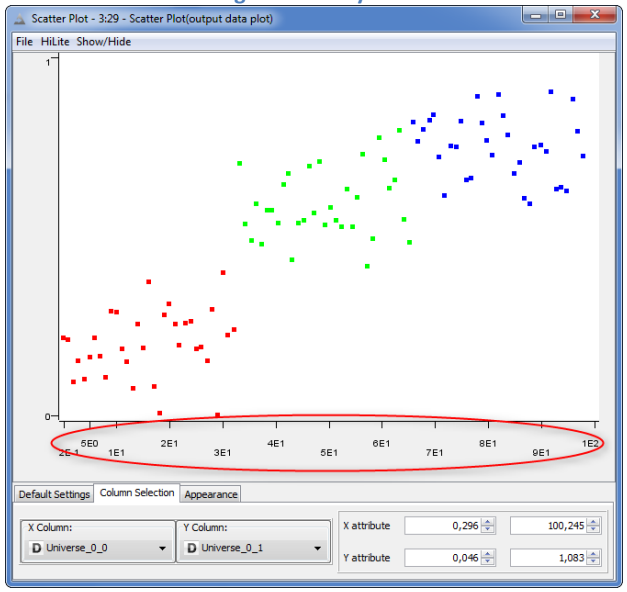
```
double val =0.0;

if($${ScurrentColumnName}$$.equals("dummy"))
   val = 100-$Universe_0_0$;
else
   val = $Universe_0_0$+$count$;

return val;
```

This “Java Snippet (simple)” node reverses the original data set and shifts it 100 units to the right when the iteration column is “dummy”; otherwise, i.e. if the iteration column is “Cluster Membership”, it shifts each data point as many units to the right as its row number. The “Java Snippet (simple)” node was named “x-shift on only one column”.

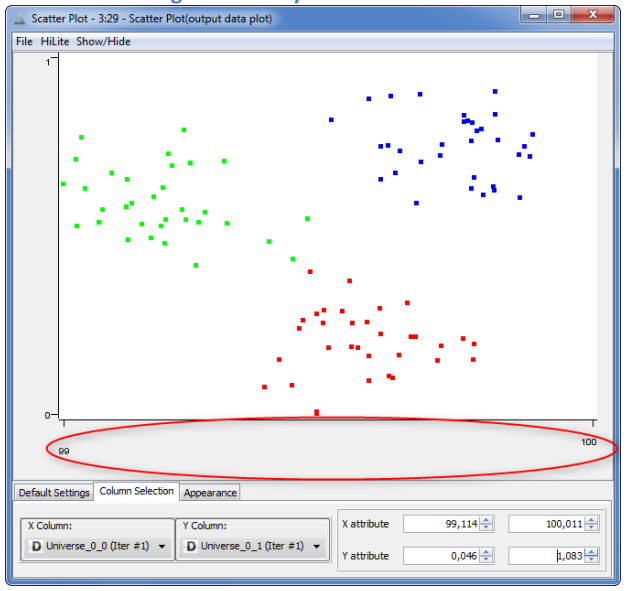
Unlike the previously seen loop start nodes, the “Column List Loop Start” node does not create a workflow variable for the iteration number. Instead, it creates a workflow variable with the iteration column name. Because of that, we could not use the iteration number in the “Java Snippet (simple)” node and we needed the new “count” column to produce the shift.

The two final data sets are reported in figure 7.25 and 7.26 respectively. The final workflow “Loop on List of Columns” is shown in figure 7.27.

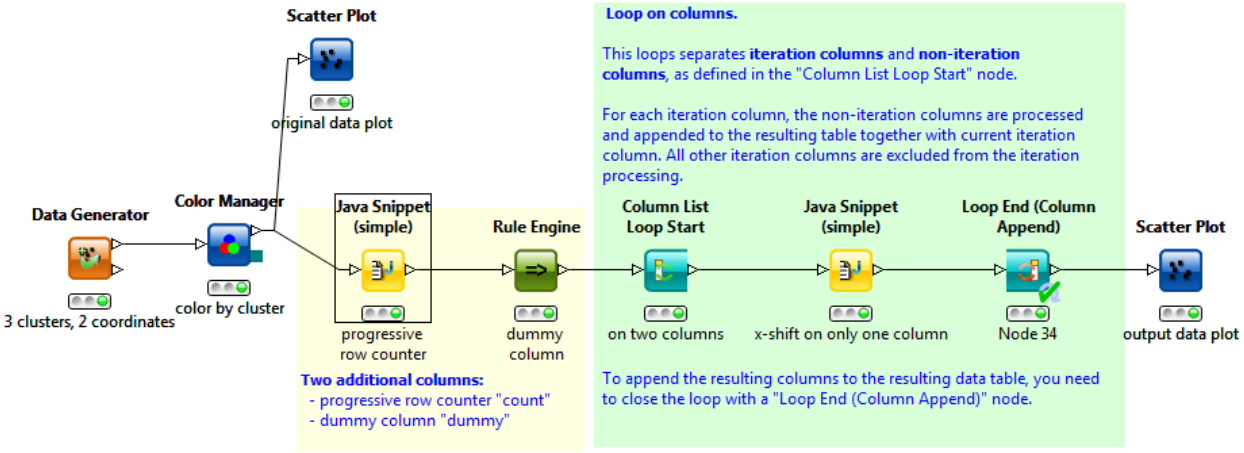
7.25. Data Set generated by iteration on column "dummy"



7.26. Data Set generated by iteration on column "Cluster Membership"



7.27. Workflow "Loop on List of Columns"



7.8. Loop on Data Groups and Data Chunks

Finally, the “Flow Control” -> “Loop Support” category includes a last type of loop: the loop on chunks or on groups. These loops iterate just once on the input data table. It divides the input data table into smaller pieces (chunks or groups) and iterates from the first one to the last piece till the end of the data table has been reached. The difference between groups and chunks is that:

- chunks are fixed pieces of data with always the same number of data rows;
- groups are variable size pieces of data grouping data rows on the basis of their values on a particular column.

So, for example, a data set with 5 clusters and 99 data rows can have 3 chunks of 33 data rows each or 5 groups, one for each cluster.

One advantage of using a chunk/groups loop is not in the number of loops on the input data set, since it only covers the input data set once, but the speed. Indeed, processing smaller chunks/groups of data at a time speeds up execution. A second advantage of using a chunk/group loop is that different processing can be applied to different subsets of data.

The Chunk Loop

A chunk loop starts with a “Chunk Loop Start” node and ends with any of the loop end nodes.

Chunk Loop Start

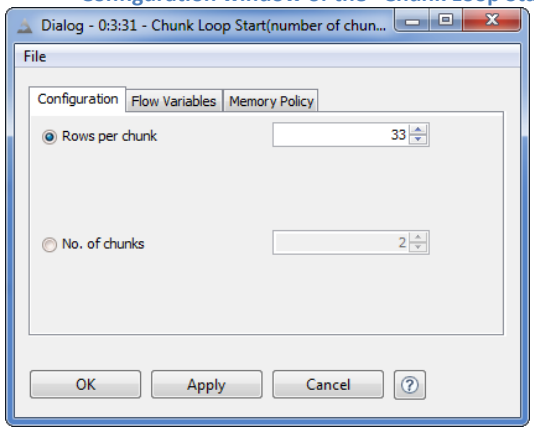
The “Chunk Loop Start” node starts a chunk loop.

It divides the input data table into a number of chunks and iterates over those chunks.

The configuration window of the “Chunk Loop Start” node requires:

- Either the (maximum)number of data rows in each chunk
- Or the number of chunks

7.28. Configuration window of the “Chunk Loop Start” node



We created a new workflow “Chunk Loop”. This workflow uses the same data generated by the “Data Generator” node named “3 clusters, 2 coordinates" as in the previous sections. In order to demonstrate how the chunk loop works, we used the “Chunk Loop” workflow to stretch the 3 clusters in the original data set in the same way along the x-axis.

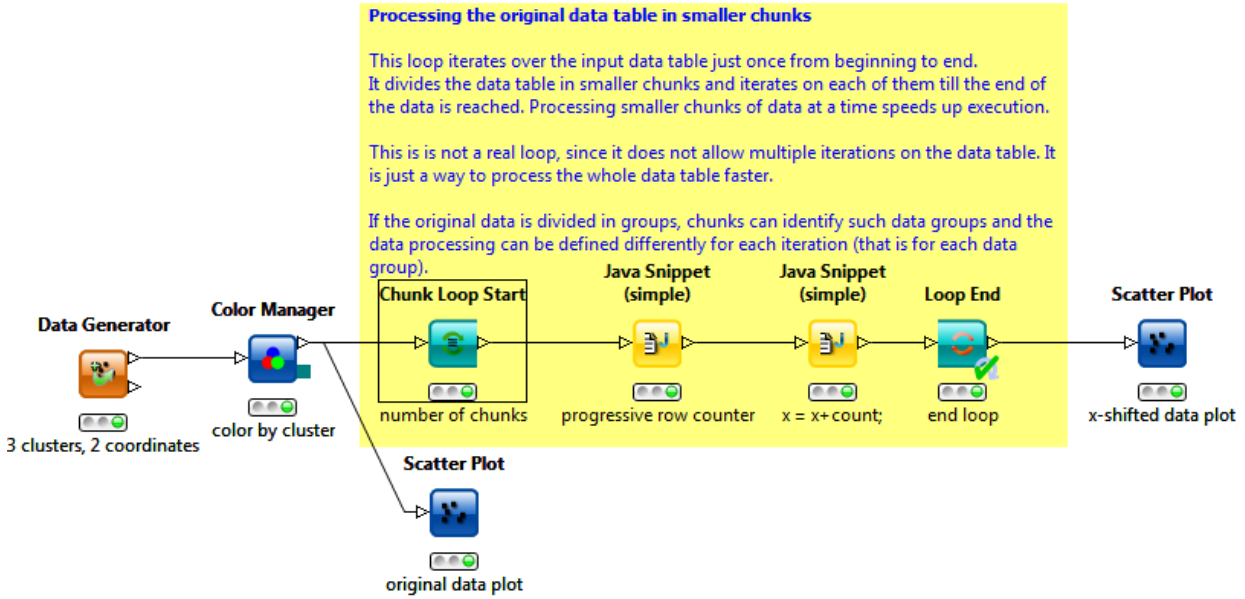
The original data set contains 33 data rows for cluster 1, 33 data rows for cluster 2, and 33 data rows for cluster 3. Defining chunks of 33 rows fits the clusters´ size perfectly. We implemented a chunk loop on chunks of 33 rows, for each chunk we defined a progressive row counter column, and we used that counter to shift the data of each cluster progressively. This translated into:

- A “Chunk Loop Start” node to start the chunk loop, with “Rows per chunk” set to 33;
- A “Java Snippet (simple)” node for the progressive row counter (Fig. 7.24);
- A “Java Snippet (simple)” node with code: “`return $Universe_0_0$+$count$;`”
- A generic “Loop End” node to close the loop and collect the results by appending the rows.

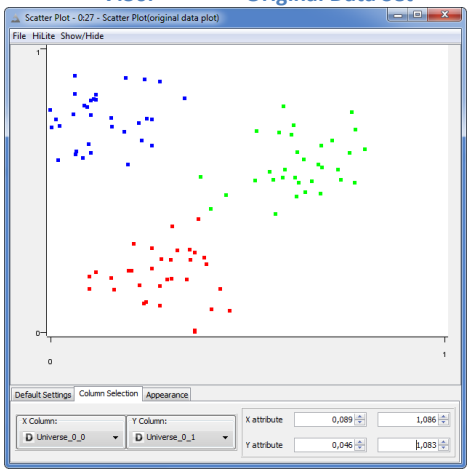
We then inserted a “Scatter Plot” node to visualize the results.

The final “Chunk Loop” workflow is shown in figure 7.29, while the data set before and after the progressive shifting is shown in figure 7.30 and 7.31 respectively.

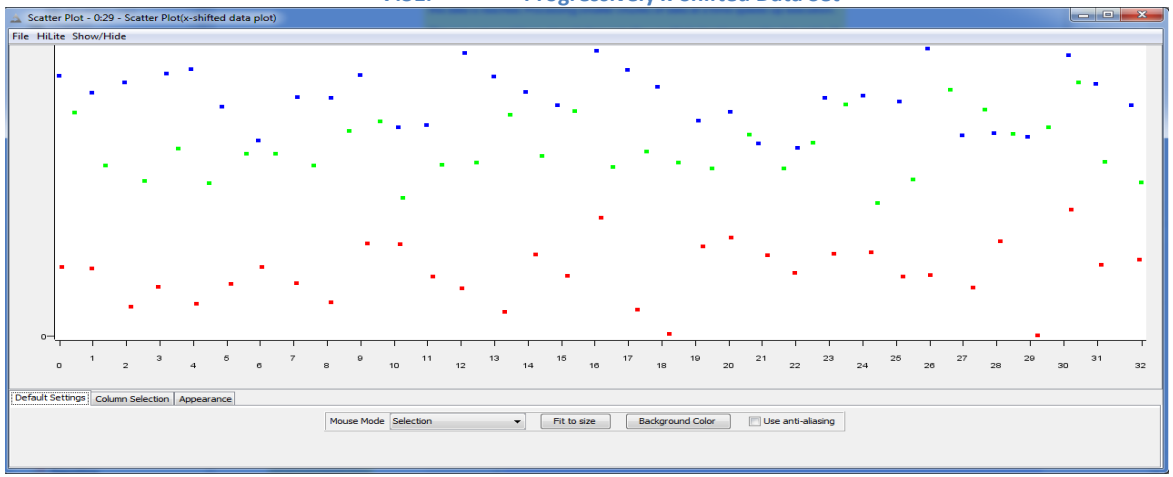
7.29. Workflow "Chunk Loop"



7.30. Original Data Set



7.31. Progressively x-Shifted Data Set



Let's now extend the "Chunk Loop" workflow, by calculating the new cluster prototypes after each data chunk shift. To do that, we introduced a "GroupBy" node at the end of the chunk loop cycle. This "GroupBy" node calculates the mean value of the two coordinates, "Universe_0_0" and "Universe_0_1", across the whole chunk of data made available by the "Chunk Loop Start" node and processed by the following "Java Snippet (simple)" nodes. The "GroupBy" node was named "x-shifted cluster prototype". Since we defined the loop chunks to contain exactly the rows with a specific cluster, each iteration of the chunk loop calculates the new cluster prototypes. At the end of the loop we have two data tables: the progressively x-shifted data and the corresponding new cluster prototypes. It would be good to export both data tables from the loop. To collect two data results at the end of a loop, KNIME has a special end loop node: the "Loop End (2 ports)" node.

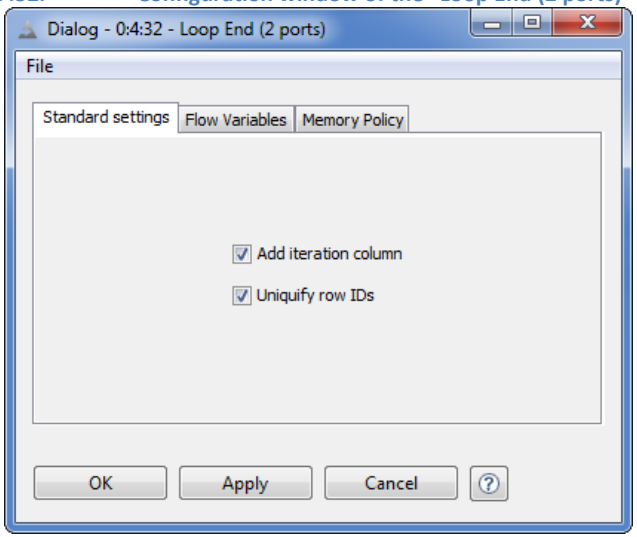
Loop End (2 ports)

The "Loop End (2 ports)" node closes a loop, collects the results from two input ports, and presents the data tables after the loop at two output ports respectively. This means that with this node we can export two data tables built with the preceding loop.

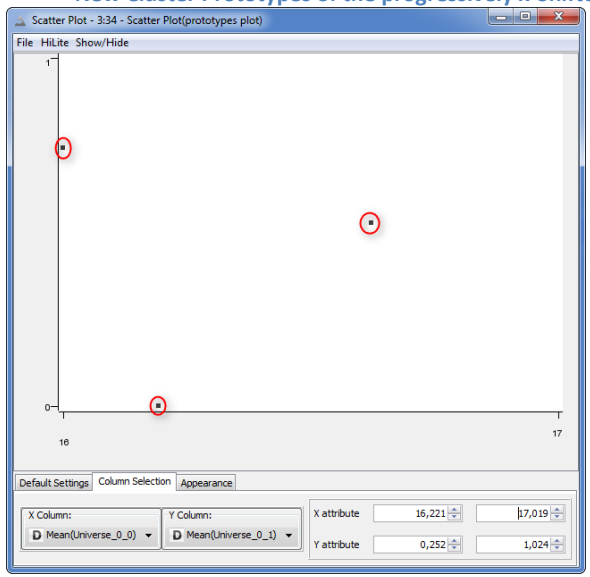
The "Loop End (2 ports)" node requires only minimal settings, that is:

- A flag specifying whether to add the iteration column at the end of the resulting data tables
- A flag specifying whether to force RowIDs to be unique

7.32. Configuration window of the "Loop End (2 ports)" node



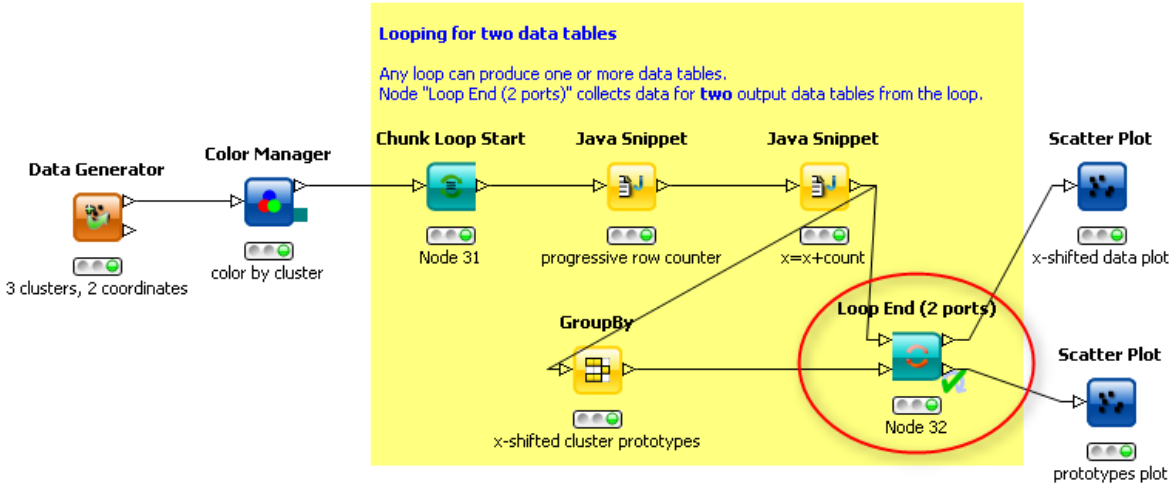
7.33. New Cluster Prototypes of the progressively x-Shifted Data Sets



We used this “Loop End (2 ports)” node to close the loop and collect the results.

We then connected it to two “Scatter Plot” nodes to visualize the progressively x-shifted data (Fig. 7.31) as in the previous workflow and the corresponding prototypes (Fig. 7.33). We named this new workflow “Chunk Loop 2 Ports”.

7.34. Workflow "Chunk Loop 2 Ports"



The Group Loop

The same loop, as shown in the previous section, could be implemented by using a group loop. The group loop group the data rows according to their cluster value in the "Cluster Membership" data column. A group loop starts with a "Group Loop Start" node and ends with a generic "Loop End" node.

Group Loop Start

The "Group Loop Start" node starts a group loop.

It defines a number of groups on the input data table and iterates over those groups.

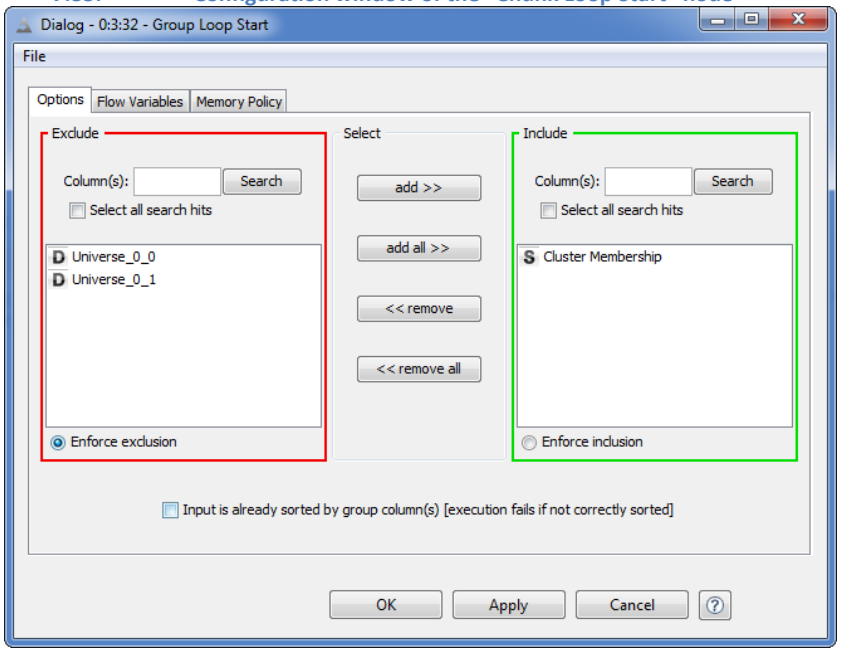
The configuration window of the "Group Loop Start" node requires the data column on which to group data rows with the same value.

This is obtained by means of an Exclude/Include framework.

Data columns in the "Include" frame will be used to group the data rows; those in the "Exclude" frame will not. To move data columns from one frame to the other, use the "add", "remove", "add all", and "remove all" buttons.

"Enforce Inclusion" and "Enforce Exclusion" add possible new data columns to the "Exclude" frame or to the "Include" frame respectively.

7.35. Configuration window of the "Chunk Loop Start" node



We added a "Group Loop Start" node to the "Chunk Loop" workflow and we configured it to group the data rows by their values in the "Cluster_Membership" data column. The body loop then was the same as in the chunk loop, that is:

- A "Java Snippet (simple)" node for the progressive row counter (Fig. 7.24);
- A "Java Snippet (simple)" node with code: "`return $Universe_0_0$+$count$;`"

Since the chunk size in the chunk loop was such as to include a full cluster, the results of the chunk loop and of the group loop are identical.

One last node in the “Loops” category that might be worth mentioning is the “Breakpoint” node. The “Breakpoint” node can be inserted in a loop body and, while it does not process data, it might give some control on the loop execution.

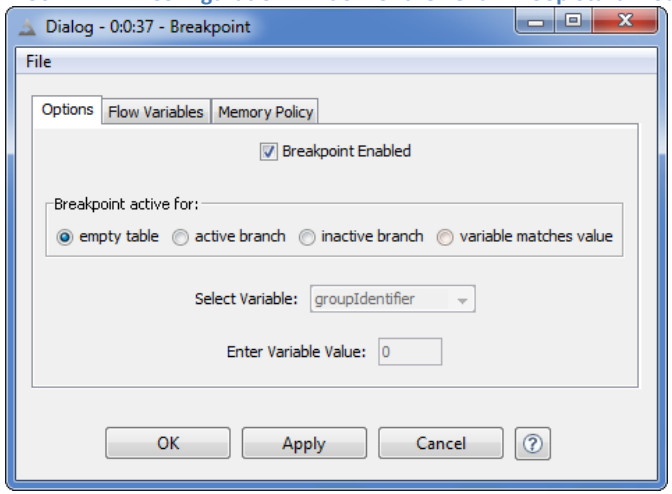
Breakpoint

The “Breakpoint” prevents the loop execution when the input table fulfills a user-specified condition, like the input data table is empty or a variable matches some value.

The configuration window then needs:

- The flag enabling the breakpoint
- The condition for when the breakpoint has to disable the loop execution (like for example an empty input data table)
- The name of the flow variable and its breakpoint value if the “variable matches value” condition has been selected.

7.36. Configuration window of the “Chunk Loop Start” node



7.9. Exercises

Exercise 1

Generate 5400 patterns in a two-dimensional space grouped in 6 clusters each with a standard deviation value of 0.1 and no noise. Assign different colors to the data of each cluster and plot the data by means of a scatter plot.

Process the data of each cluster in a different way according to the formulas below and observe how the clusters have changed by means of a scatter plot again. To make the workflow run faster and be more flexible, use a loop to process the data.

x = Universe_0_0, y = Universe_0_1

Cluster 0: **x = x**
Cluster 1: **x = sqrt(x)**

Cluster 2: x = x + iteration number
Cluster 3: x = x * iteration number
Cluster 4: x = y
Cluster 5: x = x*x

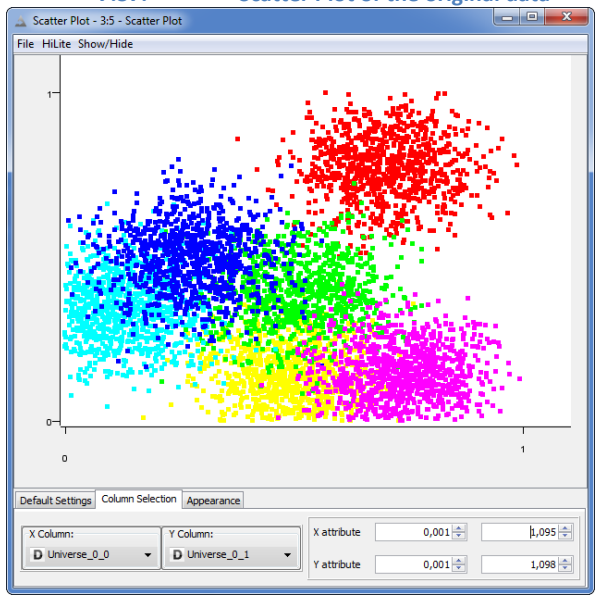
Solution to Exercise 1

To process the data in a loop, we used a "Chunk Loop Start" node. We defined the chunk size to be 900 data rows in order to have exactly one cluster in each chunk. We then closed the loop with a generic "End Loop" node which concatenates the output tables into the final result. The different processing for each cluster is implemented with a "Java Snippet (simple)" node with the following code:

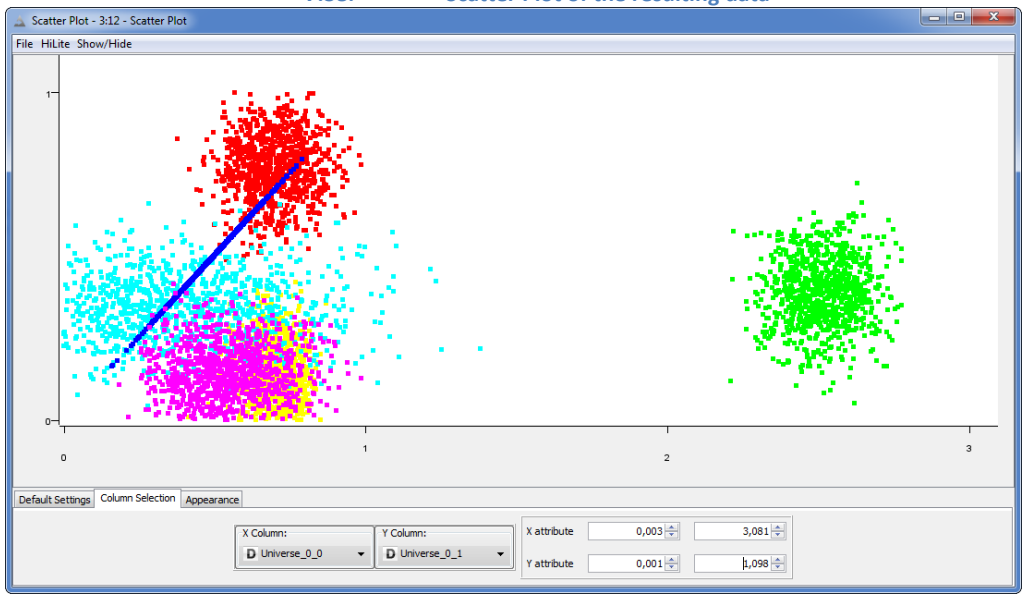
```
Double x = 0.0;
if     ($${IcurrentIteration}$$ == 0)     x = $Universe_0_0$;
else if($${IcurrentIteration}$$ == 1)     x = Math.sqrt($Universe_0_0$);
else if($${IcurrentIteration}$$ == 2)     x = $Universe_0_0$+$${IcurrentIteration}$$;
else if($${IcurrentIteration}$$ == 3)     x = $Universe_0_0$*$${IcurrentIteration}$$;
else if($${IcurrentIteration}$$ == 4)     x = $Universe_0_1$;
else                                      x = $Universe_0_0$*$Universe_0_0$;
return x;
```

The scatter plot of the original data is shown in figure 7.35 and the scatter plot of the processed data is shown in figure 7.36.

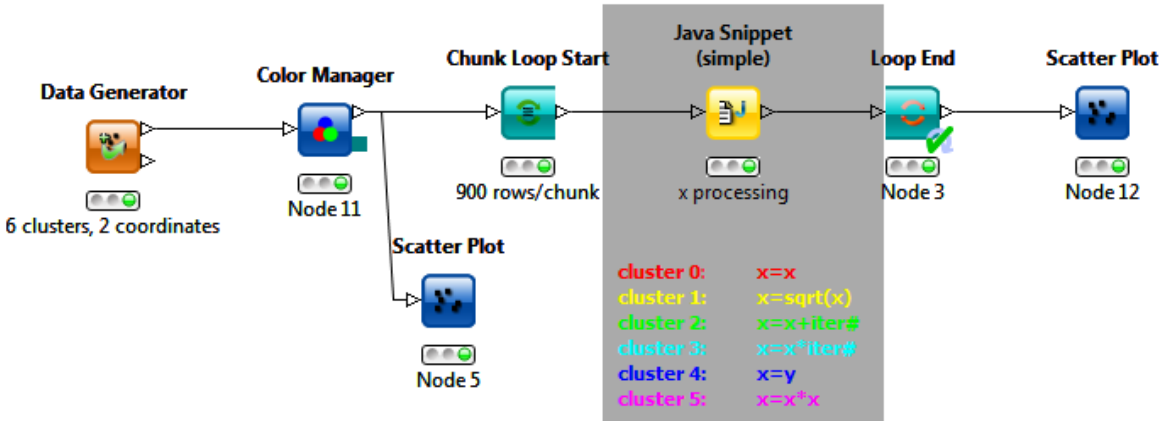
7.37. Scatter Plot of the original data



7.38. Scatter Plot of the resulting data



7.39. Exercise 1: The workflow



Exercise 2

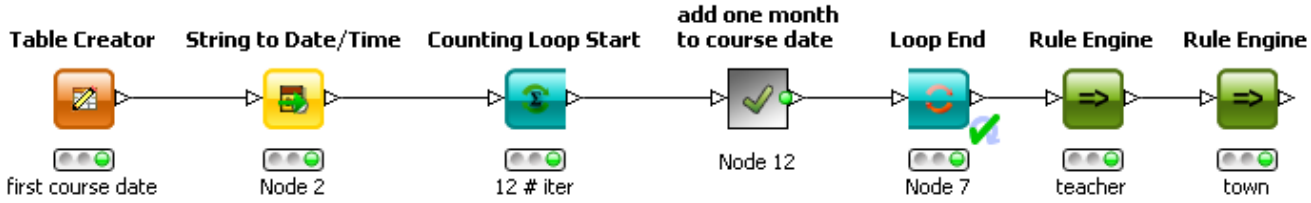
On the 15th of each month a course takes place, starting from 15.01.2011. Teacher “Maria” teaches till the end of March, teacher “Jay” till the end of September, and teacher “Michael” till the end of the year. The course is held in San Francisco from May to September and in New York the other months.

Generate the full table of the courses for the year 2011 with course date, teacher name, and town.

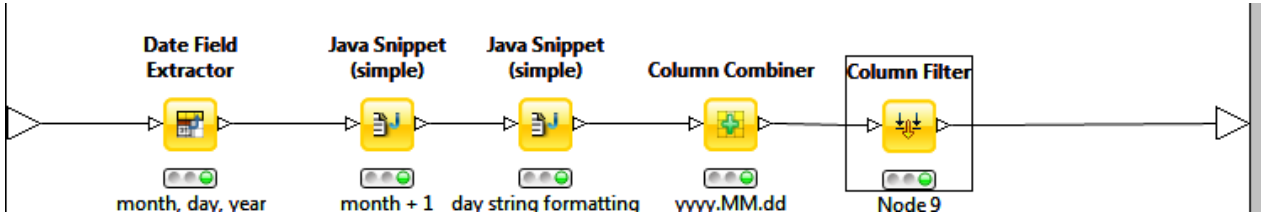
Solution to Exercise 2

To solve this problem we started from a data set with the first course date only: 15.01.2011. Then we used a “Counting Loop Start” node that iterates 12 times on the initial data set and generates a new date at each iteration. The “teacher” and “town” columns are both obtained with a “Rule Engine” node.

7.40. Exercise 2: The workflow



7.41. Exercise 2: The "add one month to course date" meta-node

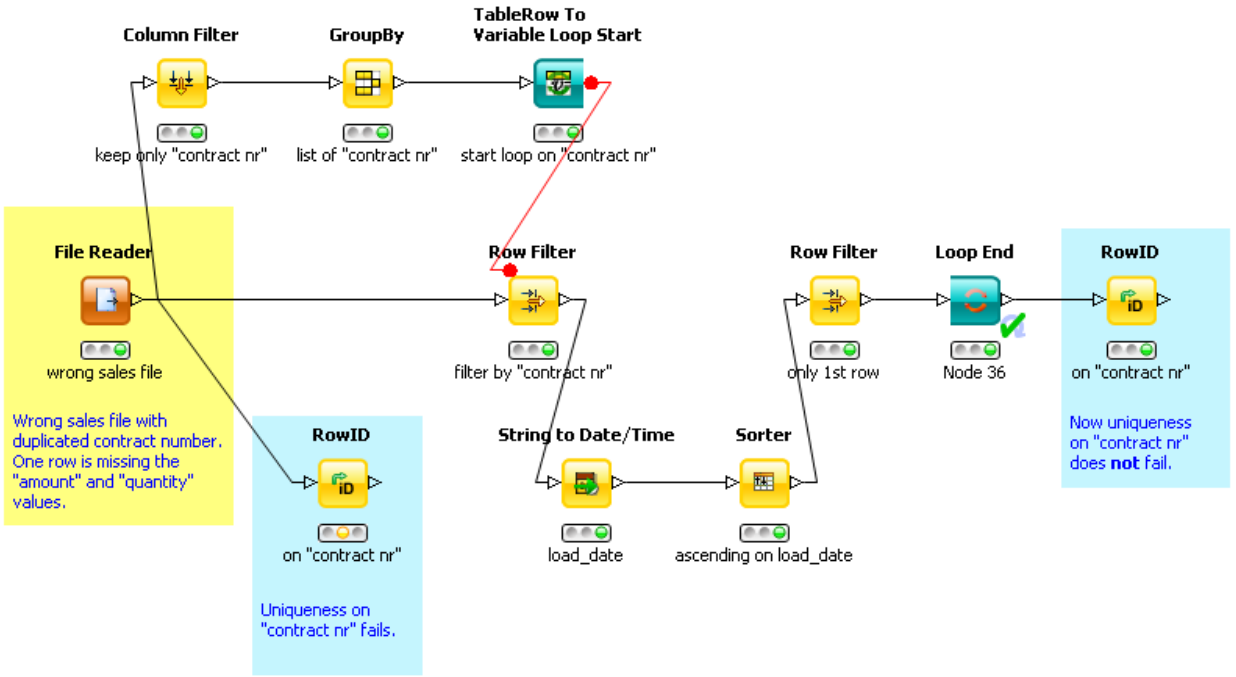


Exercise 3

This exercise gets rid of duplicated unnecessary data rows. In the Download Zone there is a file called “wrong_sales_file.txt” which contains sales records, each one with a contract number. However the file does contain duplicate records. In fact, for each sale you can find an older record with a few missing values and a more recent record with all field values correctly filled in. The column “load_date” contains the data of when the sale record was inserted into the file. Of course, we want to get rid of the old records with missing values and keep only the recent records with all fields filled in. In this way, we get a sales table with a unique record ID, i.e. the contract number, and only the most up to date values.

Solution to Exercise 3

7.42. Exercise 3: The workflow



In the solution workflow, we read the “wrong_sales_file.txt” with a “File Reader” node. If we execute a “RowID” node on the “contract nr” column and with the flag ”ensure uniqueness” not enabled, then the “RowID” node’s execution fails. This means that the “contract nr” column contains non-unique

values. Indeed, for each “contract nr” we have two records in the data table: an old one with many missing values and a recent one with all fields filled in.

In order to filter out the older records with missing values, we loop on the “contract nr” list with a “TableRow To Variable Loop Start” node. At each iteration, we keep only the records with the “contract nr” of the current iteration (“Row Filter” node), we sort the selected records by “load_date”, in descending order, and we keep only the first row (the second “Row Filter” node), which is the most recent record. The loop is then closed by a generic “Loop End” node.

If we now run a “RowID” node on the loop results, which is similarly configured to the first “RowID” node of this exercise, it should not fail.

Exercise 4

Let’s suppose that the correct file called “sales.txt” had been saved in many pieces. In particular, let’s suppose that each column of the file had been saved together with the “contract nr” in a single file under “Download Zone/sales”. This exercise tries to find all the pieces and to collect them together to form the original file “sales.txt”. In “Download Zone/sales” we find files like “sales_<column name>.txt” containing the “contract nr” and the “<column name>” columns of the sale records. There are 6 files for 6 columns: “card”, “amount”, “quantity”, “card”, “date”, “product”.

Solution to Exercise 4

Our solution to this exercise, builds a data table with a “Table Creator” node which includes only the column names in a column named “type”. Then a “TableRow To Variable Loop Start” starts a loop that:

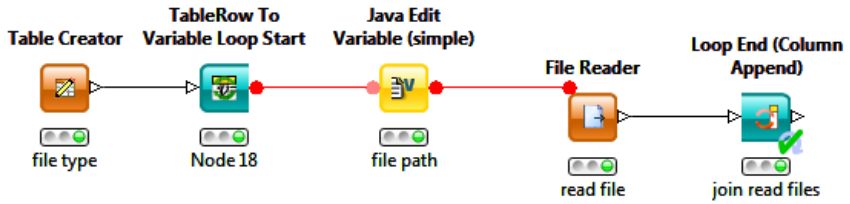
- loops over the list of column names in “type”,
- builds the file path with a “Java Edit” node with code: `return "file:\\"+$${Sfile-path}$$+"sales_"+$${Stype}$$+".csv";`
- passes the file path as a workflow variable to a “File Reader” node,
- reads the file via the “File Reader” node,
- collects the final results with a “Loop End (Column Append)” node.

If the “contract nr” values are read as RowIDs for each file, then the “Loop End (Column Append)” node joins all these columns providing the flag for the ”Loop has same rowIDs in each iteration” is enabled in its configuration window.

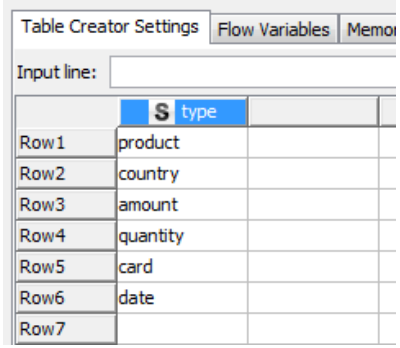
The ”File Reader” node needs to use the workflow variable called “filename” to read the file and the workflow variable named “type”, introduced by the “TableRow To Variable Loop Start”, in order to name the column read after the “contract nr” column. To set the column name with a workflow variable in a “File Reader” node, you need to:

- go to the “File Reader” node’s configuration window,
- select the “Workflow Variables” tab,
- expand the “Column Properties” node and then “0”
- assign the value to the “Column Name” field via the workflow variable.

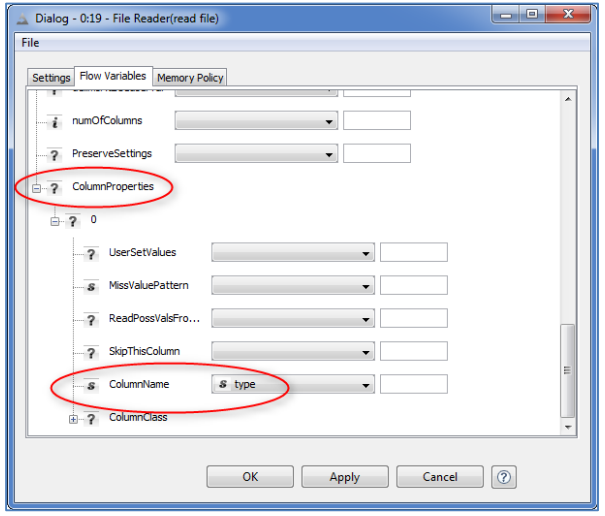
7.43. Exercise 4: The workflow



7.44. The Configuration Window of the "Table Creator" node



7.45. The “ColumnName” setting in the "Workflow Variables" Tab of the configuration window of the "File Reader" node

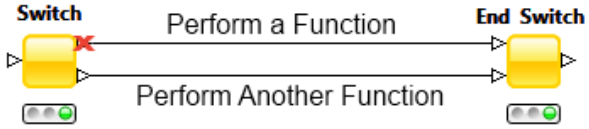


Chapter 8. Switches

8.1. Introduction to Switches

KNIME workflows consist of a sequence of nodes connected together. The sequence can be linear, where one node is connected to the next, or it can branch off to multiple parallel routes. Sometimes in some situations it might be preferable to execute only some of the parallel branches of the workflow and not the others. This is where the KNIME node group, “Switches”, becomes useful. A “Switch” node determines the flow of data via one or more workflow branches. Data flows into a “Switch” node and is then directed down one or more specific routes where a number of specific operations are performed, while all other routes remain inactive. All routes originating from a “Switch” node are finally collected by an “End Switch” node. The KNIME switch concept is illustrated in figure 8.1, where a switch with 2 alternative parallel routes has been implemented.

8.1. Generic illustration of data flow control via Switches



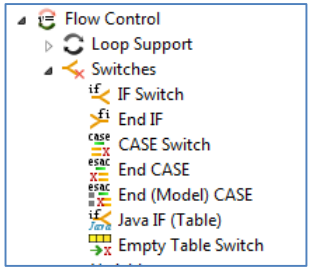
Depending on the type of “Switch” node, the data can flow out through one or more ports. In figure 8.1, for example, the “Switch” node has two output ports, of which only one is active at a time. In this particular configuration, the data has been directed to flow out from the second output port of the “Switch” node only. The top output port, in fact, is blocked, as depicted by the red cross. The data then flows through a number of nodes implementing “Another Function” till the “End Switch” node, thus completing the switch process.

Which output port of the “Switch” node is active and therefore which route the data takes can be controlled in the configuration window of each “Switch” node either manually or through flow variables.

All “Switch” and “End Switch” nodes are located in the “Flow Control” -> “Switches” category in the “Node Repository” panel. There are two main types of “Switch” nodes: the “IF Switch” node and the “CASE Switch” node. These “Switch” nodes each have an “End Switch” partner: respectively the “End IF” node and the “End CASE” node. “End Switch” nodes are depicted by the switch type written backwards on the node icon. Thus “if” and “fi” form one pair and “case” and “esac” the other (Fig. 8.2). The “CASE Switch” node has an additional END partner, the “End (Model) CASE” node, to handle model data types.

In the “Flow Control” -> “Switches” category we can also find a “Java IF (Table)” node. The “Java IF (Table)” node works similarly to an “IF Switch” node, but allows for the flow of data to be controlled by a Java method. Finally, the last node in this category is the “Empty Table Replacer” node. The output of an “Empty Table Replacer” node becomes inactive (i.e. connected nodes are on an inactive branch) if the input table is an empty data table. This avoids the execution of subsequent nodes on tables with no actual content and possibly the workflow failure.

8.2. The nodes contained in the “Switches” category



8.2. The “IF Switch”- “END IF” switch block

There are two “IF Switch” nodes: the “IF Switch” node and the “Java IF (Table)” node. The “IF Switch” node is a simple switch to change the data flow in one direction or another. The direction can be controlled manually or by means of a workflow variable. The “Java IF (Table)” node also controls the data flow but by means of a Java method. Both these nodes use the same “End IF” node to terminate the switch data flow control.

Let’s start with the simplest of the two switch nodes: the “IF Switch” node. In the Download Zone two workflows are available that demonstrate how to set the “IF Switch” node manually or automatically: the “Manual IF Switch” workflow and the “Automatic IF Switch” workflow.

The “Manual IF Switch” workflow reads the “cars-85.csv” file from the Download Zone, bins the data set using the fuel consumption in mpg for either city driving or highway usage, and calculates the statistics of “curb weight”, “engine size”, and “horse power” over the defined bins. To alternatively bin the data set on different data column values, we need two branches inside the workflow: one to bin the data set on the “city mpg” data column and the other one to bin the data set on the “highway mpg” data column.

In order to produce two parallel alternative branches in the workflow, the data flow is controlled by an “IF Switch” node. The “IF Switch” node implements two branches: the top branch is supposed to bin the “city mpg” data column, while the bottom branch is supposed to bin the “highway mpg” data column. The “IF Switch” node is configured to always enable the top branch. If the data is supposed to flow through the bottom branch, a manual change has to be made to the settings in its configuration window.

In both branches, an “Auto-Binner” node is used to bin the data set according to the desired data column based on sample quartiles. The “Auto-Binner” node returns the input data set with an additional column containing the numbered bins. This column has the header name of either “city mpg [Binned]” if the top branch is taken or “highway mpg [Binned]” if the bottom branch is selected.

The idea is to append a “GroupBy” node at the end of the switch block to calculate the statistics of “curb weight”, “engine size”, and “horse power” over the defined bins of the data table produced by the active workflow branch. Since it would be cumbersome to manually change the configuration settings of the “GroupBy” node every time the active output port of the “IF Switch” node is changed, we renamed the binned column in both branches to carry the same column header, that is “Binned.MPG”. The switch block is then closed by an “END IF” node and subsequently a “GroupBy” node calculates the required statistics over the group column “Binned.MPG”.

IF Switch

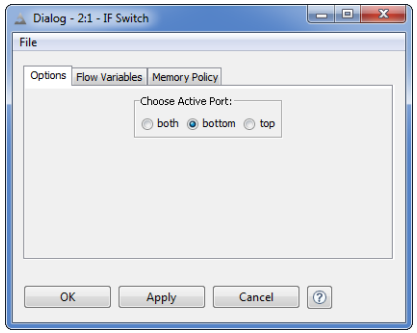
The “IF Switch” node allows data to be branched off to either one or both of the two available output ports. The direction of the data flow can be controlled by manually changing the node’s configuration settings or automatically by means of a workflow variable value.

In the configuration window:

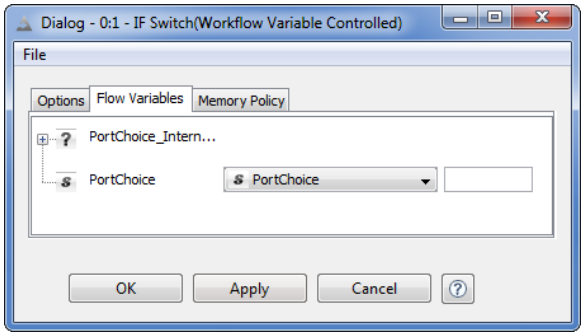
- The “Options” tab contains the options for the manual setting of the active output port;
- The “Flow Variables” tab overrides the active output port in the “Options” tab with a workflow variable value.

The “Options” tab in the configuration window contains three radio buttons to manually select the path for the data flow. By selecting “top” or “bottom” or “both” the corresponding output port(s) is/are activated. Alternatively, the “Flow Variables” tab allows overriding the configuration parameter “PortChoice”, set in the “Options” tab, with a String-type workflow variable. The allowed workflow variable values are: “top”, “bottom”, and “both”.

8.3. The configuration window of the “IF Switch” node: the “Options” tab



8.4. The configuration window of the “IF Switch” node: the "Flow Variables" tab



END IF

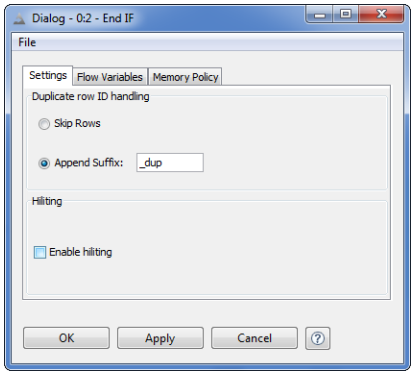
The “End IF” node closes a switch block that was started by either an “IF Switch” node or a “Java IF (Table)” node.

The “END IF” node has two input ports and accepts data from either the top, or bottom, or both input ports. If both input ports receive data from active branches, then the result is the concatenation of the two data tables.

The configuration window of the “END IF” node requires only a duplicate row handling strategy among two possible alternatives: either skip duplicate rows or append a suffix to make their RowID unique.

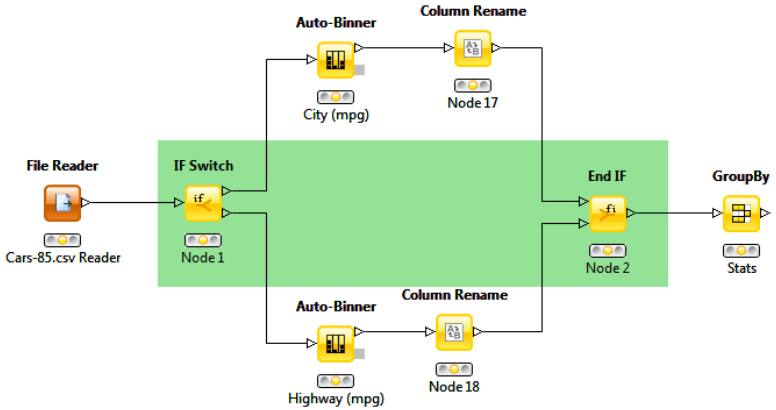
Check the “Enable hiliting” box if you wish to preserve any hiliting after this node.

8.5. The configuration window of the “END IF” node



The “Manual IF Switch” workflow is shown in figure 8.6.

8.6. The “Manual IF Switch” workflow



The “Auto-Binner” node does not belong to the “Switches” category. However, since it has been used to implement the “Manual IF Switch” workflow, we spend a few words here to describe how it works and what it can be used for.

Auto-Binner

The "Auto-Binner" node groups numeric data in intervals - called bins.

Unlike the "Numeric Binner" node, the "Auto-Binner" node:

- First divides the binning data columns into a number of equally spaced bins;
- Then labels the data as belonging to one of those bins.

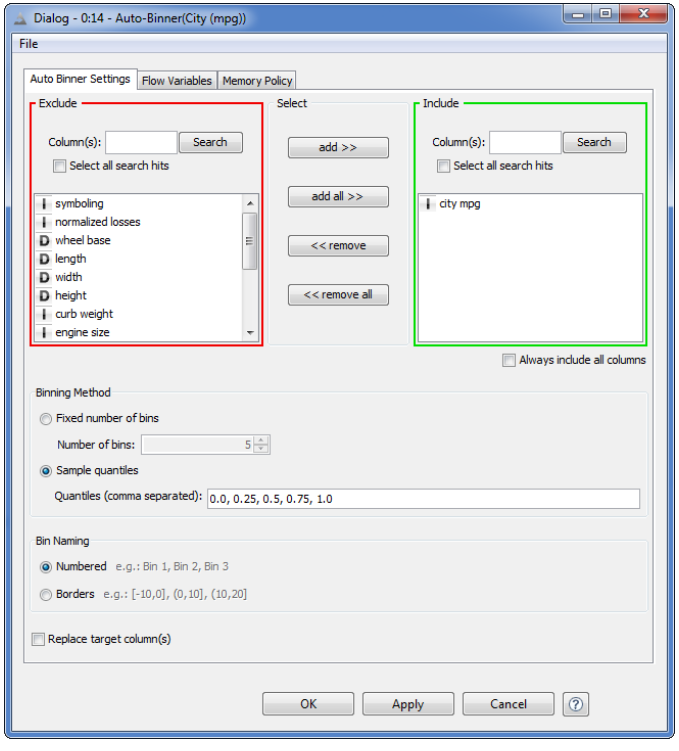
The configuration window offers the choice between two methods to build the bins:

- As a fixed number of equally spaced bins;
- As pre-defined sample quartiles.

The configuration window also offers the choice between two naming options for the bins:

- "Bin1", "Bin2", … , "BinN"
- By using the bin borders, e.g. [0.5674, 0.7899]

8.7. The configuration window of the "Auto-Binner" node



The selection of the binning column is based on an "Exclude/Include" framework:

- The columns to be used for binning are listed in the "Include" frame on the right
- The columns to be excluded from the binning process are listed in the "Exclude" frame on the left

To move single columns from the "Include" frame to the "Exclude" frame and vice versa, use the "add" and "remove" buttons. To move all columns to one frame or the other use the "add all" and "remove all" buttons.

A "Search" box in each frame allows searching for specific columns, in case an excessive number of columns impedes an easy overview of the data.

Pre-defined sample quartiles produce bins corresponding to the given list of probabilities. The smallest element corresponds to a probability of 0 and the largest to a probability of 1. The applied estimation method is Type 7 which is the default method in R, S and Excel.

Note. The “Auto-Binner” node does not allow customized bins. Please use the "Numeric Binner" node if you want to define custom bins.

Alternatively to the manual setting of the active port(s) in the “IF Switch” node configuration window, we could define a workflow variable of type String, named “PortChoice” for example, and we could use that variable to override the manual settings in the “IF Switch” node’s configuration window (Fig. 8.4). The workflow variable could be generated either via the “Workflow Variable Administration” window or via a “Java Edit" node (see chapter 4 for more details).

If we activate the output port of the “IF Switch” node via a variable workflow, we do not need to change the node’s configuration settings to enable the other branch of the switch block. In this case, it is enough to change the value of the workflow variable “PortChoice” for example from “top” to “bottom”.

The new workflow with the automatic activation of the output port(s) of the “IF Switch” node is named “Automatic IF Switch” and is available from the Download Zone. This workflow is identical to the “IF Manual Switch” workflow, except for the workflow variable “PortChoice” and the configuration settings of the “IF Switch” node. In fact, the “IF Switch” node is configured so as to override the active port choice with the value of the workflow variable “PortChoice”.

8.3. The “Java IF (Table)” node

The other IF Switch node is the “Java IF (Table)” node. The “Java IF (Table)” node works like the “IF Switch” node, only that the active port, and therefore the data flow direction, is controlled by means of a piece of Java code. The return value of the Java code determines which port is activated and which branch of the workflow is active.

The “JAVA IF (Table)” node can be found like all other switch nodes under the “Flow Control” -> “Switches” category.

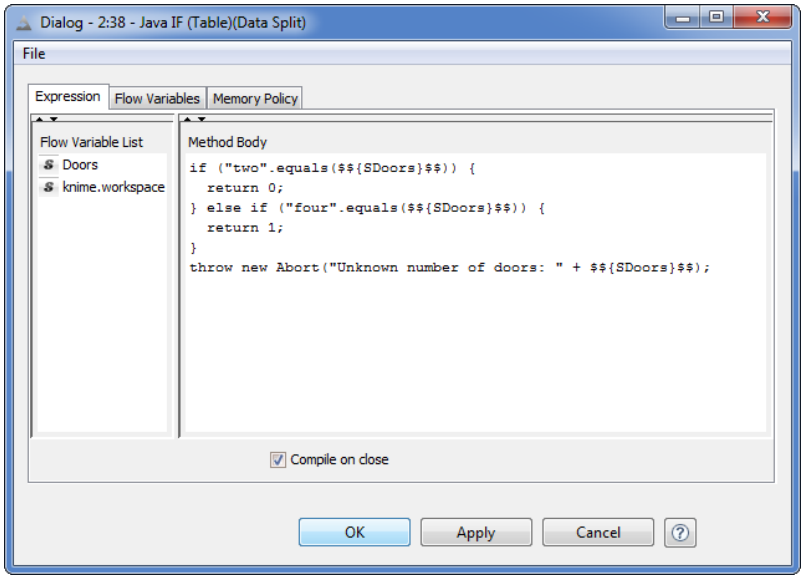
The switch block with a “Java IF (Table)” node is organized as the switch block with an “IF Switch” node. That is, it starts with a “Java IF (Table)” node, branches off into two different data flow paths, and collects the results of the two branches with an “END IF” node.

Java IF (Table)

The "Java IF (Table)" node acts like an "IF Switch" node, in the sense that it has two output ports; but it differs from the "IF Switch" node, in the sense that it can activate only one output port at a time. Unlike the "IF Switch" node, the "Java IF (Table)" node does not allow both output ports to be active at the same time.

The difference between the "IF Switch" node and the "Java IF (Table)" node resides in the management of the output port activation. The "Java IF (Table)" node executes a piece of Java code with return value 0 or 1. Return value 0 activates the top output port, while return value 1 activates the bottom output port.

8.8. The configuration window of the "Java IF (Table)" node



The configuration window of the "Java IF (Table)" node resembles the configuration window of the "Java Edit Variable" node. It contains:

- A "Method Body" panel for the piece of Java code to be executed at execution time
- A "Flow Variable List" panel where all available flow variables are listed

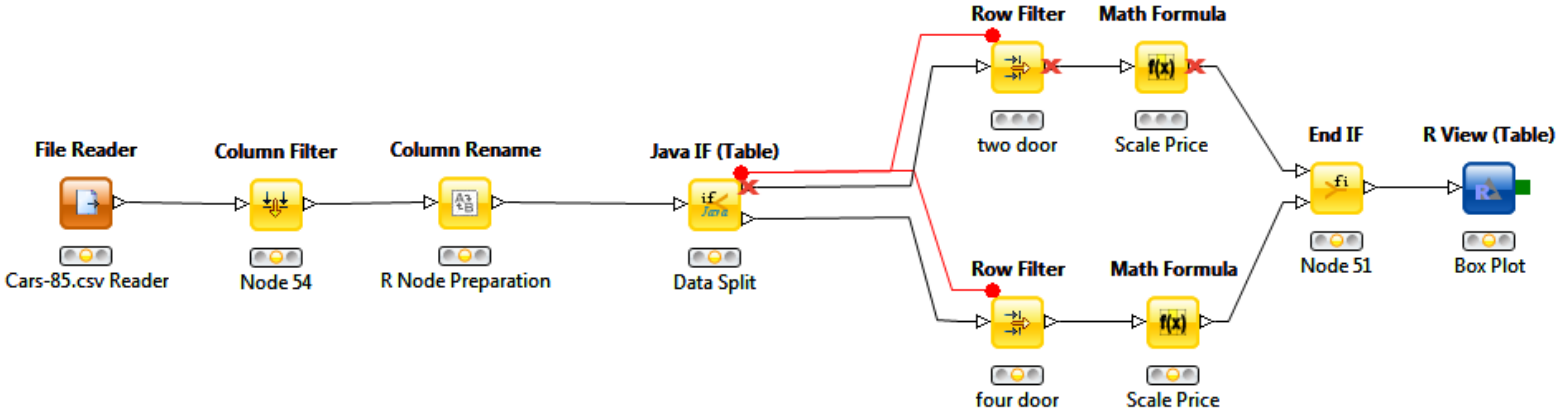
Like the "Java Edit" node, the Java code in the "Java IF (Table)" node operates only on workflow variables. A flow variable can be inserted into the "Method Body" panel by double clicking any of the entries in the "Flow Variable List" panel.

Unlike the "Java Edit Variable" node, the configuration window of the "Java IF (Table)" node does contain any return variable information: neither a return variable type nor a return variable name. This is because the return values can only be 0 or 1. Any other return value will result in an error at execution time.

Note. In "Method Body", an exception handling statement "`throw new Abort();`" is required to handle results that do not produce a 0/1 return value.

To demonstrate the use of the “Java IF (Table)” node, we created a new workflow, named “Java IF & Tables”, in the workflow group called “Chapter 8”. The goal of the workflow was to produce a box plot of the car price, engine size, and horse power for two- and four-door cars from the “cars-85.csv” file. The selection of either two-door cars or four-door cars was implemented by means of a “Java IF (Table)” switch node governed by a workflow variable called “Doors”. This workflow variable can take two string values, either “two” or “four”, and then determines the behaviour of the “Java IF (Table)” switch node. Finally, an “R View (Local)” node produces the box plot on the data coming out of the switch block. Depending on the value of the “Doors” workflow variable and therefore on the active branch in the switch block, a box plot for engine size, horse power, and scaled price of two- or four-door cars is produced.

8.9. The “Java IF & Tables” workflow



8.4. The CASE Switch Block

The “IF Switch” node and the “Java IF (Table)” node offer a choice between two and only two data flow paths. In some situations, we might wish to have more options than just two data flow paths. The “CASE Switch” node offers the choice of three possible mutually exclusive data flow paths. A “CASE Switch” node opens a switch block which is then completed by inserting an “End CASE” node or an “End (Model) CASE” node. The “END CASE” node collects data tables from the active path(s) of the switch block, while the “End (Model) CASE” node collects data models.

The “End (Model) CASE” node only accepts one data model from only one single active branch; in case of multiple active branches the execution fails and a warning is sent to the console. On the other hand, the “End CASE” node supports multiple active branches. If more than one branch is active the final data table results from the concatenation of the multiple input data tables.

CASE Switch

The “Case Switch” node implements a switch block where three optional data flow branches are available. It has three output ports, of which only one can be active at a time. The three output ports are indexed by an integer number, such as 0, 1, and 2.

The configuration window then requires only the index of the active output port.

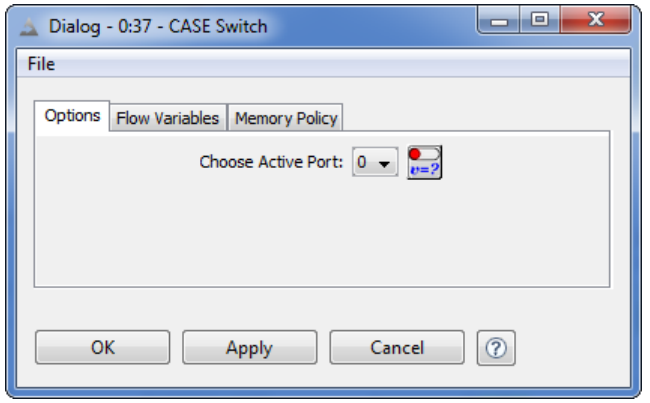
Similar to the “IF Switch” node, the active port index can be provided manually via the “Options” tab or automatically via the “Flow Variables” tab in the configuration window.

The “Options” tab offers a list of the output port indices to choose from.

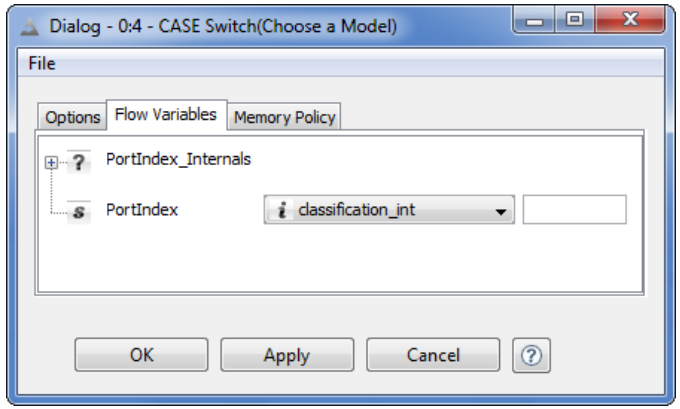
The “Flow Variables” tab overrides the active port index parameter, named “PortIndex”, with the value of the selected workflow variable.

Unlike the configuration window of the “IF Switch” node, the value of a workflow variable can be assigned to the outport index through the workflow variable button located on the right of the “Choose Active Port” list in the “Options” tab. For this node, it is not necessary to go through the “Flow Variables” tab to assign a workflow variable value to a node setting.

8.10. The configuration window of the “CASE Switch” node: “Options” tab



8.11. The configuration window of the “CASE Switch” node: “Flow Variables” tab



The “CASE Switch” node starts a switch block with up to three possible branches. This switch block can then be closed by an “End CASE” node or an “End Model CASE” node.

End CASE

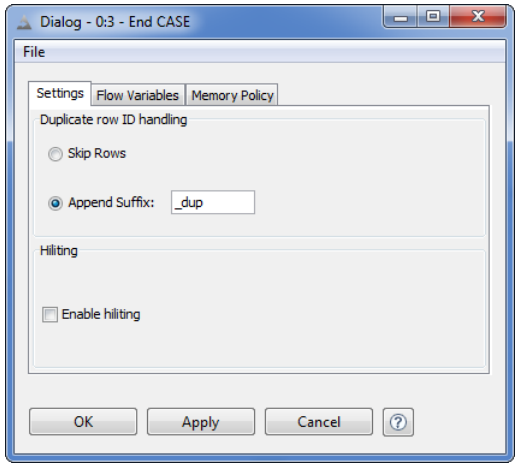
The "End CASE" node closes a switch block with multiple branches and collects the resulting data table(s). It has three input ports, which limits the multiple possibly active branches to three. If more than one branch is active at the same time, the "End CASE" node concatenates the resulting data tables together.

This is therefore a very straightforward node. The only problem being the possible existence of rows with duplicate RowIDs.

The configuration window therefore only requires a strategy to handle rows with the same RowIDs. Two options are offered:

1. Skip rows with duplicate IDs
2. Make those RowIDs unique by appending a suffix to the duplicate values.

8.12. The configuration window of the "End CASE" node



End Model CASE

The "End Model CASE" node also closes a switch block started by a "CASE Switch" node. Unlike the "End CASE" node, the "End Model CASE" node collects a data model rather than data tables.

It has three input ports, but only one at a time can be active. Thus the problem of unique RowIDs from data table concatenation is not an issue here and the node's configuration window does not require any setting of parameters.

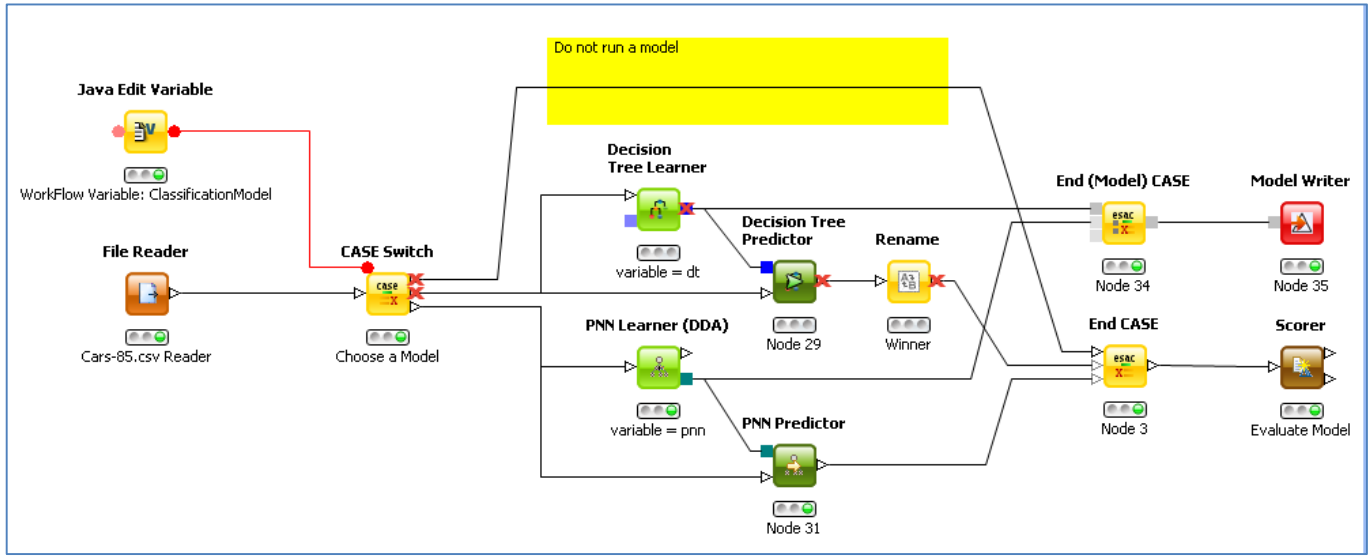
In the Download Zone there is an example of a CASE switch block in the "CASE Switch" workflow in the "Chapter 8" workflow group. This workflow was again implemented to work on the "cars-85.csv" data. In particular, it was implemented to build a few data classification models on the input data set: a Decision Tree, a Probabilistic Neural Network (PNN), and no model at all. We then needed three separate workflow branches: one to implement the decision tree, one to implement the PNN, and one to simply transfer the original data to the output. An "IF Switch" node here was not enough, since three branches were needed. We were forced to use a "CASE Switch" node. To toggle between the models, we created a new workflow variable, called

“ClassificationModel”. This workflow variable was used to indicate the type of analysis to do and could take three values only: “none” for no model, “dt” for decision tree, and “pnn” for probabilistic neural network.

However, the “CASE Switch” node cannot understand string values, like “dt” or “pnn”. It only takes integer values in order to define which output port is active. We needed to convert the string value of the “Classification Model” workflow variable to an integer value. We used a “Java Edit Variable” node, to transform “none” into 0, “dt” into 1, and “pnn” into 2 and we assigned the final value to a new workflow variable “classification_int”. The “CASE Switch” node was then governed by the workflow variable named “classification_int” and generated by the “Java Edit Variable” node on the basis of the value in the “ClassificationModel” original workflow variable.

Next, we implemented a decision tree, a pnn, and no model respectively on each one of the three branches coming out of the “CASE Switch” node. Each model produces two kinds of results: the model itself and the predicted data if a predictor node is applied. We then used an “End Model CASE” node to collect, on one side, the generated models and an “End CASE” node to collect the predicted data on the other side. The models were then saved to a file and evaluated by a “Scorer” node applied on the predicted data.

8.13. The “CASE Switch” workflow



8.5. Transforming an Empty Data Table Result into an Inactive Branch

In the Download Zone, there is another workflow that implements a CASE switch block: the “Empty Table Replacer” workflow. This workflow reads the “cars-85.csv” data, keeps only the cars of a particular “make”, as defined in a workflow variable named “CarMake”. Then it follows three different branches depending on the type of wheel drive of the car: “4wd”, “fwd”, and “rwd”. Each branch keeps only the cars with this particular wheel drive and sends the resulting data table to an “End CASE” node. Finally a “GroupBy” node counts the number of cars by fuel-type for that “make” with that type of wheel drive (Fig. 8.15).

In the “Empty Table Replacer” workflow, the “CASE Switch” node is controlled using a workflow variable, named “wheeldrive”. The workflow variable “wheeldrive” can only take “rwd”, “fwd”, and “4wd” string values. These values are then transformed into output port indices for the “CASE Switch” node by a “Java Edit Variable” node. The data selection in the switch branches is implemented by “Row Filter” nodes on the wheel drive type. The “Row Filter” node in the branch activated by workflow variable “wheeldrive” = “4wd” keeps all rows with “4wd” in the “drive wheels” data column, and so on.

What happens, though, if one of the “Row Filter” nodes returns an empty data table? The final “GroupBy” node will give a few warning messages about an empty table being created. For example, if you assign the value “rwd” to the workflow variable “wheeldrive”, the “Row Filter” node in the corresponding active branch of the CASE switch block produces an empty data table. Indeed, sometimes the execution of a workflow branch may result in an empty data table being created. Empty tables can still be processed by nodes further down in the workflow but they could result in a lot of warning messages, wasted time, and maybe even data inconsistencies. To prevent this from happening, the “Empty Table Switch” node enables two different output ports in case that an empty data table or a value-filled data table is created.

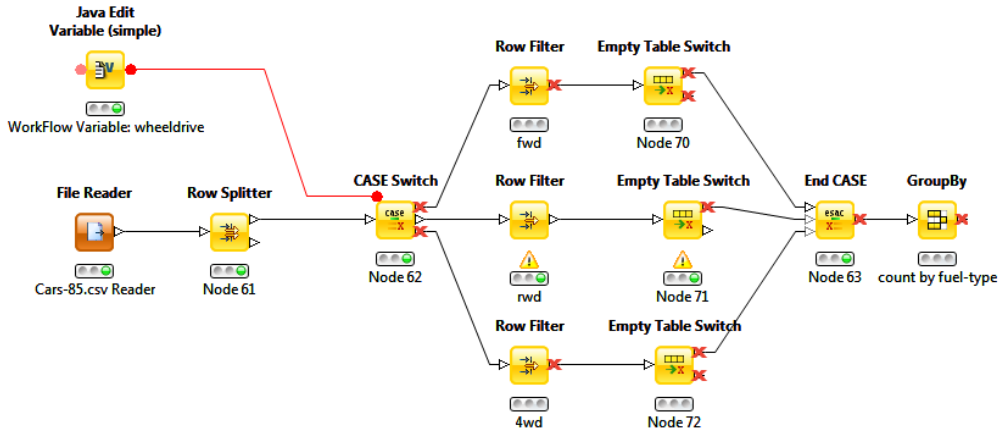
Empty Table Switch

The “Empty Table Switch” node activates the bottom output port if an empty table is the result of the node; otherwise it activates the top output port. This avoids the execution of subsequent nodes on tables with no actual content.

The configuration window of the “Empty Table Switch” node does not requires any setting.

In each branch of the CASE switch block we then introduced an “Empty Table Switch” node, to block further processing of the workflow in case the result of the branch is an empty data table. If we now run the workflow with the workflow variable “wheeldrive” = “rwd”, the “Row Filter” node of the second branch produces an empty data table, the output of the “End Case" node collects no results, and the “GroupBY” node becomes inactive and is not executed.

8.14. The “Empty Table Replacer” workflow



8.6. Exercises

Exercise 1

A situation where the “IF Switch” node is useful is when you need to perform one operation on a particular row entry and an alternate operation on the other rows. Let us examine this using a trivial scenario. Create a table containing the numbers 1-10. Then, by using an “IF Switch” node, create a workflow to generate an additional integer column produced by multiplying the even numbers by a factor of 3 and the odd numbers by a factor of 10.

Solution to Exercise 1

The first step is to create a table containing the numbers 1-10, which is achieved using the “Table Creator” node and by labeling the integer column as “Number”.

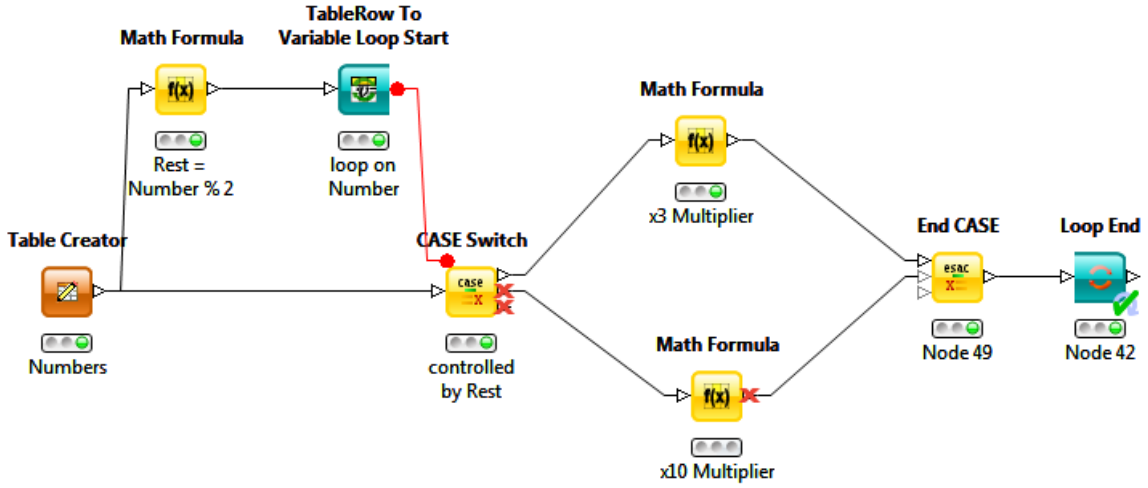
The aim here is to perform an operation on alternate lines. For each number its rest from a division by 2 is calculated by means of a “Math Formula” node. The value of this rest will define the multiplying factor, 3 or 10, for each number. An easy solution to solve this problem would then be to apply a “Rule Engine” node and transform the rest value into the corresponding multiplying factor. A “Math Formula” node then would finish the trick, multiplying each number for the newly created multiplying factor.

However, we want to use a switch block here. As this can be viewed as a row operation, we looped over the rows, performed the right multiplication, and then concatenated the final results. We then started a loop with the “TableRow to Variable Loop Start” node, turning each number and each rest value into workflow variables.

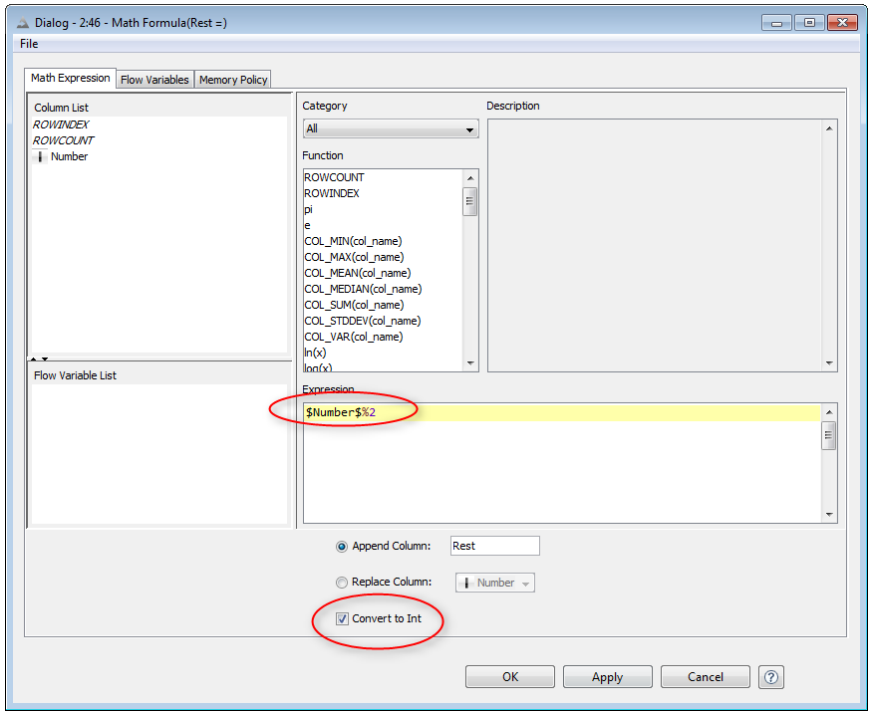
A “CASE Switch” node implemented the switch for different multipliers. Numbers are then multiplied by 3 on one of the two branches and by 10 on the other. The CASE Switch active port is controlled by the rest value: 0 enables the upper port, 1 enables the second output port. We selected a “CASE Switch” node, because we could control its output port with integer numbers. An “IF Switch” node would need Strings as “top”, “bottom”, and “both” to control its output ports.

In order to transform the rest values from Double to Integer, we enabled the “Convert to Int” flag at the bottom of the “Math Formula” node configuration window. The final workflow is shown in figure 8.15.

8.15. Exercise 1: The workflow



8.16. Exercise 1: Configuration window of the first “Math Formula” node



Exercise 2

In this exercise we show how to implement a control of the workflow processes based on time and date.

Using the “cars-85.csv” file, create a workflow that manipulates data differently depending upon the day of the week:

- On Wednesdays, write out an R box plot for engine size, horse power, and scaled price and save it as a PNG image;
- Do not perform any operations on the weekend
- For the remaining days keep only the “make”, “aspiration”, “stroke”, and “compression ratio” data columns.

Use a “CASE Switch” node to implement the branches for different daily data manipulation.

Solution to Exercise 2

First of all, we need to extract the current date and time information. We did that with a "Java "Edit Variable" node with the following Java code:

```
Calendar rightNow = Calendar.getInstance();
String now = new String(rightNow.toString());
return now;
```

This method returns a long string containing details of the exact time and date. We are interested in the DAY_OF_WEEK information, expressed as an integer 1-7, 1 being Sunday. Using a number of data manipulations, we can extract this information and generate a variable as a result, named "values_Arr[1]". All those data manipulations have been compressed into a meta-node for convenience, named "DAY_OF_THE_WEEK". The most complicated part of the workflow is now complete.

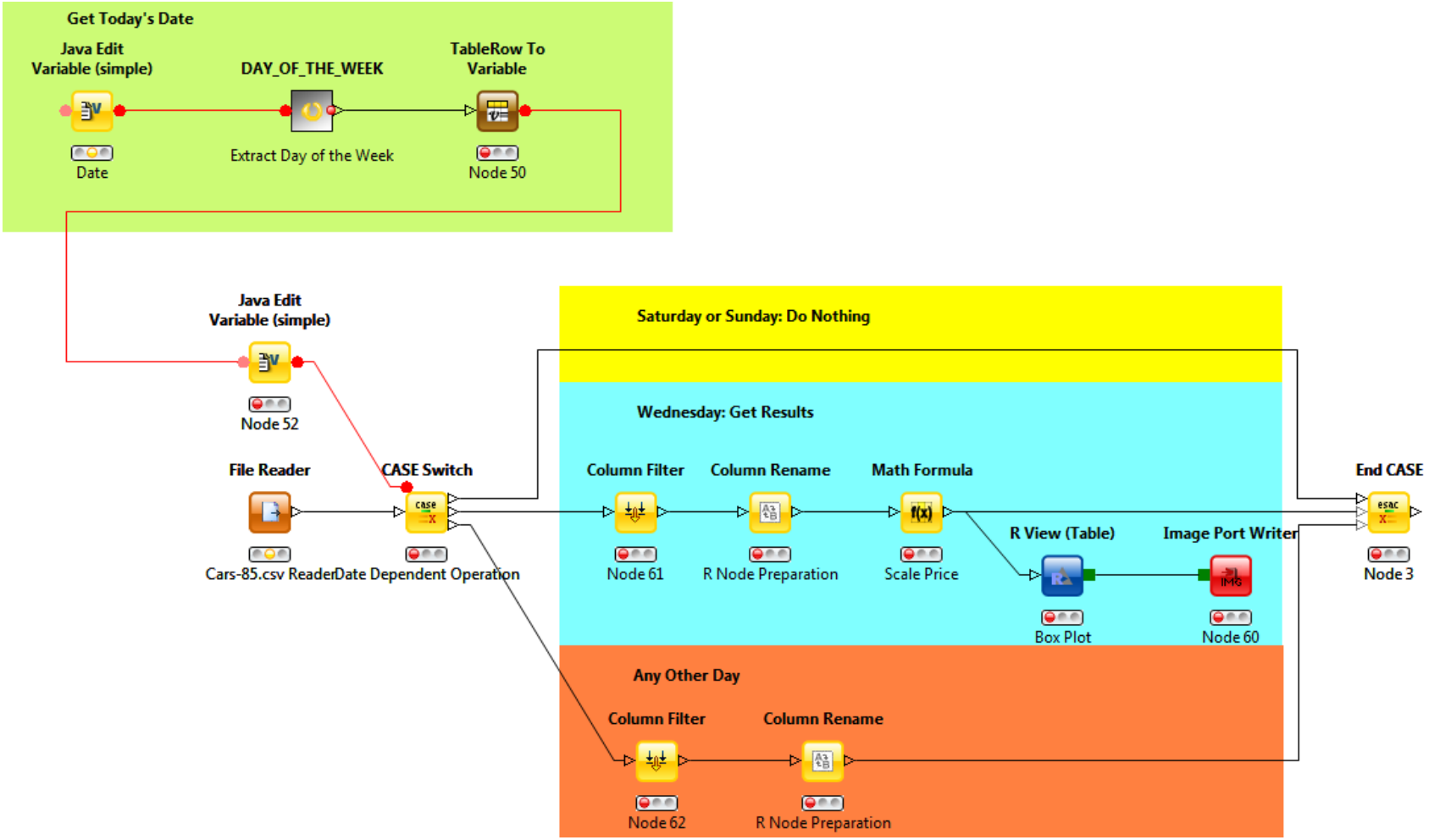
A subsequent "Java Edit Variable" node implements the following Java code:

```
if ("1".equals($${Svalues_Arr[1]}$$) || "7".equals($${Svalues_Arr[1]}$$)) {
  return 0;
} else if ("4".equals($${Svalues_Arr[1]}$$)) {
  return 1;
} else {
  return 2;
}
```

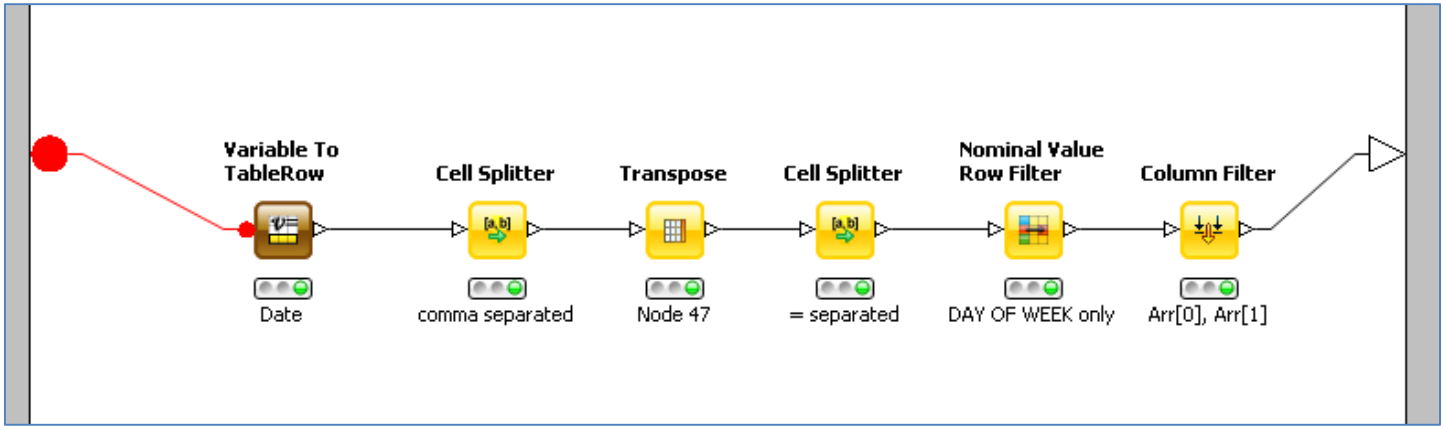
This Java code analyzes the value on "values_Arr[1]" and returns 0 if it is a weekend, 1 if it is a Wednesday, and 2 otherwise. These numbers are then used to control the active port of a subsequent "CASE Switch" node. Three branches are then generated from the "CASE Switch" node and the final results are collected by an "End CASE" node.

The solution workflow for this exercise is shown in figure 8.17.

8.17. Exercise 2: The workflow



8.18. Exercise 2: The “DAY_OF_WEEK” meta-node



Chapter 9. Advanced Reporting

9.1. Introduction

So far, we have seen many advanced KNIME features: from the workflow variables to the inclusion of R scripts in a workflow; from the implementation of loops to the calling of web services, and so on. At last, the exploration of the advanced features of the KNIME Reporting tool is covered in this chapter.

The KNIME Reporting Tool was covered in detail in chapter 6 of the first KNIME user guide ("The KNIME Beginner´s Luck" [1]). However, a few aspects of the KNIME Reporting tool were not included, since previous knowledge of a number of advanced KNIME features was required. For example, it is necessary to know something about workflow variables before using them to generate report parameters. In this chapter, we want to describe the advanced features of the KNIME Reporting tool that were ignored by the previous book. This chapter covers:

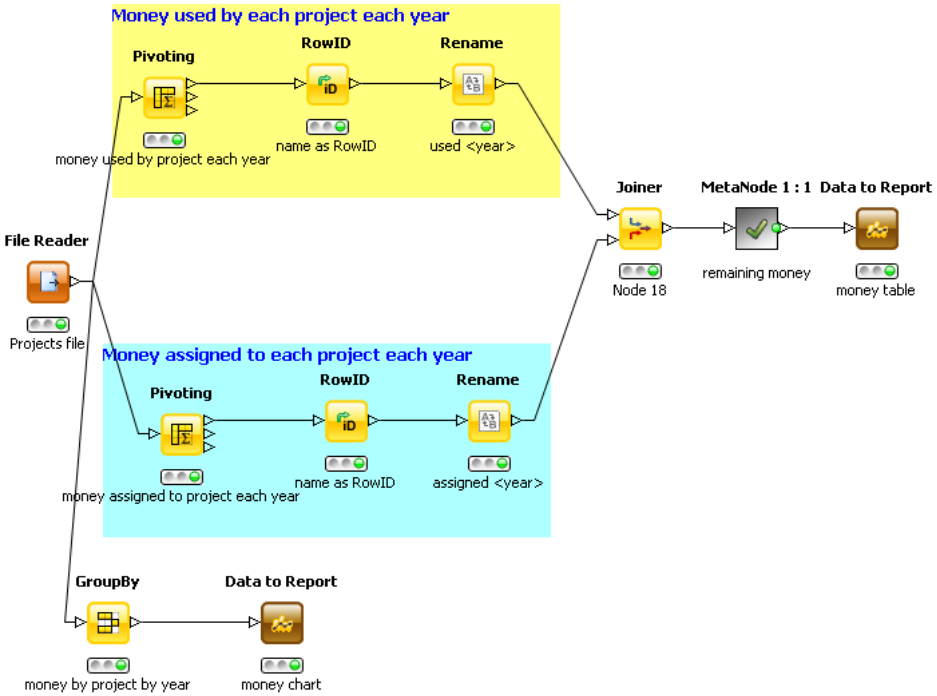
- how report parameters can be created by means of workflow variables;
- how to display parameter values in a report;
- the introduction of dynamic content into a report, by means of dedicated BIRT and Javascript functions;
- how to import images, that have been generated in the underlying workflow, into a report.

In order to give continuity to the example material for the first and this second KNIME user guide, the workflow example for this chapter is "Projects" and it is taken directly from the workflow examples developed for chapter 6 in [1].

The "Projects" workflow works on the "Projects.txt" file from the Download Zone. The "Projects.txt" file contains a list of project names with the corresponding amount of money assigned and used for each quarter of each year between 2007 and 2009. The "Projects" workflow builds a pivot table with the project names and the sum of the money assigned, the sum of the money used, and the sum of the money remaining for each project and for each year between 2007 and 2009. The "Projects" workflow also builds a data table with the total amount of used and assigned money for each year and each project. The resulting data is then used to fill a few tables and two bar charts in the associated "Projects" report.

The final version of the "Projects" workflow is displayed in figure 9.1 and the resulting pivot data table, from the "money table" node, in figure 9.2.

9.1. The "Projects" workflow



9.2. The data table, from the “money table” node, of the "Projects" workflow

Table View - 0:15 - Data to Report(money table)

File Hilite Navigation View Output

Row ID	S name	D used 2007	D used 2008	D used 2009	D assigned 2007	D assigned 2008	D assigned 2009	D remain 2009	D remain 2008	D remain 2007
Row0	Blue	1,360	1,277	1,565	1,360	1,277	1,565	0	0	0
Row1	Gobi	1,203	1,424	1,740	1,203	1,424	1,740	0	0	0
Row10	White	860	1,087	1,420	860	1,087	1,420	0	0	0
Row2	Kalahari	630	800	1,192	630	800	1,192	0	0	0
Row3	Kara Kum	800	888	1,516	800	888	1,516	0	0	0
Row4	La Guajira	1,020	1,404	1,496	1,020	1,404	1,496	0	0	0
Row5	Mojave	1,800	1,819	1,860	1,800	1,819	1,860	0	0	0
Row6	Patagonia	864	2,098	1,359	864	2,098	1,359	0	0	0
Row7	Sahara	806	1,457	1,495	806	1,457	1,495	0	0	0
Row8	Sechura	3,200	2,966	3,940	3,200	2,966	3,940	0	0	0
Row9	Tanami	453	0	453	453	0	453	0	0	0

9.2. Report Parameters from Workflow Variables

The report of the “Projects” workflow contains three tables, each one with the data of all 11 projects, and two bar charts also with the data of all 11 projects. Due to the high number of projects, the tables and the bar charts in the report can be quite hard to read.

In this chapter we reduce the number of projects to display in the tables and in the bar charts, for example, to a maximum of three projects per report. With only the data of three projects, the tables and the bar charts become much easier to read. We will even create three report parameters containing the names of the three projects to be displayed in the report.

We created three workflow variables, named respectively “project_1”, “project_2”, and “project_3”, for the “Projects” workflow. These workflow variables were created to hold the names of the three projects to be displayed in the report. How to create and use workflow variables in a workflow is described earlier on in this book in chapter 4. We set “Sahara”, “Blue”, and “White” as the three default values for the three workflow variables.

We then changed the “Projects” workflow accordingly. After the “File Reader” node named “Projects file”, we placed three “Row Filter” nodes. The first “Row Filter” node was built to select the data for the project contained in the “project_1” workflow variable; the second “Row Filter” node to select the data for the project contained in the “project_2” workflow variable: and the third “Row Filter” node to select the data for the project contained in the “project_3” workflow variable. Each “Row Filter” node was configured to use the pattern matching feature and to match the values in the data column “names” with the value in one of the three workflow variables “project_1” or “project_2”, or “project_3”. The data tables at the output ports of the three “Row Filter” nodes were then concatenated together by a “Concatenate (Optional in)” node.

The data table at the output port of the “Concatenate (Optional in)” node was subsequently fed into the two blocks that calculate the money used/assigned by each project each year, respectively characterized by a yellow and light blue annotation. For space reasons, we collapsed these two blocks into two meta-nodes. In order to do that, we selected the three nodes of the group, right-click any of them, and selected the “Collapse into a meta-node” option. The new meta-node was automatically created and filled in with the selected nodes. As a new name we assigned the corresponding annotation content to each one of the meta-nodes.

As in the previous version of the “Projects” workflow, the data tables resulting from the two meta-nodes were joined together, the amount of the remaining money was calculated, and the final data table was exported for reporting with a “Data to Report” node named “money table”. During execution, this “Data to Report” node exports its input data table into the report data set “money table” to be used to build the tables in the report.

Concatenate (Optional in)

The “Concatenate (Optional in)” node is a variation of the “Concatenate” node.

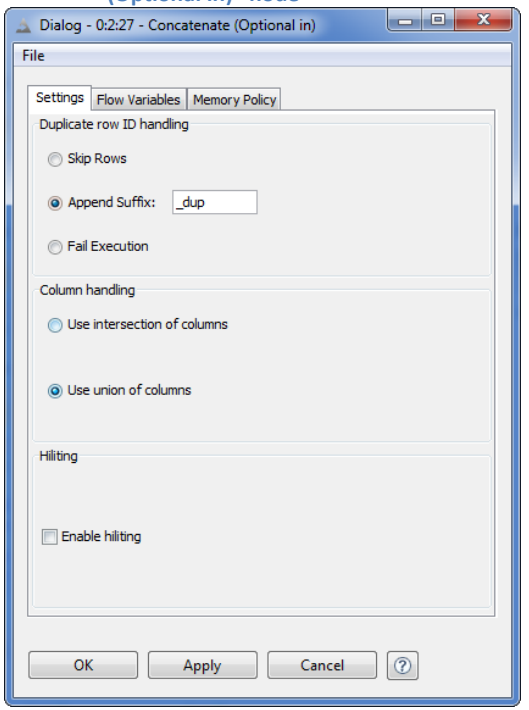
Like the “Concatenate” node, it concatenates data tables and is located in the `“Data Manipulation” -> “Row” -> “Transform”` category. The difference to the “Concatenate” node lies in the number of input data tables that can be concatenated: 2 for the “Concatenate” node and up to 4 for the “Concatenate (Optional in)” node.

The node has “optional” input ports (displayed as gray outlined triangles). Optional ports don’t have to be connected for the node to be executed.

The configuration window requires:

- A method to handle rows with duplicate IDs
- A choice between using the intersection or the union of the input data columns
- A flag to enable hiliting

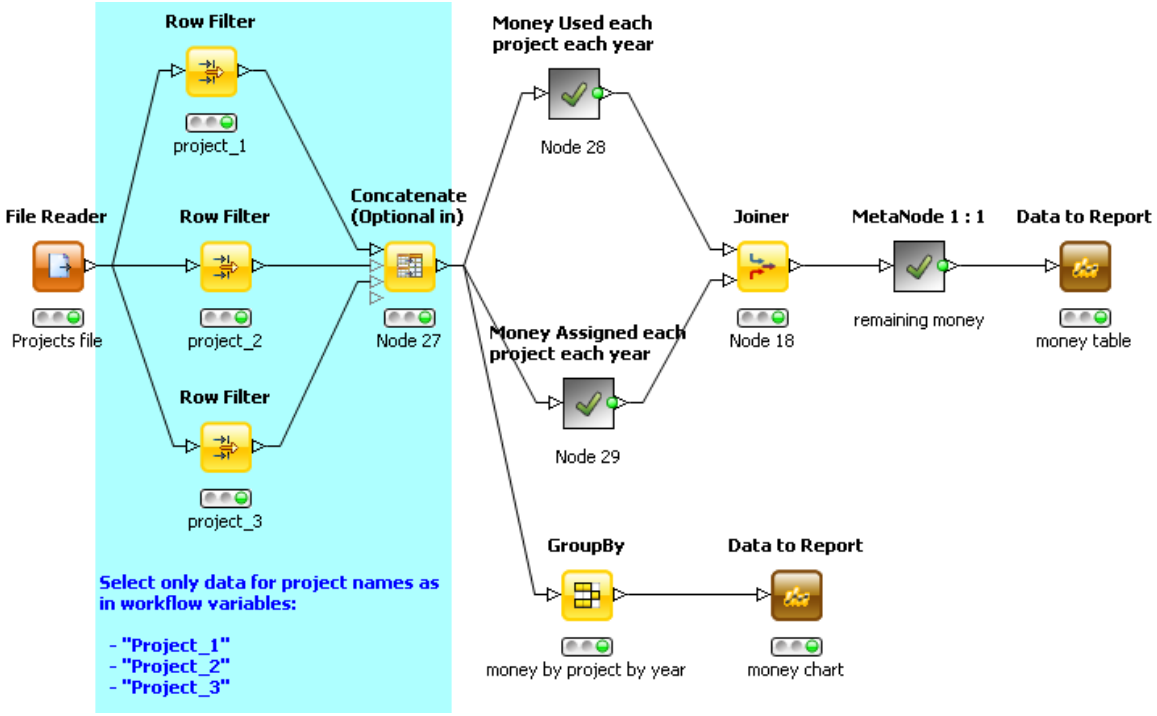
9.3. Configuration window of the “Concatenate (Optional in)” node



Finally, a “GroupBy” node named “money by project by year” was placed after the “Concatenate (Optional in)” node to calculate the total amount of money assigned (used by) each project each year. Its output data table was marked for reporting with a “Data to Report” node named “money chart”. This “Data to Report” node generates the data set called “money chart” to be used to build the bar charts in the report.

The final workflow was then renamed “New Projects”. The “New Projects” workflow is shown in figure 9.4.

9.4. The "New Projects" workflow



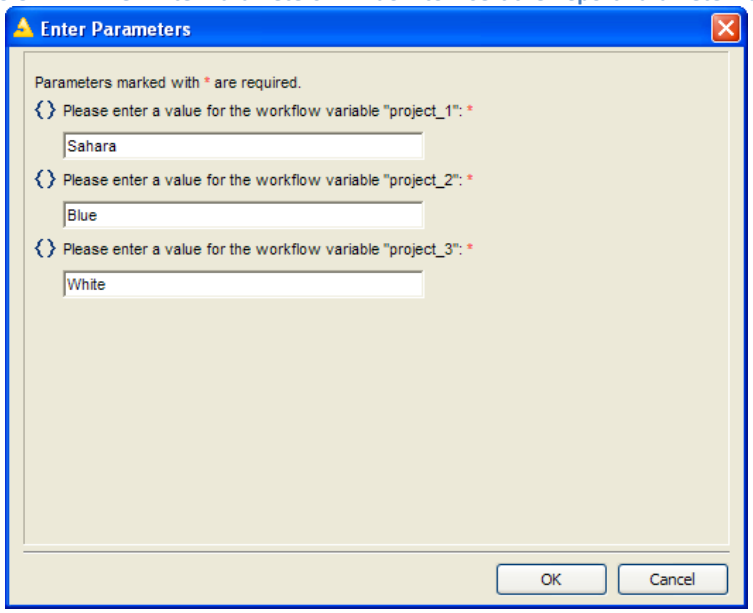
The “New Projects” workflow programmatically selects some data from the original data set, on the basis of the values in the newly created workflow variables, “project_1”, “project_2”, and “project_3”. We would like to transfer this parameterization to the report, so that we could also run the report only for a maximum of three selected projects at a time.

We now execute the “New Projects” workflow and we switch to the reporting environment. We do not change the report layout; we just run a “Preview”. A window appears asking for the report parameter values (Fig. 9.5). Indeed the report now has three parameters: one named “project_1”, one named “project_2”, and one named “project_3”.

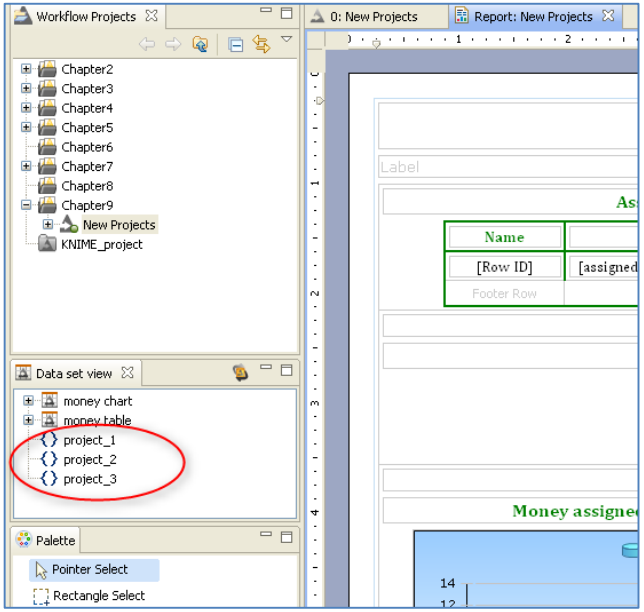
Note. When switching from the KNIME workbench to the reporting environment, the software looks for all the “Data to Report” (or “Image to Report”) nodes, transfers their output data tables into report data sets named as the original “Data to Report” node, looks for all workflow variables, and transforms the workflow variables into report parameters with the same name and the same default values.

The workflow variables found in the underlying workflow are displayed as report parameters in the panel named “Data Set View” in the middle on the left (Fig. 9.6).

9.5. The “Enter Parameters” window to insert the Report Parameter values



9.6. The "Data Set View" panel in the reporting perspective



In the “Enter Parameters” window, let’s accept the three default values for the three report parameters. The result is a report with tables and bar charts built only on the data of the three selected projects, which is exactly the kind of report we wanted to build.

If we want the report to display the data for only two projects, we give the third workflow variable/report parameter the name of a non-existing project. The third “Row Filter” node in the workflow then produces an empty data table, which does not show up after concatenation, and the report only shows the data for the 2 existing projects.

9.3. Customize the “Enter Parameters” window

The default “Enter Parameters” window is very anonymous and not very informative. It is possible to customize the “Enter Parameters” window:

- To show a nicer layout,

- To be more informative,
- To offer only a number of pre-defined answers, so that user's errors, like typos, are less likely to happen.

In order to change the request display in the "Enter Parameters" window for a specific parameter, we double-click the name of this report parameter in the "Data Set View" panel located on the left.

A new window, named "Edit Parameter" opens (Fig. 9.7). This window allows customizing the request display for a report parameter in the "Enter Parameters" window (Fig. 9.5).

In the "Edit Parameter" window, in fact, we can customize:

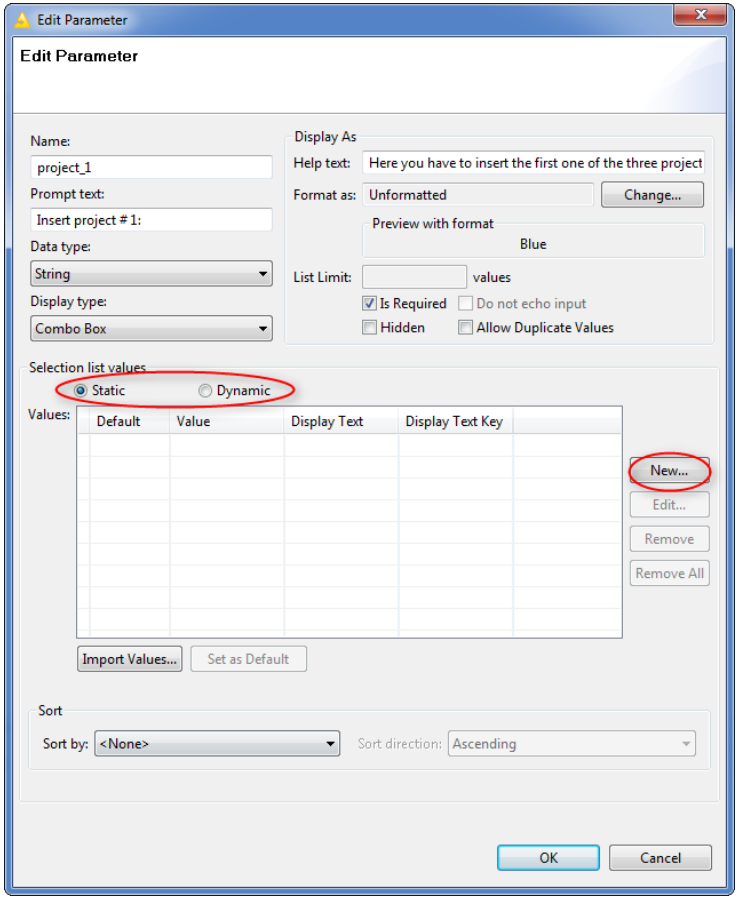
- The "Prompt Text" from the original "Please enter a value …";
- The "Display Type" from the default text box;
- The "Help Text" that pops up when hovering over the parameter box in the "Enter Parameters" window

The "Display Type" in particular offers a few options on how to enter the report parameter: via a "Text Box", via a "List Box", via a "Combo Box", and via a "Radio Button".

The "Text Box" is a box in which to type free text and therefore it is the most error prone option.

The "List Box" and the "Combo Box" both offer a list of pre-defined values to the user. List Boxes and Combo Boxes are safer against typos than a simple Text Box. The difference between the "List Box" and the "Combo Box" lies in the fact that the first one allows multiple values, while the second one does not.

9.7. The "Edit Parameter" window



Finally, the "Radio Button" only allows a mutually exclusive choice between two possible values. Very minimal error is possible here on the user's side.

In the "Edit Parameter" window, the parameters "Name" and "Data Type" can also be changed, generating in this way a new report parameter.

For our three workflow variables / report parameters, we provided:

- “Insert the first project:” as prompt text,
- A “Combo Box” as the modality to enter the parameter value,
- A very long explanation of what the parameter does for the “Help Text”.

At this point, we needed to fill in the “Combo Box” with a list of pre-typed values. There are two options to pass a list of values into a “Combo Box”: by using a static list or a dynamic list of values.

A static list of values is a list of values that was created once and can only be edited manually.

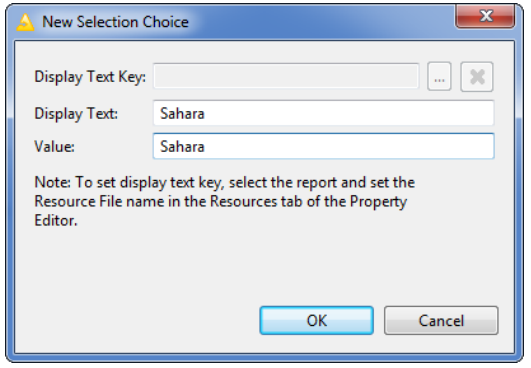
A dynamic list of values is created on the fly from the data sets produced at execution time for this report.

Static List of Values

To fill in the list with static values, in the “Edit Parameter” window (Fig. 9.7):

- Select the “Static” radio button in the “Selection List Values” frame;
- Click the “New” button on the right;
- The “New Selection Choice” window opens;
- In the “New Selection Choice” window:
    - Enter the list value and its display text;
    - Click “OK”;
- The pair (value, display text) appears in the list of values in the “Enter Parameter” window;
- Perform the same operations for all remaining values to be in the list

9.8. The "New Selection Choice" window



Note. The list value and the display text might not be the same. For example, the report parameter might refer to month abbreviations. Since month abbreviations are locale dependent, you can use the month’s short form as the “Value” list item in your preferred locale and the full month text as the “Display Text” list item, e.g. “Value” = “Mar” or “Mrz” depending on your locale and “Display Text” = “March”. “Value” is not visible in the “Enter Parameters” window.

Dynamic List of Values

To enter all values in the list manually could be time consuming and even impossible if the list of values is really long. An alternative is to fill the list dynamically from the data of one of the report data sets.

In order to fill the list of values into the “ComboBox” dynamically, that is from one of the report data sets, go to the “Edit Parameter” window (Fig. 9.9):

- Select the “Dynamic” radio button in the “Selection List Values” frame;
- Select the report data set with the desired value list;
- Select the column of the data set that contains the values for the list;
- Select the column of the data set that contains the display texts for the list;
- Click “OK”.

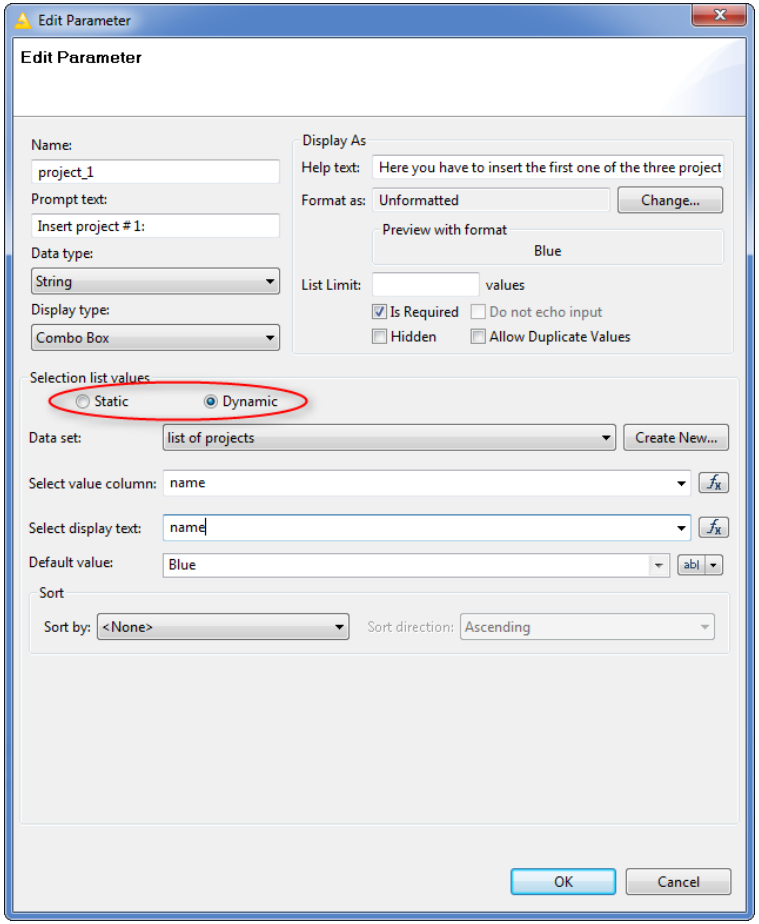
The “Enter Parameters” window can be customized for each report parameters.

For the “New Projects” report, we decided to use a “ComboBox” to select the name of the project for each one of the three report parameters. We decided to fill in the “ComboBox” dynamically. In fact, since we are not sure that the list of projects will not change in the future, we preferred the dynamic solution rather than the static one.

We introduced a new “GroupBy” node, named “list of projects”, in the “New Projects” workflow. The “GroupBy” node was supposed to collect all project names and make them available in the report for the “Combo Box” list of values. The “GroupBy” node was then connected to a “Data to Report” node named “list of projects” as well.

Going back to the report, a new “list of projects” data set in the “Data Views” panel was shown. Then for each one of the three report parameters, we selected:

9.9. The "Edit Parameter" window when the option dynamic is selected to fill the list of values for the "ComboBox"

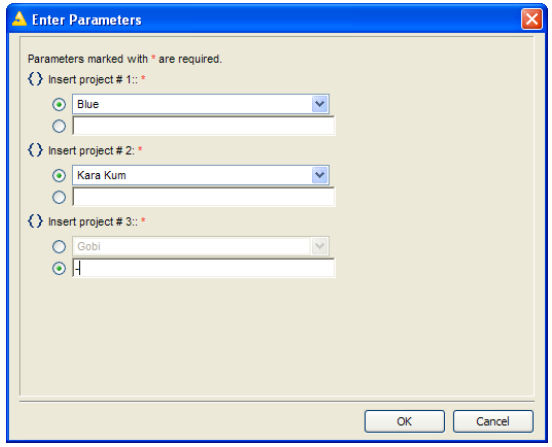


- The “ComboBox” as “Display type”;
- The “Dynamic” option to fill in the “ComboBox” with a list of values;
- “list of projects” as “data set”;
- “name” as the column containing the list of values;
- “name” again as the column containing the “Display text” strings;

The “Enter Parameters” window now looks like the one in figure 9.10. For each parameter there is a “ComboBox” with a list of pre-defined values and its “Prompt Text" says “Insert project # <n>”.

Note. Below the “ComboBox” there is still a “TextBox” where free text can be enetered. This can be used to type in a text that is not offered by the “Combo Box” list. For example, we could use it to type the name of a non-existing project to limit the report to only two or even one project.

9.10. The new "Enter Parameters" window



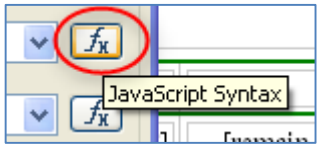
9.4. The Expression Builder

In the “Edit Parameters” window (Fig. 9.9), at the right of the “select value column” and of the “Select display text” boxes, there is a button with an “fx” symbol, as depicted in figure 9.11. This button can often be found at the side of text/combo boxes in the KNIME Reporting Tool to access the “Expression Builder” window (Fig. 9.12).

The “Expression Builder” window helps building an expression around some column values or just by itself. Expressions can be either mathematical or string-based.

For example, in the “Enter Parameters” window we might want to display the following text for each workflow variable value: “Project: <project-name>”. This expression can be built by means of the “Expression Builder” window.

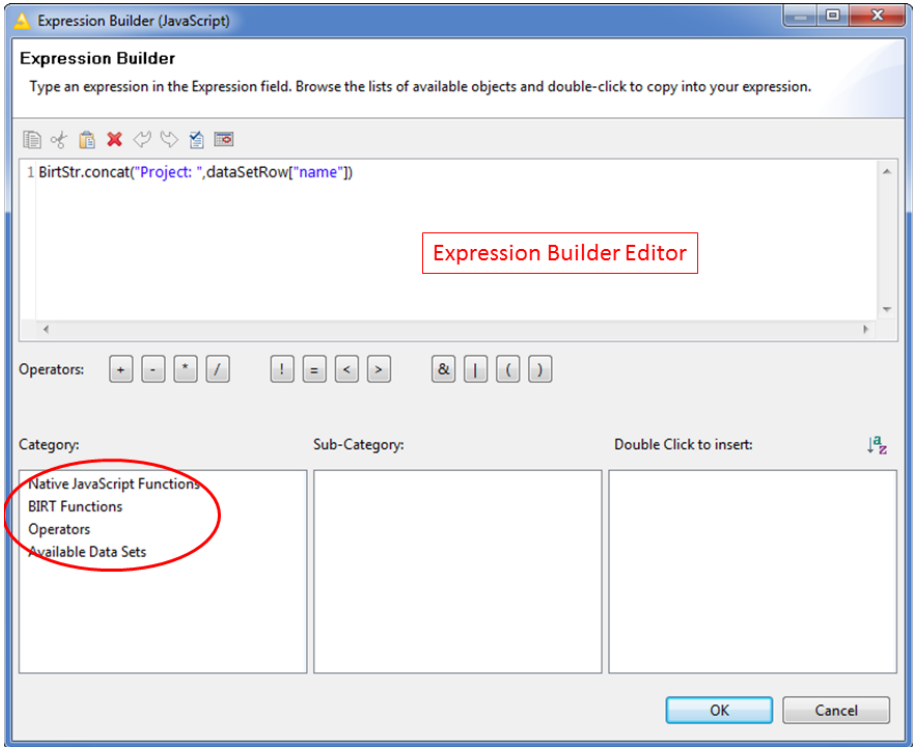
9.11. Button to the "Expression Builder" window



In the “Expression Builder” window (Fig. 9.12) we find:

- In the top panel, the “Expression Builder” editor, where expression parts can be typed in manually;
- In the middle panel, a list of operators that can be applied to mathematical expressions;
- In the bottom panel, the lists of functions, operators, and column values that can be inserted into the Expression Builder Editor.

9.12. The "Expression Builder" window



Let's have a closer look at the bottom panel. On the left, are the function categories we can choose items from.

Starting from the bottom, the "**Available Data Sets**" category offers all available data sets and their columns for this particular "Expression Builder" instance. The data sets are listed in the center sub-panel and the corresponding data columns in the right sub-panel.

Note. This Expression Builder originates from the "Edit Parameter" window (Fig. 9.9) where the "list of projects" data set was selected. Therefore the only available data set here is "list of projects". In other instances of the Expression Builder originating from different items of the KNIME Reporting tool, more report data sets might be available.

The "**Operators**" category offers a number of mathematical/logical operators. An extract with the most frequently used operators is shown in the middle panel.

The "**BIRT Functions**" category includes a number of BIRT specifically designed functions in the fields of finance, date/time manipulation, duration, mathematics, string manipulation, and string or number comparison.

The "**Native JavaScript Functions**" category includes a number of JavaScript functions. These turn out to be particularly useful when the report is created using the HTML format.

Double-clicking an item in the right sub-panel, like a column name, a BIRT function, an operator, or a Java Script function, automatically inserts the column into the Expression Builder editor above.

For example, in order to get the display text as: "Project: <project-name>":

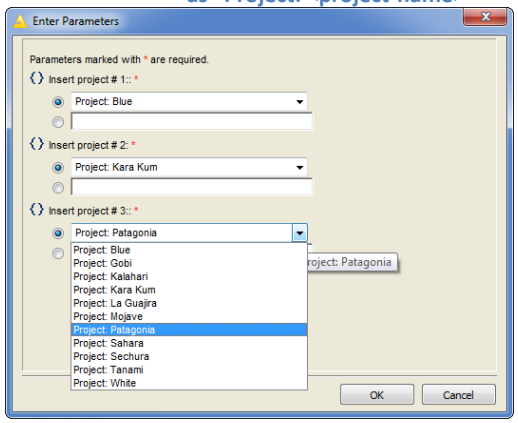
- we double-clicked one of the report parameters in the "Data Set" panel;
- we then filled the "Edit Parameter" window as in figure 9.9;
- we clicked the "fx" button at the right of the "Select display text" box;
- In the "Expression Builder" window:
    - We opened the "BIRT Functions" category in the left panel followed by the "BirtStr" sub-category in the center panel;
    - We double-clicked the `concat()` function in the right panel;
    - The text `BirtStr.concat()` appeared in the expression builder editor;
    - The cursor was positioned in between the parenthesis;
    - We typed the text `<"Project: ",>` (the quotation marks are needed for Java-style strings);
    - We opened the "Available Data Sets" category in the left panel and then the "list of projects" sub-category in the center panel;

- We double-clicked the “name” column;
- The text `BirtStr.concat("Project: ",dataSetRow["name"])` appeared in the expression builder editor (Fig. 9.12);
- We clicked “OK”

- We clicked “OK” back in the “Edit Parameter” window

Note. In the Expression Builder editor text strings must be typed in quotation marks (Java-style). Text items in a `concat()` function have to be separated by a comma.

9.13. The combo-boxes in the "Enter Parameters" window display the project names as "Project: <project-name>"



After repeating this operation for the three report parameters, “project_1”, “project_2”, and “project_3”, the new “Enter Parameters” window resembled figure 9.13.

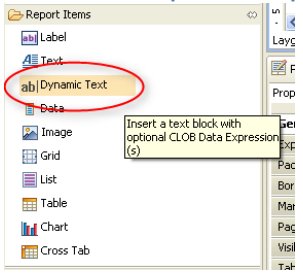
In this section we have seen how the “Enter Parameters” window can be customized, so that less errors and typos occur on the user’s side. More complex customizations are possible by exploiting the BIRT dedicated functions and the native JavaScript functions more in depth.

9.5. Dynamic Text

It is good practice to display the report parameter values at the beginning of the report. In this way, the reader knows immediately under which conditions the data was collected and processed. The report parameter values, however, are different at each run. Therefore we need a dynamic item that updates its content from the report parameter values at each run.

A dynamic report item is the “Dynamic Text”, which can be found in the “Report Items” list in the bottom left panel. The “Dynamic Text” item displays a small text, built with the “Expression Builder”. The “Expression

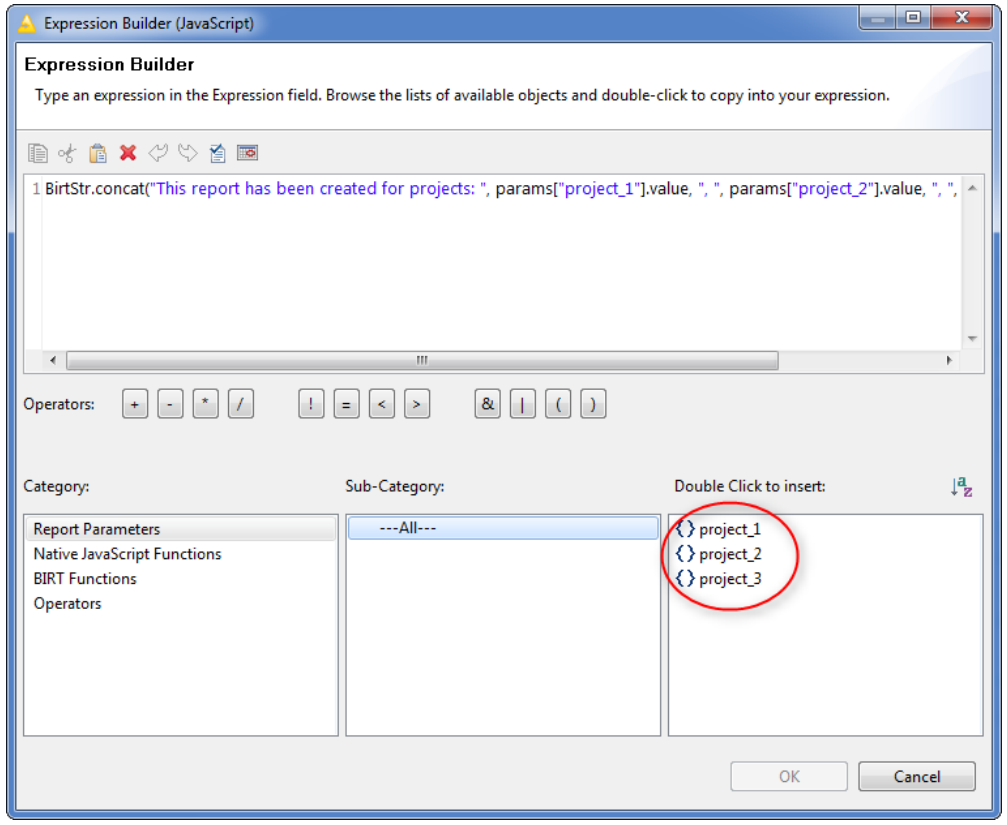
9.14. The "Dynamic Text" item in the "Report Items" panel



Builder" window generated by a "Dynamic Text" offers the list of the available report parameters in addition to the lists of operators and BIRT and Java Script functions, as seen in the previous section.

Let's drag and drop a "Dynamic Text" item from the "Report Items" list to the top of the report layout, to directly under the title. The "Expression Builder" opens. The "Expression Builder" editor is still empty. In the bottom panel, on the left, we can now see 4 categories (Fig. 9.15): "Report Parameters", "Native JavaScript Functions", "BIRT Functions", and "Operators". The "Report Parameters" category is new, while the "Data Sets" category has disappeared in comparison with the "Expression Builder" generated by the "Select Display Text" expression builder button (Fig. 9.9). The "Report Parameters" category contains the list of all available report parameters. As for the other categories, double-clicking any of the report parameters automatically places it in the "Expression Builder" editor.

9.15. The "Expression Builder" window for the "Dynamic Text" report item



We wanted to build a text, like: “This report has been created for projects: <project-name1>, <project-name2>, <project-name3>”.

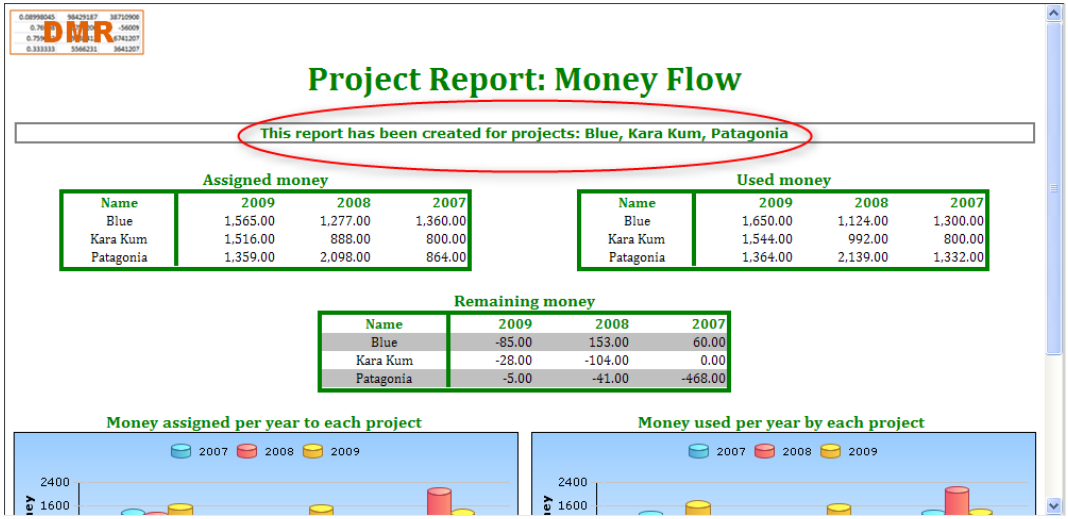
To do that, we used the BIRT function `concat()` once more. In the “BIRT Functions” category on the left, we selected the “`BirtStr`” sub-category in the center panel and then the `concat()` function in the right panel. In the “Expression Builder” editor in the top panel the text `BirtStr.concat()` appeared. Inside the parentheses, we typed the text <“`This report has been created for projects: `”, > with the text in quotation marks and followed by a comma.

Then, we selected the “Report Parameters” category, the “All” sub-category, and double-clicked “project_1”. `params["project_1"].value` appeared in the “Expression Builder” editor. We typed a comma after that and the text <“, “, > in quotation marks and again followed by a comma. The same sequence of operations was repeated for the remaining two workflow variables “project_2” and “project_3”, till the following text was present in the “Expression Builder” editor (Fig. 9.15):

```
BirtStr.concat(   "This report has been created for projects: ",
                   params["project_1"].value, ", ", params["project_2"].value, ", ", params["project_3"].value)
```

We clicked “OK”, the “Expression Builder” window closed, and the “Dynamic Text” item was set in the report with this text. The “Dynamic Text” item was then formatted as centered, with green font color and bold style.

9.16. The "Dynamic Text" item in the Report



9.6. BIRT and JavaScript Functions

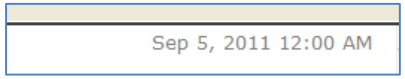
In the “BIRT Functions” and “JavaScript Functions” categories for the “Dynamic Text” item, there are other useful functions, for example the date and time functions and the mathematical functions. Indeed, it is good practice to include the creation date in the report, just to be sure not to read an excessively old report. We could insert the current date as a running title in the report, to the right of the logo.

In the reporting environment, in the “Master Page” tab, in the top header frame, we inserted a grid with two columns and one row. The left cell already included the logo [1]. In the right cell we wanted to insert a “Dynamic Text” item displaying the current date at report creation.

Let’s drag and drop a “Dynamic Text” item from the “Report Items” list panel on the bottom left to the right grid cell in the top header frame of the “Master Page” tab. The “Expression Builder” window opens. In order to display the current date we have many options.

We can choose for example a straightforward “BIRT Functions” -> “BirtDateTime” -> “Today()” function. “Today()” returns a timestamp date which is midnight of the current date. The function in the “Expression Builder” window then runs as `BirtDateTime.today()` and the current date in the running title of the report looks like in figure 9.17.

9.17. Current Date from "Today()" BIRT Function

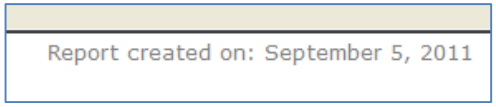


The “Today()” function offers no formatting options. If we want to have the current date in a customized format we must build that ourselves. “BIRT Functions” -> “BirtDateTime” offers a number of functions to extract components from a DateTime object, like `day(DateTime)`, `month(DateTime)`, `year(DateTime)` and so on. We could extract the date components and combine them with a `BirtStr.concat()` function to get the desired date format. After extracting date parts from the result of the `Today()` function and combining them with a `concat()` function, we get, for example, the following formula in the “Expression Builder” window:

```
BirtStr.concat(   "Report created on: ",
                  BirtDateTime.month(BirtDateTime.today(), 2), " ",
                  BirtDateTime.day(BirtDateTime.today()), ", ",
                  BirtDateTime.year(BirtDateTime.today()))
```

and the following current date format in the report running title:

9.18. Customized Current Date from "Today()" BIRT Function



The “BIRT Functions” -> “BirtDateTime” sub-category also offers a number of functions to add and subtract time from a DateTime object. For example, the running title could use the following formula with the `addQuarter()` function:

```
BirtStr.concat("Report valid from: ",
BirtDateTime.today(),
" to: ",
BirtDateTime.addQuarter(BirtDateTime.today(),1)
)
```

which produces a date on the running title as follows.

9.19. Use of the "addQuarter()" function in the Running Title

Note. Notice the change in date locale between figure 9.17 and figure 9.19. The change is due to the introduction of the `concat()` function, which automatically sets the locale for the DateTime values as well.

In this section we have only shown the BIRT functions related to the DateTime object, because they are the most commonly used. However, the "BIRT Functions" category offers many built-in functions for mathematical expressions, finance quantities, string manipulation, etc …

If the report output document is in HTML, we could also take advantage of the built-in JavaScript functions, which are more articulated and varied than the built-in BIRT functions.

9.7. Import Images from the underlying Workflow

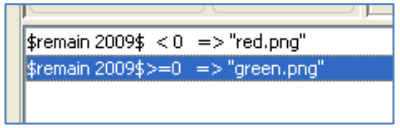
The last advanced reporting feature that we would like to discuss in this chapter is the dynamic image transfer between a workflow and its report. In this chapter, we take the data generated by the "New Projects" workflow and we attach a red or green traffic light to each project, depending on project performance in 2009. That is, if the "remain 2009" is below 0 we depict a red traffic light close to the project name in the report table; if "remain 2009" is above or equal to 0 we append a green traffic light.

To illustrate this concept, we have created a new workflow in the workflow group "Chapter9" and we named it "traffic lights". We have also created two images: one with a red traffic light (file "red.png") and one with a green traffic light (file "green.png"). These images are available in the Download Zone in the "images" folder. We also generated a new file with the data table built in the workflow developed in the previous sections. The file was named "Totals Projects.csv" and can be found in the Download Zone.

The “traffic lights” workflow starts with a “File Reader” node to read the “Totals Projects.csv” file, with the project name as RowIDs. A “Column Filter” node then removes all columns besides the RowIDs with the project names and the “remain 2009” column. The “remain 2009” column contains the difference between the money assigned to each project and the money used by each project in year 2009.

A “Rule Engine” node appends the name of the image file of the red\green traffic light in a new column named “image”, depending on the value of the “remain 2009” column. That is, column “image” contains the name of the image file of the red traffic light (“red.png”), if “remain 2009” is below zero, and the name of the image file of the green traffic light (“green.png”), if “remain 2009” is above or equal to 0.

9.20. Rules implemented by the "Rule Engine" node



The path to the folder containing the image files is implemented as a workflow variable, named “image_path” and set to:

```
file:\C:\data\book2\Download Zone\images
```

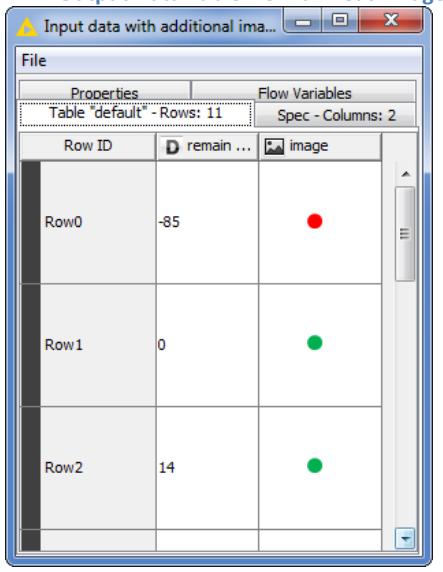
Note. Notice the “file:/” before the real path. This is required later on by the “Read Images” node, to appropriately find and read the image files.

The image file name in the “image” column is then transformed into the full path of the image file by a “Java Snippet (simple)” node. The “Java Snippet (simple)” node combines the workflow variable “image_path” with the values in the “image” column. The new values replace the old values in the “image” column.

A “Read Images” node reads the images pointed by the file paths in the “image” column. For each row, a red or a green traffic light image is read (Fig. 9.21).

Finally, the data table is fed into a “Data to Report” node to be exported into a report data set. The “Data to Report” node was named again “money table”.

9.21. Output Data Table from a "Read Images" Node



Read Images

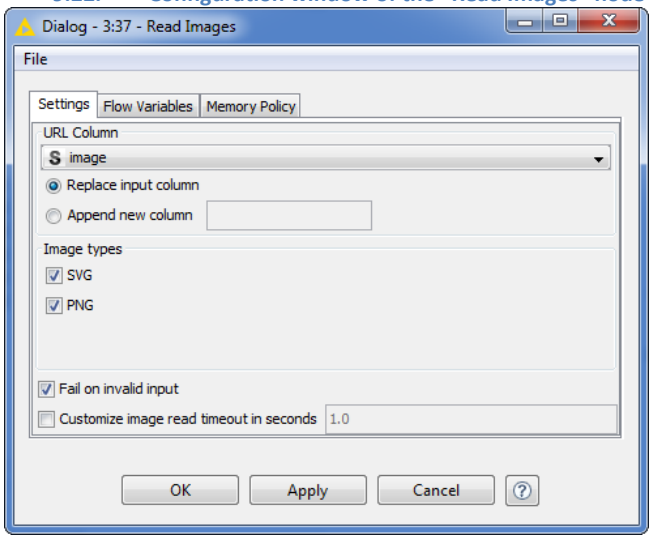
The “Read Images” node reads image files from the URLs stored in a selected column of the input data table.

The “Read Images” node is located in `“IO” -> “Read” -> “Other”.`

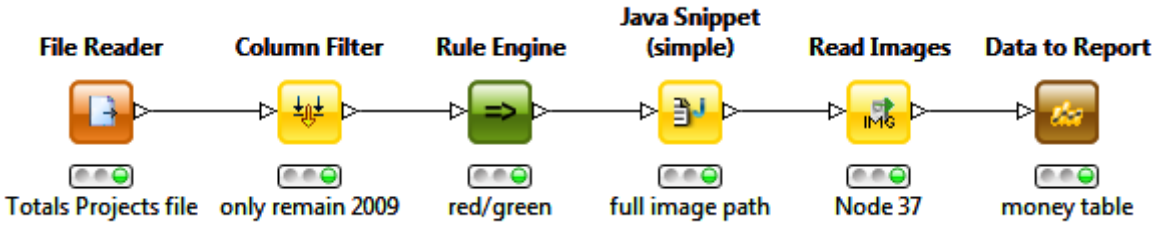
The configuration window requires:

- The column containing the URLs of the PNG image files
- A flag to stop or continue in case of a reading error
- A radio button to create a new column or replace an existing one
- The image format(s)
- A read time out

9.22. Configuration window of the “Read Images” node



9.23. The "traffic lights" workflow



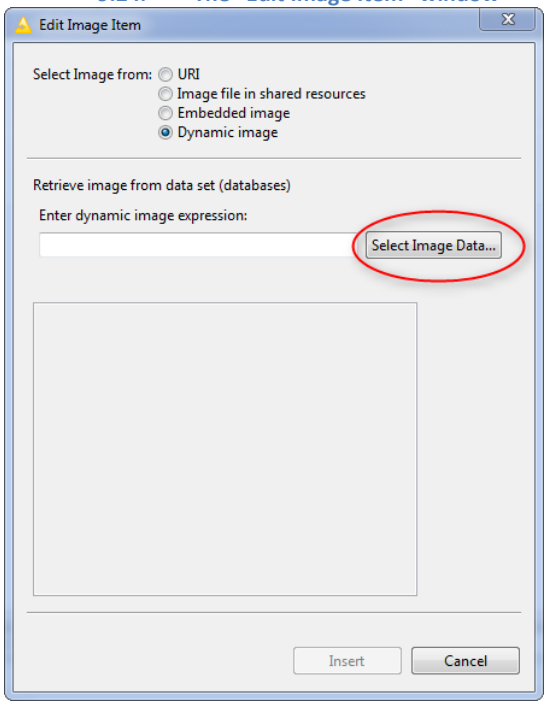
After the “traffic lights” workflow was finished, we switched to the reporting perspective and we started to build a very simple report to show the traffic light images.

In the “Master Page” tab, we set “Orientation” to “Landscape”. In the “Layout” page, we inserted a title (font color=blue, font size=16, font style=cold, alignment=center) with the text “Use of Conditional Images in Reports”. Under the title we placed a small table with three columns only: the image of the traffic light (“image”), the project name (“RowID”), and the remaining money for 2009 (“remain 2009”).

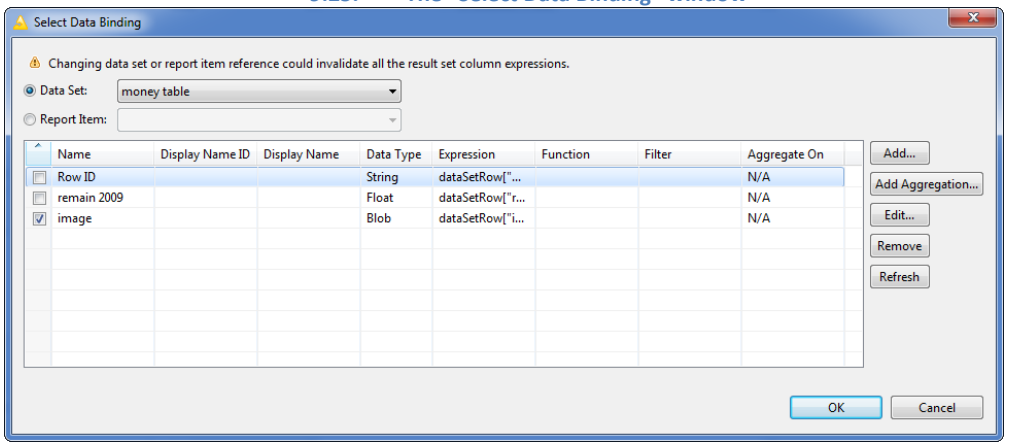
If we just dragged and dropped the “money table” data set from the “Data Set Views” panel into the report editor, the “image” data would have been rendered as a sequence of characters and not as an image. To make the report render the data in the “image” column as an image, we needed to specify that this is an image. In order to import an image value in a table cell of the report:

- Right-click the empty data cell of the table column where the images should go;
- Select “Insert” and then “Image”;
- In the “Edit Image Item” window :
    - Select the radio button, “Dynamic Image”, at the top;
    - Click the “Select Image Data …” button;
    - In the “Select Data Binding” window:
        - Select the column of the data set containing the images (“image” in our example);
        - Click “OK”
- Back in the “Edit Image Item”, click “Insert”

9.24. The "Edit Image Item" window



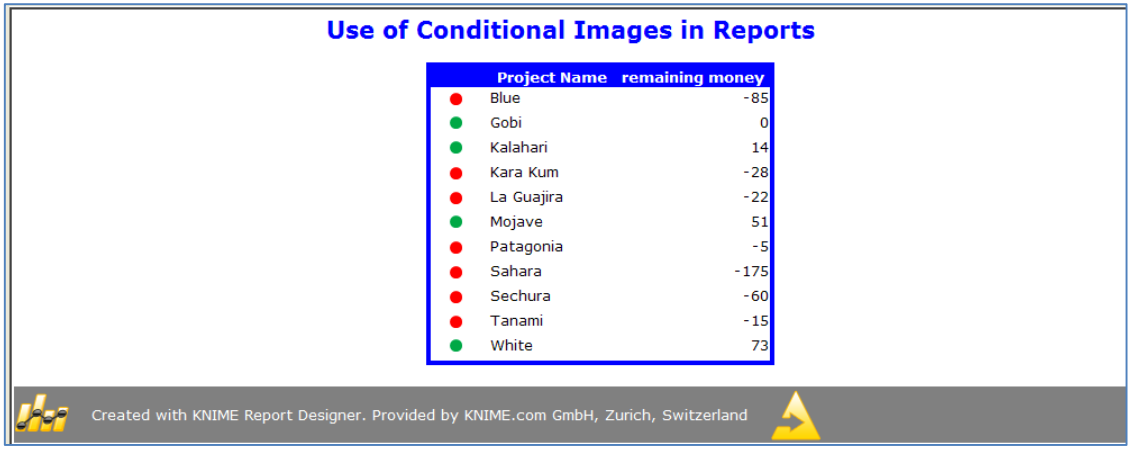
9.25. The "Select Data Binding" window



After inserting an image in your table, a square with a red cross appears in the table cell. This is because the reporting editor cannot visualize images. However, the “Preview” should show them clearly in your report layout. In our example, we have imported red and green traffic lights depending on the “remain 2009” column contents. The table in the report should then show rows with a red traffic light and rows with a green traffic light (Fig. 9.26).

Note. It is necessary to import the “image” column as an image and, in particular, as a dynamic image for the content of the “image” column to be appropriately interpreted.

9.26. The final report of the “traffic lights” workflow



9.8. Exercises

Exercise 1

Using data of the file called “Projects.txt” from the Download Zone build a report as follows:

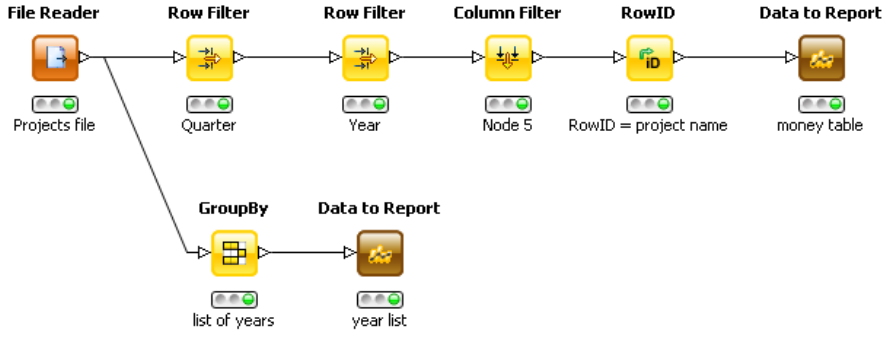
- With a title;
- With a table with “project name”, “used money”, and “assigned money”;
- Limit the data in the table to a given quarter and reference year;
- Display the report parameters used to generate the report;
- Display the current date;
- Customize the “Edit Parameters” window for all report parameters: use a static list of values for some and a dynamic list of values for other parameters.

Solution to Exercise 1

The workflow named “Exercise 1” is shown in figure 9.27. It includes a “File Reader” node to read the file “Projects.txt”, two “Row Filter” nodes to select a given reference year and a given quarter, and a “Column Filter” node to keep only the project name and the two money related columns. The final data table gets exported into a report data set named “money table”.

To make the “Row Filter” nodes work with parametric values for “reference year” and “quarter”, we introduced two workflow variables: “Year” (Integer, default value=2009) and “Quarter” (String, default value = Q1). These two workflow variables generated the two corresponding report parameters. We decided to use a static list of values for the report parameter named “Quarter” and a dynamic list of values for the report parameter named “Year”. The “GroupBy” node collects the list of available years for the dynamic list of values.

9.27. Exercise 1: Workflow



We then built a minimal report with a title and a table generated directly by drag and drop of the “money table” data set. We formatted the title and the table as shown in figure 9.29. The report inherits two report parameters from the underlying workflow: “Quarter” and “Year”.

Under the title we placed a grid with two “Dynamic Text” items: one to display the report parameters and one to display the current date.

The “Dynamic Text” displaying the report parameters uses the following expression in the “Expression Builder” window:

```
BirtStr.concat("Money Summary for Year: ", params["Year"].value, " and Quarter: ", params["Quarter"].value)
```

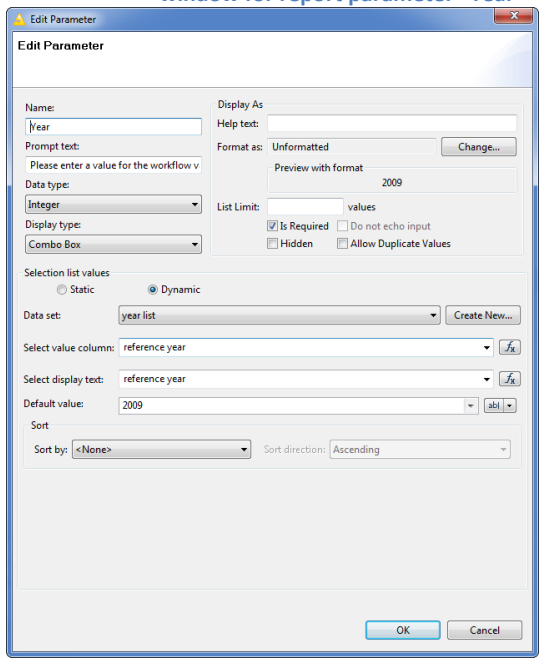
The “Dynamic Text” displaying the current date uses the following expression in the “Expression Builder” window: `BirtDateTime.today()`.

Finally, we customized the “Enter Parameters” window.

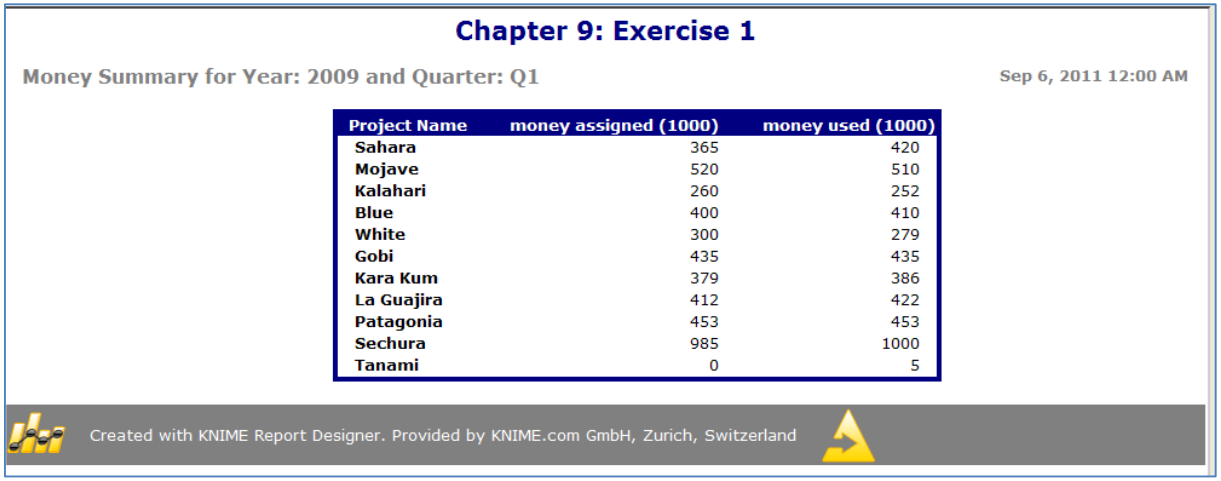
We used a static list of values for “Quarter” report parameter. There we manually inserted the following values: “Q1”, “Q2”, “Q3”, and “Q4”.

We used a dynamic list of values for the “Year” report parameter (Fig. 9.26). The list of values was taken from the report data set called “year list”.

9.28. Exercise 1: The "Edit Parameter" window for report parameter "Year"



9.29. Exercise 1: The Report



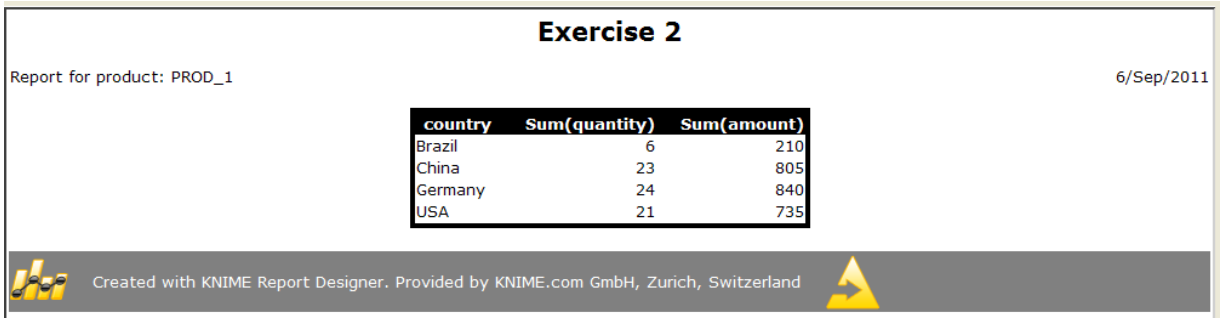
Project Name	money assigned (1000)	money used (1000)
Sahara	365	420
Mojave	520	510
Kalahari	260	252
Blue	400	410
White	300	279
Gobi	435	435
Kara Kum	379	386
La Guajira	412	422
Patagonia	453	453
Sechura	985	1000
Tanami	0	5

Exercise 2

With the data from file “sales.csv” from the Download Zone, set a report to look like figure 9.30. In particular:

- The report has to be created for either product “prod_1” or product “prod_2”;
- Write which product has been used, in the top part of the report under the title;
- Produce a current date in the format “dd/MMM/yyyy”.

9.30. Exercise 2: The target report



Exercise 2

Report for product: PROD_1 6/Sep/2011

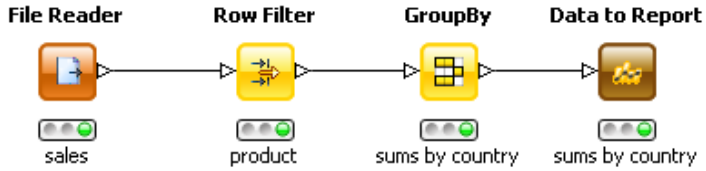
country	Sum(quantity)	Sum(amount)
Brazil	6	210
China	23	805
Germany	24	840
USA	21	735

Created with KNIME Report Designer. Provided by KNIME.com GmbH, Zurich, Switzerland

Solution to Exercise 2

The workflow to calculate the data for the report table is shown in the figure below.

9.31. Exercise 2: The workflow



In addition, one workflow variable has been created and named “product”. This workflow variable becomes the report parameter when switching to the reporting environment. The “product” workflow variable should accept two values only: “prod_1” for “product 1” and “prod_2” for “product 2”.

The dynamic text items below the title use the following expression respectively:

```
BirtStr.concat("Report for product: ",
               BirtStr.toUpper(params["product"].value))
```

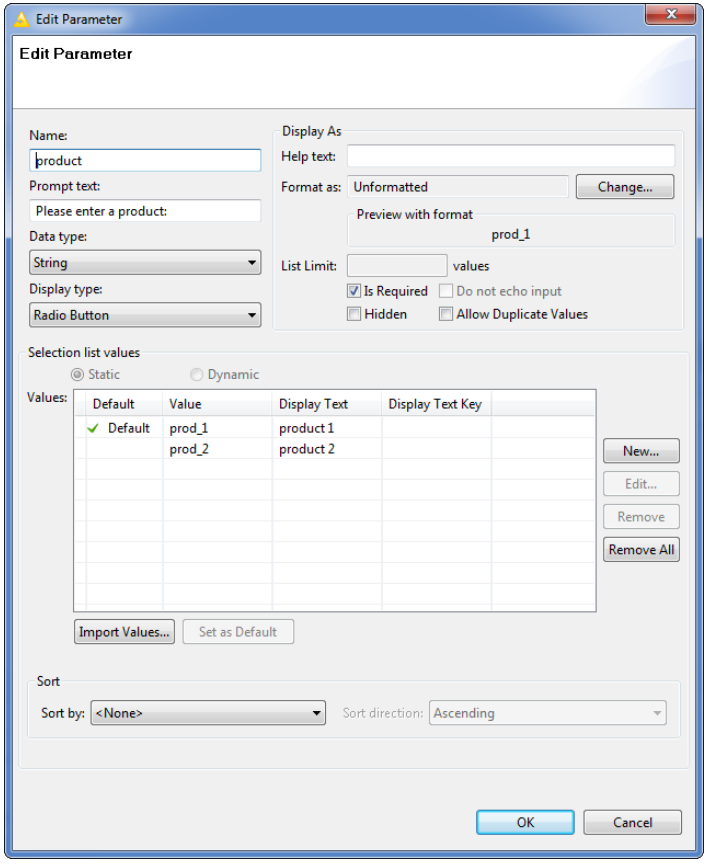
The “toUpper” function changes the argument string into a full upper case string.

```
BirtStr.concat( BirtDateTime.day(BirtDateTime.today()), "/",
               BirtDateTime.month(BirtDateTime.today(),3), "/",
               BirtDateTime.year(BirtDateTime.today()))
```

This sequence of “concat()”, “day()”, “month()”, and “year()” produces the desired date format as specified in figure 9.30.

In the “Edit Parameters” window for the “product” report parameter we have chosen a radio button with the choice between “prod_1” and “prod_2”. The list of values for a radio button is a static one and has to be filled in manually.

9.32. Exercise 2: The "Edit Parameter" window



Exercise 3

Using data in the “sales.csv” file from the Download Zone, build a report with:

- A table with the product name, the total sum of amount, and the total sum of quantity for each product;
- The image of each product at the end of the row.

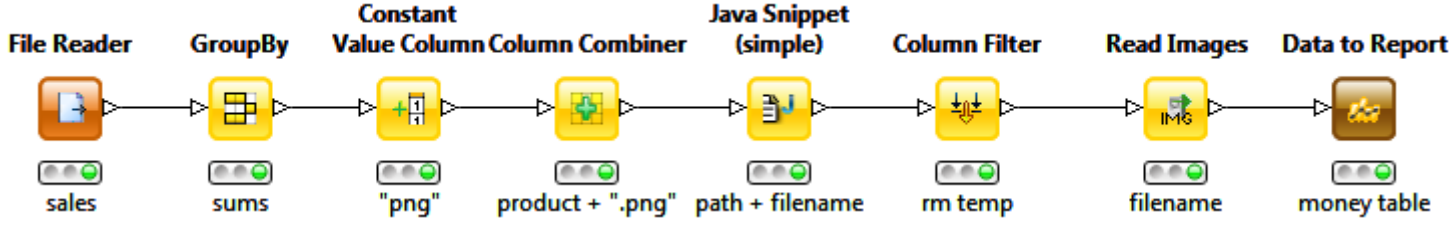
Solution to Exercise 3

The product PNG image files are located in the “Download Zone\images” folder and are named after the product they represent.

We created a workflow with the image file path as workflow variable, named “path”.

After reading the “sales” data set, we built the required table with product name, sum(amount), and sum(quantity) using a “GroupBy” node named “sums”. In a temporary column named “temp” we wrote the “png” string in each row by using a “Rule Engine” node. We combined the product name with the “png” extension with a “Column Combiner” node and subsequently with the image file path with a “Java Snippet (simple)” node and we placed the result in a column named “filename”. After removing the temporary column “temp”, we read the PNG images from the file path into the “filename” column with a “Read Images” node. Finally we exported the data table to a report by means of a “Data to Report” node named “money table”. The final workflow is shown in figure 9.33.

9.33. Exercise 3: The workflow



Note. The file path to be read by the “Read Images” node has to start with “file:/” to allow a proper reading.

In the report, we created a title and a table from the “money table” data set. However, just creating the report table from the data set by drag and drop does not render the images in the table properly. We needed to:

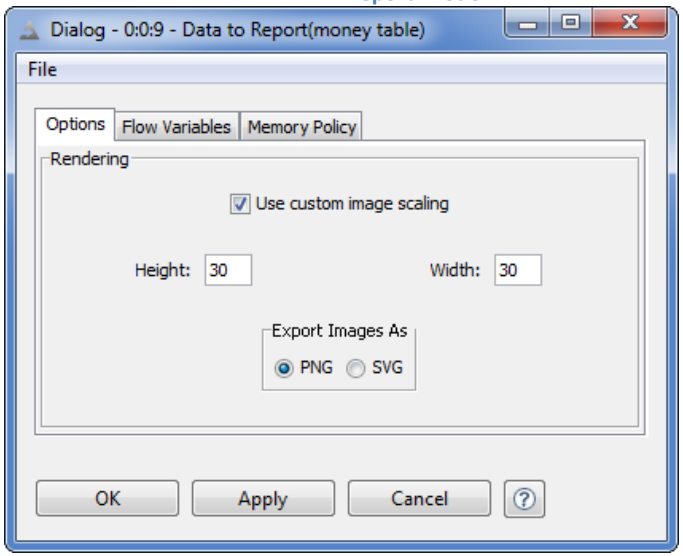
- remove the data cell “filename”;
- right-click the empty data cell and select “Insert” and then “Image”;
- in the “Edit Image Item” window:
    - select “Dynamic image”
    - click the “Select Image Data …” button

- In the “Select Data Binding” window:
  - Select the column name that contains the images
  - Click “OK”
- Click “Insert”

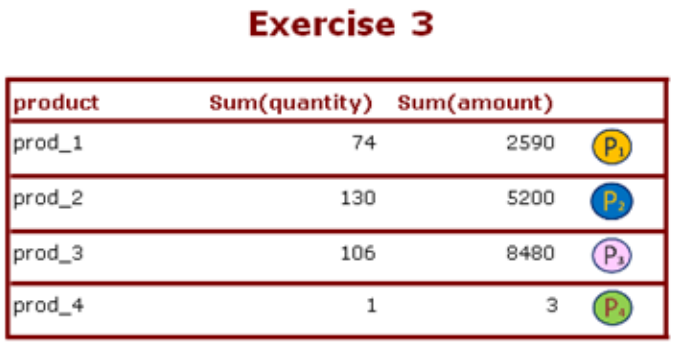
The images now display properly in the table, even though they might be a bit too large. To change the image sizes, you can:

- Either resize the image icon in the report layout manually
- Or change the “Height” and “Width” preferences in the configuration settings window of the “Data to Report” node back in the workflow (Fig. 9.34).

9.34. Exercise 3: The configuration window of the "Data to Report" node



9.35. Exercise 3: The report



Chapter 10. Memory Handling and Batch Mode

10.1. The “knime.ini” File

Sometimes some workflows require exceptional memory usage. The amount of memory available to the KNIME software is stored in the “knime.ini” file. The “knime.ini” file is located in the directory in which KNIME is installed, together with the “knime.exe” file. The “knime.ini” file contains a number of settings required by the KNIME software. Two of these settings pertain to memory usage.

- -Xmx<size> defines the maximum heap size available to run the workflows
- -XX:MaxPermSize=<size> defines the size of the separate heap space that is not garbage collected (ergo the “Perm” for permanent). Whatever is allocated to “Perm” is in addition to the heap space size set with -Xmx.

If you run into memory problems, you probably need to increase the heap space (-Xmx option) to a size compatible with the memory you have on your machine. In more rare situations, you might need to increase the MaxPermSize value.

10.1. The "knime.ini" file

```
-startup
plugins/org.eclipse.equinox.launcher_1.1.0.v20100507.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.1.R36x_v20100810
-vmargs
-XX:MaxPermSize=256m
-server
-Dsun.java2d.d3d=false
-Dosgi.classloader.lock=classname
-XX:+UnlockDiagnosticVMOptions
-XX:+UnsyncloadClass
-Dknime.enable.fastload=true
-Xmx1024m
```

10.2. Memory Usage on the KNIME Workbench

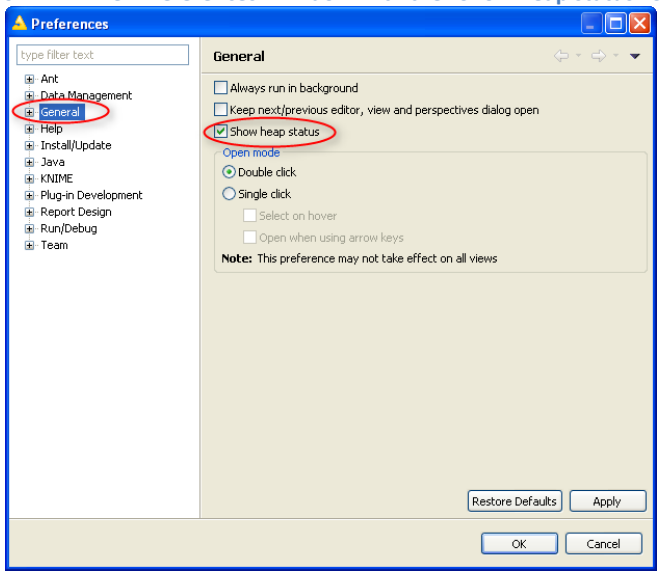
Let’s suppose that you are having a memory problem while executing a workflow. You have already changed the Xmx value in the “knime.ini” file to a higher value, but that was not enough. You are still running into memory problems when your workflow is executed. Even though this kind of problem

occurs very rarely in KNIME, it can still happen. In this case, you need to know whether the problem is due to your workflow or to some other program running on your machine at the same time.

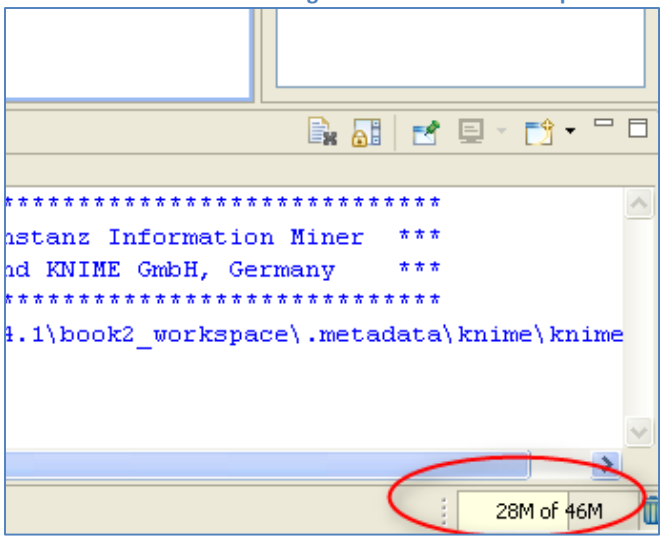
There is an easy way to monitor how much heap space is being used by the workflow and if this reaches the maximum limit assigned by the –Xmx option in the “knime.ini” file. In the KNIME Top Menu:

- Click “File”
- Select “Preferences”
- The “Preferences” window opens
- In the “Preferences” window
    - Select “General”
    - In the frame on the right, named “General”, enable the option “Show heap status”
    - Click “OK”

10.2. The "Preferences" window with the "Show heap status" option



10.3. The bottom right corner shows the heap status



Now, in the bottom right corner you can see a small number showing the heap status (Fig. 10.3).

10.3. The KNIME Batch Command

This book has shown how to run KNIME with its graphical user interface. However there may come an occasion when you need to run KNIME on the command line without graphical interaction. Running KNIME workflows via the command line is possible and it is referred to as KNIME batch mode execution.

The precise format of the command line arguments used to run a KNIME workflow depends on the operating system. The list of the available arguments can be obtained by executing one of the following command lines. Choose the command line that is appropriate for the operating system.

Linux:

```
knime -nosplash -application org.knime.product.KNIME_BATCH_APPLICATION
```

Mac:

```
/Applications/knime/knime.app/Contents/MacOS/knime -nosplash -application org.knime.product.KNIME_BATCH_APPLICATION
```

Windows:

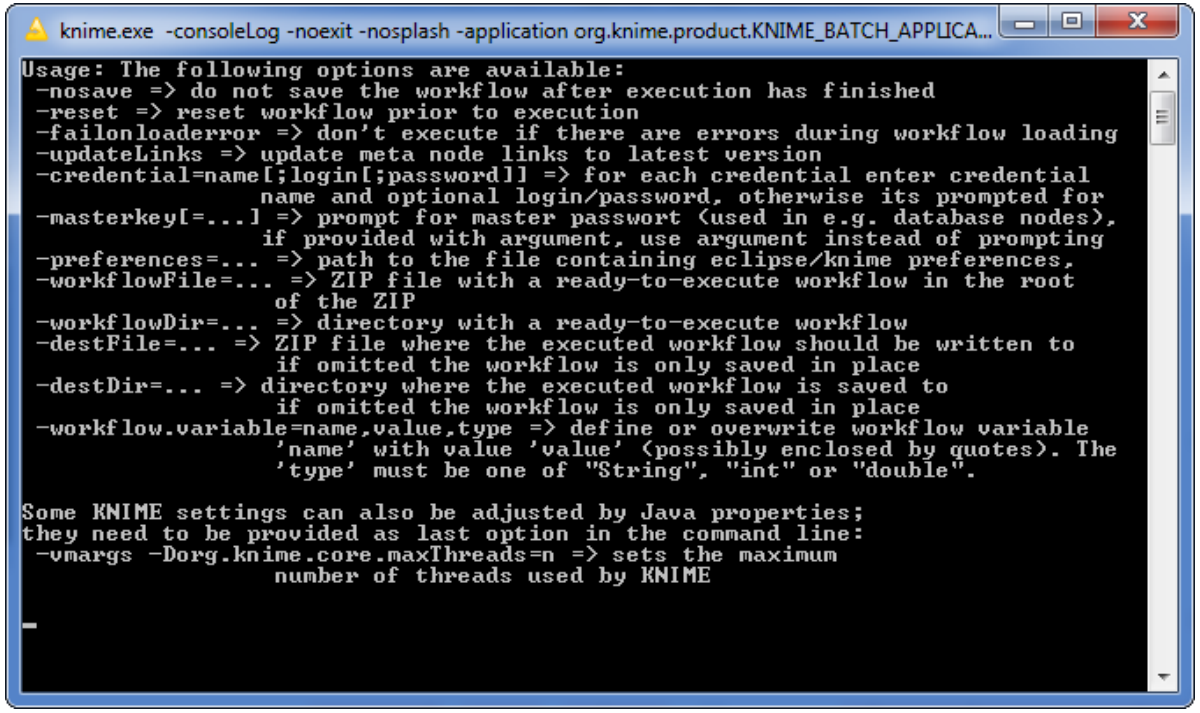
```
knime.exe -consoleLog -noexit -nosplash -application org.knime.product.KNIME_BATCH_APPLICATION
```

The KNIME executable on the Mac is located in the `knime.app/Contents/MacOS/` subfolder. Other than this, the command line usage is the same for the Linux operating system. In the above example, we have assumed that KNIME has been copied into the `/Application` directory.

In the Windows operating system two additional options are required at the command line to enable system messages that would otherwise be suppressed by default. The `-consoleLog` option causes a new window to be opened containing the log messages and `-noexit` keeps the window open after the execution has finished.

The arguments common to all operating systems are the "`-application org.knime.product.KNIME_BATCH_APPLICATION`", which launches the KNIME batch application, and `-nosplash`, which prevents the initial splash window from being shown.

The KNIME executable also returns a code: 0 for a successful execution, 1 for an execution with warnings, 2 for an error before the workflow has been loaded, 3 for errors during the load phase, and 4 for errors during the workflow execution.



10.4. Run a Workflow in Batch Mode

In this section, we show how to run a workflow in batch mode. Taking into account the differences in command line arguments across the different operating systems, as discussed in the previous section, we will continue with examples of workflows running in batch mode under Windows in this and the following sections.

For this chapter we have prepared a number of example workflows to show how to run them in batch mode with different options.

The first workflow is named “Simple Batch Example” and is shown in figure 10.5. This workflow is very simple: it simply reads in the “cars-85.csv” file and writes out a file called “C:\testbatch.csv”.

Note. The assumption is that C:\ is a writable location. Otherwise amend the file output path to an appropriate location.

10.5. A Simple Batch Example Workflow

In order to run a pre-configured workflow contained in the workspace directory, execute the following command line (all in one line) with the argument **`-workflowDir`** to indicate the path of the workflow directory.

```
knime.exe -consoleLog -noexit -nosplash -workflowDir="C:/knime_2.x.y/workspace/Book/Chapter 10/Simple Batch Example" -application org.knime.product.KNIME_BATCH_APPLICATION
```

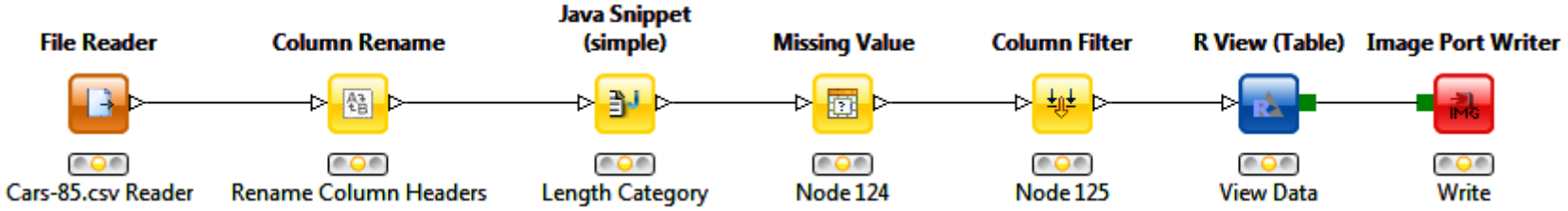
Note. Make sure that you do not have the same workflow open in the KNIME GUI when trying to execute it in batch mode. Otherwise a locked error is thrown. Therefore, the workflow needs to be configured and ready to execute as configuration dialogs are not accessible.

The workflow is saved in place (unless –nosave is specified) and not reset (unless –reset is specified). Thus the workflow should be in a reset state (at least all nodes that should be executed) prior to running the batch execution.

10.3 Batch Execution Using Flow Variables

In many cases you may want to set the node settings dynamically via the workflow variables. A workflow with flow variables can also run in batch execution mode and the flow variable values can be set with the `-workflow.variable` argument in the command line.

In order to show how to run a workflow in batch execution mode using workflow variables, we created an example workflow named “Batch Mode” (Fig. 10.6). This workflow reads the usual “cars-85.csv” file, produces a box plot using an “R View” node, and writes the result to a PNG file with a “Image Port Writer” node. The output file path for the “Image Port Writer” node is passed via a workflow variable named “file_output” of type String.



Let's suppose now that we want to run the workflow in batch mode and that we want to use an output path for the PNG image which is different from the default. The `-workflow.variable` option can be used to pass a new workflow variable value to the workflow executing in batch mode. The `-workflow.variable` option requires three arguments: name, value, and type of the flow variable. The command line is as follows:

```
knime.exe -consoleLog -noexit -nosplash
-workflowDir="C:/knime_2.x.y/workspace/Book/Chapter 10/Batch Mode"
-workflow.variable=file_output,"C:/somepath.png",String
-preferences="C:/knime_2.x.y/preferences.epf"
-application org.knime.product.KNIME_BATCH_APPLICATION -nosave
```

Successfully running this command line opens up a new window with the log information akin to "Finished in 1 sec (1045ms)".

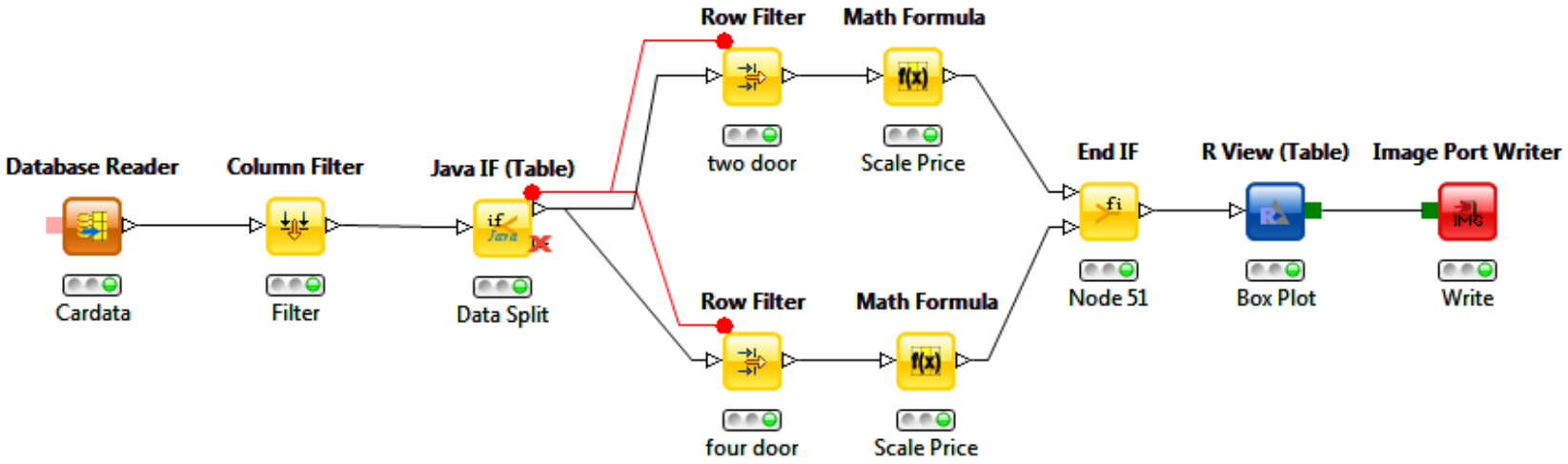
We have used the `-nosave` argument in some examples of this chapter, so that the changes made to the workflow are not saved. The `-preferences` flag is often important to specify, particularly when dealing with databases. Therefore you need to export the preferences prior to running the batch execution of the workflow.

10.4 Batch Execution with Database Connections and Flow Variables

We are now going to expand our knowledge of batch commands by looking at running a workflow that contains a database connection and multiple flow variables. Also for this section we have built an example workflow, named "Batch Mode Databases" which is shown in figure 10.7. The workflow reads car data taken from the "cars-85.csv" file from a database (see the "Create Database" workflow in the "Chapter 10" workflow group in the Download Zone), it splits the analysis for two- or four-doors cars, providing a box plot for engine size, horse power, and price by means of an "R View" node, and finally writes the box-plot image into a PNG image file.

The “Database Reader” node accesses the database by means of credentials. The “Java IF(Table)” node is controlled by a workflow variable named “Doors” and the output path in the “Image Port Writer” node is contained in a second workflow variable named “C:\data\output_2.png”.

10.7. The “Batch Mode Databases” Workflow



To access the database we need the appropriate workflow credential. Similarly, we require knowledge of the workflow variables to control the “Java IF (Table)” node and the “Image Port Writer” node. The database credential can be supplied to the workflow by the `-credential` argument. This argument requires the credential name and optionally the login/password. If the login/password is not supplied the user will be prompted for it.

The workflow credentials needed to access the database are named “RS”, with username “knime_user” and password “knime_user”. The following is the command line to analyze four-door cars.

```
knime.exe -consoleLog -noexit -nosplash
-workflowDir="C:/knime_2.x.y/workspace/Book/Chapter 10/Batch Mode Databases"
-workflow.variable=file_output,"C:/somepath_four.png",String -workflow.variable=Doors,"four",String
-credential=RS;knime_user;knime_user -preferences="C:/knime_2.x.y/preferences.epf" -application
org.knime.product.KNIME_BATCH_APPLICATION -nosave
```

And the following is the command line to analyze two-door cars.

```
knime.exe -consoleLog -noexit -nosplash
```

```
-workflowDir="C:/knime_2.x.y/workspace/Book/Chapter 10/Batch Mode Databases"
-workflow.variable=file_output,"C:/somepath_two.png",String -workflow.variable=Doors,"two",String
-credential=RS;knime_user;knime_user -preferences="C:/knime_2.x.y/preferences.epf"
-application org.knime.product.KNIME_BATCH_APPLICATION -nosave
```

10.5 Exercise

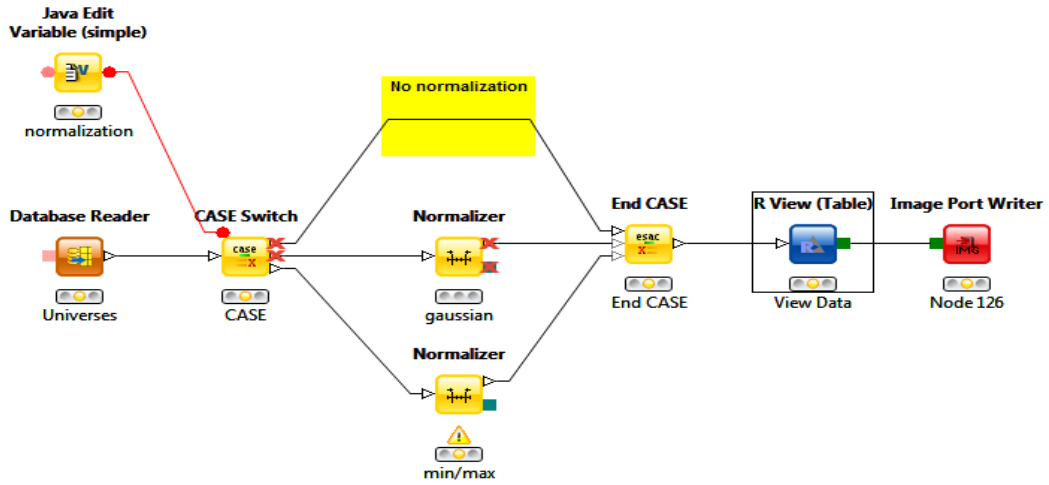
Exercise 1

First of all, create a new table in your "Book2" database with the "Create Database" workflow in the "Chapter10" workflow group.

Write the batch mode command line to run the "Exercise 1" workflow (Fig. 10.8). The batch mode command line needs:

- The credentials to access the table "Universe" created in the database by the "Create Database" workflow (credential name "RS", username "knime_user", and password "knime_user");
- The workflow variable to generate a Gaussian normalization on the data (see the "CASE Switch" node);
- The workflow variable for the output path of the "Image Port Writer" node;
- The workflow preferences exported.

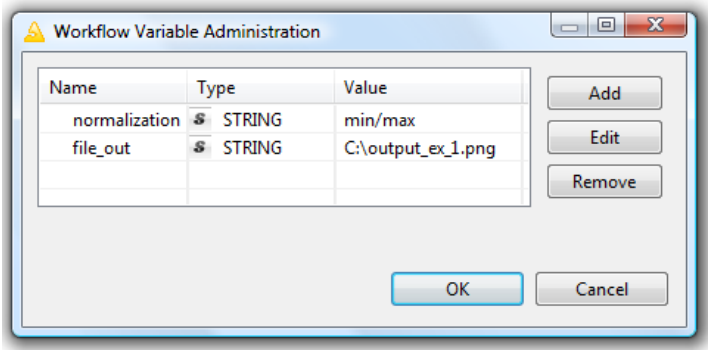
10.8. The "Exercise 1" Workflow



Solution to Exercise 1

The workflow variables can be identified in the workflow variable administration window (Fig. 10.9).

10.9. The Workflow Variable Administration Window of the "Exercise 1" workflow



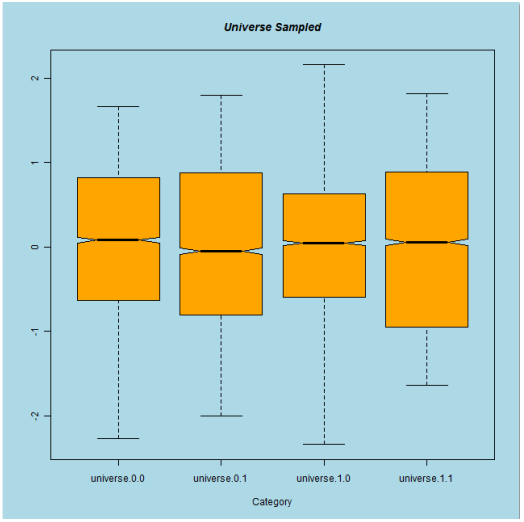
Thus, the following command line gives the desired result as seen in figure 10.10:

```
knime.exe -consoleLog -noexit -nosplash
-workflowDir="C:/knime_2.x.y/workspace/Book/Chapter 10/Exercises/Exercise 1"
-workflow.variable=file_out,"C:/gauss.png",String -preferences="C:/knime_2.x.y/preferences.epf"
-application org.knime.product.KNIME_BATCH_APPLICATION -nosave
-workflow.variable=normalization,"gaussian",String
-credential=RS;knime_user;knime_user
```

It sends the desired output to "`C:/gauss.png`". The console should show a log message similar to:

```
WARN  Normalizer  All numeric columns are used for normalization. Mode: Z -Score Normalization
Finished in 4 sec (4166ms)
```

10.10. The “gauss.png” output from the “Exercise 1” workflow



References

[1] Silipo R., “KNIME Beginner’s Luck”, KNIME Press, 2010
[2] Frank A. and Asuncion A., “UCI Machine Learning Repository [http://archive.ucs.uci.edu/ml], Irvine, CA: University of California, School of Information and Computer Science (2010)
[3] NIST/SEMATECH, “e-Handbook of Statistical Methods”, (http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4.htm)
[4] R Project, “An Introduction to R”, (http://cran.r-project.org/doc/manuals/R-intro.pdf)
[5] Web Services Activity (http://www.w3.org/2002/ws/)

Node and Topic Index

H

I

J

K

L

M

N

P

Q

R

S

T

U

V

W

X

The KNIME Cookbook: Recipes for the Advanced User

This book is the much awaited sequel to the introductory text "KNIME Beginner's Luck". Building upon the reader's first experience with KNIME, this book presents some more advanced features, like looping, selecting workflow paths, workflow variables, reading and writing data from and to a database, running R scripts from inside a workflow, and more.

All new concepts, nodes, and features are demonstrated through worked examples and the learned knowledge is reinforced with exercises. All example workflows, exercise solutions, and data sets are available on line.

The goal of this book is to elevate your data analysis from a basic exploratory level to a more professionally organized and complex structure.

About the Authors

***Dr Rosaria Silipo** has gained her PhD in biomedical engineering, at the University of Florence, Italy, in 1996. She has been analyzing data ever since, during her postdoctoral program at the University of Berkeley and during most of her following job positions (Nuance Comm., Viseca). She has many years of experience in data analysis in many different application fields. In particular, she is interested in customer care, finance, and text mining. In the last few years she has been using KNIME for her consulting work, becoming a KNIME certified trainer and an expert in the KNIME Reporting tool.*

***Dr Michael P. Mazanetz** gained his PhD in medicinal and computational chemistry at the University of Nottingham, UK. He has worked in the pharmaceutical sector for over a decade. After several years working as a medicinal chemist at Eli Lilly, he took a post at Evotec as a computational chemist. His interests lie in the development and in the application of computational methods to drive medicinal chemistry projects forward. He has been using KNIME for several years and is presently involved in coordinating the KNIME node development and deployment at Evotec. Michael is also a KNIME certified trainer.*

ISBN: 978-3-9523926-0-7