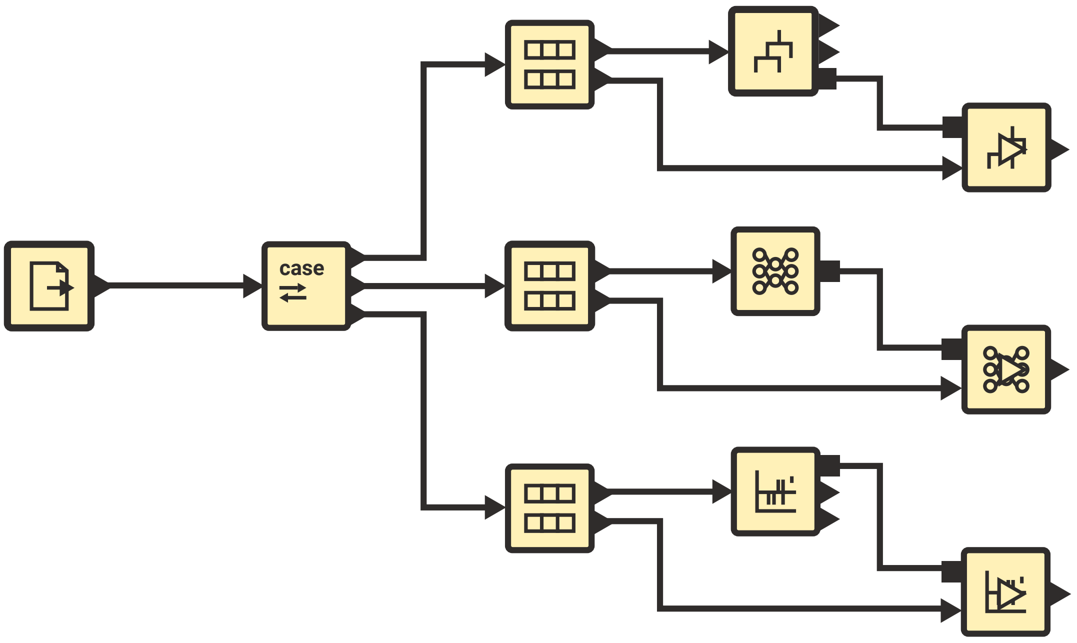
KNIME Textbooks

ROSARIA **SILIPO** SANKET **JOSHI**

KNIME Advanced Luck

KNIME v5.2



Open for Innovation
KNIME

Copyright © 2024 by KNIME Press

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording or likewise.

This book has been updated for **KNIME 5.2**.

For information regarding permissions and sales, write to:

KNIME Press
Talacker 50
8001 Zurich
Switzerland

knimepress@knime.com

ISBN: 978-3-9523926-6-9

www.knime.com

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 Bernd Wiswedel for answering our endless questions about calling external REST services from inside a workflow, Iris Adae for explaining the most advanced features of the Date&Time nodes, Andisa Dewi for significantly contributing to the Database Operations chapter, and Roberto Cadili and Elisabeth Richter for creating examples for the latest extension: the Geospatial Analytics Extension.

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 for their support in publishing and advertising this book.

Table of Contents

Chapter 1: Introduction

1.1. Purpose and Structure of this Book

KNIME Analytics Platform 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 Analytics Platform is open source, has led a large number of professionals to use it. In addition, third-party software vendors develop KNIME extensions in order to integrate their tools into it. KNIME nodes are now available that reach beyond customer relationship management and business intelligence, extending into the field of finance, life sciences, biotechnology, pharmaceutical, and chemical industries. Thus, the archetypal KNIME user is no longer necessarily a data science 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 software. It expands on the topics that were covered in the first KNIME user guide ("KNIME Beginner's Luck") and introduces more advanced functionalities. In the first guide[1], we described the basic principles of KNIME Analytics Platform and showed how to use it. We demonstrated how to build a basic workflow to manipulate, visualize, and model 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 connect to a database, import data, build an appropriate SQL query to select a subset of the data or for some required processing, and finally to write data back into the database. Accessing a database, importing data, and building SQL queries are the basic operations necessary for any, even very simple, data warehousing strategy.

Of course, the largest source of data is nowadays the Internet. Chapter 3 is dedicated to alternative ways of getting data besides files and databases, i.e. web data sources. Chapter 3 starts with the connectors to Google Sheets and continues with access to REST services. Those are definitely powerful tools to search for data elsewhere.

[1] *Silipo R., Joshi S.; "KNIME Beginner's Luck", KNIME Press (2023) (https://www.knime.com/knimepress/beginners-luck)*

Chapter 4 introduces the Date&Time object and the nodes to turn a String column into a Date&Time column, to format it, to extract a time difference, and in general to perform date/time based operations. The Date&Time object provides the basis for working with time series. The last section of chapter 4 briefly describes a few nodes dedicated to time series analysis.

A very important concept for the KNIME workflows is the concept of “flow variables”. Flow variables enable external parameters to be introduced into a workflow to control its execution. Chapter 5 describes what a flow variable is, how to create it, and how to edit it inside the workflow, if needed. Capitalizing on the concept of flow variables, we introduce the components and explain how to make them configurable via Configuration nodes and Widget nodes.

Widget nodes especially can be helpful in the creation of dashboards, reports, and data apps. Chapter 6 shows how to build a rich dashboard through the composite view of an advanced component.

Most data operations in KNIME Analytics Platform are executed on a data matrix, named data table. This means that an operation is executed on all data rows. This is a big advantage in terms of speed and programming compactness. However, from time to time, a workflow also needs to run its rows, one after the other, through an operation. That is, sometimes it needs a real loop. Chapter 7 introduces a few nodes that implement loops: from a simple “for” cycle to more complex loops, such as looping on a list of values or feeding the current iteration results into the next iteration.

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

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

1.2. Data and Workflows for this Book

This book builds a few examples and provides the solutions to the exercises. The workflows are accessible via the KNIME Community Hub and are stored in the KNIME Press space – look for the respective book and KNIME version. To download material from the KNIME Community Hub, you need to be logged in with your KNIME account (see how to create a KNIME account), the same as for the KNIME Forum. After entering the KNIME Community Hub, in order to download the workflows, just click on the cloud icon. Download the whole folder onto your machine from the KNIME Advanced Luck space, which will result in a .knar file. Then double click it OR import it into the KNIME Explorer via "Import workflow".

At the end of the import operation, in the Space Explorer panel, you should find a *KNIME Advanced Luck v5.2 – Exercises* folder containing Chapter2, Chapter3, Chapter4, Chapter5, Chapter6, Chapter7, and Chapter8 subfolders, each one with workflows and exercises to be

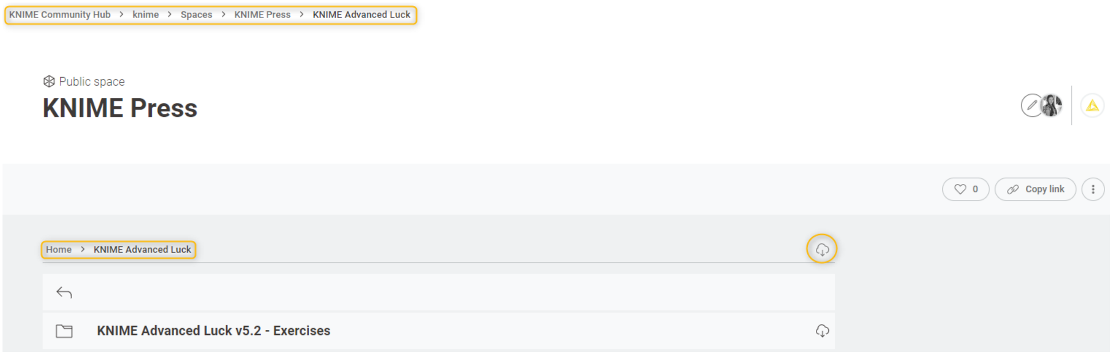

KNIME Community Hub > knime > Spaces > KNIME Press > KNIME Advanced Luck
Public space
KNIME Press
0
Copy link
Home > KNIME Advanced Luck
KNIME Advanced Luck v5.2 - Exercises

Figure 1.1. Workflows and data for this book on the KNIME Community Hub.

implemented in the next chapters. You should also find a KALdata folder containing the required data.

The data used for the exercises and for the demonstrative workflows of this book 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). If the data set belongs to the UCI Repository, a full link is provided here to download it. Data generated by the author, that is not public data, are located only in the KALdata folder.

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

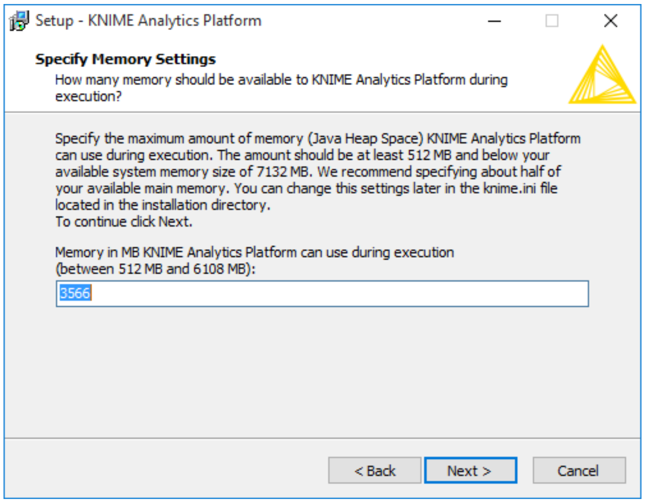
- Automobile: http://archive.ics.uci.edu/ml/datasets/Automobile
- Slump_test: http://archive.ics.uci.edu/ml/datasets/Concrete+Slump+Test

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

[2] *Frank A. and Asuncion A., "UCI Machine Learning Repository", Irvine, CA: University of California, School of Information and Computer Science (2010) (https://archive.ics.uci.edu/datasets)*

1.3. Memory Usage in KNIME Analytics Platform

When installing KNIME Analytics Platform via the Windows installer, it suggests suitable memory settings at installation time. However, in all other installation procedures, or if you want to change the set number of MB later, you will need to set yourself the maximum amount of memory available to KNIME Analytics Platform.


Setup - KNIME Analytics Platform

Specify Memory Settings

How many memory should be available to KNIME Analytics Platform during execution?

Specify the maximum amount of memory (Java Heap Space) KNIME Analytics Platform can use during execution. The amount should be at least 512 MB and below your available system memory size of 7132 MB. We recommend specifying about half of your available main memory. You can change this settings later in the knime.ini file located in the installation directory.
To continue click Next.

Memory in MB KNIME Analytics Platform can use during execution (between 512 MB and 6108 MB):
3566

Figure 1.2. Specifying the Memory Setting on install when using the Windows installer.

The amount of memory available to KNIME Analytics Platform is stored in the *knime.ini* file. The knime.ini file is located in the directory in which KNIME Analytics Platform has been installed, together with the knime.exe file. The knime.ini file contains a number of required settings.

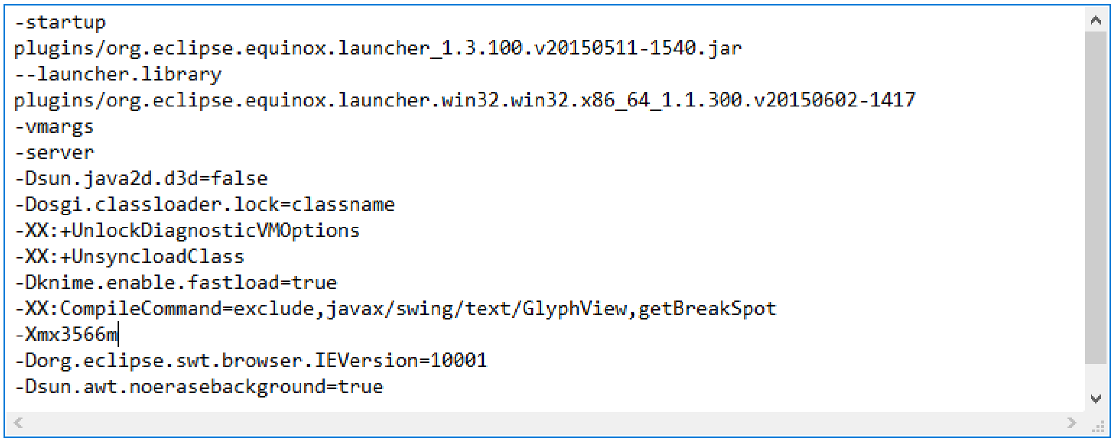

```
-startup
plugins/org.eclipse.equinox.launcher_1.3.100.v20150511-1540.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.300.v20150602-1417
-vmargs
-server
-Dsun.java2d.d3d=false
-Dosgi.classloader.lock=classname
-XX:+UnlockDiagnosticVMOptions
-XX:+UnsyncloadClass
-Dknime.enable.fastload=true
-XX:CompileCommand=exclude,javax/swing/text/GlyphView,getBreakSpot
-Xmx3566m
-Dorg.eclipse.swt.browser.IEVersion=10001
-Dsun.awt.noerasebackground=true
```

Figure 1.3. The "knime.ini" file.

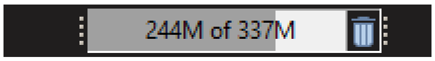


Figure 1.4. The bottom right corner shows the heap status.

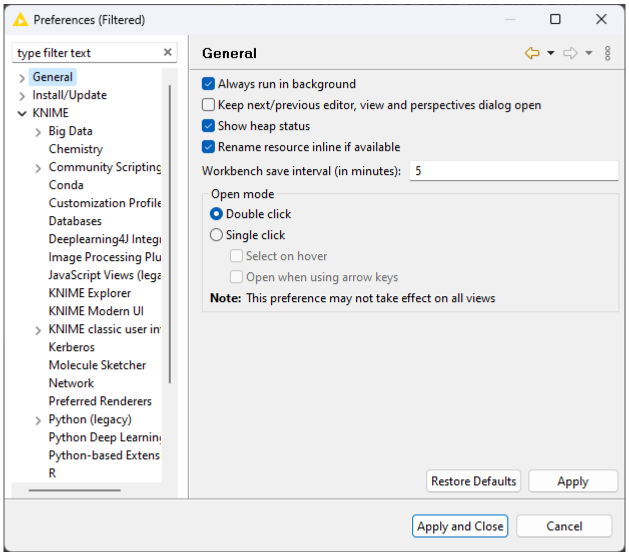
-Xmx<size> is the setting that defines the maximum heap size available to run workflows. You can define this value by editing the knime.ini file or at installation time. If you run into memory problems, you probably need to manually increase the heap space (-Xmx option) directly in the knime.ini file to a size compatible with the memory you have on your machine (like 4G for 4 Gigabytes).

There is also an easy way to monitor how much heap space is being used by a workflow and if this reaches the maximum limit assigned by the –Xmx option. From within KNIME Analytics Platform

- Access the “Preferences” in the upper right corner
- In the “Preferences” window, select “General” and enable the “Show heap status” option; Click “Apply and Close”
- Now you can see a number showing the heap status in the lower right corner of the KNIME Workbench

To run the example workflows and the exercises provided in this book, you will need to install the whole “KNIME & Extensions” group. In order to install a KNIME Extension:

- In the top-right corner of the screen, select “Menu” option -> “Install Extensions”
- In the “Install” window:
    - Make sure “Group items by category” is checked
    - Open the group containing your extension, like for example “KNIME & Extensions” group
    - If you do not know where your extension package is located, just run a search by inserting a few related keywords in the top textbox
    - Select your extension
    - Click “Next” and follow installation instructions



General

- [✓] Always run in background
- [ ] Keep next/previous editor, view and perspectives dialog open
- [✓] Show heap status
- [✓] Rename resource inline if available

Workbench save interval (in minutes): 5

Open mode

- [✓ Double click]
- Single click
    - [ ] Select on hover
    - [ ] Open when using arrow keys

Note: This preference may not take effect on all views

Figure 1.5. The "Preferences" window with the "Show heap status" option.

Chapter 2: Database Operations

2.1. Database Nodes

We proceed with the exploration of the advanced features of KNIME Analytics Platform 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]. Here, though, we investigate the multiple possibilities for connecting, reading, writing, and selecting data from and to a database in much greater detail.

For KNIME Analytics Platform 4.0, a full rewrite of the Database support was performed. The old Database nodes are now marked as “Legacy”; the new nodes reside in the DB category in the Node Repository, where you can find a number of nodes for database access, manipulation, and writing.

First of all, we want to create the workflow group “Chapter2”, to host all workflows for this chapter. Then, in this new workflow group, we want to create an empty workflow with the name “Database_Operations”. The goal of this workflow is to show how to connect to a database, retrieve data from the database, and write data into the database.

In the newly created workflow named “Database_Operations”, we read the data from the “sales” table in the “KCBBook.sqlite” database. SQLite is a file-based database software. Since it requires neither server nor authentication, it is suitable to show the KNIME database nodes without the hassle of setting up a full-blown database. In SQLite the only thing you need is the path to the file containing the database, “KCBBook.sqlite” in this case and available in folder KALdata. The “sales” table inside the database contains the sale records for a fictitious company. Such 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) generated by 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.

To read data from a database into a KNIME workflow, we start with one node to establish the connection to the database; then a node to select the table to work on; a few nodes, one after the other, to build the SQL query to extract the required data set; and finally one last node to run the SQL query on the database and import the results into the KNIME workflow. There are many ways to build a SQL query, each one fitting a given level of SQL expertise.

2.2. Connect to a Database: DB Connector Nodes

The first step is to just establish a connection to a database, nothing more, and to do that you just need to know the specs of your database: *server URL, credentials, JDBC driver*. In the DB/Connection category we find all connector nodes; that is those nodes that just connect to a database. We find dedicated nodes for selected databases and one generic node to connect to any database.

The generic connector node is named "DB Connector" and produces a database connection at the output port, indicated with a red square port. It can connect to any database, as long as you provide the correct JDBC driver. The JDBC driver is a file, usually provided by the database software distributor, interfacing the SQL script with the database software.

If you have chosen an uncommon or a strictly licensed database, it is possible that the JDBC driver you need is not part of the pre-loaded JDBC driver set. In this case, you need to upload your own JDBC driver file onto KNIME Analytics Platform via the "Preferences" window.

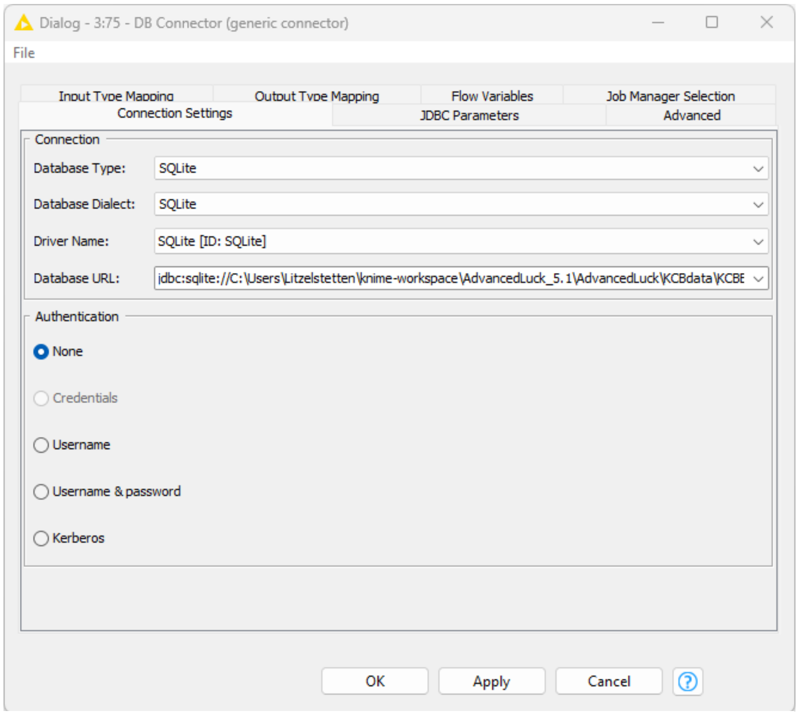
Note. The Oracle JDBC driver for example is not available in the set of pre-loaded JDBC drivers, due to Oracle's licensing restriction. If you want to connect to an Oracle database via the "DB Connector" node or even the dedicated "Oracle Connector" node, you need to register the Oracle JDBC driver first in the Preferences window.

DB Connector

The "DB Connector" node just establishes a connection to a database, i.e. it can connect to arbitrary JDBC compliant databases. To do that, the following most important node settings are required:

- ***Database Type.*** Select the type of database the node will connect to. For example, if the database is a PostgreSQL derivative select Postgres as database type. If you don't know the type select the default type.

- ***Database Dialect.*** Select the database dialect.
- ***Driver Name.*** Select the appropriate driver for your specific database. JDBC drivers for most commonly used databases have been pre-loaded in the node. If there is no matching JDBC driver it first needs to be registered, as described in “Register your own JDBC driver” later in this book. Only JDBC drivers that have been registered for that database type, will be available for selection.
- ***Database URL.*** A driver-specific JDBC URL is required. Please consult the vendor documentation for the URL representation according to the JDBC driver you are using.



Connection

Database Type: SQLite

Database Dialect: SQLite

Driver Name: SQLite [ID: SQLite]

Database URL: jdbc:sqlite://C:\Users\Litzelstetten\knime-workspace\AdvancedLuck_5.1\AdvancedLuck\KCBdata\KCBE

Authentication

● None

○ Credentials

○ Username

○ Username & password

○ Kerberos

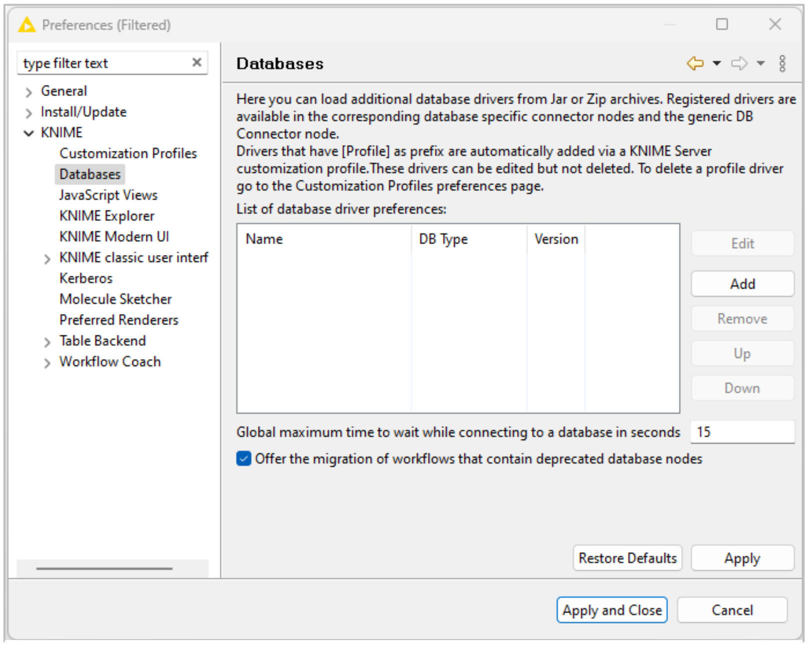
Figure 2.1. Configuration window of the DB Connector node.

- ***Authentication.*** Login credentials – when required - can either be provided via credential flow variables, or directly in the configuration dialog in the form of username and password. Kerberos authentication is also provided for databases that support this feature, e.g., Hive or Impala.

Register your own JDBC Driver

In the top menu:

- Click “Open preferences” icon at the top-right corner of the screen
- In the “Preferences” window:
    - In the left frame: Open “KNIME” -> Select “Databases”
    - In the right frame: Click button “Add” and a new database driver settings window will open where you can provide all necessary information about the JDBC driver. Load the database driver file. After that, the driver will appear in the list of database drivers. Click “Apply and Close” to apply the changes.



Databases

Here you can load additional database drivers from Jar or Zip archives. Registered drivers are available in the corresponding database specific connector nodes and the generic DB Connector node.
Drivers that have [Profile] as prefix are automatically added via a KNIME Server customization profile.These drivers can be edited but not deleted. To delete a profile driver go to the Customization Profiles preferences page.
List of database driver preferences:

Name	DB Type	Version

Global maximum time to wait while connecting to a database in seconds 15

☑ Offer the migration of workflows that contain deprecated database nodes

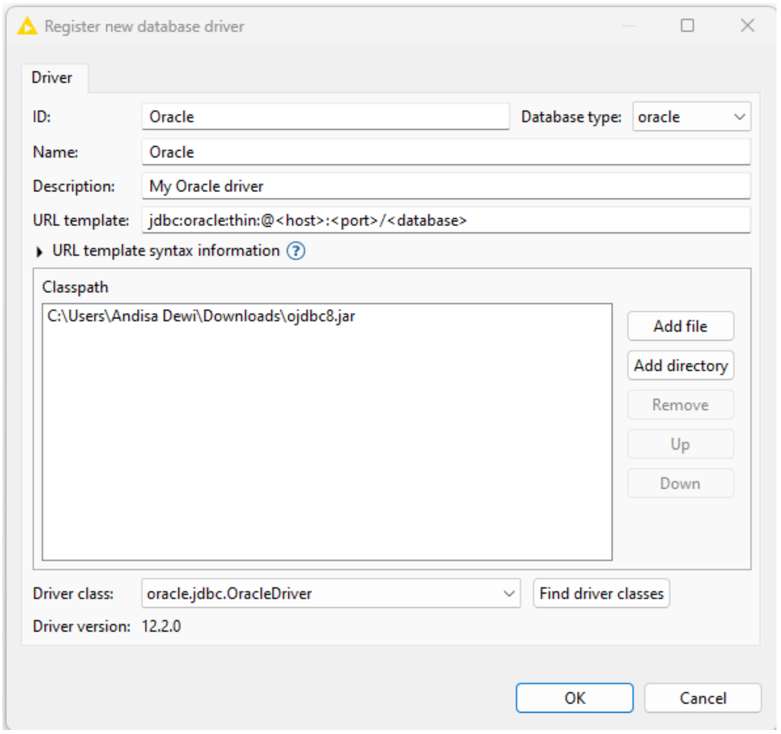
Figure 2.2. Register a JDBC driver file in KNIME Preferences.

Edit Database Driver Settings Window

Clicking “Add” in the previous window opens a new database driver window where you can provide the JDBC driver path and all necessary information, such as:

- ***ID.*** The unique ID of the JDBC driver consisting only of alphanumeric characters and underscore.
- ***Name.*** The unique name of the JDBC driver.
- ***Database type.*** If your database is not on the list, you can choose “default”.

- ***Description.*** Optional description of the JDBC driver.
- ***URL template.*** The JDBC driver connection URL format. Please consult your database vendor to find the appropriate format.
- ***Classpath.*** The path to the JDBC driver. Click "Add file" if the driver is provided as a single .jar file, or "Add directory" if the driver is provided as a folder that contains several .jar files. Some vendors offer a .zip file for download, which needs to be unpacked to a folder first.



ID: Oracle
Database type: oracle
Name: Oracle
Description: My Oracle driver
URL template: jdbc:oracle:thin:@<host>:<port>/<database>
Classpath: C:\Users\Andisa Dewi\Downloads\ojdbc8.jar
Driver class: oracle.jdbc.OracleDriver
Driver version: 12.2.0

Figure 2.3. Edit database driver settings.

- ***Driver class.*** The JDBC driver class and version will be detected automatically by clicking "Find driver classes". Please select the appropriate class after clicking the button.

After filling in all the information, click "Ok", and the newly added driver will appear in the list of database drivers.

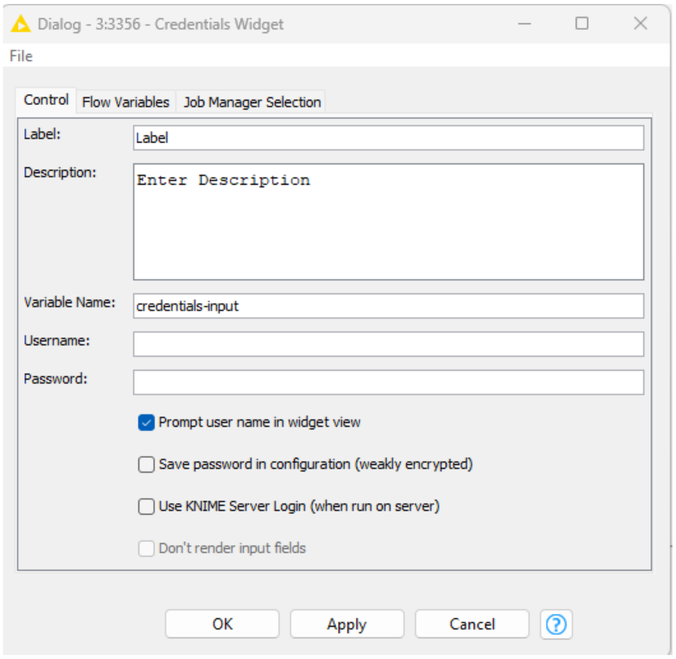
After loading the JDBC database driver file in the "Preferences" window, the database driver becomes a general KNIME feature and is available for all nodes in all workflows in all workspaces.

Credentials Widget

The *Credentials Widget* node is a KNIME node that allows you to create a credentials input widget for use in component composite views. It provides a user-friendly way for users to enter their credentials, such as usernames and passwords, when running a workflow.

It outputs a credentials flow variable, which can be used with other nodes that require authentication. It provides a secure way to enter the credentials, which eventually enhances the security of the workflows.

Figure 2.4. The Credentials Widget node.



Dialog - 3:3356 - Credentials Widget

File

Control | Flow Variables | Job Manager Selection

Label: Label

Description: Enter Description

Variable Name: credentials-input

Username:

Password:

[✓] Prompt user name in widget view

[ ] Save password in configuration (weakly encrypted)

[ ] Use KNIME Server Login (when run on server)

[ ] Don't render input fields

OK | Apply | Cancel

Figure 2.5. Configuration window of the Credentials Widget node

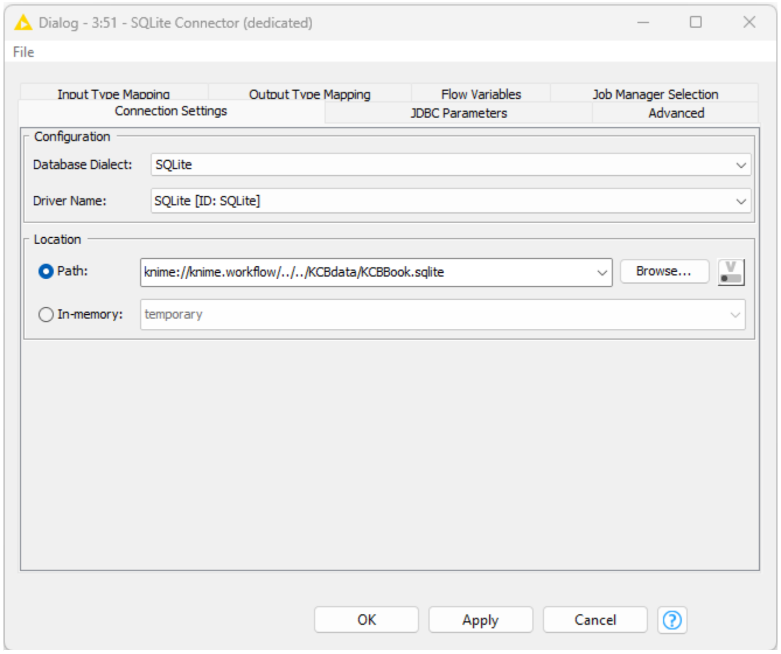
SQLite Connector

The “SQLite Connector” node establishes the connection to an SQLite database (file) and requires the following connection settings:

- ***Database Dialect.*** Choose the registered database dialect here.
- ***Driver Name.*** Select the registered database drive.
- ***Path.*** The path to the SQLite database file.

- ***In-memory.*** The in-memory SQLite database name. Use this option to create a temporary database if the database supports this feature.

Note. The option Path accepts the knime:// protocol and therefore the relative path to the workflow location, which enhances the workflow portability.



Database Dialect: SQLite
Driver Name: SQLite [ID: SQLite]
[✓ Path:] knime://knime.workflow/../../KCBdata/KCBBook.sqlite
In-memory: temporary

Figure 2.6. Configuration window of the "SQLite Connector" node.

In the workflow "Database_Operations" we used an SQLite Connector node and a generic DB Connector node to connect to the SQLite database file KCBBook.sqlite. In the DB Connector node, we selected the SQLite driver and provided the path to the SQLite file in the node configuration window. The configuration window of the SQLite Connector node is shown above.

Note. Unlike in an "SQLite Connector" node, in a generic "DB Connector" node, the URL for the sqlite file cannot use the knime:// protocol and the relative path. It needs the absolute URL path of the database file, which might make things complicated when moving the workflow into another environment.

As an example, for all dedicated connector nodes, we have shown the "SQLite Connector" node. Other dedicated database connector nodes differ from this one in the specific settings required by their database master.

2.3. Select the Table to work on: the DB Table Selector Node

Once we have a connection to the database, we need to start working on the data. The next step is to select the table to work on. This is the task for the “DB Table Selector” node. This node transforms a database connection (red square input port) into a database SQL query (brown square output port) to be executed later on the input database connection.

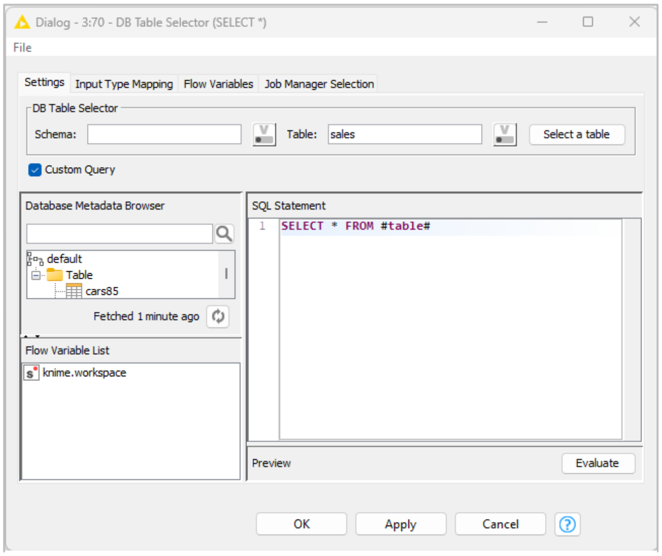
DB Table Selector

The “DB Table Selector” node takes a database connection at the input port and allows to select a table, or a view, interactively based on the input database connection.

At the top part you can enter the schema and the table/view name that you want to select.

Pressing the “Select a table” button opens a “Database Metadata Browser” window (see below) that lists available tables/views in the database.

In addition, ticking the “Custom Query” checkbox allows you to write your own custom SQL query to narrow down the result. It accepts any SELECT statement, and the placeholder #table# can be used to refer to the selected table. It also shows the “Database Metadata Browser” and “Flow Variable List” panels on the left.


Dialog - 3:70 - DB Table Selector (SELECT *)

File

Settings | Input Type Mapping | Flow Variables | Job Manager Selection

DB Table Selector

Schema:

Table: sales

Select a table

☑ Custom Query

Database Metadata Browser

default
Table
cars85

Fetched 1 minute ago

Flow Variable List

knime.workspace

SQL Statement

```
SELECT * FROM #table#
```

Preview

Evaluate

OK | Apply | Cancel

Figure 2.7. Configuration window of the DB Table Selector window

Database Metadata Browser

The “Database Metadata Browser” window shows the database schema, including all tables / views and their corresponding columns and column data types.

At first opening, it fetches the metadata from the database and caches it for subsequent use. By clicking on an element (schema/table/view) it shows the contained elements. To select an element, select the name and click “OK” or double-click it.

The search box at the top of the window allows you to search for any table or view inside the database.

At the bottom there is a refresh button to re-fetch the schema list, including a time reference on how long ago the schema was last refreshed.

Note that if you have just created a table and you cannot find it in the schema list, it might be that the metadata browser cache is not up to date, so please try to refresh the list by clicking the refresh button in the lower right corner.

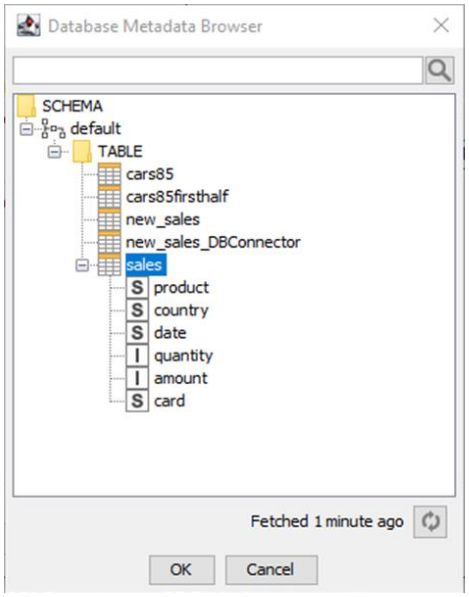


Figure 2.8. Database Metadata Browser.

This modular approach separating database connection from table selection allows to process different database tables with different SQL queries on different workflow branches. It also allows to deal with special table selection statements for the many different databases, big data platforms, SQL dialects, and NoSQL scripts.

In the workflow “Database_Operations” we use the DB Table Selector node after the SQLite Connector node to upload the whole “sales” table using the default SQL query `SELECT * FROM #table#`.

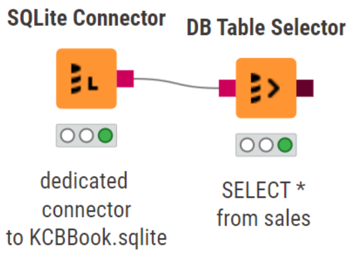


Figure 2.9. Sequence of SQLite Connector node and TB Table Selector node.

2.4. In-Database Processing

For the non-SQL savvy, KNIME Analytics Platform offers several database manipulation nodes. These nodes implement SQL queries through a graphical user interface bypassing the whole SQL script. They take a SQL query as input (brown square port) and produce a SQL query as output, which consists of the input SQL query augmented with the SQL query implemented in the node itself.

For example, using the *DB Table Selector* node to select the table "sales" simply loads the whole dataset. However, what we really want to do is to get rid of the rows that pertain to product "prod_4", to keep only those with country "Germany", and to get rid of the "card" column, before we pull in the data set from the database.

The SQL query to perform those operations would be something like:

```
SELECT product, country, date, quantity, amount from sales WHERE product!=‘prod_4’
AND country = ‘Germany’
```

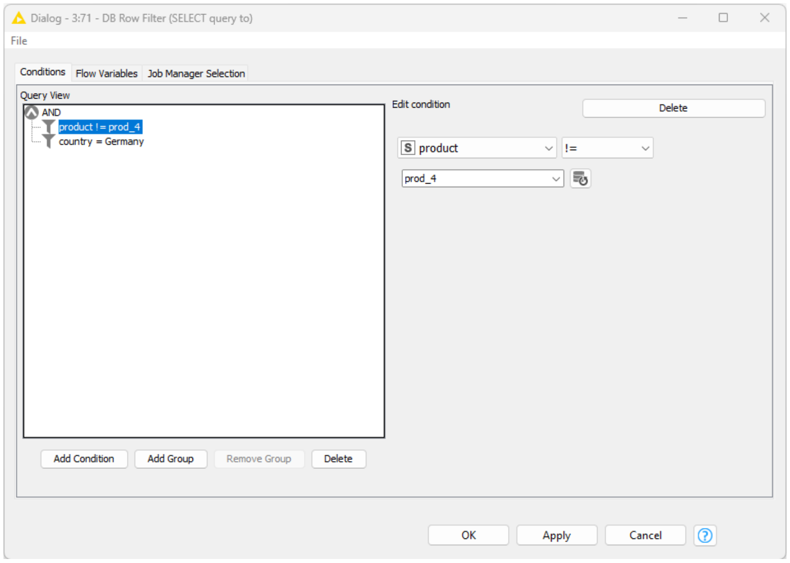
In order to implement the SQL query above, we just need a *DB Row Filter*, to filter out records with "prod_4" and filter in records with country "Germany" and a "DB Column Filter" node to remove the field named "card".

The *DB Row Filter* node customizes a SQL query, according to a filtering criterion, to keep only the matching data records. The filtering criterion consists of one or more conditions grouped together in `AND` or `OR` mode. The configuration window of this node allows you to interactively build the single conditions necessary to obtain the desired filtering criterion.

DB Row Filter

On the left side is a "Query View" panel. Here all implemented filtering conditions are listed. On the right is the condition editor. At the bottom there are buttons to:

- ***Add Condition.*** Add a new condition to the list.
- ***Add Group.*** Group together two or more conditions via a logical operator (AND or OR). Clicking the logical operator at the top of the group in the "Query View" panel allows to select whether AND or OR or to delete tout court the group.
- ***Remove Group.*** Ungroup a set of conditions via their grouping logical operator.
- ***Delete.*** Delete the selected condition from the list.



AND
product != prod_4
country = Germany

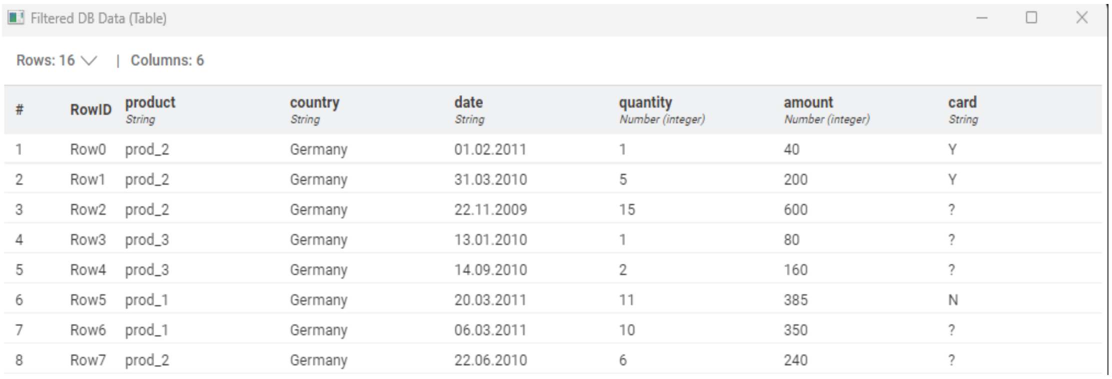
product != prod_4

Figure 2.10. Configuration window of the DB Row Filter node.

We connected a *DB Row Filter* node to the *DB Table Selector* node after the *SQLite Connector* node. The *DB Row Filter* node was set to keep all rows where the column “product” was different (operator “!=”) from the value “prod_4” and the column “country” was equal to “Germany” (operator “=”).

After execution, if we look at the “Filtered DB Data” tab in the Node Monitor, we see no data table, since the node produces a SQL query and no data. However, it is possible to see a temporary preview of the results of the SQL query. Clicking the button “Fetch 100 data rows” fetches the first 100 rows from the database according to the implemented query.

In the “Database_Operations” workflow, a *DB Column Filter* node was also introduced to follow the *DB Row Filter* node and to remove column “card” from the dataset.


Filtered DB Data (Table)

Rows: 16 | Columns: 6

#	RowID	product (String)	country (String)	date (String)	quantity (Number (integer))	amount (Number (integer))	card (String)
1	Row0	prod_2	Germany	01.02.2011	1	40	Y
2	Row1	prod_2	Germany	31.03.2010	5	200	Y
3	Row2	prod_2	Germany	22.11.2009	15	600	?
4	Row3	prod_3	Germany	13.01.2010	1	80	?
5	Row4	prod_3	Germany	14.09.2010	2	160	?
6	Row5	prod_1	Germany	20.03.2011	11	385	N
7	Row6	prod_1	Germany	06.03.2011	10	350	?
8	Row7	prod_2	Germany	22.06.2010	6	240	?

Figure 2.11. Output of DB Row filter node

DB Column Filter

Th DB Column Filter node customizes a SQL query to exclude or include some of the fields in the original data 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.

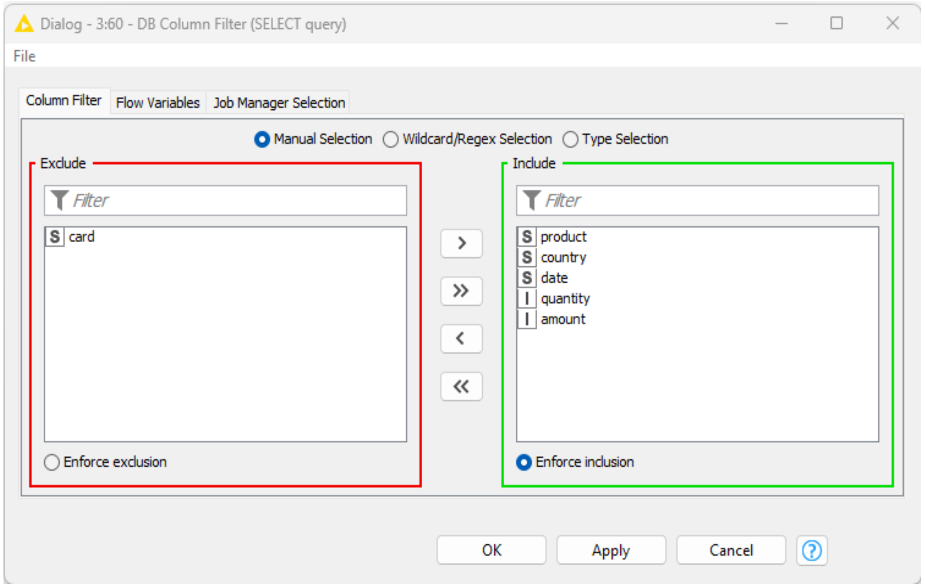
- The columns to be kept are listed in the "Include" frame on the right
- The columns to be removed are listed in the "Exclude" frame on the left

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

A "Filter" 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 add some SQL free code to any existing SQL query, with the node "DB Query". This node takes a SQL statement as input, adds the SQL query written in its configuration window, and exports the total SQL query at the output port.

If you know your way around SQL, but you are not a SQL wizard, you can write smaller SQL queries and pile them up using a sequence of "DB Query" nodes.



[✓ Manual Selection] Wildcard/Regex Selection Type Selection

Exclude:
S card

Include:
S product
S country
S date
I quantity
I amount

Enforce exclusion [✓ Enforce inclusion]

Figure 2.12. Configuration window of DB Column filter node

Let’s now build the same SQL query, the one built before with the DB Row Filter node, using a DB Query node. The “DB Query” node has the task of adding SQL instructions to the SQL query at its input port.

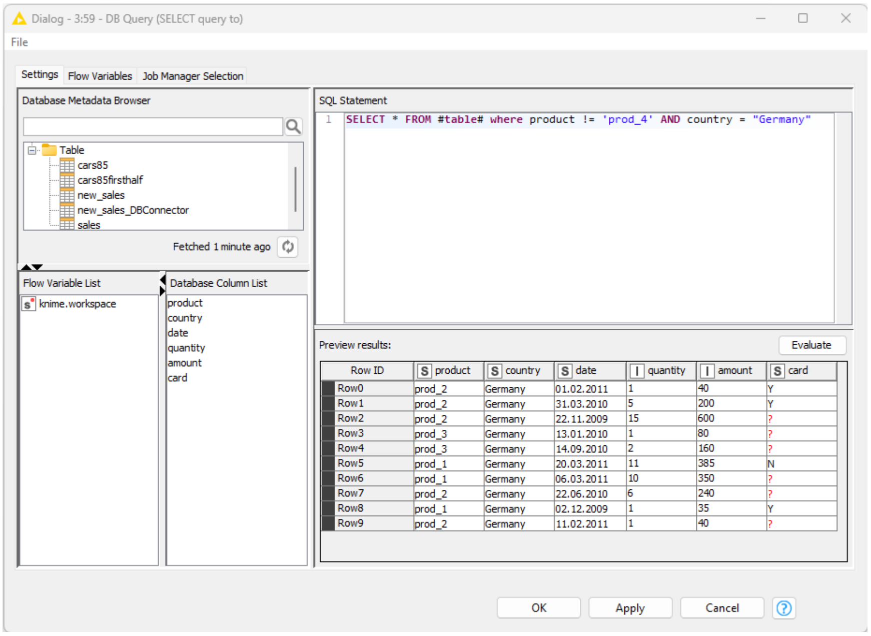
DB Query

On the right, the SQL Statement editor allows to insert the new SQL instruction.

The “Database Metadata Browser” window on the left allows you to browse the database metadata such as the tables and views and their corresponding columns.

The “Database Column List” contains the columns that are available from the connected database table. Double clicking any of the items will insert its name at the current cursor position in the SQL statement area.

The button “Evaluate” allows you to test the syntax and the top 10 results of your query, avoiding surprises after execution.



```
SELECT * FROM #table# where product != 'prod_4' AND country = "Germany"
```

Row ID	product	country	date	quantity	amount	card
Row0	prod_2	Germany	01.02.2011	1	40	Y
Row1	prod_2	Germany	31.03.2010	5	200	Y
Row2	prod_2	Germany	22.11.2009	15	600	?
Row3	prod_3	Germany	13.01.2010	1	80	?
Row4	prod_3	Germany	14.09.2010	2	160	?
Row5	prod_1	Germany	20.03.2011	11	385	N
Row6	prod_1	Germany	06.03.2011	10	350	?
Row7	prod_2	Germany	22.06.2010	6	240	?
Row8	prod_1	Germany	02.12.2009	1	35	Y
Row9	prod_2	Germany	11.02.2011	1	40	?

Figure 2.13. Configuration window of the DB Query node.

Note. The notation #table# is a placeholder for the input table. Do not remove it! Some database software also require the statement “as <new-table-name>” to work. Click the “Evaluate” button to evaluate the SQL statement and return the first 10 rows of the result. If there is an error in the SQL statement, an error message will be shown in the Evaluate window.

In the “Database_Operations” workflow, we introduced a “DB Query” node to implement our target query, including a row filter and a column filter, as described above.

```
SELECT * FROM #table# as tem234 where product != 'prod_4' AND country = “Germany”
```

Note. The “DB Row Filter” node, as the “DB Column Filter” node and the “DB Query” node, do not operate directly on data, they simply customize the input SQL query without executing it. In fact, all database processing nodes do not have a data output port (black triangle), but instead a database output port (brown square). This is because they do not output a data table, but just a SQL query.

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

2.5. Utility Nodes for Databases

If you need to execute a SQL statement on the database before pulling out the data, then the “DB SQL Executor” is your node. This node implements and executes a SQL statement on the database connection available at its input port. It then re-presents the same database connection at the output port for further database operations.

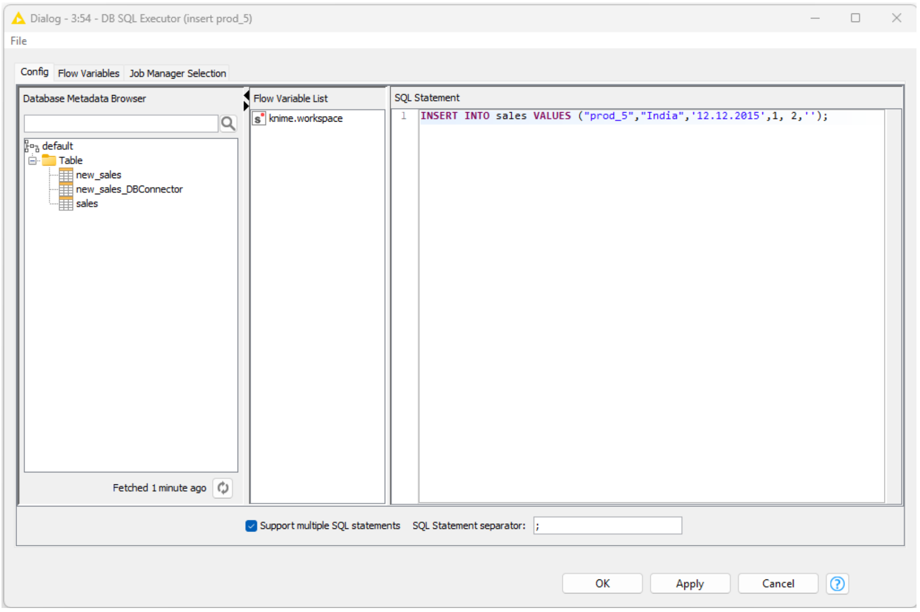
The task of the “DB SQL Executor” node is to allow the execution of any SQL statements on the connected database. While the “DB Query” node produces a SQL statement that gets appended to the input SQL statement after the node execution, the “DB SQL Executor” node creates the SQL statement and already runs it against the selected database during the node execution.

Since later on, in the “Database_Operations” workflow, we would like to use the “DB DELETE (TABLE)” and “DB UPDATE” node, that physically alter the content of the database, we insert a spurious record with product ‘prod_5’ to be removed later. Such insertion is executed via a “DB SQL Executor” node. As we have already said, execution of this node physically executes the SQL statement on the connected database. It is then useful to make changes into the underlying database, such as deletions and insertions, for example. In our example workflow “Database_Operations” we use it to insert spurious records in the database table before proceeding with the next workflow operations.

DB SQL Executor

The “DB SQL Executor” node executes a SQL query on the database connection available at its input port. The most important configuration setting required is the SQL statement. If you want to execute multiple SQL statements, the checkbox “Support multiple SQL statements” has to be checked. In this case, the default SQL statement separator is “;”.

Finally, two last nodes complete the landscape of the database processing nodes: “DB Query Injector” and “DB Query Extractor”.



```
INSERT INTO sales VALUES ("prod_5","India",'12.12.2015',1, 2,'');
```

[✓] Support multiple SQL statements   SQL Statement separator: ;

Figure 2.14. Configuration window of the DB SQL Executor node.

DB Query Injector

The “DB Query Injector” node takes a database connection and a flow variable as input and produces a SQL query at the output port. The flow variable contains the SQL query that will be produced at the output port.

The goal of this node is similar to the goal of the “DB Table Selector” node. The only difference is in the way the new SQL query is defined: the “DB Table Selector” node builds the SQL query in an SQL editor (if “custom query” option is checked) in the configuration window, while the “DB Query Injector” node takes the SQL query from the input flow variable of type String.

Since the SQL query comes from the input port and this is all the settings needed, the node requires no configuration besides the name of the input flow variable containing the SQL query.

DB Query Extractor

The inverse path of the “DB Query Injector” node is implemented by the “DB Query Extractor” node.

The “DB Query Extractor” node connects to a SQL query running on a database connection (brown square) and extract the SQL statement, which is output as a flow variable and as a data table. The SQL query in the flow variable port can feed a “DB Query Injector” node, therefore allowing for extraction and re-execution of complex SQL statements.

The “DB Query Extractor” node also requires no configuration besides the name of the output flow variable.

In workflow “Database_Operations”, we used the “DB Query Extractor” node to extract the SQL query resulting from the cascade of the “DB Row Filter” node and the “DB Column Filter” node. The final result was stored in a flow variable named “sql” and also presented at the output port of the node. The extracted SQL query is the following:

```
SELECT product,country,date,quantity,amount FROM (SELECT * FROM (SELECT * FROM sale
s) table_1133717088 WHERE product != 'prod_4' AND country = 'Germany') table_765435
881
```

Where the outer `SELECT` statement is the result of the “DB Column Filter” node and the inner `SELECT` statement is the result of the “DB Row Filter” node.

2.6. Reading Data resulting from a SQL Query

So far we have connected to the database and built a SQL query that fits our purposes. How do we run the SQL query on the database and get the results into the KNIME workflow? We need a reader node. There are a number of such database reader nodes, each one operating a slightly different task. The simplest – and yet the most powerful one - is the “DB Reader” node.

DB Reader

The “DB Reader” node executes the SQL query at its input port on the database and produces the resulting data table at its output port.

This node does not need any configuration settings. Everything that is needed, such as the database connection and the SQL query to execute, is contained in the input SQL query.

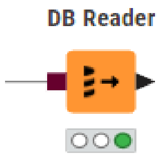


Figure 2.15. The DB Reader node.

Note. The “DB Reader” node has a dark brown square as input port (SQL statement) and a black triangle as output port, i.e. it takes a database connection with a SQL query as input and produces a KNIME data table as output.

We introduced a few “DB Reader” nodes into the “Database_Operations” workflow, each one connected to a different branch. Since all branches though are implementing the same SQL query, either as free code, or created with the help of database processing nodes, or injected from a flow variable, you can compare the SQL results by comparing the output data tables from all these “DB Reader” nodes.

However, sometimes the SQL query at the input port is not pre-defined and therefore not easy to build statically. It can change from one execution run to the next. In this case, we need to parameterize it: either we build it dynamically, for example via a *DB Query* node or we use a “Parameterized DB Query Reader” node.

Let’s start with an example. The SQL SELECT statement extracted in the previous section is equivalent to the following SQL statement:

```
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, change for each execution run, or even unknown.

In cases like this, it can be necessary to use the values of another column as the matching patterns for the `WHERE` condition in the `SELECT` query. In our workflow, for example, we could create a data table with the values “prod_1”, “prod_2” and “prod_3” in one column and use this column’s values as the matching patterns for the `WHERE` condition in the `SELECT` query.

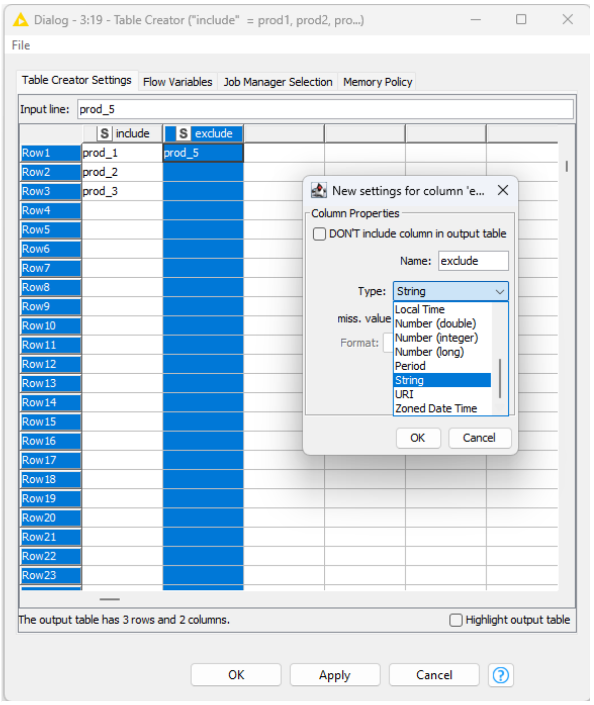
To create a data table from inside a workflow we can use the “Table Creator” node. The “Table Creator” node simulates an Excel Sheet and is often used to create temporary small data sets. We have introduced it into the workflow “Database_Operations” to create two data columns: one named “include” containing “prod_1”, “prod_2”, and “prod_3” and one named “exclude” containing “prod_5”. The idea would be to loop through all values in column “include” and extract the matching records from the database.

Now that we have a data column containing all values we want to match in the database, we need a node to loop on all those values and search for possible matches: the “Parameterized DB Query Reader“ node. This node has two input ports and one output port. At one input port the node expects a data table and at the other input port a database connection. A data table is also produced at the output port.

Table Creator

The “Table Creator” node provides a small editor to manually generate data from inside a workflow. It does not belong to the database category, 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.
- Selecting a column and right-clicking its column header displays a menu to change the column’s properties (name, type, missing values, and format) and allows to insert or delete columns
- Selecting a row and right-clicking its RowID allows to change the RowID’s properties and to insert or remove rows
- “Copy and paste” of cells from Excel sheets is also enabled.


Dialog - 3:19 - Table Creator ("include" = prod1, prod2, pro...)

Input line: prod_5

	S include	S exclude
Row1	prod_1	prod_5
Row2	prod_2	
Row3	prod_3	

New settings for column 'e...

Column Properties

☐ DON'T include column in output table

Name: exclude

Type: String

The output table has 3 rows and 2 columns.

☐ Highlight output table

Figure 2.16. Configuration window of Table Creator node.

The “Parameterized DB Query Reader” node connects to a database, and implements and executes a SQL query like:

```
SELECT * FROM #table# AS “table” WHERE <database-column-name> = <value from $<colum
n-name>$>
```

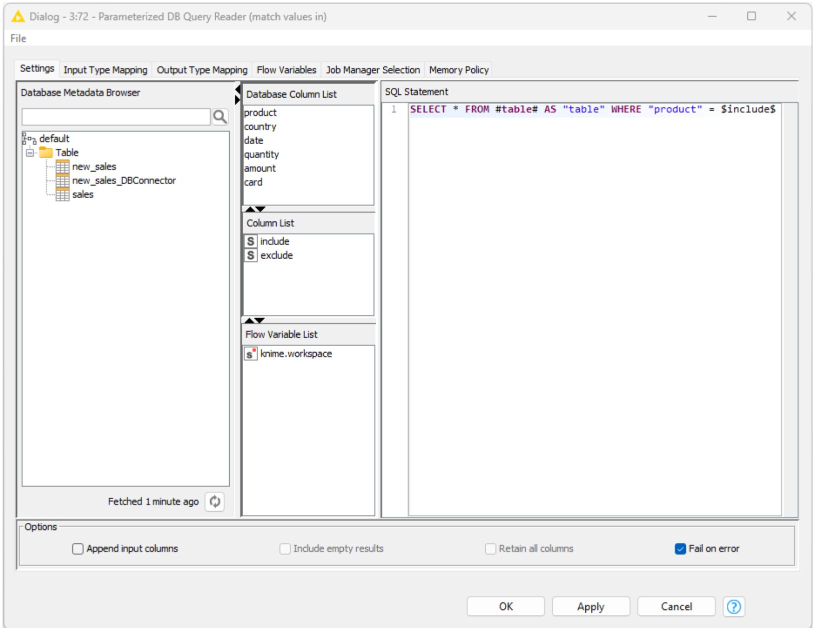
During execution, this SQL query will be executed as many times as the number of values in the selected column of the input data table, where in each of the query $<column-name>$ is substituted with a value from the selected column of the input data table. The resulting data are concatenated together and imported from the database into the KNIME workflow.

Parameterized DB Query Reader

The configuration window contains:

- The editor for the SQL statement

- The list of Database Columns from the input database.
- The list of columns from the input data table whose distinct values will be looped over
- The "Database Metadata Browser" to explore the tables and table structures in the database and help to build the SQL statement in the editor.



```
SELECT * FROM #table# AS "table" WHERE "product" = $include$
```

Options

- [ ] Append input columns
- [ ] Include empty results
- [ ] Retain all columns
- [x] Fail on error

Figure 2.17. Configuration window of Parameterized DB Query Reader.

Again, double-clicking one element here, automatically inserts it in the SQL statement editor.

At the bottom is a series of additional options:

- "Include empty results" to append all empty results to the output table where they will be represented as a Missing Cell.
- "Append input columns" to append the input columns used in the looping
- "Retain all columns" to append all input columns including those not used in the looping
- "Fail on error" to make the node fail when an error is encountered in the SQL query execution

We have seen that, if we can write little SQL, we do not need many extra nodes to build a SELECT query. We have seen, for example, that after connecting to a database using one of the Connector nodes, we can write the SELECT query directly into a "DB Table Selector" node and then read the data into the workflow by means of a "DB Reader" node. We can reduce the number of nodes even further by using a "DB Query Reader" node.

The “DB Query Reader” node has a database connection at the input port and a data table at the output port. This node allows you to execute a SQL statement, including table selection, and import the result into the KNIME workflow. The “DB Query Reader” node performs almost all database operations:

- Reads and executes the SQL statement in its configuration window, including the database table selection
- Pulls the resulting data from the database into a KNIME data table

DB Query Reader

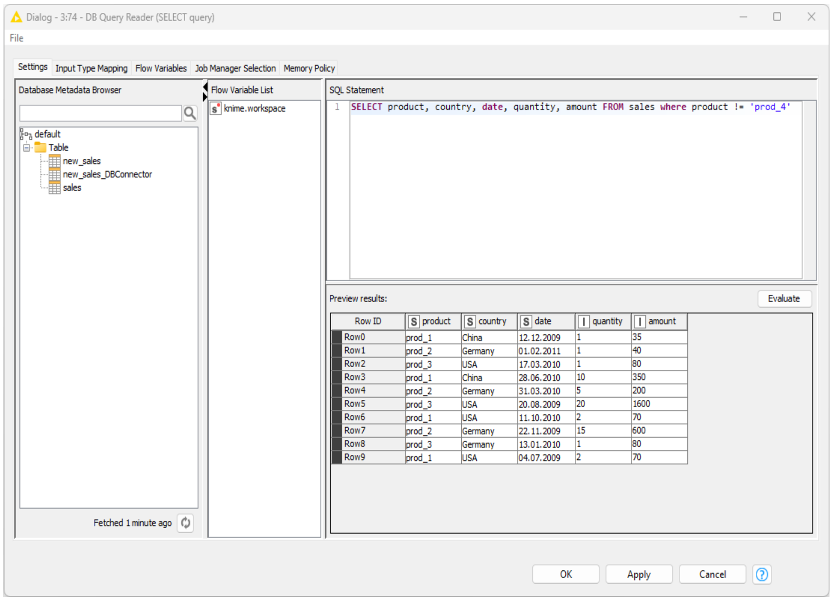
The configuration window of the “DB Query Reader” node only requires the SQL statement to retrieve the data from the database table.

Here the “Database Metadata Browser” panel can help with building the SQL query. Just hit the Refresh button at the bottom of the panel to update the database structure and double-click table and field names to make them appear automatically in the SQL query with the right syntax.

At the bottom right there is the “Evaluate” button where you can evaluate the SQL statement and return the first 10 rows of the result. If there is an error in the SQL statement, then an error message will show in the Evaluate window.

In the lower part of the “Database_Operations” workflow we introduced a “DB Query Reader” node with a database connection at its input port and using:

- The “KCBBookCopy.sqlite” database located in the “KALdata” folder in the “KNIME Explorer” panel
- The SQL statement `SELECT product, country, date, quantity, amount FROM sales where product != 'prod_4'`



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

Row ID	product	country	date	quantity	amount
Row0	prod_1	China	12.12.2009	1	35
Row1	prod_2	Germany	01.02.2011	1	40
Row2	prod_3	USA	17.03.2010	1	80
Row3	prod_1	China	28.06.2010	10	350
Row4	prod_2	Germany	31.03.2010	5	200
Row5	prod_3	USA	20.08.2009	20	1600
Row6	prod_1	USA	11.10.2010	2	70
Row7	prod_2	Germany	22.11.2009	15	600
Row8	prod_3	Germany	13.01.2010	1	80
Row9	prod_1	USA	04.07.2009	2	70

Figure 2.18. The configuration window of the DB Query Reader node.

2.7. Writing Data resulting from a SQL Query

Similarly, the “DB Reader” node, KNIME offers a “DB Writer” node. The “DB Writer” node writes data from a data table input port into a database table. The “DB Writer” node has two output ports, one containing the input table with additional columns providing the writing status for each row and warnings (if any), and the other is the database written table.

DB Writer

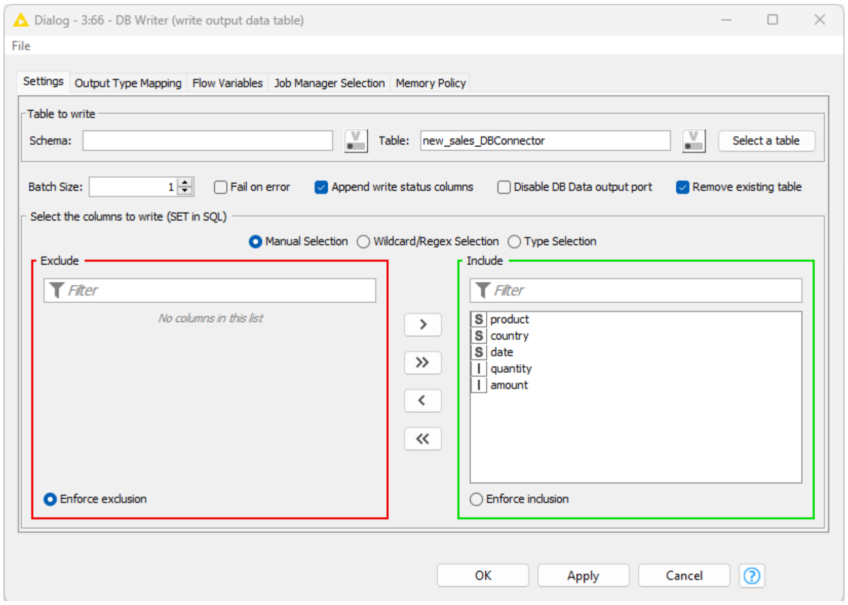
The configuration window requires the following values:

- The name of the table to write into. The “Select Table” button helps with locating the right table, if already existing.
- The “Batch size” with the number of rows to be written in each batch job. For higher performance you should choose a higher number. However, a too high number might need more memory.
- An Exclude/Include panel to set the columns to write or to exclude.

Additional options, to:

- “Remove existing table” in case the table already exists.
- “Append write status columns” for each written row in the output table.
- “Disable DB Data output port” to avoid problems with databases that do not support subqueries.
- “Fail on error” in case an error is encountered when writing on the database.

Sometimes we do not even need to pass through the KNIME Analytics Platform. We connect to the database, we select the table, we build the SQL query, and we execute it against the selected database and just write the results into a table in the same database. This is done with the “DB Connection Table Writer” node.


Dialog - 3:66 - DB Writer (write output data table)

Settings | Output Type Mapping | Flow Variables | Job Manager Selection | Memory Policy

Table to write
Schema:
Table: new_sales_DBConnector

Batch Size: 1
[ ] Fail on error
[✓] Append write status columns
[ ] Disable DB Data output port
[✓] Remove existing table

Select the columns to write (SET in SQL)
[✓ Manual Selection] Wildcard/Regex Selection Type Selection

Exclude: No columns in this list
[✓ Enforce exclusion]

Include:
S product
S country
S date
I quantity
I amount
Enforce inclusion

Figure 2.19. Configuration window of DB Writer node.

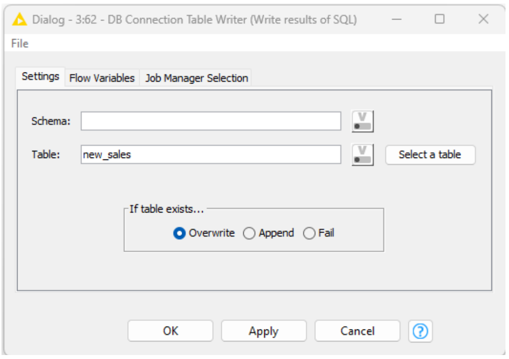
The “DB Connection Table Writer” node reads and executes the SQL query at the input port and writes the resulting data into a database table. At the input port (brown square) we find the database connection with the SQL query. The output port (brown square) contains the database connection with the SQL query used to write data into the table.

DB Connection Table Writer

The only setting required in the configuration window is the name of the database table to which the data is written. The database name is already known to the node since it is part of the database connection parameters.

Another option is to determine how the node should behave if the specified table already exists:

- "Overwrite" overwrites the existing table (i.e. it will be dropped and recreated)
- "Append" adds the new data to the existing table (no new table is created)
- "Fail" makes the node execution fail



Schema:
Table: new_sales
If table exists...
[✓ Overwrite] Append Fail

Figure 2.20. Configuration window of DB Connection Table Writer node.

We introduced one "DB Connection Table Writer" node to write the data resulting from the sequence "DB Row Filter" + "DB Column Filter" into a new database table named "new_sales".

2.8. Database UPDATE and DELETE Commands

There are a couple of database manipulation nodes available in the "Database" category: the "DB Update", "DB Delete (Filter)", "DB Delete (Table)", "DB Merge", "DB Row Manipulator" node, among other things. In this section, we will cover two nodes, the "DB Update", and "DB Delete (Table)" 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 "Select WHERE Columns" panel in their configuration window. The "Select WHERE Columns" panel uses the values in the selected column(s) to match the values in the corresponding database table field(s) for the WHERE clause.

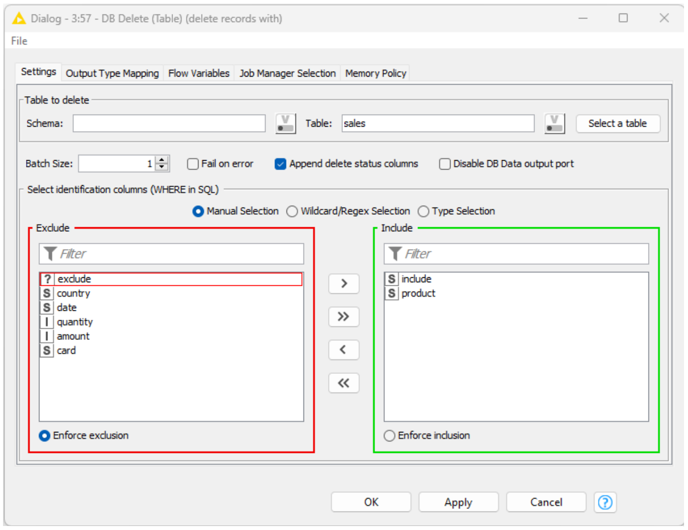
Both nodes have two input ports – one for a data table, the other one for a database connection – and two output ports – one containing the input table with additional columns providing the deletion and update status and one referencing the SQL query. The data table at the input port is used for the `WHERE` and `SET` conditions in the `DELETE` and `UPDATE` statements.

DB Delete (Table)

The “DB Delete (Table)” node deletes all records in the selected table that match the `WHERE` clause.

The `WHERE` clause identifies the table field(s) with the same name as the selected column(s) in the Include/Exclude frame.

All records in the table, with value(s) in the field(s) matching the values in the selected column(s), are deleted during execution through a `DELETE` statement.


Dialog - 3:57 - DB Delete (Table) (delete records with)

Settings | Output Type Mapping | Flow Variables | Job Manager Selection | Memory Policy

Table to delete
Schema: Table: sales

Batch Size: 1 ☐ Fail on error ☑ Append delete status columns ☐ Disable DB Data output port

Select identification columns (WHERE in SQL)
[✓ Manual Selection] Wildcard/Regex Selection Type Selection

Exclude: ? exclude, S country, S date, I quantity, I amount, S card
[✓ Enforce exclusion]

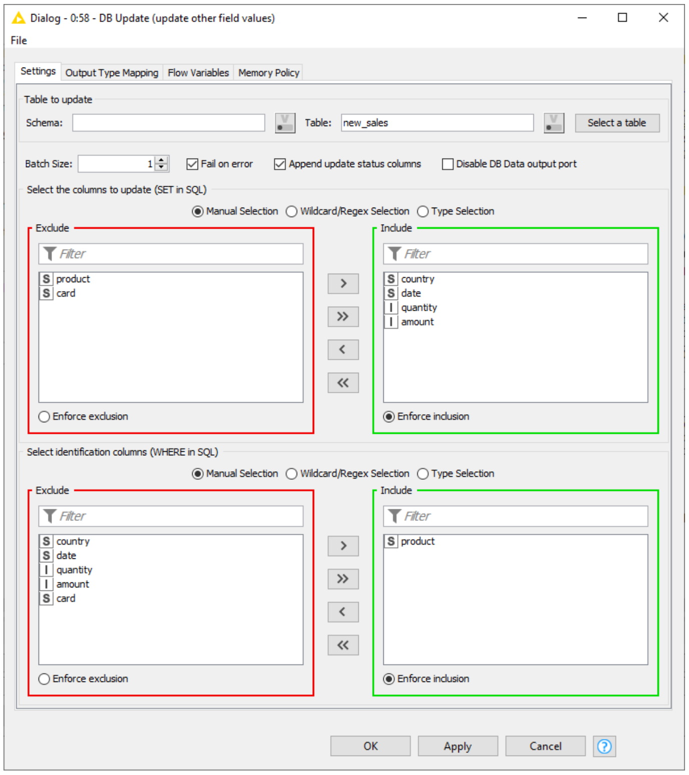
Include: S include, S product
Enforce inclusion

Figure 2.21. Configuration window of the DB Delete (Table) node.

DB Update

The “DB Update” node updates the value(s) of some table field(s) through a `SET WHERE` statement. The `WHERE` clause is built like for the “DB Delete” node. That is, the `WHERE` clause identifies the table field with the same name in the database as the selected column in the “Select identification columns (`WHERE` in SQL)” frame.

Record value(s) of rows identified in the `WHERE` set are then changed, according to the `SET` condition. All field(s) with the same name as the selected column(s) in the “Select the columns to update (`SET` in SQL)” frame, take on the value(s)


Dialog - 0:58 - DB Update (update other field values)

Settings | Output Type Mapping | Flow Variables | Memory Policy

Table to update
Schema: Table: new_sales

Batch Size: 1 ☑ Fail on error ☑ Append update status columns ☐ Disable DB Data output port

Select the columns to update (SET in SQL)
[✓ Manual Selection] Wildcard/Regex Selection Type Selection

Exclude: S product, S card
Enforce exclusion

Include: S country, S date, I quantity, I amount
[✓ Enforce inclusion]

Select identification columns (WHERE in SQL)
[✓ Manual Selection] Wildcard/Regex Selection Type Selection

Exclude: S country, S date, I quantity, I amount, S card
Enforce exclusion

Include: S product
[✓ Enforce inclusion]

Figure 2.22. Configuration window of the DB Update node.

of the `SET` columns, through an `UPDATE` statement.

The final workflow, named “Database_Operations”, is shown in the figure below.

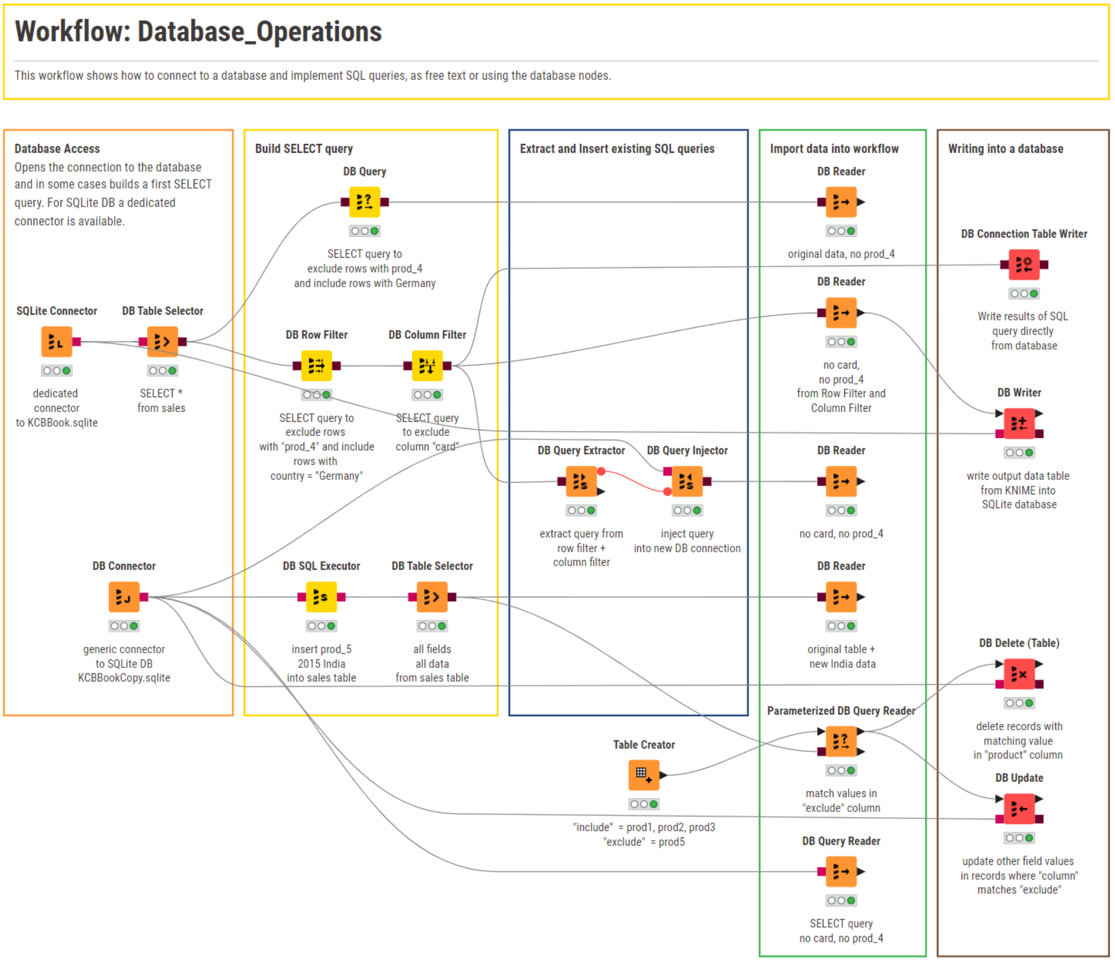

Workflow: Database_Operations
This workflow shows how to connect to a database and implement SQL queries, as free text or using the database nodes.
Database Access
Opens the connection to the database and in some cases builds a first SELECT query. For SQLite DB a dedicated connector is available.
SQLite Connector
dedicated connector to KCBBook.sqlite
DB Table Selector
SELECT * from sales
DB Connector
generic connector to SQLite DB KCBBookCopy.sqlite
Build SELECT query
DB Query
SELECT query to exclude rows with prod_4 and include rows with Germany
DB Row Filter
SELECT query to exclude rows with "prod_4" and include rows with country = "Germany"
DB Column Filter
SELECT query to exclude column "card"
DB SQL Executor
insert prod_5 2015 India into sales table
DB Table Selector
all fields all data from sales table
Extract and Insert existing SQL queries
DB Query Extractor
extract query from row filter + column filter
DB Query Injector
inject query into new DB connection
Table Creator
"include" = prod1, prod2, prod3 "exclude" = prod5
Import data into workflow
DB Reader
original data, no prod_4
DB Reader
no card, no prod_4 from Row Filter and Column Filter
DB Reader
no card, no prod_4
DB Reader
original table + new India data
Parameterized DB Query Reader
match values in "exclude" column
DB Query Reader
SELECT query no card, no prod_4
Writing into a database
DB Connection Table Writer
Write results of SQL query directly from database
DB Writer
write output data table from KNIME into SQLite database
DB Delete (Table)
delete records with matching value in "product" column
DB Update
update other field values in records where "column" matches "exclude"

Figure 2.23. The “Database_Operations” workflow in folder “Chapter2” shows several nodes for database operations. From right to left: Database Connectors (dedicated and generic), in-database processing nodes, utility nodes, readers, and writers.

2.9. Database Type Mapping

The database framework allows you to define rules to map from database types to KNIME types and vice versa. This is necessary because databases support different sets of types. For example, Oracle only has one numeric type with different precision to represent integer and floating-point numbers whereas KNIME uses different types (integer, long, double) to represent them.

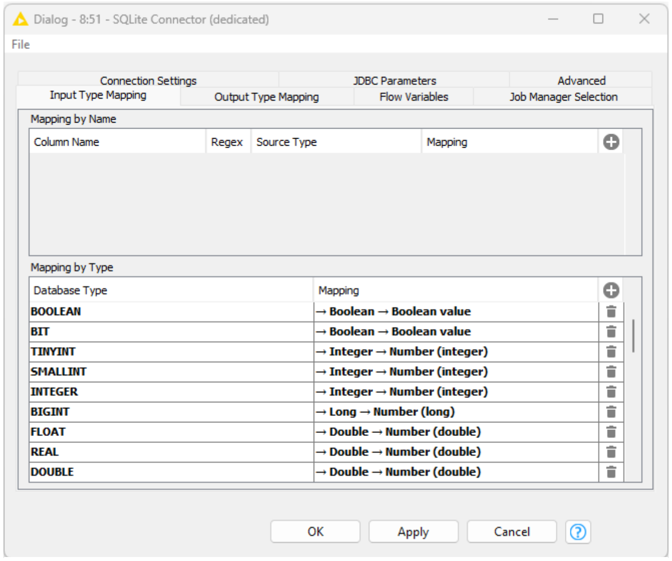
Especially, date and time formats are supported differently across different databases. The zoned Date&Time type that is used in KNIME to represent a time point within a defined time zone is only supported by few databases. With the type mapping framework, you can force KNIME to automatically convert the zoned date time type to String before writing it into a database table and to convert the String back into a zoned date time value when reading it.

The type mapping framework consists of a set of mapping rules for each direction specified from the KNIME Analytics Platform view point:

- ***Output Type Mapping:*** The mapping of KNIME types to database types
- ***Input Type Mapping:*** The mapping from database types to KNIME types

Each of the mapping directions has two sets of rules:

- ***Mapping by Name:*** Mapping rules based on a column name (or regular expression) and type.
- ***Mapping by Type:*** Mapping rules based on a KNIME or database type. All columns of the specified data type are considered.



Mapping by Name

Column Name	Regex	Source Type	Mapping

Mapping by Type

Database Type	Mapping
BOOLEAN	→ Boolean → Boolean value
BIT	→ Boolean → Boolean value
TINYINT	→ Integer → Number (integer)
SMALLINT	→ Integer → Number (integer)
INTEGER	→ Integer → Number (integer)
BIGINT	→ Long → Number (long)
FLOAT	→ Double → Number (double)
REAL	→ Double → Number (double)
DOUBLE	→ Double → Number (double)

Figure 2.24. The "Input Type Mapping" tab in the configuration window of the SQLite Connector node.

The type mapping can be set and altered at various places in a workflow. All database nodes with a KNIME data table as input provide the “Output Type Mapping” tab in the configuration window to map the types of the input KNIME columns to the types of the corresponding database fields.

2.10. Big Data Platforms and MongoDB

Amongst the dedicated connectors, there are a few dedicated to big data platforms, such as Apache Hive, Impala Cloudera, and more. These nodes can be obtained by installing the [KNIME Big Data Extension]().

As for all databases, you can connect to a big data platform using either the dedicated connector or the generic “DB Connector” node. If you use the generic “DB Connector” node, you need to supply the JDBC driver file, usually provided by the big data platform vendor.

As for all databases, after the connection has been established, you can use a “DB Table Selector” node for table selection and a sequence of in-database processing nodes and “DB Query” nodes to build the desired SQL query.

However, writing to a big data platform is not possible through a “DB Writer”, “DB Update”, or “DB Delete” node. For big data platforms we need to use one of the following nodes.

- A “HDFS Connector” node or its related node (“HDFS Connector (KNOX)”). This node supports HDFS, WebHDFS, and HTTPFS. This node can be followed by a “Transfer Files” node to upload data onto the HDFS platform, or to read data from an HDFS platform. Similarly, it is possible to load data directly on a Hive or Impala database using a “DB Loader” node right after this node. All these nodes are available in “IO/Connectors” and “IO/File Folder Utility” categories.

- An alternative way to load data onto a big data platform is to go through Spark and use the Parquet format. These nodes are available in the “Apache Spark” category.

If you have opted for NoSQL databases, such as MongoDB, CouchDB, or NewSQL, in general you need to rely on REST service nodes in the REST Web Services category (see chapter 3) to extract information. Indeed, for now only nodes to read, write, update, save, and remove records in a MongoDB database are available in KNIME Analytics Platform under the “Tools & Services/MongoDB” category.

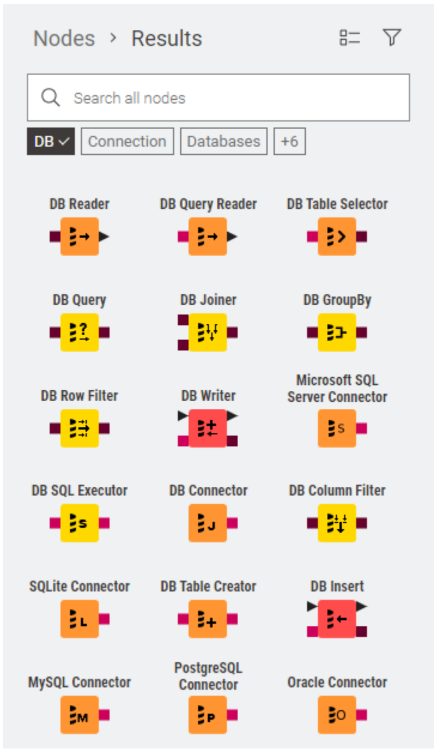


Figure 2.25. Dedicated connector nodes to big data platforms available through the KNIME Big Data extension.

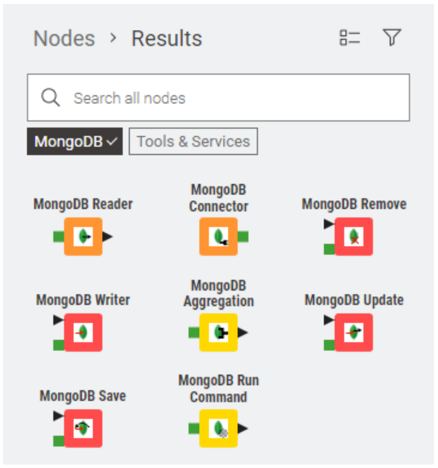


Figure 2.26. The MongoDB nodes in "Tools & Services".

2.11. Exercises

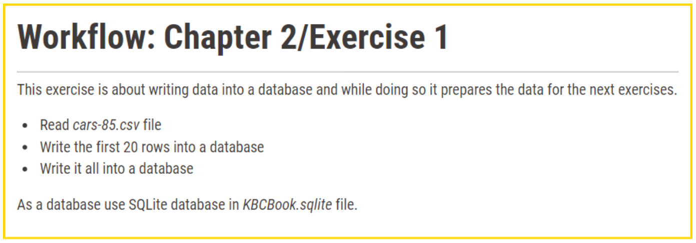
Exercise 1

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 “KALdata” folder);
- Write the data to an SQLite database table named “cars85” in KCBBook.sqlite database;
- Write only the first 20 rows of the data into a table called “cars85firsthalf” in the same KCBBook.sqlite database.

Solution to Exercise 1



Workflow: Chapter 2/Exercise 1

This exercise is about writing data into a database and while doing so it prepares the data for the next exercises.

- Read *cars-85.csv* file
- Write the first 20 rows into a database
- Write it all into a database

As a database use SQLite database in *KBCBook.sqlite* file.

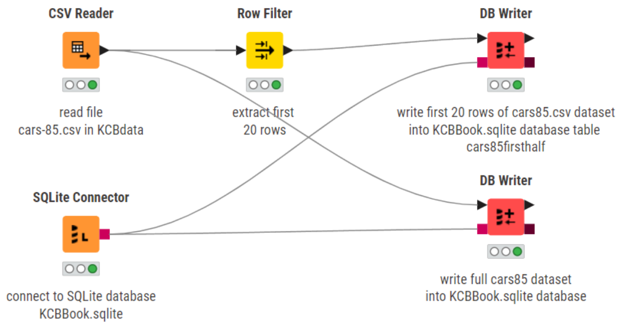

CSV Reader
read file
cars-85.csv in KCBdata
Row Filter
extract first
20 rows
DB Writer
write first 20 rows of cars85.csv dataset
into KCBBook.sqlite database table
cars85firsthalf
SQLite Connector
connect to SQLite database
KCBBook.sqlite
DB Writer
write full cars85 dataset
into KCBBook.sqlite database

Figure 2.27. Exercise 1: The workflow.

The “DB Writer” node is used in conjunction with a “SQLite Connector” node.

Exercise 2

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

- Connect to “KBCBook.sqlite” database and read “cars85” table
- Remove the first two columns: “symboling” and “normalized_losses”
- Only keep rows where “make” is “bmw” or “audi”

Solution to Exercise 2

There are many ways to implement this exercise. We propose four of them:

- “DB Connector” + “DB Table Selector” + in-database processing nodes
- “DB Connector” + “DB Table Selector” + in-database processing nodes for column selection + SQL query for row filter (WHERE)
- “SQLIte Connector” + “DB Reader” + full SQL query

The full SQL SELECT query for the last two options is:

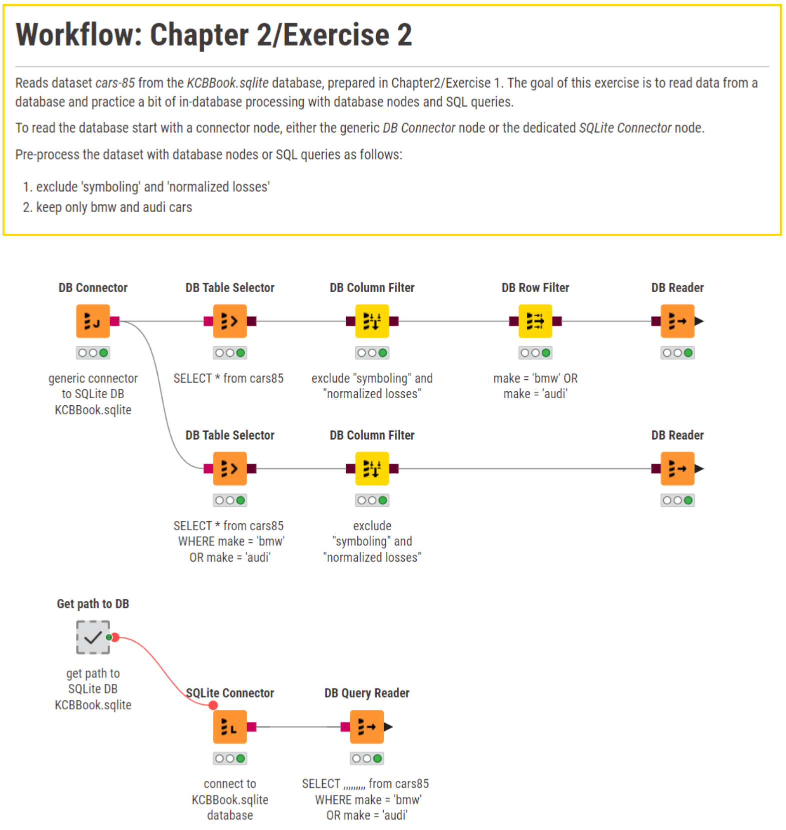
```
SELECT make, fuel_type, aspiration, nr_doors, body_style, drive_wheels, engine_loca
tion, wheel_base, length, width, height, curb_weight, engine_type, cylinders_nr, en
gine_size, fuel_system, bore, stroke, compression_ratio, horse_power, peak_rpm, cit
y_mpg, highway_mpg, price
FROM cars85 where make = 'bmw' OR make = 'audi'
```

The `SELECT` query is a bit tedious to write mainly because of all the columns of the table we want to keep. This same SQL query can be implemented with a combination of “DB Column Filter” and “DB Row Filter” nodes (first approach from the list).

Alternatively, we can use a simple SQL query for the row filter and a “DB Column Filter” node (second approach in the list). The simple SQL query for the row filtering part then takes the shape:

```
SELECT * FROM cars85 where make = 'bmw' OR make= 'audi'
```

All described approaches are shown in the figure below. Notice that the metanode named “Get path to DB” uses a flow variable connection (the red connection) to parameterize the sqlite file path. We will see flow variables later in this book.



Workflow: Chapter 2/Exercise 2

Reads dataset *cars-85* from the *KCBBook.sqlite* database, prepared in Chapter2/Exercise 1. The goal of this exercise is to read data from a database and practice a bit of in-database processing with database nodes and SQL queries.

To read the database start with a connector node, either the generic *DB Connector* node or the dedicated *SQLite Connector* node.

Pre-process the dataset with database nodes or SQL queries as follows:

1. exclude 'symboling' and 'normalized losses'
2. keep only bmw and audi cars

DB Connector
generic connector to SQLite DB KCBBook.sqlite
DB Table Selector
SELECT * from cars85
DB Column Filter
exclude "symboling" and "normalized losses"
DB Row Filter
make = 'bmw' OR make = 'audi'
DB Reader
DB Table Selector
SELECT * from cars85 WHERE make = 'bmw' OR make = 'audi'
DB Column Filter
exclude "symboling" and "normalized losses"
DB Reader
Get path to DB
get path to SQLite DB KCBBook.sqlite
SQLite Connector
connect to KCBBook.sqlite database
DB Query Reader
SELECT ,,,,,,,, from cars85 WHERE make = 'bmw' OR make = 'audi'

Figure 2.28. Exercise 2: Accessing and filtering data with SQL queries and/or in-database processing nodes.

Exercise 3

In the “Chapter2\Exercises” workflow group create a workflow called “Exercise3”. The goal of this exercise is to practice with the Parameterized DB Query Reader node.

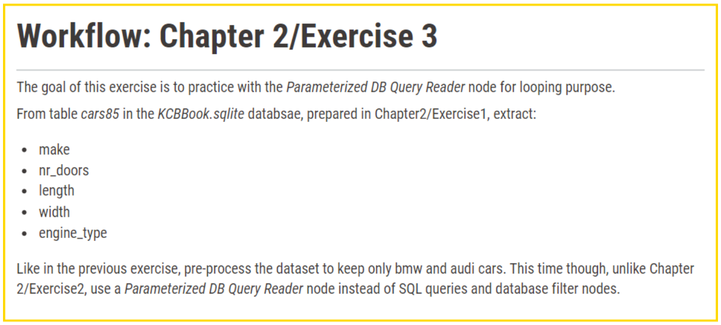
Extract “make”, “nr_doors”, “length”, “width”, and “engine_type” from table “cars-85” in KCBBook.sqlite database, as prepared in exercise1. Like in exercise 2, pre-process the dataset to keep only “bmw” and “audi” cars. This time though use a “Database Looping” node instead of SQL queries and database filter nodes.

Solution to Exercise 3

For the solution of this exercise, we need the list of the values to keep for attribute “make”: “audi” and “bmw”. We build this with a “Table Creator” node in a column named “include-make”.

Afterwards a “Parameterized DB Query Reader” node loops on all the distinct values in column “include-make” and keeps the rows in the database table where “make” matches any of these values.

The “Parameterized DB Query Reader” node is used in standalone mode and in conjunction with a dedicated connector node.



Workflow: Chapter 2/Exercise 3

The goal of this exercise is to practice with the *Parameterized DB Query Reader* node for looping purpose.

From table *cars85* in the *KCBBook.sqlite* databsae, prepared in Chapter2/Exercise1, extract:

- make
- nr_doors
- length
- width
- engine_type

Like in the previous exercise, pre-process the dataset to keep only bmw and audi cars. This time though, unlike Chapter 2/Exercise2, use a *Parameterized DB Query Reader* node instead of SQL queries and database filter nodes.

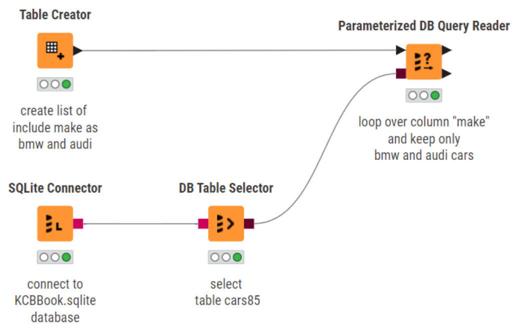


Table Creator
create list of include make as bmw and audi
Parameterized DB Query Reader
loop over column "make" and keep only bmw and audi cars
SQLite Connector
connect to KCBBook.sqlite database
DB Table Selector
select table cars85

Figure 2.29. Exercise 3: The workflow.

Chapter 3: Accessing Information from the Web

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). Accessing information on the web through KNIME Analytics Platform allows us to retrieve and manipulate data in real-time.

An important repository of information for public and private documents resides in the whole Google environment: docs, sheets, drives, and so on. Similar to Excel sheets, KNIME Analytics Platform can also access Google Sheets.

3.1. Accessing Google Sheets

One powerful and simple way to create and edit data online while collaborating with other users in real-time is via Google Sheets. KNIME Analytics Platform offers a full set of Google Sheets dedicated nodes. A number of such nodes are available to connect, read, write, update, and append cells, rows, and columns into private or public Google Sheets. They can be found in the Node Repository under:

- "IO/Connectors/Google"
- "IO/Read"
- "IO/Write"

Let's start with connecting generically to the Google API via the Google Authenticator node. This node provides a generic connection to the Google API, to connect to a Google Sheet or other Google services.

Notice that user credentials to access any Google service are never seen by KNIME. All KNIME keeps in memory is the token derived from the authentication operation on the Google access page. Even the authentication key is not saved anywhere on your hard-disk, unless an explicit instruction has been set.

Note. KNIME Analytics Platform does not store your Google credentials, just the authentication token (if at all). Your Google credentials are entered in the Google sign-in page and not in the KNIME software.

If you do not want to share the Google authentication token, remember to make the workflow forget about it by clicking the button “Clear Selected Credentials” in the configuration window of the Google Authenticator node.

Google Authenticator

This node authenticates to Google services specified in the node settings. Authentication happens in the configuration window.

After choosing the scopes, one must authenticate using the "Login" button. A pop-up will appear that asks the user to grant access to the selected scopes. If you have already authenticated, the "Login" will test the stored credentials in the selected location. Access can be revoked at any time by visiting `myaccount.google.com/permissions`.

By default, the authentication key is kept in memory for that particular instance unless differently specified in the configuration window. the authentication key is kept in memory, the user must authenticate again at each new KNIME session. The authentication key can be stored in a file via the “Custom” option. In this case, a folder must be specified where to store the key. This option is useful in case a shared authentication key is used.

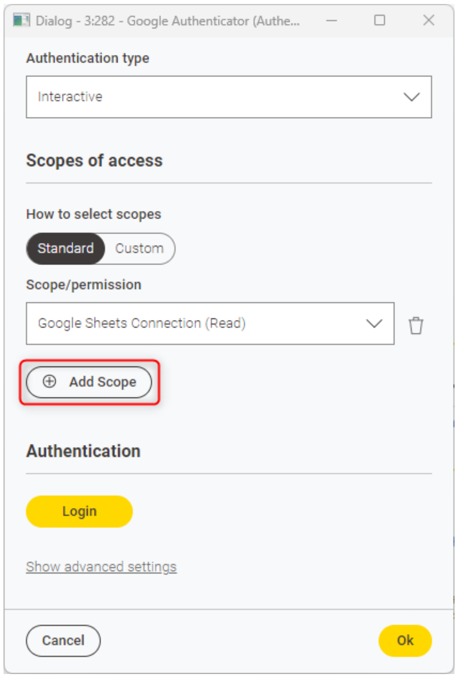


Figure 3.1. Configuration window of the Google Authenticator node.

Only in the case where authentication happened via the API key (Authentication type = API Key), which could be a JSON file or a P12 file stored in some location, the user does not need to re-authenticate the next instance of KNIME Analytics Platform.

The extent of the number of Google services accessible via the authentication key is defined in the lower part of the configuration window under “Scopes of access”. The user can select from the standard scopes or add a custom scope.

Among all available Google services, after authentication, in this example we want to access the Google Sheets service. The node to do that is the “Google Sheets Connector” node.

Google Sheets Connector

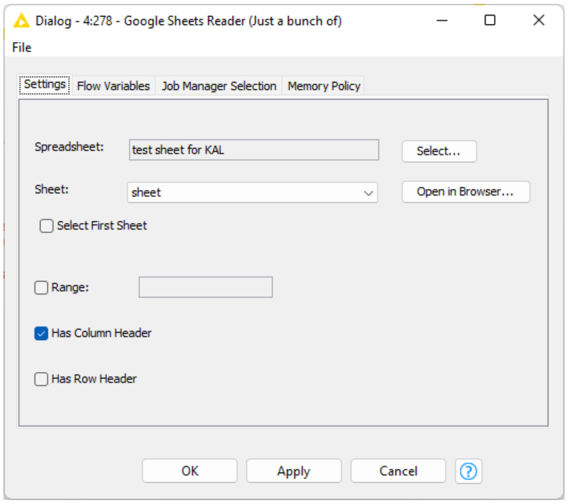
The Google Sheets Connector node creates a connection to Google Sheets, given an existing Google API connection at its input port. Since the whole required information is already contained in the authentication key at its input port, no other configuration setting is required, and no configuration window is provided. It is clear that the input authentication key must be enabled to access the Google Sheets service even in Read/Write mode, if required.

Now, let's read a tab of a Google sheet. To read a tab ("sheet") of a Google spreadsheet, we use the "Google Sheets Reader" node. In the "Google Sheets Reader" node, the spreadsheet can be selected among all spreadsheets with access permission.

Inside the selected sheet, you can specify a specific range of cells to be read. The range must be entered in A1 notation (e.g., "A1:G10"). For more information about A1 notation visit: https://developers.google.com/sheets/api/guides/concepts#a1_notation.

Google Sheets Reader

The Google Sheets Reader accesses data from one tab "sheet" of a Google spreadsheet. The following settings can be configured:



Spreadsheet: test sheet for KAL

Sheet: sheet

☐ Select First Sheet

☐ Range:

☑ Has Column Header

☐ Has Row Header

Figure 3.2. Configuration window of the Google Sheets Reader node.

- ***Spreadsheet:*** Selects one spreadsheet from the list of spreadsheets available on Google Drive. Clicking on button "Select…" opens a dialog with the list of available spreadsheets from Google drive. If a document doesn't appear in this list, make sure that you have permissions to access it and that you have opened it at least once within a browser to associate it with your Google account.
- ***Sheet:*** Selects the sheet from the spreadsheet that should be read. Available sheets can be selected from the drop-down menu. The button "Open in Browser…" opens the selected spreadsheet in the browser. This is useful to ascertain whether that is the sheet of interest.

- ***Select First Sheet:*** When selected, the first sheet of the spreadsheet will be read instead of the one selected from the drop-down menu.
- ***Range:*** The range of cells that should be read from the sheet can be specified here in A1 notation. (E.g. "A1:G20").
- Two checkboxes that specify if the sheet includes column names and row ids, respectively.

To translate these notions into a practical workflow, we built a workflow in “Chapter3” named “Access_GoogleSheets”. This workflow accesses the sheet named “sheet” from a public Google spreadsheet named “Test Sheet for KAL”. This sheet contains just randomly generated numbers. We first connected to the Google Spreadsheet using the “Google Sheets Connector” and retrieved the data from the Spreadsheet “Test Sheet for KAL” and the Sheet named “sheet” using the “Google Sheets Reader” node.

There are more nodes covering different functionalities on Google Sheets. We have not introduced these nodes in the example workflow, but we will cover them briefly in the rest of this section.

For example, the node “Google Sheets Appender” adds a new sheet to an existing spreadsheet in Google Sheets. The configuration settings are very similar to the settings of the “Google Sheets Reader” node, plus a few additional writing settings.

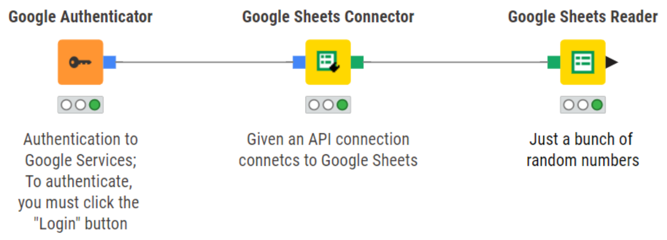
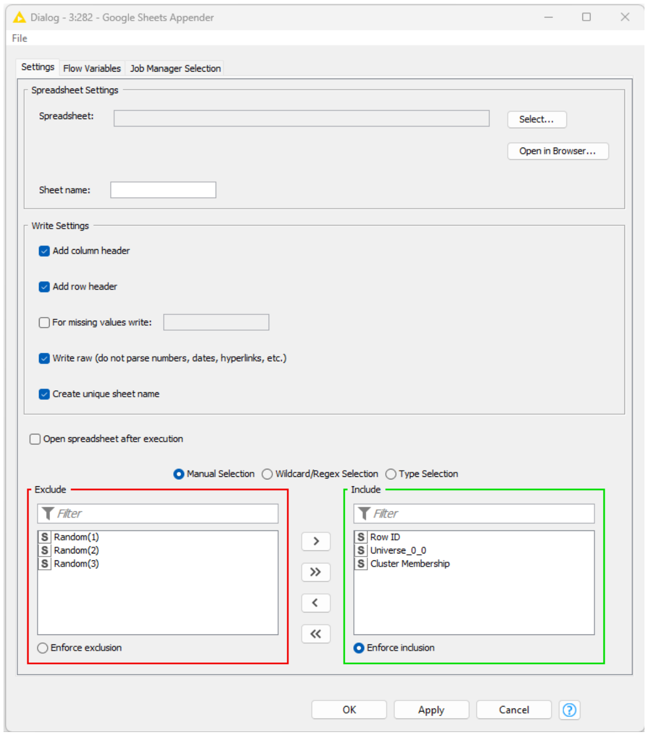

Google Authenticator
Google Sheets Connector
Google Sheets Reader
Authentication to Google Services; To authenticate, you must click the "Login" button
Given an API connection connetcs to Google Sheets
Just a bunch of random numbers

Figure 3.3. Workflow "Access_GoogleSheets" accessing and reading content from a Google Spreadsheet.

Google Sheets Appender

- ***Spreadsheet*** and ***Sheet name*** require the selection of the spreadsheet among one of the available spreadsheets on Google drive and the selection of the sheet inside the spreadsheet.
- ***Add column/Add row header:*** Specifies, whether the column names should be written in the first row and whether the row ID's should be written in the first column of the spreadsheet.
- ***For missing values write:*** By selecting this option, you can specify a string you want to substitute for missing values. If the option is left unchecked, the cells with missing values remain empty.



Spreadsheet Settings
Spreadsheet:
Sheet name:

Write Settings
[✓] Add column header
[✓] Add row header
[ ] For missing values write:
[✓] Write raw (do not parse numbers, dates, hyperlinks, etc.)
[✓] Create unique sheet name

[ ] Open spreadsheet after execution

[✓ Manual Selection] Wildcard/Regex Selection Type Selection

Exclude: Random(1), Random(2), Random(3) — Enforce exclusion
Include: Row ID, Universe_0_0, Cluster Membership — [✓ Enforce inclusion]

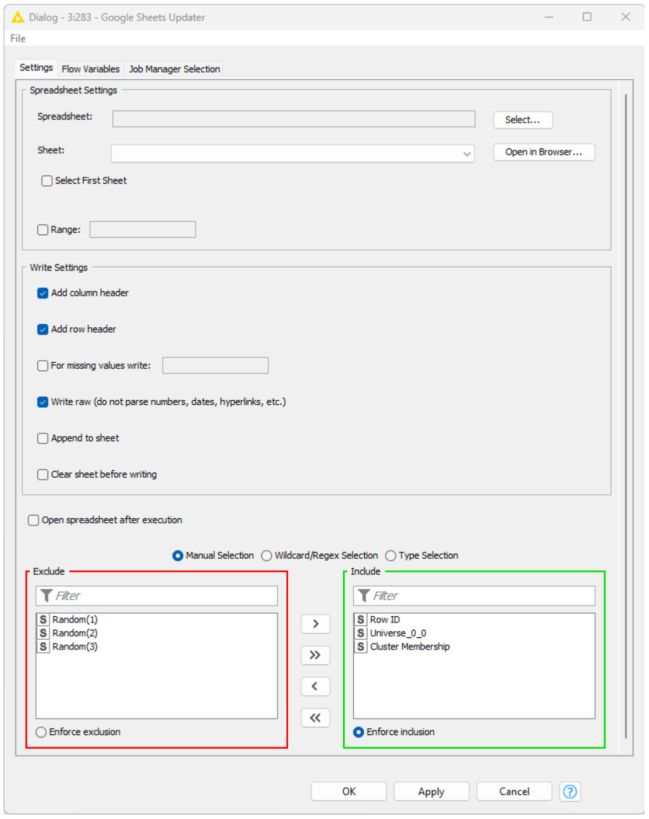
Figure 3.4. Configuration window of the Google Sheets Appender. node.

- ***Write Raw (do not parse numbers, dates, hyperlinks, etc):*** Values are written into the spreadsheet as-is ("raw"), i.e. they will not be parsed. For example, strings like *=hyperlink("example.com", "example")* will be parsed to hyperlinks if this option is unchecked.
- ***Create unique sheet name:*** The node will create a unique sheet name based on the given sheet name. (Example: Should 'SheetOne' already exist, the unique sheet name will be 'SheetOne (#1)')
- ***Open spreadsheet after execution:*** Opens the spreadsheet after it has been written successfully. The spreadsheet will be opened in the system's default browser.
- ***Exclude/Include columns:*** Uses a classic Include/Exclude frame to select the columns that will be written to the sheet file.

Instead of appending a new sheet, we might want to update an existing one. For that, we use the "**Google Sheets Updater**" node.

Google Sheets Updater

The "Google Sheet Updater" node enables us to write an input data table to an existing Google sheet. It can overwrite some or all of the content of a sheet or append to the content of a sheet. The dialog settings are the same as in the "Google Sheets Appender" node, with a few additional options:



Dialog - 3:283 - Google Sheets Updater

File

Settings | Flow Variables | Job Manager Selection

Spreadsheet Settings

Spreadsheet: [ ] Select...

Sheet: [ ] Open in Browser...

☐ Select First Sheet

☐ Range: [ ]

Write Settings

☑ Add column header

☑ Add row header

☐ For missing values write: [ ]

☑ Write raw (do not parse numbers, dates, hyperlinks, etc.)

☐ Append to sheet

☐ Clear sheet before writing

☐ Open spreadsheet after execution

● Manual Selection ○ Wildcard/Regex Selection ○ Type Selection

Exclude	Include
Random(1)	Row ID
Random(2)	Universe_0_0
Random(3)	Cluster Membership
○ Enforce exclusion	● Enforce inclusion

OK | Apply | Cancel

Figure 3.5. Configuration window of the Google Sheets Updater node

- ***Range:*** If only part of the content of a sheet is to be overwritten, the relevant range can be specified in A1 notation. The size of the input table must not exceed the size of the selected range, otherwise execution will fail.
- ***Append to sheet:*** When this option is selected, the data table content will be appended to the selected sheet. This means we append data to an existing sheet and do not add a new sheet into an existing spreadsheet as we did with the "Google Sheets Appender" node.
- ***Clear sheet before writing:*** When this option is selected, the sheet or the selected range of the sheet will be cleared before writing. This deletes the content in the specified sheet/range.

If you want to write the input data table to a new Google Sheets spreadsheet instead of overwriting or appending data to an existing sheet, you can use the "Google Sheets Writer" node.

Taken together, these nodes enable us to access and manipulate data in Google Sheets through a KNIME workflow.

Google Sheets Writer

The "Google Sheets Writer" node enables us to write data into a new Google sheet.

The configuration settings are the same as for the "Google Sheets Updater" node.

However, ***Spreadsheet*** and ***Sheet name*** cannot be selected from a dropdown menu of existing spreadsheets but must be entered manually.

3.2. Accessing REST Services

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.

Nowadays a number of web services are available, such as financial estimations, weather reports, chemistry related data. This means that in many data related fields, tools and data have been made 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 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 very powerful way to do this is using the REST Web Services nodes.

Web services that follow the REpresentational State Transfer (REST) architectural principles[3] offer a robust and flexible way to access and manipulate textual representations of web resources. Since version 3.2, KNIME Analytics Platform provides a category, named "REST Web Services", which contains nodes that enable the user to interact with REST services.

The "REST Web Services" category contains 6 nodes - "GET Request", "POST Request", "PUT Request", "DELETE Request", "PATCH Request", and "Webpage Retriever" – to deal with REST operations. The "REST Web Services" category gets installed onto the "Node Repository" panel through the REST Client Extension. As to install any KNIME Extension Package, select the "go to Info page" icon at the top-right of the workbench:

- Select "Install Extensions"

[3] *Roy T. Fielding, Richard N. Taylor, "Principled design of the modern Web architecture", Journal ACM Transactions on Internet Technology (TOIT) Vol. 2 Issue 2, 115-150 (2002)*

- From the “Available Software” window, in the top box containing “type filter text”, type “web”
    - Inside “KNIME & Extensions” select the package called “KNIME REST Client Extension”
    - Click “Next”
    - Follow the installation instructions

After the package has been successfully installed, you should find a category named “REST Web Services” in the “Tools & Services” category in the “Node Repository” panel.

In order to show the potentialities of this “REST Web Services” nodes, we created two empty workflows in “Chapter3” folder and we named them “GET Request” and “POST Request”.

The “GETRequest” workflow accesses the JSONPLaceholder API to retrieve fake post content from fake user IDs. JSONPlaceholder API is a development test utility for developers to test their requests to REST APIs. The request takes the form:

```
https://jsonplaceholder.typicode.com/posts?userId=<userID>
```

The value of parameter userID has to be passed in the request message and the list of posts with their content is returned with the response message in JSON format.

The list of required userID is created in a Table Creator node. Then the REST request is built using a “String Manipulation” node as:

```
join("https://jsonplaceholder.typicode.com/posts?userId=", $userID$)
```

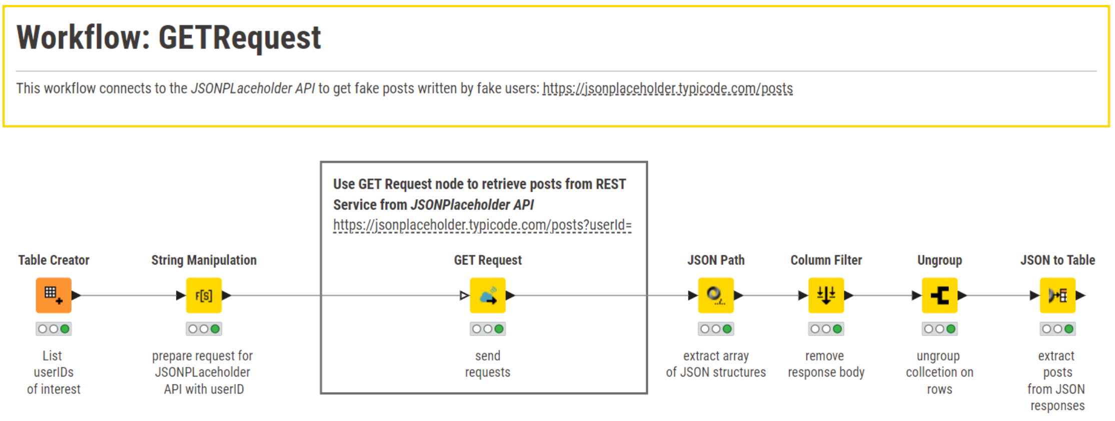
where $userID$ comes from the userID input column and contains the userID values.

The "GET Request" node is introduced to submit the GET request to the REST server. The "GET Request" node has 6 tabs in its configuration window:

- "Connection" contains all REST service settings, including the URL source;
- "Authentication" sets the authentication credentials, if required;
- “Proxy” allows to set proxy settings;
- “Error Handling” defines how to handle in case errors happen;
- “Request Headers” allows for customization of the request headers;
- “Response Headers” allows to interpret customized response headers.

When executed, the “GET Request” node sends all requests in batches of N, as defined in the “Concurrency” setting, to the REST service specified in the URL parameter and presents the subsequent response at its output port. The size of N can be specified in the Connection Settings under Concurrency (Number of concurrent requests). The JSONPlaceholder API REST service does not require any authentication and therefore we selected option “None” in the “Authentication” tab in the configuration window of the GET Request node.

The response is usually encapsulated in an XML or JSON structure, which needs to be interpreted to extract the value(s) we were looking for, in our case the body and title values. If the response is returned in XML format, you probably need to use the “XPath” node from the “XML” category in the “Node Repository” to retrieve the values you are interested in. If the response is returned in JSON format, you need to resource to one of the nodes in the “JSON” category, including “JSON Path”, “JSON To XML”, and especially “JSON to Table”. In our example, we used “JSON Path” and “JSON to Table” nodes. JSON Path is used to extract the array of posts from the response JSON structure; while the JSON To Table node does all the remaining parsing work for us, exposing at the output port all values contained in the input JSON structure.



Workflow: GETRequest

This workflow connects to the *JSONPLaceholder API* to get fake posts written by fake users: https://jsonplaceholder.typicode.com/posts

Use GET Request node to retrieve posts from REST Service from *JSONPlaceholder API*
https://jsonplaceholder.typicode.com/posts?userId=

Table Creator
List userIDs of interest
String Manipulation
prepare request for JSONPLaceholder API with userID
GET Request
send requests
JSON Path
extract array of JSON structures
Column Filter
remove response body
Ungroup
ungroup collcetion on rows
JSON to Table
extract posts from JSON responses

Figure 3.6. Workflow "GETRequest".

GET Request: “Configuration Settings” Tab

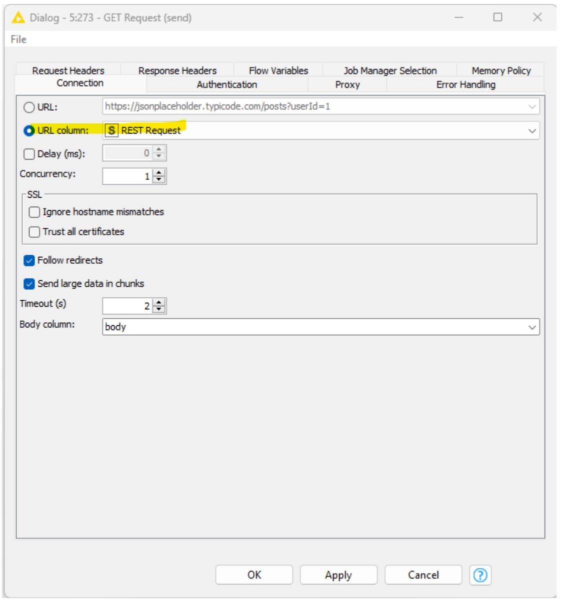
GET request URL(s) that are sent to a REST service is (are) identified either through one single manually inserted fixed URL or through a list of dynamic URLs in an input data column.

Two options in the “Connection” tab respectively enable these two modes: “URL” and “URL column”.

“Delay (ms)” and “Concurrency” both deal with multiple requests. “Delay (ms)” specifies a delay between two consecutive requests, e.g. in order to avoid overloading the web service. “Concurrency” allows for N parallel GET requests to be sent, if the REST service allows it.

The flags in the SSL section push for a higher tolerance in security when checking the REST host SSL certificates. When enabled, even if some SSL certificates are not perfect the returned response is accepted.

The flag “Follow Redirects” forces the GET request to be redirected, if so specified in the REST service.



URL: https://jsonplaceholder.typicode.com/posts?userId=1
[✓ URL column: S REST Request]
Delay (ms): 0
Concurrency: 1
SSL
Ignore hostname mismatches
Trust all certificates
[✓ Follow redirects]
[✓ Send large data in chunks]
Timeout (s): 2
Body column: body

Figure 3.7. Get Request node configuration: the "Connecton" tab.

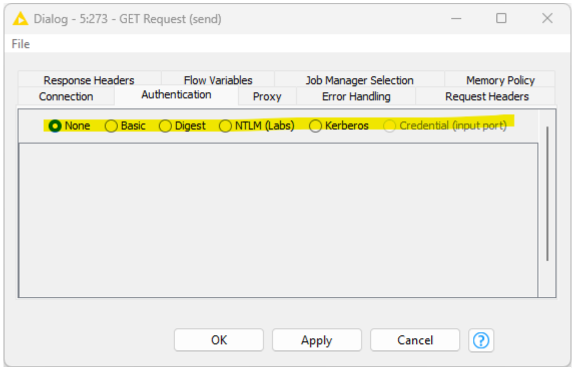
“Timeout” sets the number of seconds to wait before declaring a connection timed out.

“Body column” contains the name of the response column in the output data table.

GET Request: The Other Tabs

The “Authentication” tab sets the authentication credentials, if required by the REST service.

The node supports a number of authentication methods, e.g. None, BASIC, DIGEST, NTLM (Labs), or Kerberos. Username and password can be provided manually or via the *Credentials Widget* node. As for the database nodes, workflow credentials are automatically encrypted, while manual typing of username and password requires the definition of a Master Key for the encryption process. The Master Key can be set in the “Preferences” page.



[✓ None] Basic Digest NTLM (Labs) Kerberos Credential (input port)

Figure 3.8. GET Request node configuration: the other tabs.

In the “Proxy” tab, you can disable proxy, or set up your own proxy, if required.

When something fails in the REST service, the output data table will contain the rows with the status of the requests. In some cases, we might desire the workflow to just stop and signal the REST error. In this case, under the “Error Handling” tab we can enable “Fail node execution” or “Output missing value” depending on the type of errors.

Every request being shipped off to a REST service may contain a header. By default, in the “GET Request” node, requests are shipped with no header. However, custom request headers can be defined in the “Request Headers” tab. A request header consists of many parameters and every parameter consists of 3 fields: key, value, and kind. Three request headers have been pre-loaded as templates: none, generic REST headers, and web page related headers.

The response object that comes back can also contain headers, at least the status of the request and the content-type. Other headers can be imported if the flag named “Extract all Headers” at the very top of the “Response Headers” tab is enabled. If you prefer not to extract all headers from the response, but just some, you can set the key names one by one in the tab table. The value associated with the keys will be extracted from the response object and placed in a data column of the output table. The name of this data column is also set in the “Response Headers” tab.

JSON Path

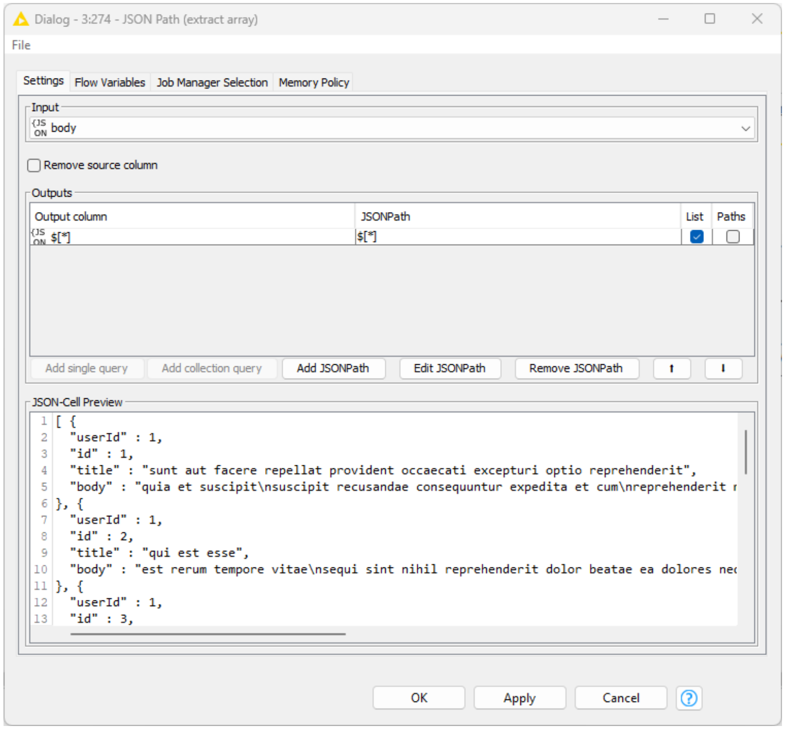
The "JSON Path" node parses a JSON structure via a custom query. In order to do that, it needs the following settings in the configuration window:

- The input data column containing the JSON structures to parse
- The editor for the custom query.
- A flag to optionally remove the original JSON column from the output table

In the lower part of the configuration window a preview frame is available.

It is possible to interactively click JSON items in the preview frame and automatically define the JSON custom parsing query.

The buttons above the preview frame allow to add/edit/remove the resulting custom JSON parsing query into the editor.



Input: body

☐ Remove source column

Outputs

Output column	JSONPath	List	Paths
$[*]	$[*]	✓	☐

JSON-Cell Preview

```
[ {
  "userId" : 1,
  "id" : 1,
  "title" : "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body" : "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit r
}, {
  "userId" : 1,
  "id" : 2,
  "title" : "qui est esse",
  "body" : "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor beatae ea dolores nec
}, {
  "userId" : 1,
  "id" : 3,
```

Figure 3.9. JSON Path node configuration.

JSON to Table

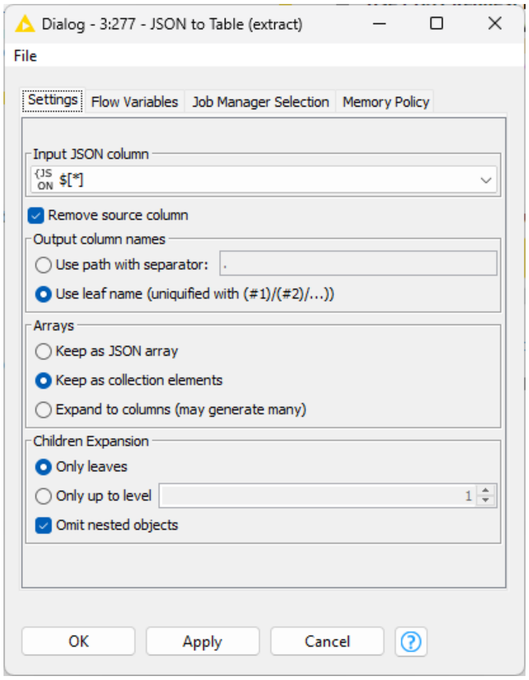
The "JSON to Table" node parses a JSON structure, extracts all values, and puts them in a KNIME data table at the output port. In order to do that, it needs the following settings in the configuration window:

- The input data column containing the JSON structures to parse
- Customized or default names for the data columns resulting from the parsing operation.
- The structure of the resulting columns (it is good practice to keep the result as a collection column, since we do not know a priori how many columns will result from this blind operation).
- The expansion level of the JSON structure (again, it is good practice to expand as little as possible the original structure, since we do not know a priori how many levels it will contain).

The result at the output port is a KNIME data table, containing the values from the REST response, including the REST URL Request and the Request Status.

The same post request to the JSONPlaceholder API REST service could be sent via POST Request. “REST Web Services” category offers the “POST Request” node to send POST requests to a service.

As a practical example to illustrate the functionality of the “POST Request” node, we altered the previous workflow to send a POST request to add a new ID to the object in the request. The POST request is the same as the previous GET Request. The associated functionality is different. Thus, instead of having a “GET Request” node, we introduced a “POST Request” node.


Dialog - 3:277 - JSON to Table (extract)

File

Settings | Flow Variables | Job Manager Selection | Memory Policy

Input JSON column: $[*]

[✓] Remove source column

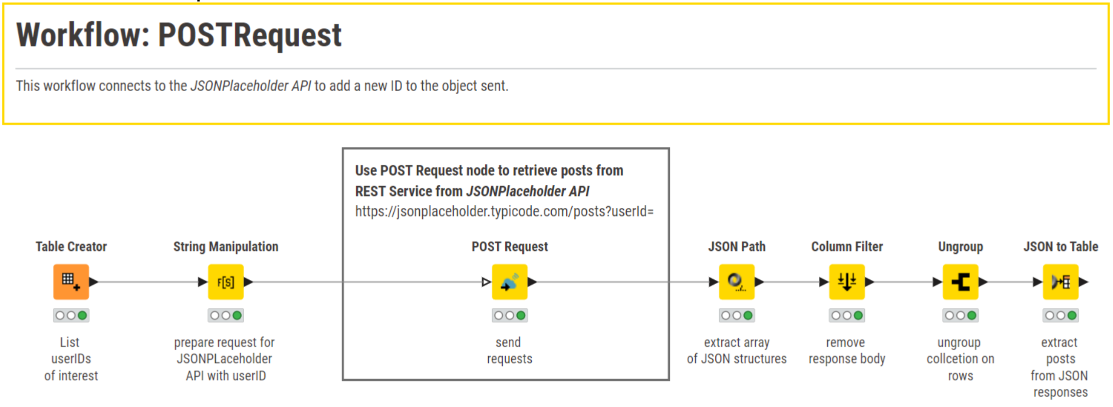
Output column names
- ( ) Use path with separator: .
- (●) Use leaf name (uniquified with (#1)/(#2)/...))

Arrays
- ( ) Keep as JSON array
- (●) Keep as collection elements
- ( ) Expand to columns (may generate many)

Children Expansion
- (●) Only leaves
- ( ) Only up to level 1
- [✓] Omit nested objects

Figure 3.10. JSON to Table node configuration.

The “POST Request” node submits a POST


Workflow: POSTRequest

This workflow connects to the *JSONPlaceholder API* to add a new ID to the object sent.

Use POST Request node to retrieve posts from REST Service from *JSONPlaceholder API*
https://jsonplaceholder.typicode.com/posts?userId=

Figure 3.11. Workflow "POSTRequest".

request to the REST server. The "POST Request" node has 5 tabs in its configuration window:

- "Connection" contains all REST service settings, including the URL source;
- "Authentication" sets the authentication credentials, if required;
- “Proxy” allows to set proxy settings;
- “Error Handling” defines how to handle in case errors happen;

- "Request Header" allows for customization of the request headers;
- "Request Body" to send the data to be transmitted through the POST request;
- "Response Header" allows to interpret customized response headers.

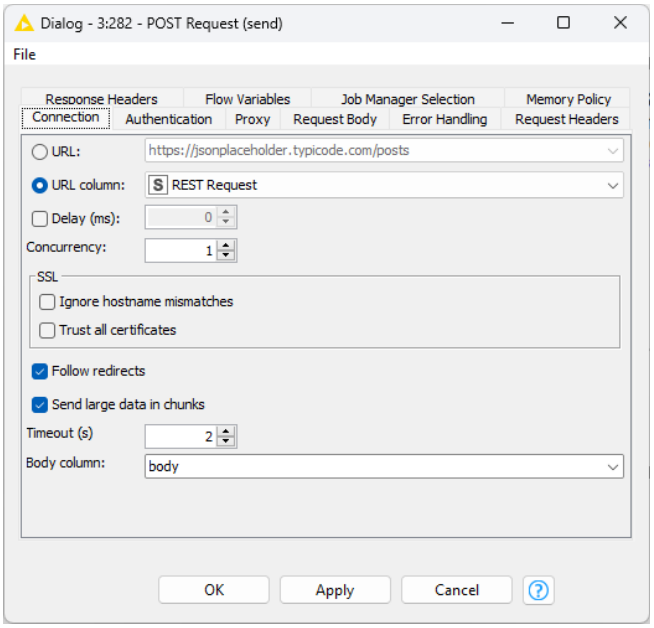
Similarly to the "GET Request" node, when executed, the "POST Request" node sends all requests in batches of N, as defined in the "Concurrency" setting, to the REST service specified in the URL parameter and presents the subsequent response at its output port. Depending on the response format, XML or JSON, you can use either an "XPath" node or a "JSON to Table" node. Here we used again a "JSON Path" and a "JSON to Table" node.

POST Request: "Connection" Tab

POST request URL(s) that are sent to a REST service is (are) identified either through one single manually inserted fixed URL or through a list of dynamic URLs in an input data column.

Two options in the "Connection Settings" tab respectively enable these two modes: "URL" and "URL column".

"Delay (ms)" and "Concurrency" both deal with multiple requests. "Delay (ms)" specifies a delay between two consecutive requests, e.g. in order to avoid overloading the web service. "Concurrency" allows for N parallel GET requests to be sent, if the REST service allows it.


Dialog - 3:282 - POST Request (send)

File

Response Headers | Flow Variables | Job Manager Selection | Memory Policy
Connection | Authentication | Proxy | Request Body | Error Handling | Request Headers

URL: https://jsonplaceholder.typicode.com/posts
[✓ URL column: REST Request]
Delay (ms): 0
Concurrency: 1
SSL
Ignore hostname mismatches
Trust all certificates
[✓ Follow redirects]
[✓ Send large data in chunks]
Timeout (s) 2
Body column: body

OK | Apply | Cancel

Figure 3.12. POST Request node configuration: the "Configuration Settings" tab.

The flags in the SSL section push for a higher tolerance in security when checking the REST host SSL certificates. When enabled, even if some SSL certificates are not perfect the returned response is accepted.

The flag "Follow Redirects" forces the GET request to be redirected, if specified so in the REST service.

"Timeout" sets the number of seconds to wait before declaring a connection timed out.

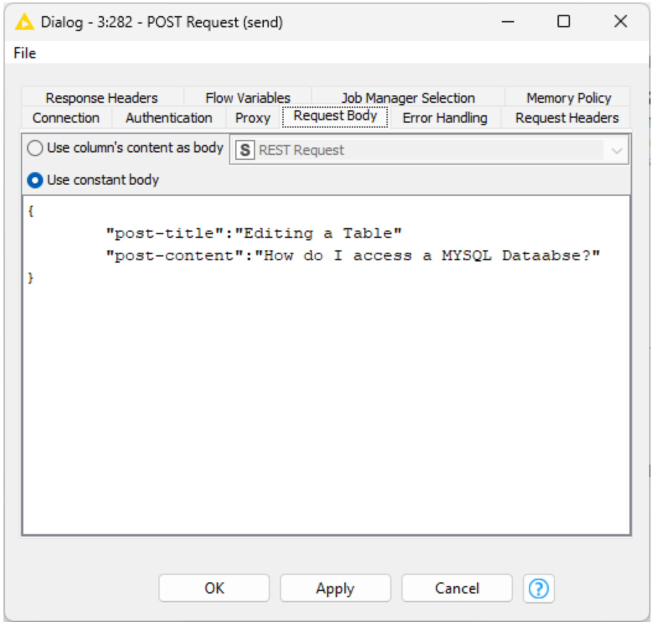
“Body column” contains the name of the response column in the output data table.

POST Request: “Request Body” Tab

POST request(s) sometimes need a body of data. The data for the request body is passed via the “Request Body” tab in the configuration window of the “POST Request” node.

The request body can either be a manually inserted fixed text or the content of a data cell from the input table. This is decided in the radio button options “Use column’s content as body” or “Use constant body”.

If you need to connect to a SOAP based web service via WSDL file, this is also possible from within a KNIME workflow. Here you would need to use the “Generic Web Service Client” node.



( ) Use column's content as body — S REST Request

(●) Use constant body

```
{
        "post-title":"Editing a Table"
        "post-content":"How do I access a MYSQL Dataabse?"
}
```

Figure 3.13. POST Request node configuration: the "Request Body" tab.

3.3. Exercises

Exercise 1

Retrieve information about selected products from fakestoreapi using a GET Request. This resource is a freely available repository of fake products, and all information can be retrieved via REST services.

Before using the service, obtain a list of product categories from Google Sheet “Demo Data” and sheet “Products” as described in section 3.1.

Using this list, then use the following URL to access information for the specific products listed in the Google sheet with:

```
https://fakestoreapi.com/products/category/<product category>
```

Solution to Exercise 1

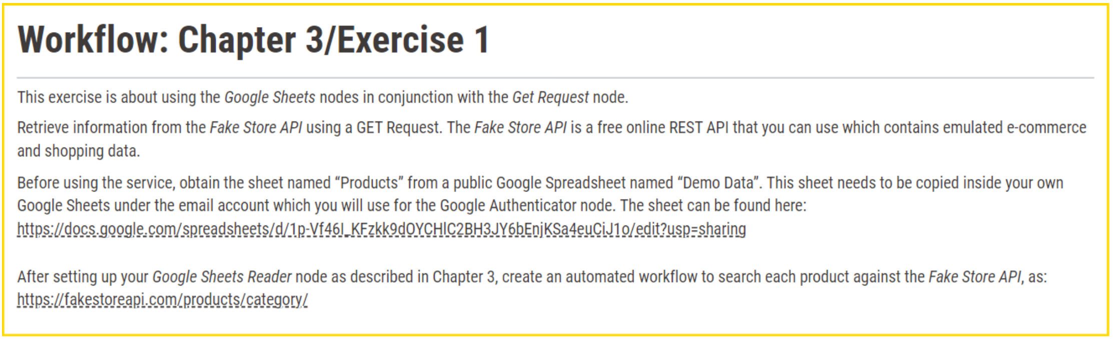
We first connect to Google Sheets using the sequence: “Google Authentication” node, “Google Sheet Connection” node, and “Google Sheets Reader” node to retrieve the product category list from the Google Sheet “Demo Data”.

We create the GET Request in a “String Manipulation” node as:

```
join("https://fakestoreapi.com/products/category/",$products$)
```

and execute it using the “GET Request” node.

The “JSONPath” node enables us to extract id and price for each product in the category. The final result should be a table containing IDs and prices for the products in the selected categories.



Workflow: Chapter 3/Exercise 1

This exercise is about using the *Google Sheets* nodes in conjunction with the *Get Request* node.

Retrieve information from the *Fake Store API* using a GET Request. The *Fake Store API* is a free online REST API that you can use which contains emulated e-commerce and shopping data.

Before using the service, obtain the sheet named “Products” from a public Google Spreadsheet named “Demo Data”. This sheet needs to be copied inside your own Google Sheets under the email account which you will use for the Google Authenticator node. The sheet can be found here:
https://docs.google.com/spreadsheets/d/1p-Vf46I_KFzkk9dOYCHlC2BH3JY6bEnjKSa4euCiJ1o/edit?usp=sharing

After setting up your *Google Sheets Reader* node as described in Chapter 3, create an automated workflow to search each product against the *Fake Store API*, as:
https://fakestoreapi.com/products/category/

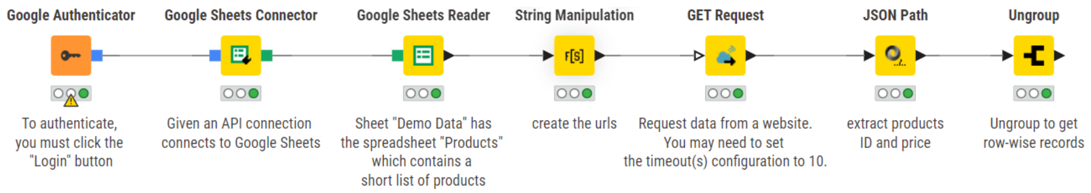
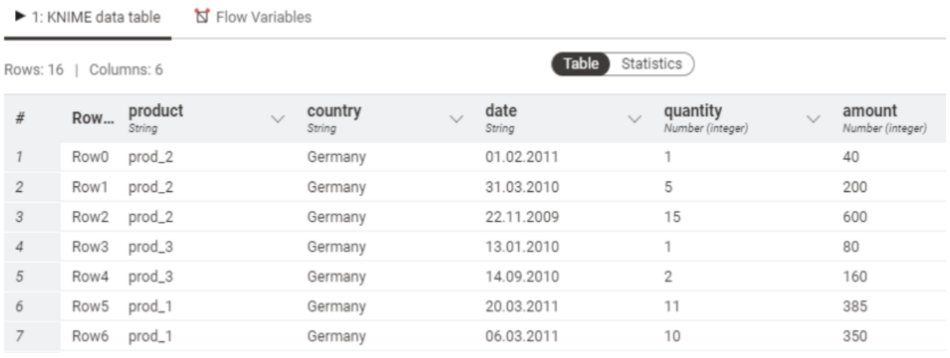

Google Authenticator
To authenticate, you must click the "Login" button
Google Sheets Connector
Given an API connection connects to Google Sheets
Google Sheets Reader
Sheet "Demo Data" has the spreadsheet "Products" which contains a short list of products
String Manipulation
create the urls
GET Request
Request data from a website. You may need to set the timeout(s) configuration to 10.
JSON Path
extract products ID and price
Ungroup
Ungroup to get row-wise records

Figure 3.14. Solution workflow for Exercise 1.

Chapter 4: Date&Time Manipulation

4.1. The Date&Time Type



Rows: 16 | Columns: 6

#	Row...	product (String)	country (String)	date (String)	quantity (Number (integer))	amount (Number (integer))
1	Row0	prod_2	Germany	01.02.2011	1	40
2	Row1	prod_2	Germany	31.03.2010	5	200
3	Row2	prod_2	Germany	22.11.2009	15	600
4	Row3	prod_3	Germany	13.01.2010	1	80
5	Row4	prod_3	Germany	14.09.2010	2	160
6	Row5	prod_1	Germany	20.03.2011	11	385
7	Row6	prod_1	Germany	06.03.2011	10	350

Figure 4.1. The data table produced by the "Database_Operations" workflow implemented in chapter 2.

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. We have three columns of String type ("product", "date", and "country") and two columns of Integer type ("quantity" and "amount").

However, the data column "date" contains the contract date for each record and should be treated as a date type variable. KNIME Analytics Platform has indeed the following dedicated data types for date and time data:

- Date
- Time
- Date & Time
- Date & Time with zone

Note. In very early KNIME Analytics Platform versions, a single Date&Time type was available. This date type is still available as a legacy type ("Legacy Date&Time). In order to convert from a "Legacy Date&Time" type to the new Date&Time type, you will need to use the "Legacy Date&Time to Date&Time" node. The "Date&Time to legacy Date&Time" node moves the data cell into the opposite format direction.

Date&Time 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

These dedicated digits combined together produce a string representation of the date and/or time (with or without time zone). The table below sh ows a few examples for 3:34pm on the 21st of March 2011 in Berlin.

The new Date&Time type carries a number of dedicated nodes implementing a large variety of operations. A full category “Other Data Types”/“Time Series” contains a large number of nodes implementing Date&Time manipulation functionalities.

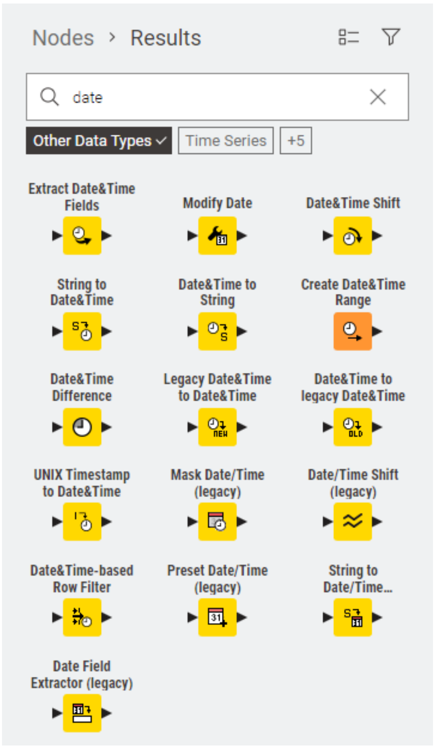


Figure 4.2. Manipulation functionalities for Date&Time objects.

Type: format	**String representation**
Date: dd-MM-yyyy	21-03-2011
Date: MMM/dd/yyyy	Mar/21/2011
Time: hh:mm	15:34
Time: ss.SSS	00.000
Date&Time: dd.MM.yyyy hh.mm.ss.SSS	21.03.2011 15:34:00.000
Date&Time: dd.MMM.yy:hh.mm	21.Mar.11:15.34
Date & Time with zone: ss.SSSyyyy-MM-dd'T'HH:mmSVV'['zzzz']	00.0002011-03-21T03:34+02:00[Europe/Berlin]

4.2. How to produce a Date&Time Column

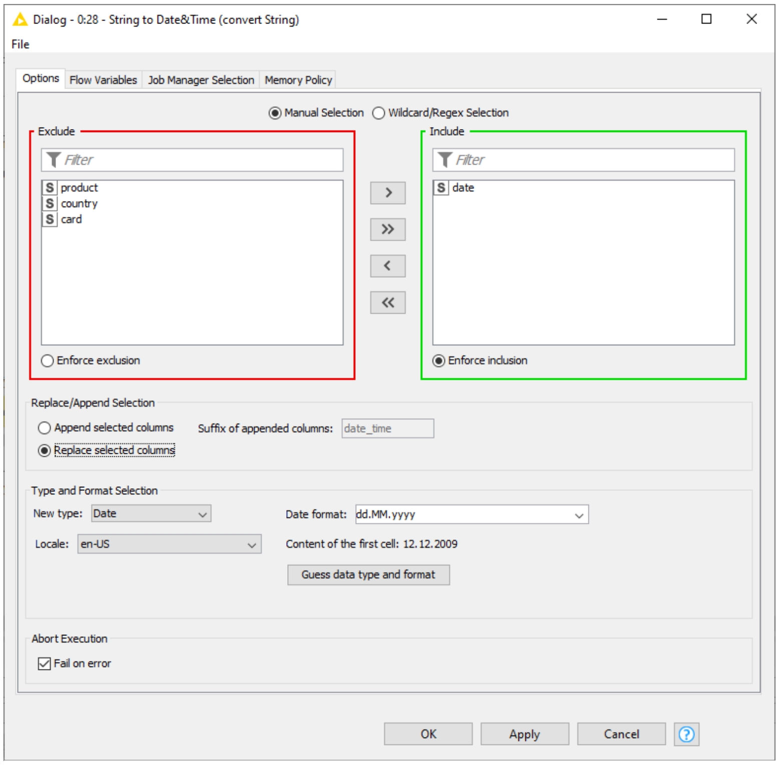
How can we generate a Date (or Time, Date & Time, Date & Time with zone) type data column? There are many possibilities to do that. A “DB Reader” node, for example, automatically reads a SQL timestamp field into a Date&Time column. Another possibility is to read in a String data column, for example with a “File Reader” node, and then convert it into a Date&Time type. This section explores the “String to Date&Time” and “Date&Time to String” conversion nodes.

String to Date&Time

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

- The String column(s) containing the date/time objects to be converted. The column selection is performed through an include/exclude column selection framework
- The option for the resulting column to replace the original String column (“Replace selected column” option) or be appended to the input data table (“Replace selected column” option and suffix for appended new column)
- The new column type (Date, Time, Date&Time, or Date&Time with zone) and locale to express it
- The Date format to be used. Here you can choose from a number of pre-defined Date&Time formats available in the “Date format” menu; or you can edit the Date&Time format manually; or you can let the node guess the Date&Time format through the button “Guess data type and format”)
- A check box that indicates whether reading errors are tolerated or not. If checked, the node will fail on errors.

Note. All Date&Time nodes support multiple columns. Columns can be easily included or excluded in the options using manual selection or wildcard/regex selection.



Dialog - 0:28 - String to Date&Time (convert String)

File

Options | Flow Variables | Job Manager Selection | Memory Policy

[✓ Manual Selection] Wildcard/Regex Selection

Exclude: product, country, card — Enforce exclusion

Include: date — [✓ Enforce inclusion]

Replace/Append Selection

Append selected columns — Suffix of appended columns: date_time

[✓ Replace selected columns]

Type and Format Selection

New type: Date

Date format: dd.MM.yyyy

Locale: en-US

Content of the first cell: 12.12.2009

Guess data type and format

Abort Execution

[✓ Fail on error]

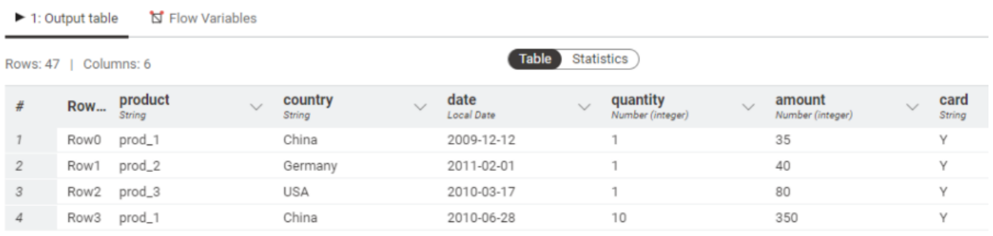
OK | Apply | Cancel

Figure 4.3. Configuration window of the String to Date&Time node.

In order to show the tasks implemented by these and more nodes in the “Time Series” sub-category, let’s create a new workflow, to be named “DateTime_Manipulation”. In the newly created workflow, we use a “File Reader” node to read the “sales.csv” file from the KALdata folder. The “File Reader” node does not automatically assign the Date&Time type to date and time values. It just reads them in as String. Before proceeding with more complex manipulations, let’s convert the “date” column from the default String to the Date&Time type by using a “String to Date&Time” node. For that, 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 Date type according to “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 Date&Time column.

After executing the node, the resulting data table should be as shown in the following, where the data column “date” is now of Date type. The little icon showing a calendar and a clock indicates a Date&Time type.



1: Output table | Flow Variables

Rows: 47 | Columns: 6 | Table | Statistics

#	Row...	product (String)	country (String)	date (Local Date)	quantity (Number (integer))	amount (Number (integer))	card (String)
1	Row0	prod_1	China	2009-12-12	1	35	Y
2	Row1	prod_2	Germany	2011-02-01	1	40	Y
3	Row2	prod_3	USA	2010-03-17	1	80	Y
4	Row3	prod_1	China	2010-06-28	10	350	Y

Figure 4.4. Column "date" after the conversion to DateTime with the String to Date&Time node.

Finally, we wrote the new data table with the Date Type column into a CSV file, with a “CSV Writer” node. If instead of writing to a CSV file we want to save the new data table into a database with a “DB Writer” node, we can make use of the type mapping framework to efficiently map the desired date&time format.

Sometimes, in contrast, it might be necessary to convert a Date&Time column into a String column. The “Date&Time to String” node walks the data in the opposite direction to the “String to Date&Time” node. In the “DateTime_Manipulation” workflow, we introduced a “Date&Time to String” node to convert the Date&Time column “date” back to a String type with format “yyyy/MM/dd”.

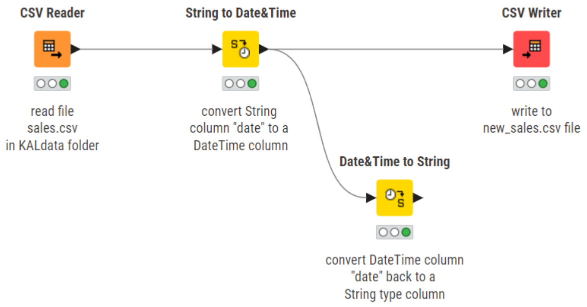
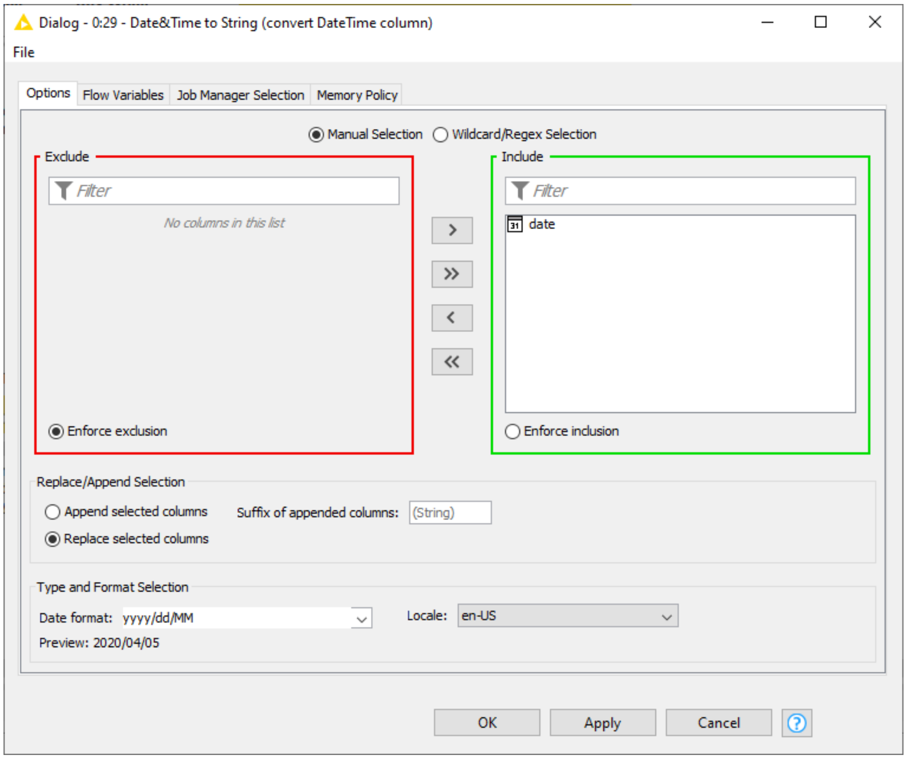

CSV Reader
read file sales.csv in KALdata folder
String to Date&Time
convert String column "date" to a DateTime column
CSV Writer
write to new_sales.csv file
Date&Time to String
convert DateTime column "date" back to a String type column

Figure 4.5. The upper part of the "DateTime_Manipulation" workflow.

Date&Time to String

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

- The Date&Time type column(s) to be converted into String type
- The option for the resulting column to replace the original String column (“Replace selected column” option) or be appended to the input data table (“Replace selected column” option and suffix for appended new column)
- The format to build the String pattern
    - A number of pre-defined Date&Time formats are available in the “Date format” menu
    - The “Date format” box can be edited manually in order to create the required Date&Time format, if the one you want is not available in the menu
    - The locale to express the Date&Time object



[✓ Manual Selection] Wildcard/Regex Selection

Exclude: No columns in this list — [✓ Enforce exclusion]

Include: date — Enforce inclusion

Replace/Append Selection
Append selected columns — Suffix of appended columns: (String)
[✓ Replace selected columns]

Type and Format Selection
Date format: yyyy/dd/MM — Locale: en-US
Preview: 2020/04/05

Figure 4.6. Configuration window of the Date&Time to String node.

Another way to create a Date&Time type column is to use the “Create Date&Time Range” node. This node generates a number of equally spaced date, time, or date and time values. You can:

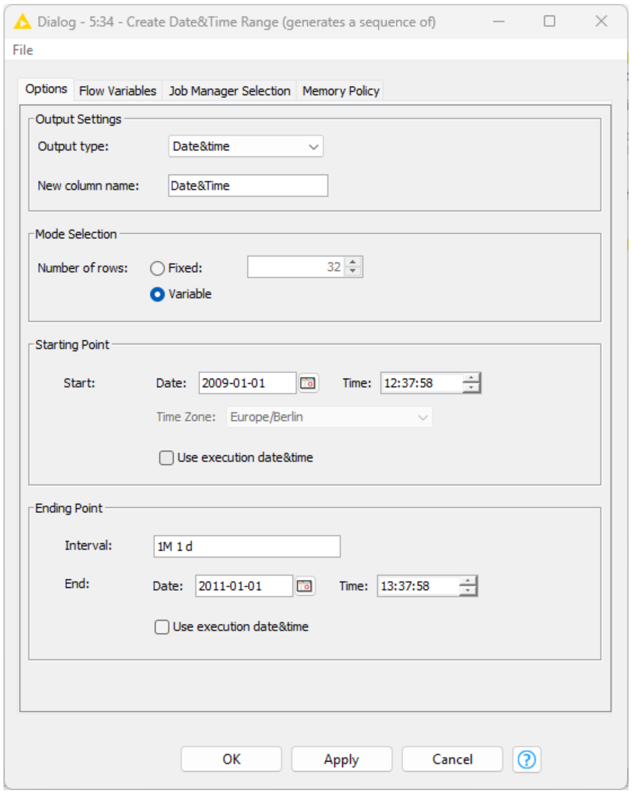
- set the number of values, the starting point in time, and the ending point in time, and calculate the corresponding interval between values
- set the number of values, the starting point, and the interval size, and calculate the corresponding end point in time
- set the interval, the starting point, and the end point and calculate how many values to generate.

Create Date&Time Range

The “Create Date&Time Range” node creates a number of Date&Time values from a start point in time to an end point in time. Values are equally spaced. This means that you set the number of values, the start, and the end and determine the interval; or set the number of values, the interval, and the start, and determine the end; or set the interval, the start, and the end, and determine the number of points.

The configuration window requires :

- The output column containing the new values: name and type
- The starting and end points in time. These two points will have the same format as defined for the new output column: just Date, just Time, Date&Time, or Date&Time with zone
- One of the three strategies to generate the values: a fixed number of rows or a variable number of rows.
    - If you have selected a fixed number of rows, also select the starting point and then the end point or the interval
    - If you have selected a variable number of rows, then select the starting point, the end point, and the interval size. Interval size can be either: time or date ISO-8601 based representation, short letter representation (e.g. '2y 3M 1d'), or long word representation (e.g. '2 years 3 months 1 day')
    - As a special for an end date, there is the current execution time. This can turn out to be useful to generate timestamps.



Output Settings
Output type: Date&time
New column name: Date&Time

Mode Selection
Number of rows: Fixed: 32 [✓ Variable]

Starting Point
Start: Date: 2009-01-01 Time: 12:37:58
Time Zone: Europe/Berlin
Use execution date&time

Ending Point
Interval: 1M 1 d
End: Date: 2011-01-01 Time: 13:37:58
Use execution date&time

Figure 4.7. Configuration window of the Create Date&Time Range node.

ISO-8601 uses H for hours, m for minutes, s for seconds, d for days, M for months, y for years, e.g. '2y 3M 1d' means an interval of 2 years 3 months and 1 day.

In the same workflow named “DateTime_Manipulation”, we introduced a “Create Date&Time Range” node to generate n Date&Time type values between Jan 1, 2009, 12:37 and Jan 1, 2011, 13:37 spaced equally spaced as 1 month and 1 day. The total number of rows generated by this node was then 24.

4.3. Refine Date&Time Values

To each one of these Date&Time values we decided to change time to 15:00. The node that generally changes time is the “Modify Time” node. This node works only on time. It can append a preset time value if none is present, change the current time value to a preset time value, or remove the current time value in the data cell.

Now we could move all Date&Time values one day ahead. That is we would like to add +1 day to all Date&Time values we have created. The node that adds and subtracts a duration from a Date&Time value is the “Date&Time Shift” node. The duration can be expressed as a number for a given granularity, like n days or k months, or as a duration value according to the ISO-8601 date and time duration standards.

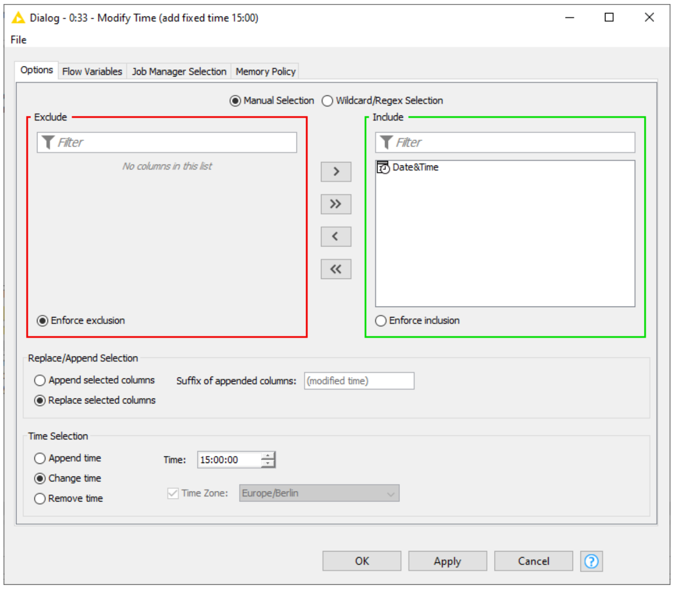
Modify Time

The “Modify Time” node modifies the time value in a data cell by:

- Appending a preset time value
- Changing current time value with a preset time value
- Removing current time value.

The configuration window requires:

- The column(s) on which to modify the time. Columns are selected via an Include/Exclude column selection framework.
- Whether to replace the selected input column or to create a new one to append to the input data table



Dialog - 0:33 - Modify Time (add fixed time 15:00)

File

Options | Flow Variables | Job Manager Selection | Memory Policy

[✓ Manual Selection] Wildcard/Regex Selection

Exclude: Filter — No columns in this list — [✓ Enforce exclusion]

Include: Filter — Date&Time — Enforce inclusion

Replace/Append Selection

Append selected columns — Suffix of appended columns: (modified time)

[✓ Replace selected columns]

Time Selection

Append time — Time: 15:00:00

[✓ Change time]

Remove time — [✓ Time Zone:] Europe/Berlin

OK | Apply | Cancel

Figure 4.8. Configuration window of the Modify Time node.

- The modification strategy: “Append”, “Change”, “Remove”
- In case of “Append” or “Change”, the preset time value
- In case of “Append” also the time zone to associate to the new time value.

Date&Time Shift

The “Date&Time Shift” node shifts a Date&Time value of a defined amount. The configuration window requires:

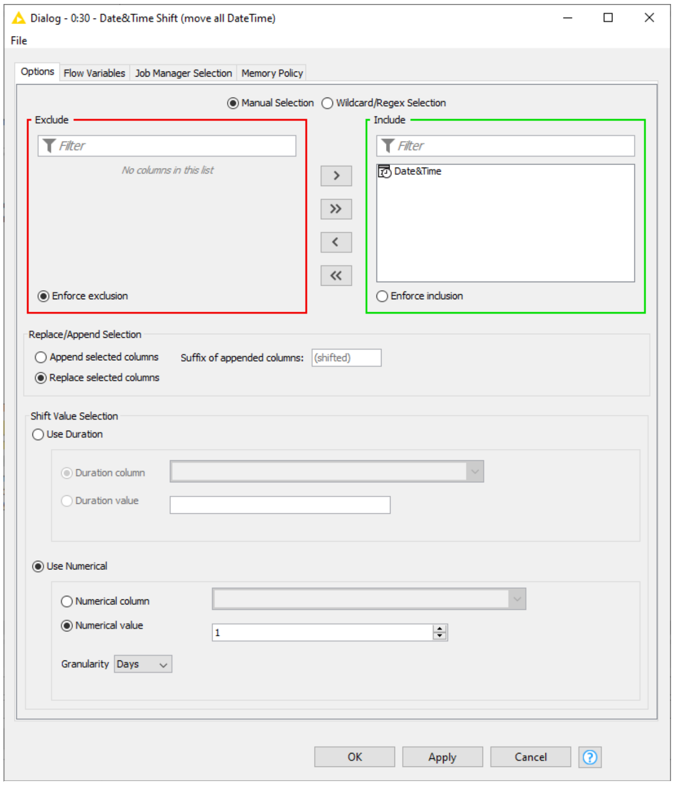
- The input Date&Time column(s) to shift. Those are selected via an Include/Exclude framework.
- Whether to create a new Date&Time column and append it to the input data table or to replace the selected input column. The suffix for the appended columns can be provided in the text field to the right.
- The shift value expressed as duration or as number:
    - Use Duration:

      Duration column. Takes the shift value from a String input columns.

      Duration value. Adds/substracts this constant shift value. The String duration value can be either an ISO-8601 representation of date and time, a short letter representation (e.g. '2y 3M 1d'), or a long word representation (e.g. '2 years 3 months 1 day')
    - Use Numerical:

      Numerical column. Takes the shift value from a numerical input column. A positive value will be added to the reference date or time, a negative one subtracted from it.

      Numerical value. Adds/subtracts this constant shift value. In case “Numerical Value” option is selected, Granularity defines the shift value granularity (day, hour, month, week, etc…).



[✓ Manual Selection] Wildcard/Regex Selection

Exclude
Filter
No columns in this list
[✓ Enforce exclusion]

Include
Filter
Date&Time
Enforce inclusion

Replace/Append Selection
Append selected columns
Suffix of appended columns: (shifted)
[✓ Replace selected columns]

Shift Value Selection
Use Duration
Duration column
Duration value

[✓ Use Numerical]
Numerical column
[✓ Numerical value] 1
Granularity Days

Figure 4.9. Configuration window of the Date&Time Shift node.

Positive and negative shift values are possible to move forward and backward in time.

In this section we have shown only two Date&Time nodes to change Date&Time values. There are many more. For example, similarly to the "Modify Time" node, a "Modify Date" and Modify Time Zone" node are available.

The figure on the right shows the lower part of the "DateTime_Manipulation" workflow as developed in this section.

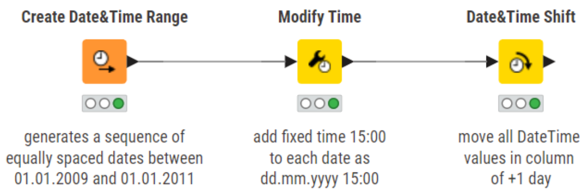

Create Date&Time Range
Modify Time
Date&Time Shift
generates a sequence of equally spaced dates between 01.01.2009 and 01.01.2011
add fixed time 15:00 to each date as dd.mm.yyyy 15:00
move all DateTime values in column of +1 day

Figure 4.10. The lower part of the "DateTime_Manipulation" workflow.

4.4. 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.

The most common Date&Time based data selection is the one extracting a time window. This kind of data row filtering requires setting explicit initial and final date/time objects. Only the data rows falling inside this time window are kept, while the remaining data rows are filtered out. The "Date&Time-based Row Filter" node performs exactly this kind of row filtering based on the explicit definition of a time window.

In order to show practically how such time based filtering criteria can be implemented, we used the "File Reader" node to read the "sales.csv" file from the KALdata folder in a new workflow, named "DateTime_Manipulation_2". After the "File Reader" node, a "String to Date&Time" node converted all String type dates to Date&Time objects. Then a "Date&Time-based Row Filter" node was introduced to select all sales that happened in a pre-defined time range. For the time range we used "2009-01-01" and "2011-01-01". We set these dates as starting and end points, respectively, in the configuration window of the "Date&Time-based Row Filter" node and we obtained a data set with 37 sales data rows at the output port, covering the selected 2-year time span.

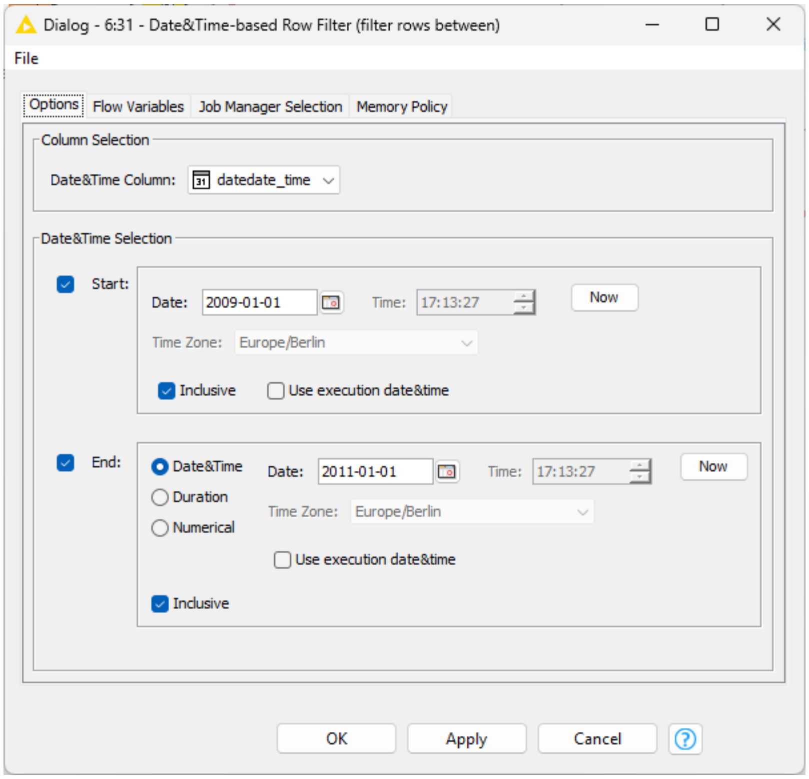
Date&Time-based Row Filter

The "Date&Time based Row Filter" node implements row filtering based on a 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 input Date&Time column to which the filtering criterion should apply
- The time window, i.e.:
    - The time window starting point, in terms of date and/or time
    - The time window end point, in terms of date and/or time. It is possible to assign the end of the time window using a duration formatted String (according to ISO-8601 date and time duration standards) or a numerical value with its granularity (days, months, …)
    - Execution time can be used as a starting point and as an end point for the time window
    - The "Inclusive" flag includes the extremes or the time window in the filter criterion.

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 Date&Time type to the String type and work on it with the



Column Selection
Date&Time Column: datedate_time

Date&Time Selection
[✓] Start:
Date: 2009-01-01 Time: 17:13:27
Time Zone: Europe/Berlin
[✓] Inclusive [ ] Use execution date&time

[✓] End:
[✓ Date&Time] Duration Numerical
Date: 2011-01-01 Time: 17:13:27
Time Zone: Europe/Berlin
[ ] Use execution date&time
[✓] Inclusive

Figure 4.11. Configuration window of the Date&Time-based Row Filter node.

String Manipulation node. There is, of course, a much faster way with the "Extract Date&Time Fields" node.

The "Extract Date&Time Fields" node decomposes a Date&Time 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

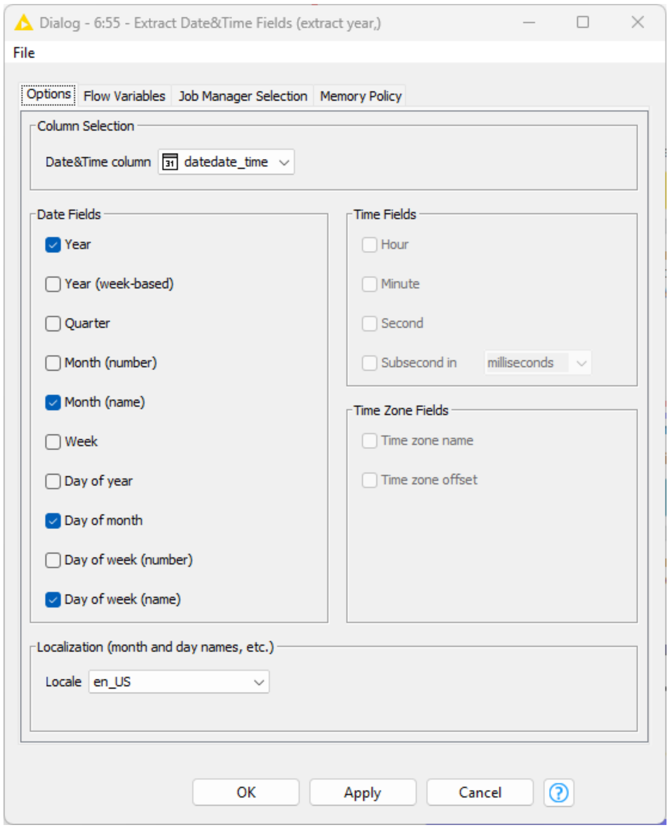
Time	**Hours**	**Minutes**	**Seconds**	**Milliseconds**
15:23:10.123	15	23	10	123
04:02:56.987	4	2	56	987

Extract Date&Time Fields

The "Extract Date&Time Fields" node extracts the components (fields) of a Date&Time object. The configuration window requires:

- Column Selection. A Date, a Time, a Date Time, or a Zoned Date Time column whose fields to extract.
- Date Fields. Pick and choose which fields to extract (year, week, hour, time zone name ...)
- The locale. This sets the locale to express the output Strings

Note. The "Extract Date&Time Fields" node is particularly useful when combined with data aggregation nodes ("GroupBy", "Pivot", etc.)


Dialog - 6:55 - Extract Date&Time Fields (extract year,)

File

Options | Flow Variables | Job Manager Selection | Memory Policy

Column Selection

Date&Time column: datedate_time

Date Fields
- [x] Year
- [ ] Year (week-based)
- [ ] Quarter
- [ ] Month (number)
- [x] Month (name)
- [ ] Week
- [ ] Day of year
- [x] Day of month
- [ ] Day of week (number)
- [x] Day of week (name)

Time Fields
- [ ] Hour
- [ ] Minute
- [ ] Second
- [ ] Subsecond in milliseconds

Time Zone Fields
- [ ] Time zone name
- [ ] Time zone offset

Localization (month and day names, etc.)

Locale: en_US

Figure 4.12. Configuration window of the Extract Time Window node.

In the “DateTime_Manipulation_2” workflow, we introduced an “Extract Date&Time Fields” node after the “Date&Time-based Row Filter” node. The configuration settings were set to extract year, month (name), day of week (name), and day number in the month from each Date&Time cell in the column named “datedate_time”.

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 at the output port, thus, had 11 columns, 4 more than the original 7 columns. The 4 additional columns are: “Year”, “Month”, Day of month”, and “Day of week (name)”.

In order to isolate the 2010 sales, we added a “Row Filter” node after the “Extract Date&Time Fields” node in the “DateTime_Manipulation_2” workflow to keep all rows with “Year” = 2010. The resulting data table had all rows 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, 2010 and 2011. 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 to exclude all rows with a sale date older than one year. The "Date&Time Difference" node calculates the time difference between two Date&Time values. The two Date&Time values can be:

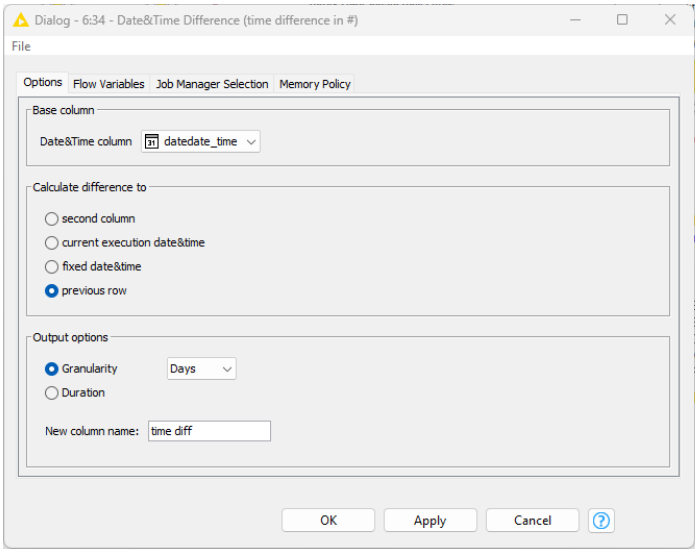
- two Date&Time cells on the same row,
- one Date&Time cell and the current execution date/time,
- one Date&Time cell and a fixed Date&Time value,
- or finally one Date&Time cell and the corresponding Date&Time cell in the previous row.

The time difference can be calculated in terms of days, months, years, etc., even milliseconds.

Date&Time Difference

The "Date&Time Difference" node calculates the time elapsed between two Date&Time values. The configuration window requires :

- The input column to use for the first Date&Time values
- The second Date&Time value, that is:
    - another Date&Time value in the same data row (Use "second column" option). In this case the name of the second column is required.
    - the current date/time at execution (Use "current execution date&time" option)
    - a fixed date/time (Use "fixed date&time" option). In this case the fix Date&Time value is required.
    - the Date&Time cell in the previous row in the same column (Use "previous row" option)
- The output options:
    - Granularity to output the difference in number of days, months, years, etc.
    - Duration to express the difference as date-based or time-based duration, according to ISO-8601 date and time duration standards
    - Name of the output column


Dialog - 6:34 - Date&Time Difference (time difference in #)

File

Options | Flow Variables | Job Manager Selection | Memory Policy

Base column

Date&Time column: datedate_time

Calculate difference to

- ○ second column
- ○ current execution date&time
- ○ fixed date&time
- ● previous row

Output options

- ● Granularity: Days
- ○ Duration

New column name: time diff

OK | Apply | Cancel

Figure 4.13. Configuration window of the Time Difference node.

In the "DateTime_Manipulation_2" workflow, we connected a "Date&Time Difference" node to the "File Reader" node to calculate the time elapsed between today ("Current execution date&time" option) and the Date&Time 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, 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 "Date&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 special event, for example some marketing initiative. In this case we need to sort the sale records by sale date with a "Sorter" node. The "Sorter" node can manage Date&Time types and can sort them appropriately. After that, we use a "Date&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. Notice that we used two nodes: the simple Line Plot node and the Line Plot (Plotly) node. They perform the same task, i.e., they plot a line for the data. However, the Line Plot (Plotly) node integrates the line plot from the Plotly libraries.

4.5. Moving Average and Aggregation

In the “Other Data Types”/“Time Series”/”Smoothing” sub-category you also find a node that is more oriented towards time series analysis rather than just Date&Time manipulation: this is the “Moving Average” node.

The “Moving Average” node calculates the moving average[4] on a 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 n-th sample of the time series. Then the moving window moves over to the next *(n+1)-th* sample of the time series, and so on.

A number of slightly different algorithms can 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 is calculated. Thus, 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 n-th sample is calculated on the basis of the (n-1)-th sample

[4] *NIST/SEMATECH, “e-Handbook of Statistical Methods”, (http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4.htm)*

If *v(i)* is the value of the original sample at position i inside the k-sample moving window, the algorithm to calculate the average value for the n-th sample 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	$avg(n) = \frac{n}{\sum_{i=0}^{k-1} \frac{1}{\mathrm{v}\left(\mathrm{n+i-\frac{k-1}{2}}\right)}}$	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 with the n-th sample 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 n-th 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

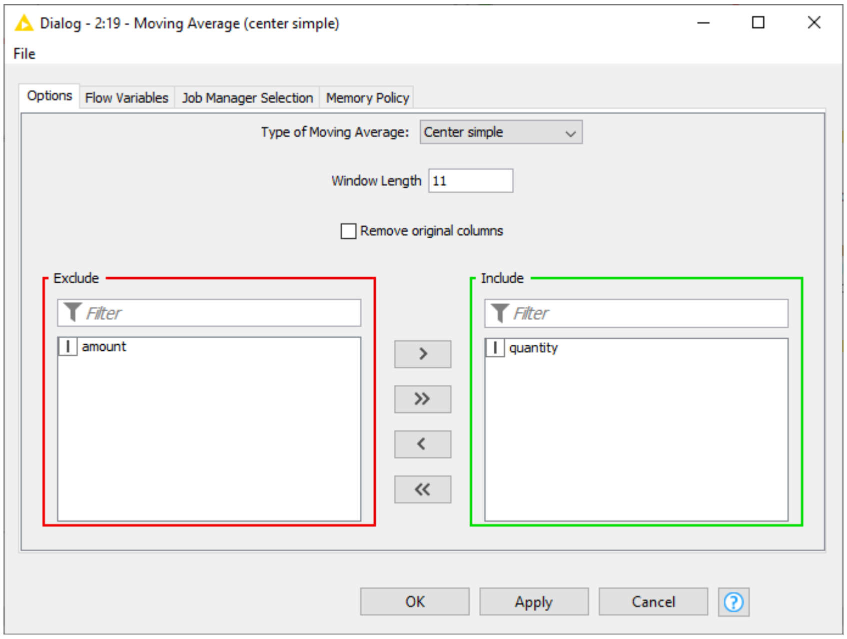
The selection of the data column(s), to which the moving average should be applied, is based on an “Exclude/Include” framework.

- 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

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. The first (n-1)/2 values are replaced with missing values.


Dialog - 2:19 - Moving Average (center simple)

File

Options | Flow Variables | Job Manager Selection | Memory Policy

Type of Moving Average: Center simple

Window Length 11

☐ Remove original columns

Exclude: amount

Include: quantity

Figure 4.14. Configuration window of the Moving Average node.

In the “DateTime_Manipulation_2” workflow we applied a “Moving Average” node to the output data table of the “File 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 (Figure 4.16).

The “Moving Aggregation” node extends the “Moving Average” node. Indeed, it calculates a number of additional statistical measures, besides 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 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.

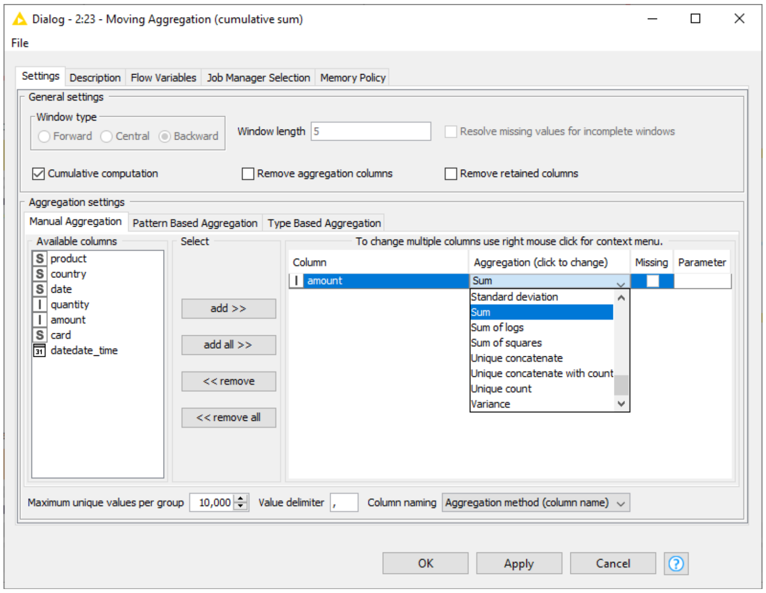
Moving Aggregation

The “Moving Aggregation” node calculates statistical and aggregation measures on a moving window. The “Settings” tab in the configuration window requires:

- The statistical or aggregation measure to use
- The input data column for the calculation
- The 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.


Dialog - 2:23 - Moving Aggregation (cumulative sum)

File

Settings | Description | Flow Variables | Job Manager Selection | Memory Policy

General settings

Window type: ◯ Forward ◯ Central [✓ Backward]

Window length: 5

☐ Resolve missing values for incomplete windows

☑ Cumulative computation ☐ Remove aggregation columns ☐ Remove retained columns

Aggregation settings

Manual Aggregation | Pattern Based Aggregation | Type Based Aggregation

Available columns: S product, S country, S date, I quantity, I amount, S card, datedate_time

Select: add >>, add all >>, << remove, << remove all

To change multiple columns use right mouse click for context menu.

Column	Aggregation (click to change)	Missing	Parameter
I amount	Sum	☐	

Standard deviation, Sum, Sum of logs, Sum of squares, Unique concatenate, Unique concatenate with count, Unique count, Variance

Maximum unique values per group: 10,000   Value delimiter: ,   Column naming: Aggregation method (column name)

OK | Apply | Cancel

Figure 4.15. 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 above.

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 in financial accounting for year to date measures.

In the “DateTime_Manipulation_2” workflow, we added a “Moving Aggregation” node to the output data table of the “File Reader” node. The cumulative sum was calculated for the “amount” data column, over the whole time series. The plot of the resulting time series is obtained through a “Line Plot” node.

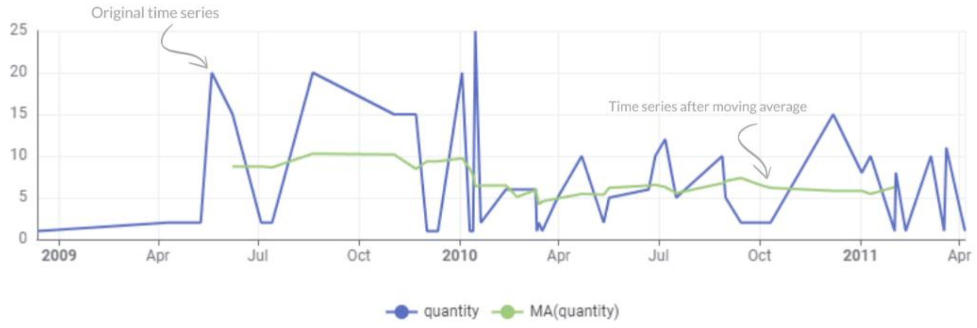


Figure 4.16. Moving Average effect on time series "quantity".

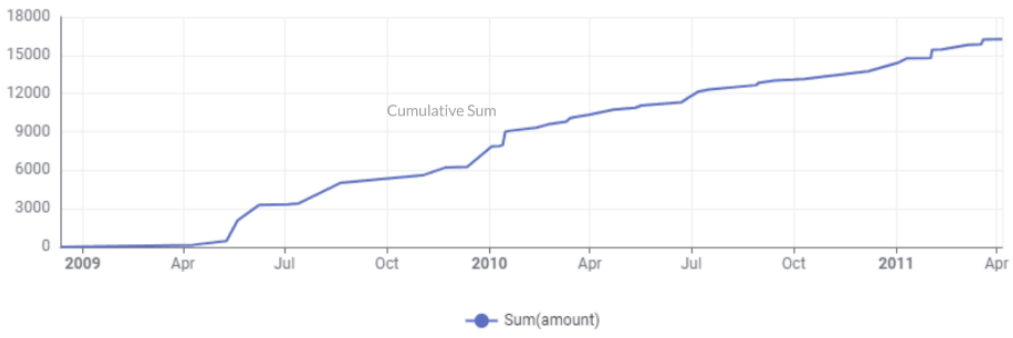
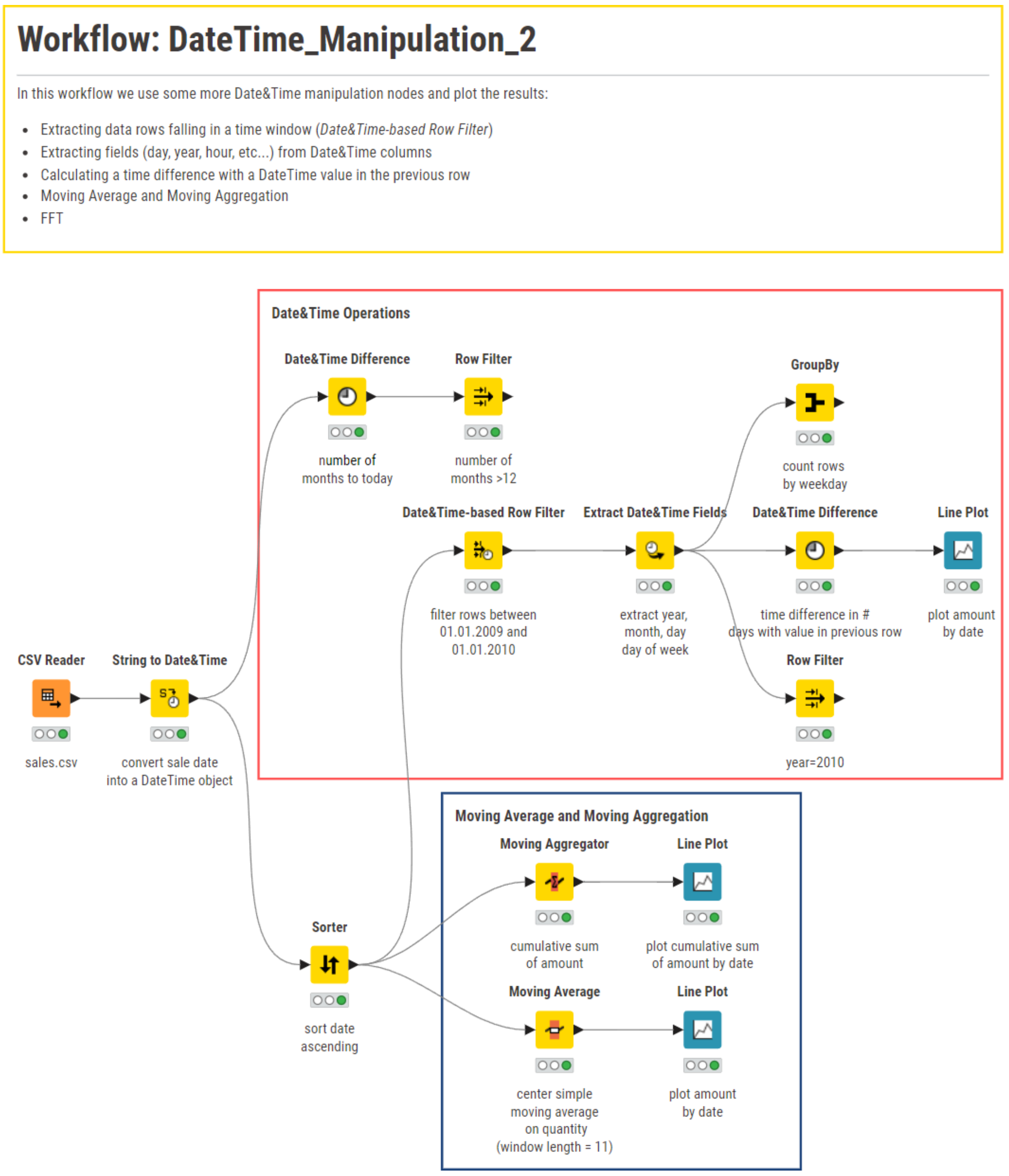


Figure 4.17. Cumulative Aggregation of time series "amount".

Note. The “Fast Fourier Transform (FFT)” node implements the Fast Fourier Transform of a signal. The node is part of the “AI.Associates Signal Processing Nodes” extension under “KNIME Community Contributions – Other”.



Workflow: DateTime_Manipulation_2

In this workflow we use some more Date&Time manipulation nodes and plot the results:

- Extracting data rows falling in a time window (*Date&Time-based Row Filter*)
- Extracting fields (day, year, hour, etc...) from Date&Time columns
- Calculating a time difference with a DateTime value in the previous row
- Moving Average and Moving Aggregation
- FFT

Date&Time Operations
Date&Time Difference
number of months to today
Row Filter
number of months >12
GroupBy
count rows by weekday
Date&Time-based Row Filter
filter rows between 01.01.2009 and 01.01.2010
Extract Date&Time Fields
extract year, month, day day of week
Date&Time Difference
time difference in # days with value in previous row
Line Plot
plot amount by date
Row Filter
year=2010
CSV Reader
sales.csv
String to Date&Time
convert sale date into a DateTime object
Sorter
sort date ascending
Moving Average and Moving Aggregation
Moving Aggregator
cumulative sum of amount
Line Plot
plot cumulative sum of amount by date
Moving Average
center simple moving average on quantity (window length = 11)
Line Plot
plot amount by date

Figure 4.18. The "DateTime_Manipulation_2" workflow.

4.6. Time Series Analysis

KNIME Analytics Platform also offers some components for time series analysis. All of them rely on one simple node: the *Lag Column* node. The *Lag Column* node is the key node for time series analysis.

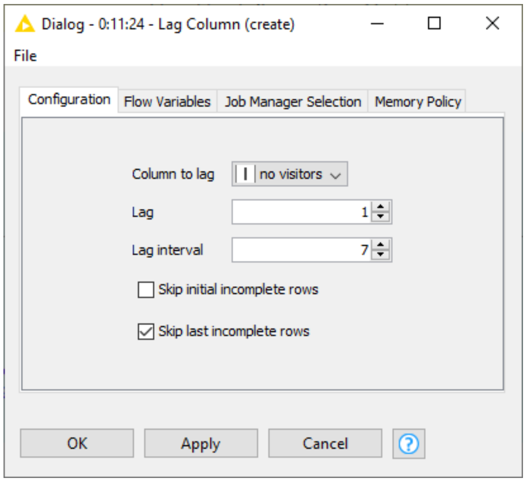
Lag Column

The *Lag Column* node copies a data column *x(t)* and shifts it: n times 1, 2, …, n step each time or one time only p steps down. It can work in three different ways, producing:

- one copy shifted p steps x(t), x(t-p)
- n copies, each shifted 1, .. n steps resp. x(t), x(t-1), …, x(t-n)
- n*p copies, each shifted p*(1, …n) steps 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.



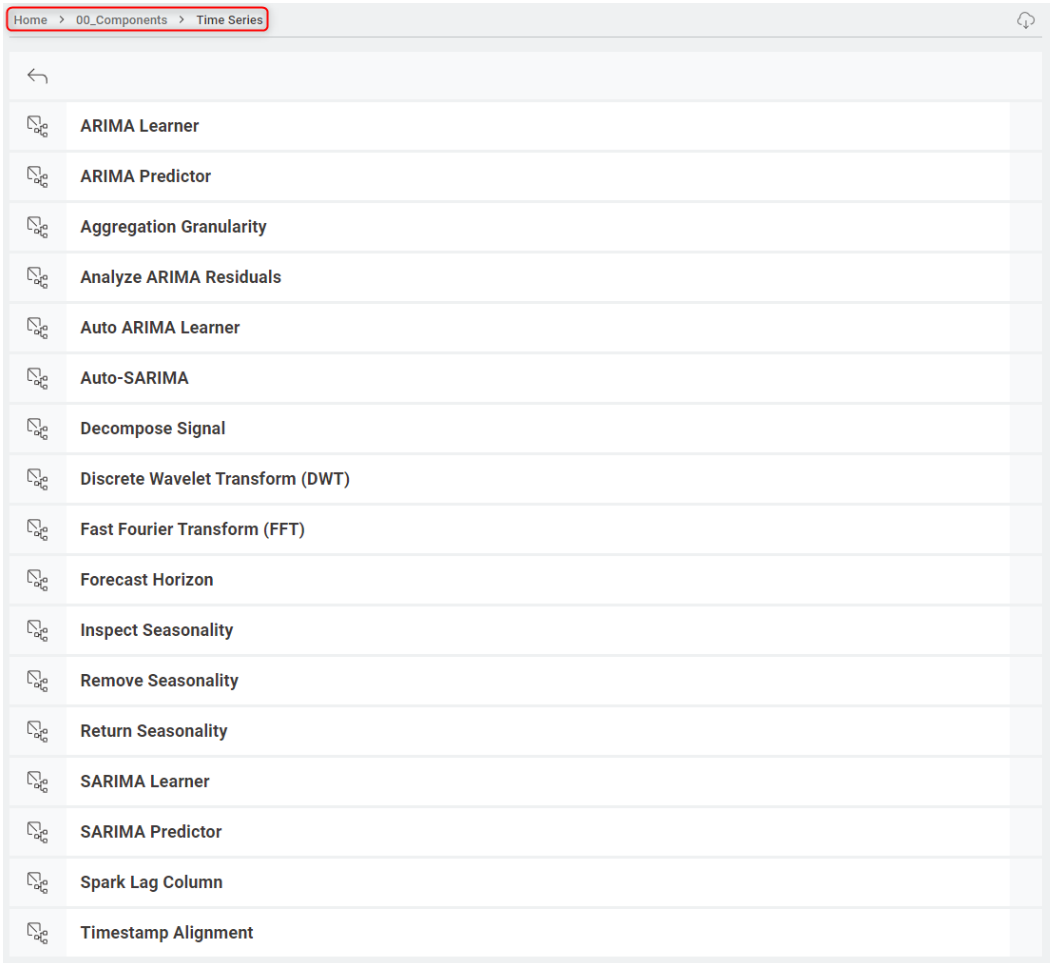
Column to lag: I no visitors
Lag: 1
Lag interval: 7
☐ Skip initial incomplete rows
☑ Skip last incomplete rows

Figure 4.19. Configuration window of the Lag Column node.

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

Note. If the column values are sorted in time ascending order, shifting the values up or down means shifting the values in the past or in the future. Vice versa if sorted in time descending order.

Other nodes, dedicated to the analysis of time series, can be found in the KNIME Examples space on the Community Hub in the 00_Components > Time Series folder, and the *Lag* node is now also a part of KNIME Base nodes. This folder hosts some special nodes, named components, each one implementing one of the many steps required in time series analysis. Some of these components rely on Python libraries and therefore they require setting up the KNIME Python extension. We will see more about components in the next chapter. For now, however, it is enough to know that they behave as regular nodes. To create an instance of the node, just drag & drop them from the KNIME Community Hub into the workflow editor. Then configure their settings via the usual configuration window, and finally execute them. They will



ARIMA Learner
ARIMA Predictor
Aggregation Granularity
Analyze ARIMA Residuals
Auto ARIMA Learner
Auto-SARIMA
Decompose Signal
Discrete Wavelet Transform (DWT)
Fast Fourier Transform (FFT)
Forecast Horizon
Inspect Seasonality
Remove Seasonality
Return Seasonality
SARIMA Learner
SARIMA Predictor
Spark Lag Column
Timestamp Alignment

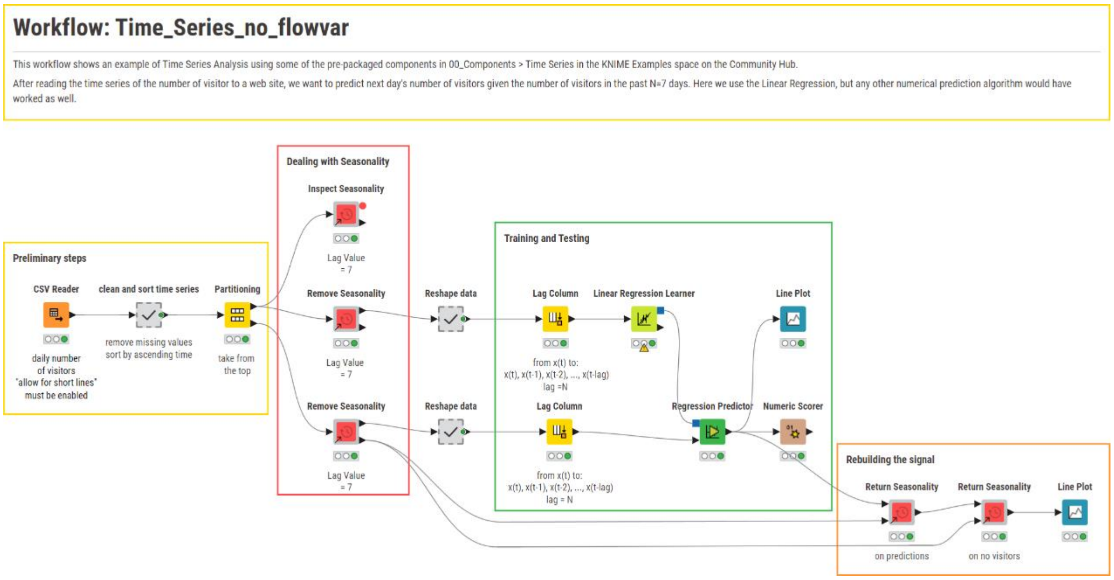
Figure 4.20. The "00_Components/Time Series" category on the public EXAMPLES Server.

present the results at their output port.

When you create a component in your workflow, this is just a link to the component template residing in the KNIME Examples space. Every time you open the workflow, with this link to the component, you will be asked if you want to look whether a new version of this same component has been uploaded to the repository in the meantime.

Notice the node "Aggregation Granularity" for time aggregation, the node "Inspect Seasonality" to extract the lag value for the dominant seasonality in the input time series; the node "Remove Seasonality" feeding from the Lag Value of the node "Inspect Seasonality" and removing the seasonality for that lag, the node "Return Seasonality" to rebuild the predicted signal from the residual predictions, the "ARIMA Learner" and "ARIMA Predictor" nodes to train an ARIMA model and to apply it respectively. Keep an eye on this folder, because it is being constantly filled with new and updated nodes.

As an example, in the workflow “Time_Series_no_flowvars”, we read the file “website.txt” from the KALdata folder containing the daily number of visitors to a website. We want to predict the next day's number of visitors given the number of visitors in the past N days, for N=3, 5, 7.



Workflow: Time_Series_no_flowvar

This workflow shows an example of Time Series Analysis using some of the pre-packaged components in 00_Components > Time Series in the KNIME Examples space on the Community Hub.

After reading the time series of the number of visitor to a web site, we want to predict next day's number of visitors given the number of visitors in the past N=7 days. Here we use the Linear Regression, but any other numerical prediction algorithm would have worked as well.

Figure 4.21. The "Time_Series_noflowvars" workflow.

After reading the data, we clean up the dataset and we order it by date in ascending order. After that, we partition the data. Notice that, in order to avoid data leakage, data partitioning for time series is performed from the top: the past is for the training set and the future is for the test set. They should not mix. After these preliminary steps, let’s start to deal with seasonality.

Seasonality and trend are a big issue in time series analysis, since can make the work too complicated for the upcoming predictive algorithms. It is good practice to detect the seasonality in the time series and, if any, to remove it before training the model. The component “Inspect Seasonality” calculates the AutoCorrelation Function(ACF) for lagged copies of time series through the Pearson Correlation for lags between 0 and a maximum lag, using a pre-define lag step. It then detects the first local maximum of correlation for sign of dominant seasonality, if any above the cut-off value defined in the node settings. The node produces, at the output port, the sequence of local ACF maxima and their corresponding lag value. The lag value for the highest maximum in this sequence is the lag for the dominant seasonality. Additionally, the component produces an interactive view that displays the Autocorrelation Function (ACF) Plot and Partial Autocorrelation Function (PACF) Plot.

From the list of ACF maxima in the output data table of the “Inspect Seasonality” component, we see the highest local maximum at lag = 7. Let’s remove this seasonality then from the original column “no visitors”. To do that we apply the “Remove Seasonality” component to input column “no visitors” and lag value 7. Notice that the seasonality index was calculated just on the training set and that the seasonality was removed from both the training time series and the test time series. The output table of the “Remove Seasonality” component contains the original data column and the same data column without seasonality.

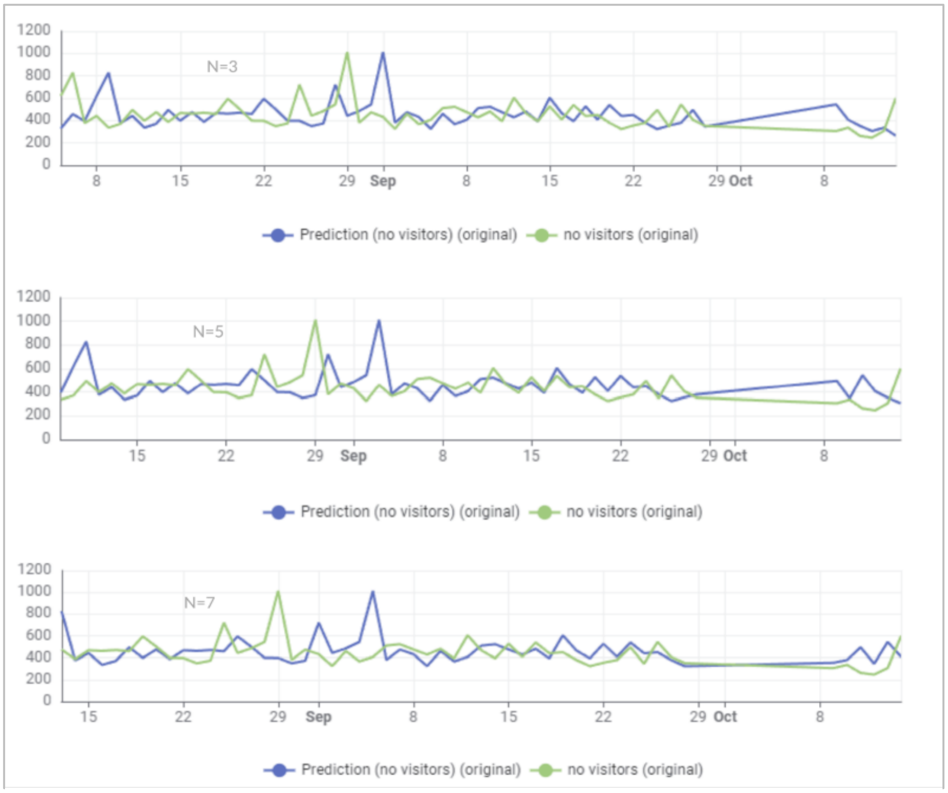


Figure 4.22.Original time series (green) vs. predicted time series (blue) using N= 3, 5, 7 previous samples.

After removing the seasonality, we finally train the model. We could train and test an ARIMA model with the ARIMA Learner and the ARIMA Predictor nodes. We could also pair a machine learning regression algorithm with the Lag Column node to train the model on the past N samples of the series to predict the next sample in the series. We trained a Linear Regression model with N= 3, 5, and finally 7 past samples to predict the value of the sample in the time series. Finally, the model is evaluated with a Numeric Scorer node.

To investigate visually how well the predictions of the time series values worked, we reinserted the seasonality in both the time series “no visitors” that had been de-seasonalized earlier on and into the predictions. Above you can see the line plots of the original time series and the predicted time series when using N=3, 5, and 7 samples.

Of course, it is not necessary to calculate the seasonality lag and then manually insert it into the "Remove Seasonality" component. This can all be done automatically using the flow variables, as we will see in the next chapter.

4.7. Exercises

Create a workflow group "Exercises" under the existing "Chapter4" 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;
- Remove the time;
- 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

The rows generated by the "Time Generator" node are only 170. The "Joiner" node is then 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".

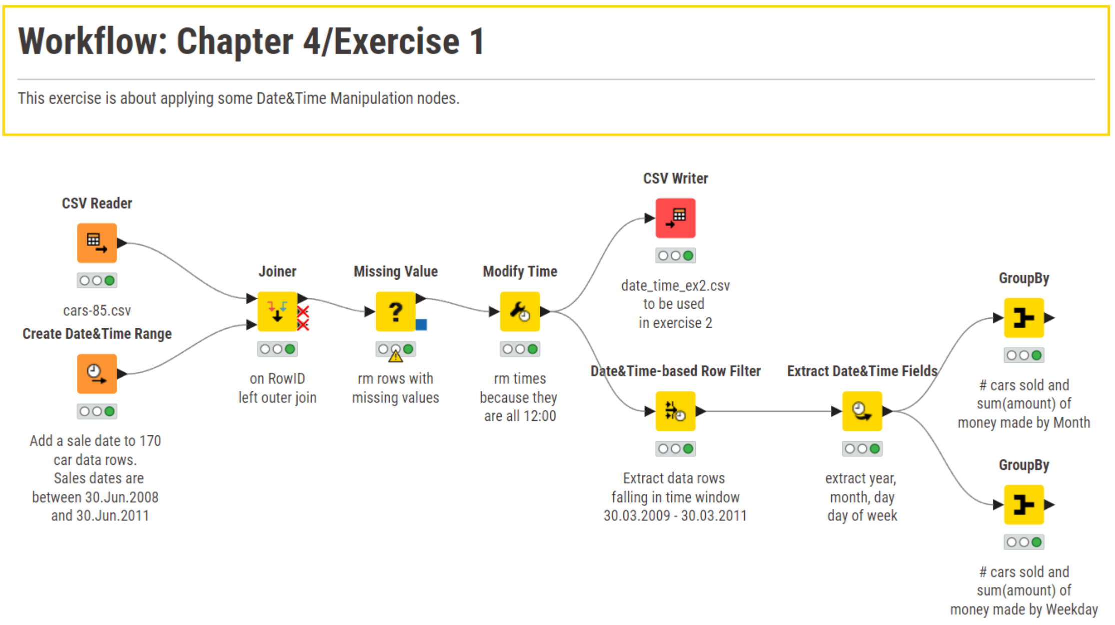
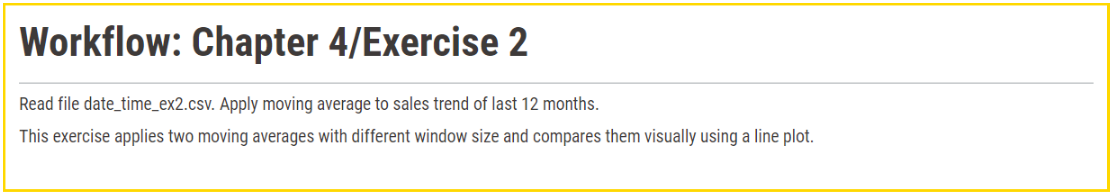

Workflow: Chapter 4/Exercise 1
This exercise is about applying some Date&Time Manipulation nodes.
CSV Reader
cars-85.csv
Create Date&Time Range
Add a sale date to 170 car data rows. Sales dates are between 30.Jun.2008 and 30.Jun.2011
Joiner
on RowID left outer join
Missing Value
rm rows with missing values
Modify Time
rm times because they are all 12:00
CSV Writer
date_time_ex2.csv to be used in exercise 2
Date&Time-based Row Filter
Extract data rows falling in time window 30.03.2009 - 30.03.2011
Extract Date&Time Fields
extract year, month, day day of week
GroupBy
# cars sold and sum(amount) of money made by Month
GroupBy
# cars sold and sum(amount) of money made by Weekday

Figure 4.23. Exercise 1: The workflow.

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.

Solution to Exercise 2



Workflow: Chapter 4/Exercise 2

Read file date_time_ex2.csv. Apply moving average to sales trend of last 12 months.

This exercise applies two moving averages with different window size and compares them visually using a line plot.

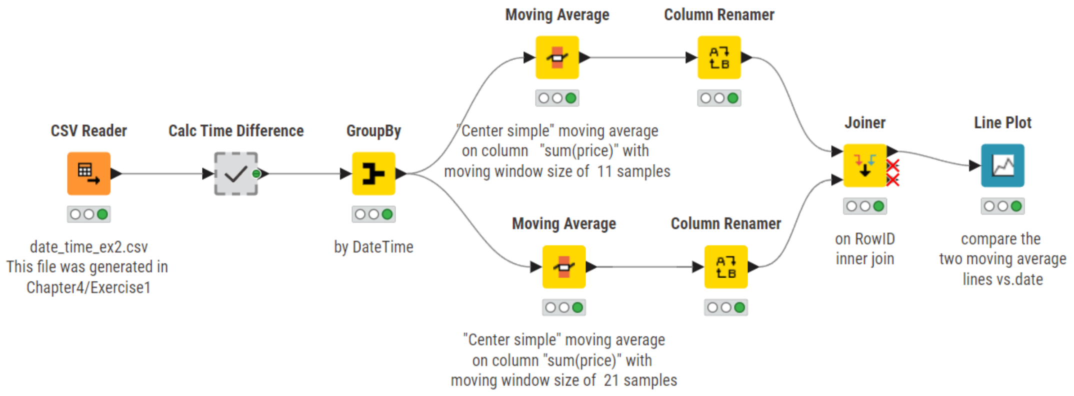


Figure 4.24. Exercise 2: The workflow.

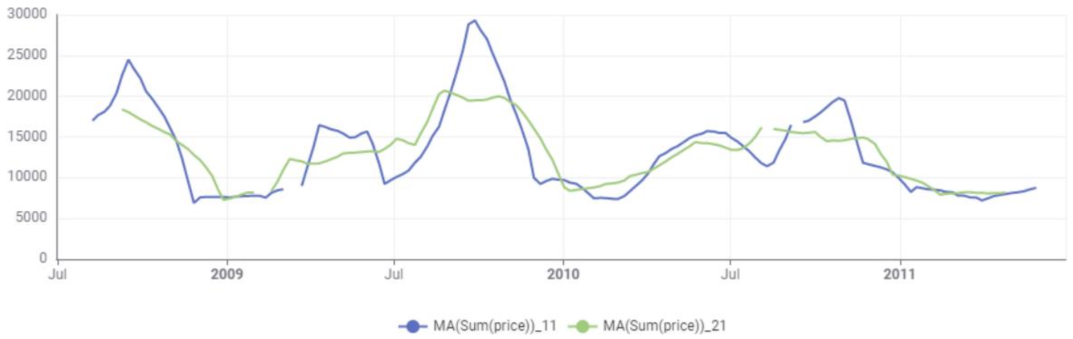
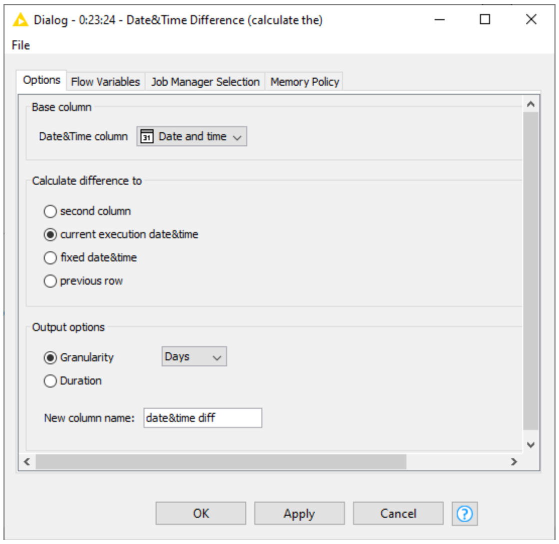


Figure 4.25. Plots of the original time series, after a center simple moving average on 11 samples, and after a center simple moving average on 21 samples.



Base column

Date&Time column: Date and time

Calculate difference to

- ( ) second column
- (●) current execution date&time
- ( ) fixed date&time
- ( ) previous row

Output options

- (●) Granularity: Days
- ( ) Duration

New column name: date&time diff

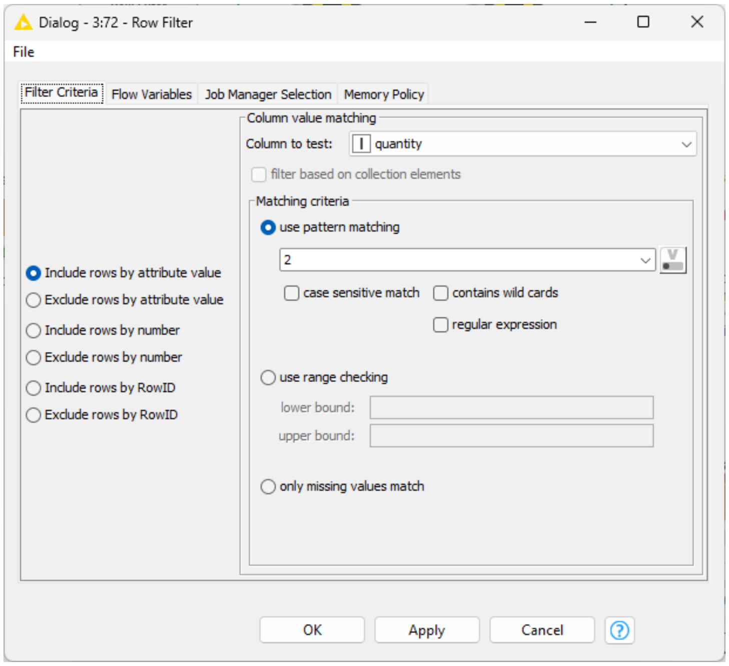
Figure 4.26. Settings for the "Date&Time Difference" node in "Calc Time Difference" metanode.

Chapter 5: Flow Variables

5.1. What is a Flow Variable?

In KNIME Analytics Platform it is possible to define external parameters to be used throughout the entire workflow: these parameters are called "Workflow Variables", or shortly "Flow Variables", and can be string, integer, double, arrays, or Path values. Flow variable values can be updated during each workflow execution though dedicated nodes and features. Therefore, such parameters can be used to avoid manually changing settings within the nodes of a workflow when a new execution with different settings is required.

Let's create a workflow group called "Chapter5" to host the workflows implemented in the course of this chapter. Let's also create an empty workflow named "Flow_Vars" as its first example workflow. First thing, we read the "sales.csv" file from the KALdata folder, then we convert column "date" from type "String" to type "DateTime", and finally we apply the *Date&Time-based Row Filter* node to filter the rows with "date" between "01.Jan. 2009" and "01.Jan. 2011".



Column value matching
Column to test: quantity
filter based on collection elements

[✓ Include rows by attribute value]
Exclude rows by attribute value
Include rows by number
Exclude rows by number
Include rows by RowID
Exclude rows by RowID

Matching criteria
[✓ use pattern matching]
2
case sensitive match
contains wild cards
regular expression
use range checking
lower bound:
upper bound:
only missing values match

Figure 5.1. The configuration window of the Row filter node to find the sales records where exactly 2 items have been sold.

On the remaining rows, we want to find all those sales where a given number of items ("quantity" column) has 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 (see configuration window above).

Now, let's suppose that the number of sold items we are looking for is not always the same. Sometimes we might want to know which sales sold 3 items, sometimes which sales sold 10 items, sometimes all we need to know is which sales sold more than n items, and so on. In theory, at each run we should open the *Row Filter* node and update the filtering criterion. But this is a very time-consuming approach, especially if more than one *Row Filter* node is involved, and it is not well suited to a remote workflow execution on a KNIME Business Hub.

We can parameterize the pattern matching in the row filtering criterion by using a flow variable. In this example, we could define a flow variable as an integer with name "number items" and initial (default) value 2. We could then change the matching pattern in the filtering criterion in the *Row Filter* node to take on the flow variable value rather than the fixed number set in the configuration window. That is, we would like to have a filtering criterion like `quantity = "number items"` rather than `quantity = 2`. At each workflow execution, the value of the flow variable can be changed to retrieve only those sale records with the newly specified number of sold items.

The following sections explore how to create and use flow variables.

5.2. Creating a Flow Variable

Flow variables are created locally inside the workflow and are available only for the downstream nodes in the workflow. To create flow variables, you have the following possibilities:

- Convert a table row into flow variables
- Export a node configuration as flow variable
- Use *Configuration* and *Widget* nodes
- Combine or modify existing flow variables

Let's have a look at the different options in the following.

Suppose that we do not know the matching pattern for the filtering criterion ahead of time. In other words, the matching pattern becomes known only during the workflow execution. Hence, we need to create the flow variable "on the fly". In our case, the flow variable will be created, and its value will be assigned as soon as it becomes know.

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 "Flow_Vars" workflow, we connect a *GroupBy* node to the output port of the *Date&Time-based Row Filter* node. The *GroupBy* node is set to group the data rows by country and to calculate the sum of values in the "quantity" column. A *Sorter* node sorts the resulting aggregation by "sum(quantity)" in descending order. Thus, the country reported in the first row of the final data table is the country with the highest number of sold items.

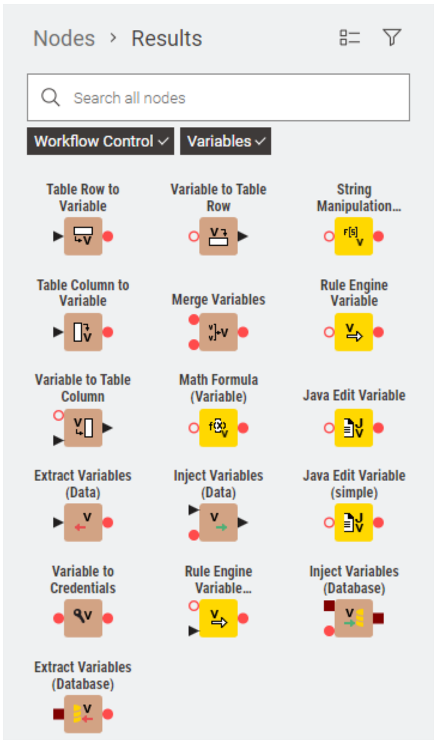


Figure 5.2. The nodes in the "Workflow Control" category that deal with flow variables.

On another branch in the workflow, a *Row Filter* node should retain all data rows where "country" is that country with the highest number of sold items, as reported in the first row of the output data table of the *Sorter* node.

Depending on the selected time window, we do not know ahead of time which country has the highest number of sold items. Therefore, we cannot assign the country name to a flow variable before running the workflow. We should first find that country, then transfer it into a flow variable, and finally use it to execute the *Row Filter* node. The node that transfers data table cells into an equal number of flow variables is the *Table Row to Variable* node. The *Table Row to Variable* node is located in the category "Workflow Control" → "Variables".

The "Workflow Control" category contains nodes that help to handle the data flow in the workflow. In particular, the sub-category "Variables" contains a number of utility nodes to deal with flow variables from inside the workflow, such as to transform data cells into variables and vice versa.

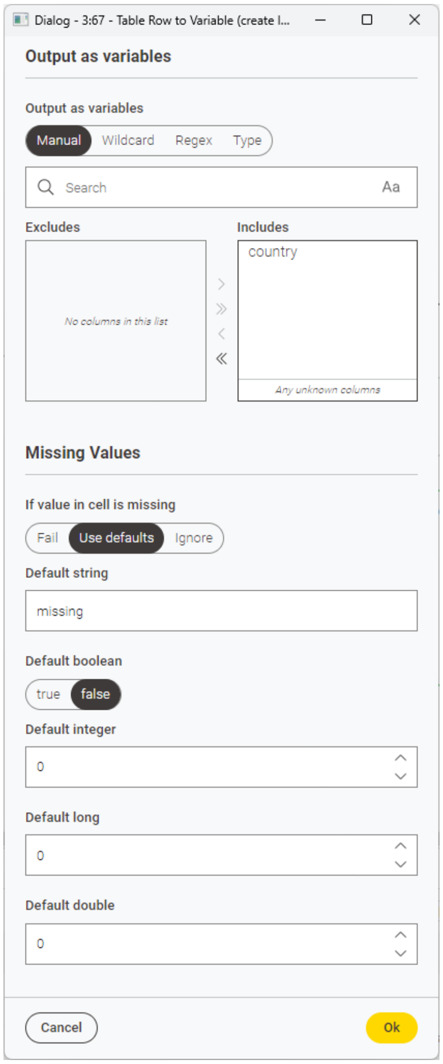
The *Table Row to Variable* node takes a data table at the input port (black triangle), transforms the data cells into flow variables, and places the resulting set of flow variables at its output port (red circle).

Table Row to Variable

The *Table Row to Variable* node requires the following settings:

- a handling strategy if the field to be transformed into a variable has a missing value. Such strategy can be drastic, such as:
  - fail the node execution or
  - omit the creation of the variable
  - or can be more tolerant and keep the node running and still create the desired flow variable. It just fills it with a fictitious value, such as a fixed string value, or a number fixed value
- the selection of the input fields to be transformed into flow variables. This is achieved as usual via an exclude/include frame.

Going back to the "Flow_Vars" workflow, we isolated the "country" data cell of the first row in the output data table of the *Sorter* node, by using an additional *Row Filter* node to keep the first row only and a *Column Filter* node to keep the "country" column only. The resulting data table has only one cell containing the name of the country with the highest number of sold items. This one-cell data table was then fed into a *Table Row to Variable* node and therefore transformed into a flow variable named "country".


Dialog - 3:67 - Table Row to Variable (create l...

Output as variables

Output as variables: [✓ Manual] Wildcard Regex Type

Search

Excludes: No columns in this list

Includes: country

Any unknown columns

Missing Values

If value in cell is missing: Fail [✓ Use defaults] Ignore

Default string: missing

Default boolean: true [✓ false]

Default integer: 0

Default long: 0

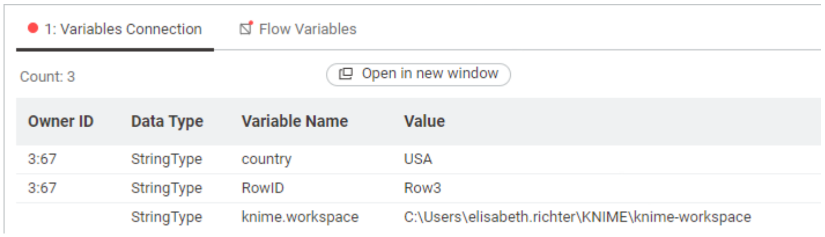
Default double: 0

Figure 5.3. Configuration window of the Table Row to Variable node.

Note. The output port of the *Table Row to Variable* node is a red circle. Red circle ports carry flow variables either as input or as output. The node creates as many flow variables as input columns, each flow variable carrying the name of the original input column. Also, it only works on the first row of the input data table. If more than one row is presented to the node, the newly created flow variables will take the values available in the first row of the corresponding input columns.

The list of flow variables available after the *Table Row to Variable* node can be seen by selecting the last item of the node context menu. This view contains the flow variables available for subsequent nodes. This includes all previously defined flow variables plus all the flow variables created by the *Table Row to Variable* node itself.

The view shows the new "country" variable with value "USA" for this execution run and the "RowID" variable which was also created by the *Table Row to Variable* node.

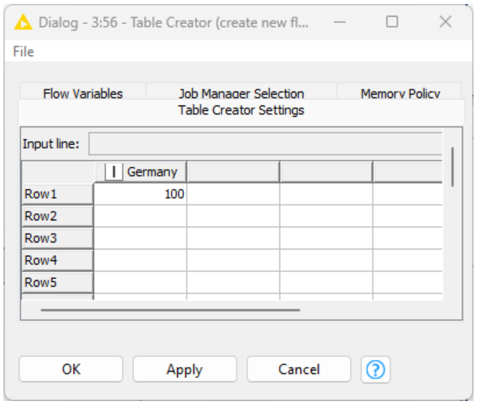


Owner ID	Data Type	Variable Name	Value
3:67	StringType	country	USA
3:67	StringType	RowID	Row3
	StringType	knime.workspace	C:\Users\elisabeth.richter\KNIME\knime-workspace

Figure 5.4. Flow Variables available in the workflow "Flow_Vars" after the Table Row to Variable node.

Export a Node Configuration as a Flow Variable

Let's suppose now to have a data table containing some values about specific countries. We simulated that table with a *Table Creator* node with just one column of values for the desired country (see figure on the right). The column name is the country name. We would like to extract all data rows for that country from the original input data table from file *sales.csv*. We should then use the column name as a matching pattern in the *Row Filter* node. It is actually possible to transfer the value of any configuration setting into the value of a new flow variable.



	Germany		
Row1	100		
Row2			
Row3			
Row4			
Row5			

Figure 5.5. The Table Creator node configuration window: the "Settings" tab.

The "Flow Variables" tab in each configuration window shows two boxes close to each setting name. One is the menu combo box that is used to overwrite the setting value with an existing flow variable, the other box is a textbox and implements the opposite pass. That is, it creates a new flow variable named according to the string in the textbox and containing the value of the corresponding configuration setting.

In the *Table Creator* node, the column name is actually a configuration setting. So, in the "Flow Variables" tab of its configuration window, we found the setting "columnProperties" > "0" > "ColumnName" which corresponds to the column name of the column number 0 in the configuration window of the *Table Creator*. We then decided to transfer this setting value into a new flow variable named “var_country” () by filling the second textbox with the desired flow variable name.

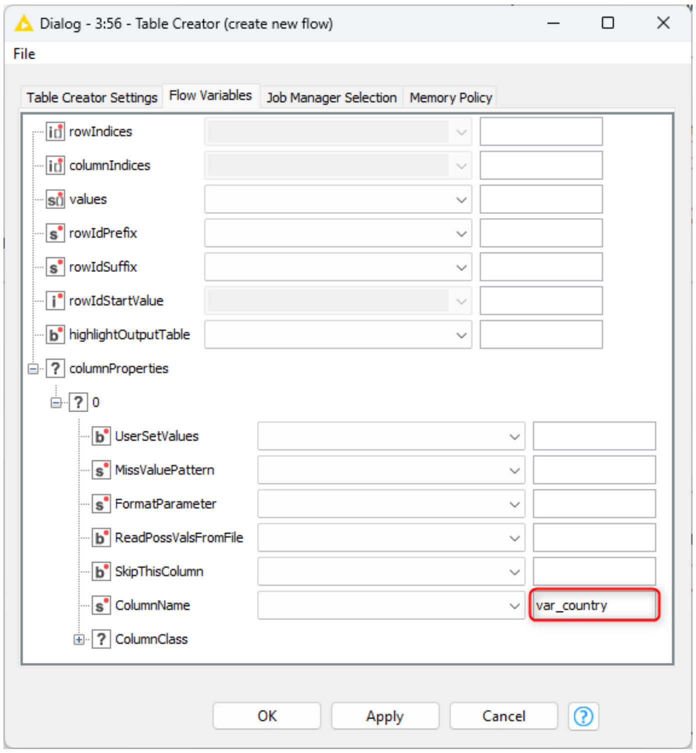
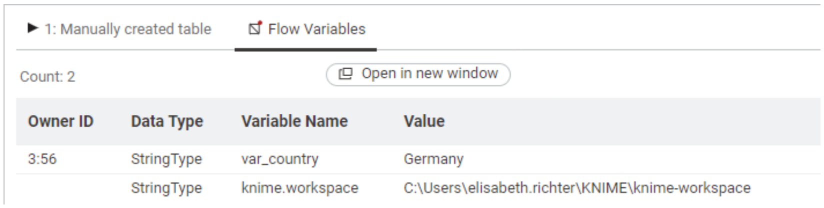


Figure 5.6. The Table Creator node configuration window: The “Flow Variables” tab.

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” with value “Germany”, exactly the name of column 0 in the *Table Creator* node.



Owner ID	Data Type	Variable Name	Value
3:56	StringType	var_country	Germany
	StringType	knime.workspace	C:\Users\elisabeth.richter\KNIME\knime-workspace

Figure 5.7. Output Flow Variables of the Table Creator node in the "Flow_Vars" workflow.

Configuration & Widget Nodes to Create Flow Variables

Another way to create a flow variable in the middle of a workflow is to use a Configuration node. Configuration nodes are located in the category "Workflow Abstraction" → "Configuration", can create flow variables, and provide interactive forms for a variety of tasks. The simplest Configuration nodes to generate flow variables are *Boolean Configuration*, *Double Configuration*, *Integer Configuration*, and *String Configuration*.

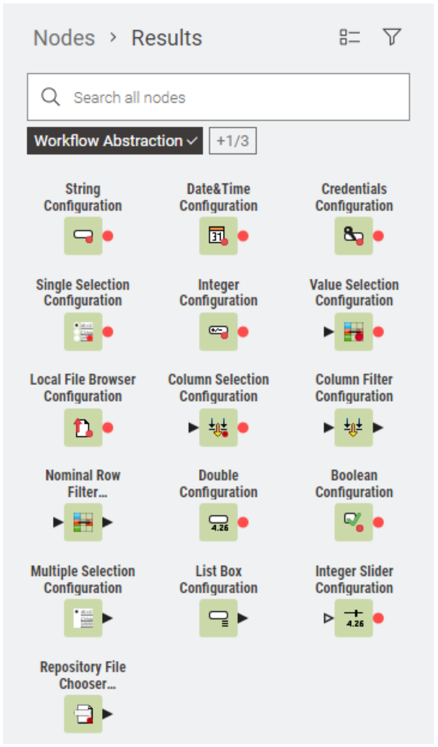

Nodes › Results
Search all nodes
Workflow Abstraction
+1/3
String Configuration
Date&Time Configuration
Credentials Configuration
Single Selection Configuration
Integer Configuration
Value Selection Configuration
Local File Browser Configuration
Column Selection Configuration
Column Filter Configuration
Nominal Row Filter...
Double Configuration
Boolean Configuration
Multiple Selection Configuration
List Box Configuration
Integer Slider Configuration
Repository File Chooser...

Figure 5.8. The simplest "Configuration" nodes to generate flow variables.

Configuration nodes can create a flow variable of a specific type, with a name and a value. Since they do no processing, they need no input. Thus, these nodes have no input and just one output port of flow variable type (red circle). They also share the same kind of configuration window, requiring: the name and the default value of the flow variable to be created, optionally some description of the flow variable purpose, and an explanation label to help the user for the assignment of new values.

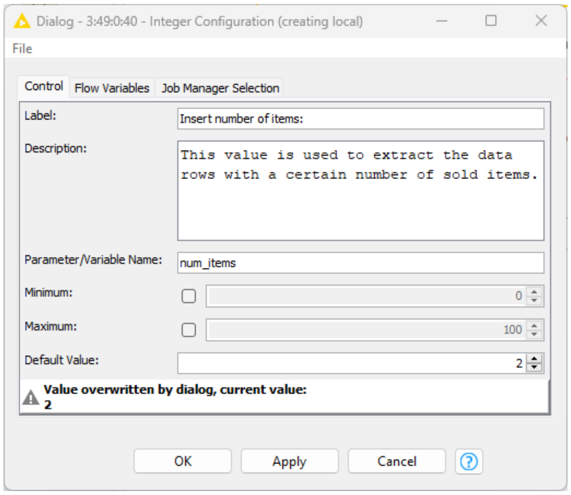
Let's have a look again at the example workflow "Flow_Vars" and particularly at the *Row Filter* node. We can use a Configuration node to create a flow variable containing the number of sold items for the filtering criterion called "num_items". This flow variable must be of type Integer to match the values in data column "quantity". Hence, we use an *Integer Configuration* node and configure is so that it produces a flow variable of type Integer named "num_items". We set the default value to 2.

The *Boolean Configuration* node, the *Double Configuration* node, and the *String Configuration* node are structured the same way as the *Integer Configuration* 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 Configuration* node.

Configuration nodes can be assembled inside a component, which is a special type of metanode. The component would then acquire a configuration window requiring the values of the flow variables generated within it. You can learn more about that later on in this chapter.

Integer Configuration

The *Integer Configuration* 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.



Label: Insert number of items:

Description: This value is used to extract the data rows with a certain number of sold items.

Parameter/Variable Name: num_items

Minimum: 0

Maximum: 100

Default Value: 2

Value overwritten by dialog, current value: 2

Figure 5.9. Configuration window of the "Integer Configuration" node.

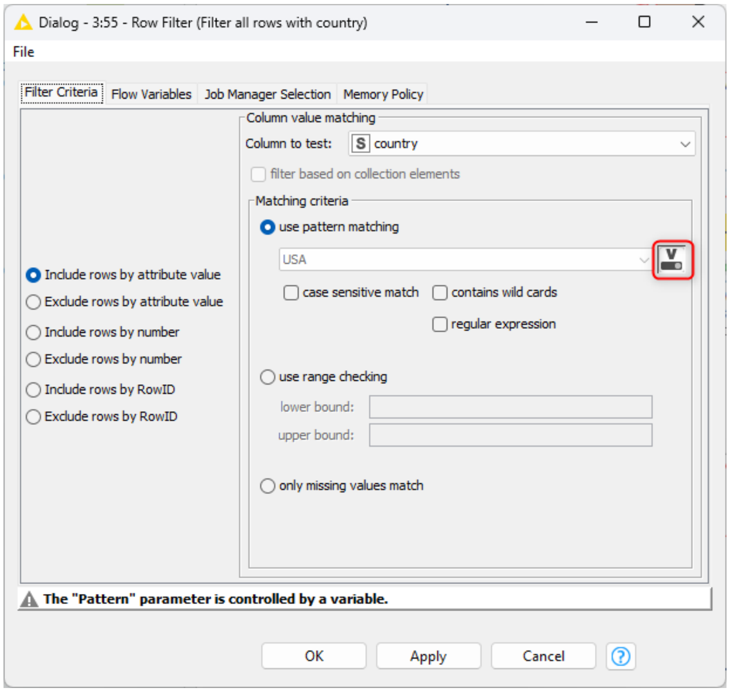
Its configuration window requires:

- A label. This label will help the user when updating the flow variable values via the configuration window of the component
- An optional description of the purpose of this flow variable
- The flow variable name. This will also be used as parameter name for external identification, for example when running the workflow in batch mode
- The flow variable default value
- An integer range (Minimum and Maximum) to verify and accept valid values

5.3. Flow Variable Values as Node Settings

Once a flow variable has been created, it can be used to overwrite the settings in a node configuration window. In our example, in the workflow named "Flow_Vars", the flow variable "country" overwrites the value of the matching pattern in the filtering criterion of the *Row Filter* node.

In the configuration window of some nodes, a "Flow Variable" button is displayed on the side of some of the settings. For example, you can find it in the *Row Filter* node, in the "matching criteria" panel for the "use pattern matching" option, to the right of the "pattern" textbox (see figure below).


Dialog - 3:55 - Row Filter (Filter all rows with country)

File

Filter Criteria | Flow Variables | Job Manager Selection | Memory Policy

Column value matching

Column to test: S country

☐ filter based on collection elements

Matching criteria

● use pattern matching

USA

☐ case sensitive match ☐ contains wild cards

☐ regular expression

○ use range checking

lower bound:

upper bound:

○ only missing values match

● Include rows by attribute value
○ Exclude rows by attribute value
○ Include rows by number
○ Exclude rows by number
○ Include rows by RowID
○ Exclude rows by RowID

The "Pattern" parameter is controlled by a variable.

Figure 5.10. The "Flow Variable" button in the configuration window of the Row Filter node.

The “Flow Variable” Button

The “Flow Variable” button allows you to use the value of a flow variable to overwrite the value of the corresponding node setting.

By clicking the “Flow Variable” button, the “Variable Settings” window opens and asks whether:

- The value from an existing flow variable should be used for this setting. If yes, then:
    - Enable the “Use Variable” flag
    - Select one of the existing flow variables from the menu
    - The value of the selected flow variable now defines the value of this setting at execution

OR

- A new flow variable should be created with the current value of this node setting. In this case:
    - Enable the “Create Variable” flag

- Provide a name for the new flow variable
- A new flow variable with the selected name and with that setting value is created and made available to all subsequent nodes in the workflow

In the "Flow_Vars" workflow, we selected the existing flow variable "country" – created in section 5.2 – to overwrite the value of the pattern matching setting in the *Row Filter* node named "country = "country" via pattern matching criterion".

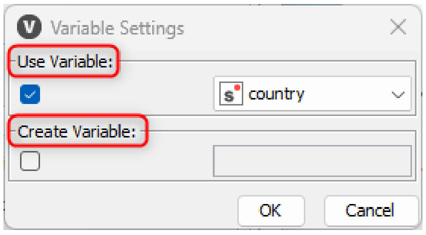


Figure 5.11. The "Variable Settings" window.

You can check which flow variables are available at a given point in a workflow by selecting the "Flow Variables" tab in the Node Monitor. The "Flow Variables" tab contains the full list of flow variables, with their current values, available for that node.

The "Flow Variables" Tab in the Configuration Window

Not all configuration settings display a "Flow Variable" button. The "Flow Variable" button has been implemented for only some settings in some nodes. A more general way to overwrite the value of a configuration setting through the value of a flow variable involves the "Flow Variables" tab in the node's configuration window. To access the "Flow Variables" tab, open the configuration window of a node and select the "Flow Variables" tab.

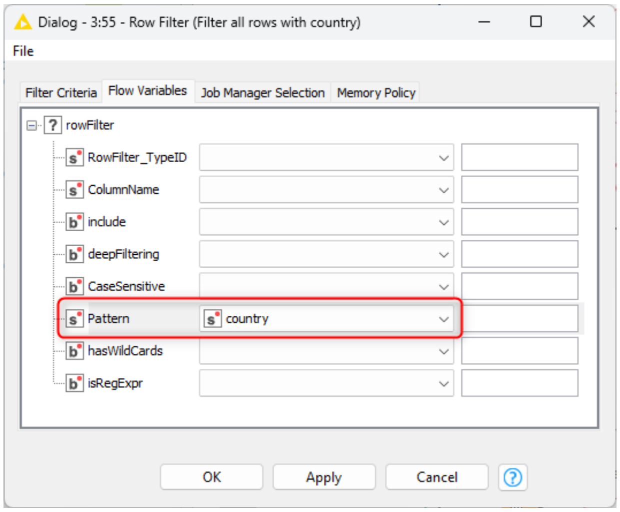
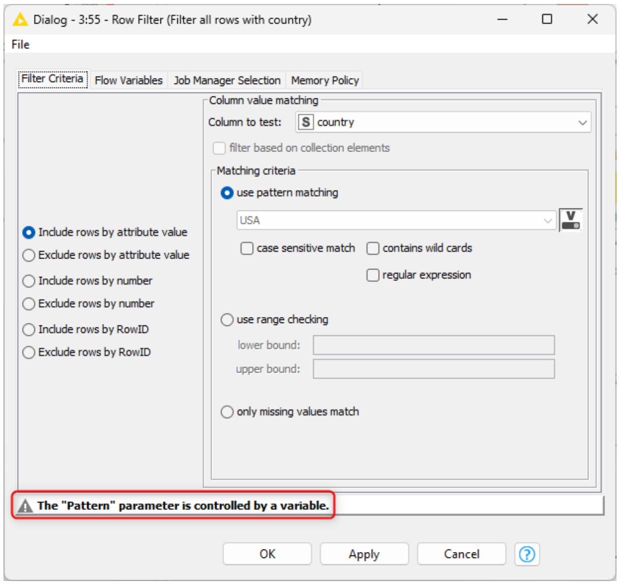


Figure 5.12. The "Flow Variables" tab in the configuration window of the Row Filter node.

The “Flow Variables” tab allows to overwrite each of the node configuration settings with the value of a flow variable. Find the desired configuration setting and open the menu on the right containing the list of available flow variables. Then simply select the respective flow variable to overwrite the setting. At execution time, the node will assign the value of the selected flow variable to that setting.

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



The "Pattern" parameter is controlled by a variable.

Figure 5.13. The warning message indicating that a particular parameter is controlled by a flow variable.

In order to make effective the manual change of the parameter value, the user needs to disable the flow variable for this setting. This can be obtained by either disabling both options in the “Variable Settings” window accessible through the “Flow Variables” button (Figure 5.11), or by selecting the first empty option in the combo box menu in the “Flow Variables” tab of the configuration window.

Note. In rare cases, it is still possible that some settings are not accessible even through the “Flow Variables” tab.

Inject a Flow Variable through the Flow Variable Ports

At this point, in our example workflow, we have four flow variables to feed a possible *Row Filter* node:

- "country" which contains the name of the country with the highest number of sold items
- "var_country" which contains the name of the country in the reference table from the "Table Creator" node
- "num_items" generated by the "Integer Configuration" node.

Let's concentrate for now on the flow variable "country". This flow variable is at the output of the *Table Row to Variable* node. How do we make a *Row Filter* node aware of the existence of this new flow variable? How do we connect the *Table Row to Variable* node to the *Row Filter* node? We need to insert, or inject, the new flow variables back into the workflow, to make them available for all subsequent nodes.

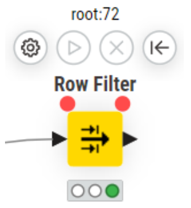


Figure 5.14. The flow variable ports become visible when you hover over the node with the cursor.

All KNIME nodes have visible data ports and hidden flow variable ports which become visible when you hover over the node with your cursor (two red circle ports on the top). We use these flow variable ports to inject flow variables from one node to the next or from one workflow branch to another.

Note. 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.

In order to inject flow variables into a node, connect the output variable port of the preceding node to the input variable port (i.e., the left one of the two flow variable ports) of the current node.

The "Flow Variables" tab in the Node Monitor contains the full list of available flow variables and after injecting a new flow variable the list should include the newly created flow variable as well as the flow variable that existed beforehand.

The flow variables injected into a node are only available for this and the subsequent nodes in the workflow.

Note. Flow variable injection becomes necessary only when we need to transfer a flow variable from one branch of the workflow to another or when we need to connect a flow variable output port to the next node. In all other cases, new flow variables are automatically transferred from one node to the next through the data flow (black triangle ports).

In "Flow_Vars" workflow, we first introduced a *Row Filter* node after the *Table Row to Variable* node. Then, we connected the flow variable output port of the *Table Row 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 group of 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 value of the flow variable “country” as the matching pattern.

Note. Flow Variable ports can also be used as a barrier point to control the execution order of nodes, i.e. nodes connected through a flow variable line will not start executing until all upstream nodes have been executed.

Merge Variables

Sometimes we might want both flow variables, “country” and “var_country”, to be available to a *Row Filter* node. In this case, we need to first merge the two (or more) flow variables from different branches of a workflow before injecting them into the following node. This can be done with the *Merge Variables* node.

The *Merge Variables* node merges flow variables into one stream. If flow variables with the same name are to be merged, the standard conflict handling is applied: the topmost inputs have higher priority and define the value of the post-merging variable. This node needs no configuration settings.

5.4. Configurations, Widgets, Components

Let’s go back to that part of the “Flow_Vars” workflow that used a Configuration node (an *Integer Configuration* node) to overwrite the value of the upper and lower bound of the range checking criterion in the *Row Filter* node. Let’s add here a second Configuration node, a *String Configuration* node, to define the name of the output CSV file where to write the row filtering results.

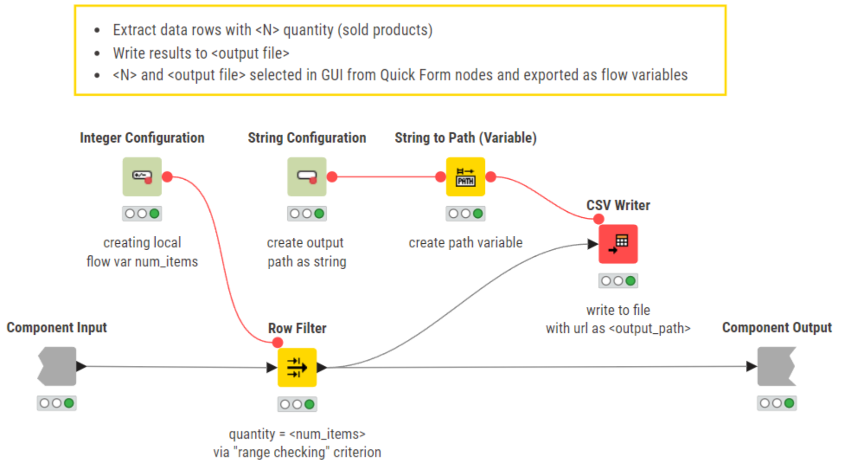
Often workflows become quickly too crowded with too many nodes. In order to clean up the workflow and collect together nodes belonging to the same logic unit, metanodes can be introduced. Metanodes are nodes containing other nodes. To create a new metanode from a set of existing nodes, select the nodes of interest, then right-click and select option “Create metanode”. The metanode is automatically created including all selected nodes and with the right number of input and output ports. Double-clicking a metanode opens its content.

An evolution of a simple metanode is a component. A component is a building block that can have sophisticated dialogs and composite views. It can be created by selecting the nodes of interest, right-clicking and selecting the option “Create component”.

- Ctrl-double-click on a component opens its content.
- If a component contains one or more Configuration nodes, it acquires a configuration window, that is, the option "Configure" in the component context menu becomes active. The configuration window is filled with the textboxes and menus from the Configuration nodes in it.
- Similarly, if a component encloses JavaScript views, they become visible as composite view of the component.

A component limits the input and output of flow variables in and out of the component. This vacuum environment is useful against the quick proliferation of flow variables. Content isolation ensures safer coding inside the component with lower risk of mixing up flow variables and their values. It is still possible to import or export a flow variable into or out of a component, if we set this explicitly.

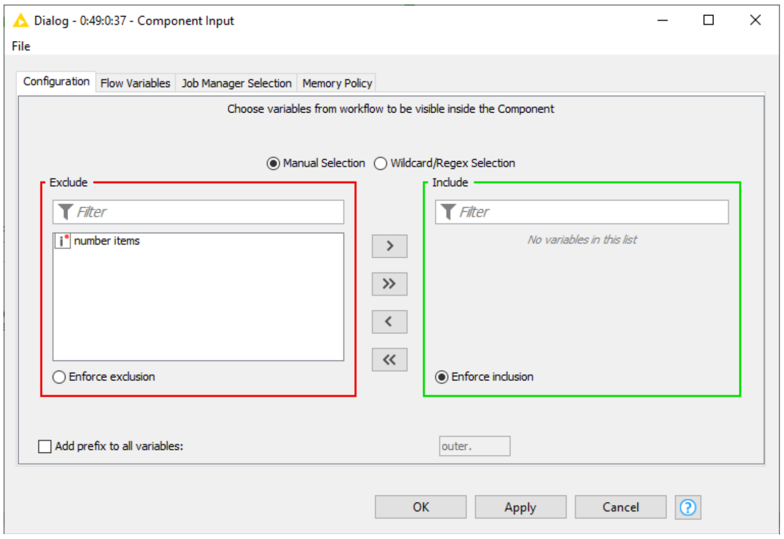
In the workflow "Flow_Vars" we enclosed the *Integer Configuration* node and the *String Configuration* node into a component named "Filter data rows and write results to output file". Ctrl+Double-click accesses its content (Figure 5.15).



- Extract data rows with <N> quantity (sold products)
- Write results to <output file>
- <N> and <output file> selected in GUI from Quick Form nodes and exported as flow variables
Integer Configuration
creating local flow var num_items
String Configuration
create output path as string
String to Path (Variable)
create path variable
CSV Writer
write to file with url as <output_path>
Component Input
Row Filter
quantity = <num_items> via "range checking" criterion
Component Output

Figure 5.15. "Integer Configuration" and "String Configuration" nodes inside a component to define the row filtering criterion and the output file path.

You can see two gray nodes: the Component Input and the Component Output node. The Component Input and the Component Output nodes in the component have a configuration window. Double-clicking them opens their configuration window, which offers an Include/Exclude framework to allow flow variables to enter or exit the component.



Choose variables from workflow to be visible inside the Component

[✓ Manual Selection] Wildcard/Regex Selection

Exclude: number items

Enforce exclusion

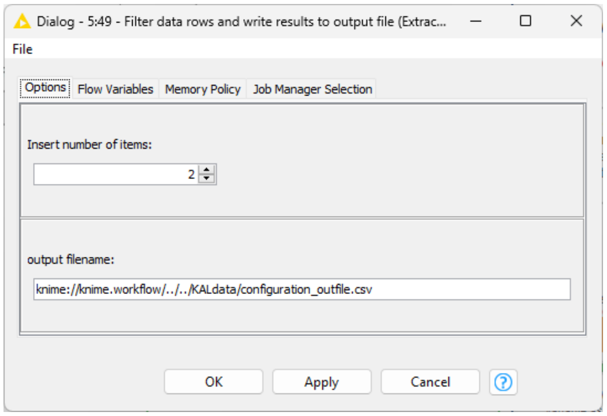
Include: No variables in this list

[✓ Enforce inclusion]

Add prefix to all variables: outer.

Figure 5.16. Configuration window of the "Component Input" node in the component shown in Figure 5.15.

The component, shown in Figure 5.15, containing an *Integer Configuration* node and a *String Configuration* node, produces the configuration window in Figure 5.17, requiring the output file path and the integer value for the flow variable "num_items" controlling the *Row Filter* node.


Dialog - 5:49 - Filter data rows and write results to output file (Extrac...
File
Options
Flow Variables
Memory Policy
Job Manager Selection
OK
Apply
Cancel

Insert number of items: 2

output filename: knime://knime.workflow/../../KALdata/configuration_outfile.csv

Figure 5.17. Newly acquired configuration window of the component containing the "Integer Configuration" node and the "String Configuration" node. The box to enter integer values comes from the Integer Configuration node. The textbox to insert the file path comes from the String Configuration node.

Note. Unlike meta-nodes, double-clicking a component leads to its configuration window and not the sub-workflow.

Another way of creating a new flow variable in a component is via the Widget nodes. Similar to the Configuration nodes, they reside in the category “Workflow Abstraction”. Also, here we find dedicated Widget nodes for different types of flow variables such as Boolean Widgets, String Widgets and Integer Widgets. The key difference to the Configuration nodes is that Widget nodes alter the view of a component (composite view) and not the configuration window. When running on a KNIME Business Hub, each component produces a web page displaying the same view as in the composite view of the component. We will talk about Widget nodes later in this chapter.

5.5. Transform a Flow Variable into a Data Value

Sometimes it is necessary to use the flow variable values as data values. To transform flow variables into data cells, we can use the *Variable to Table Row* node. The *Variable to Table Row* node mirrors the *Table Row to Variable* node. That is, it transforms a number of selected flow variables into cells of a data table.

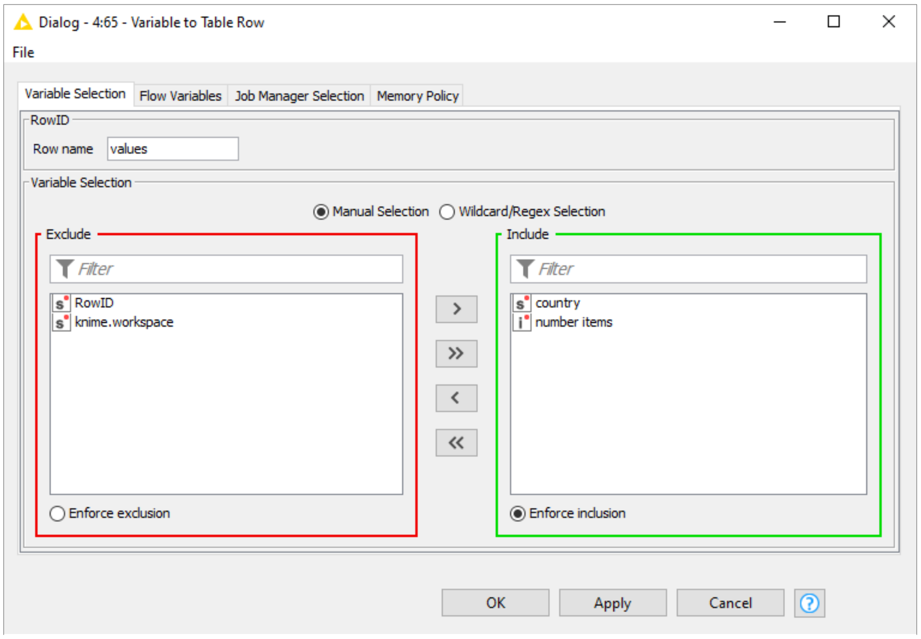
Variable to Table Row

The *Variable to Table Row* node transforms selected flow variables at the input port (red circle) into cells of a data table at the output port (black triangle). The configuration settings require the selection of the flow variables to be included in the output data table through an include/exclude panel.

The output data table contains two columns:

- One column with the name of the flow variable;
- The second column with the flow variable values at execution time.

In our example workflow, “Flow_Vars”, we connected a “Variable to Table Row” node to the “Table Row to Variable” node, in order to perform the opposite operation and transform all current flow variables into data cells.


Dialog - 4:65 - Variable to Table Row

File

Variable Selection | Flow Variables | Job Manager Selection | Memory Policy

RowID
Row name: values

Variable Selection
(● Manual Selection) ○ Wildcard/Regex Selection

Exclude: RowID, knime.workspace
○ Enforce exclusion

Include: country, number items
● Enforce inclusion

Figure 5.18. The configuration window of the Variable to Table Row node.

5.6. Modifying Flow Variable Values

For the country with the highest number of sales, we now want to restrict the analysis scope to only those records with a high value in the "amount" column. This requires an additional *Row Filter* node implementing a filtering criterion on "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 should set `x = 500` and in the other cases `x = 100`. Therefore, the threshold x cannot be set as the static value of a global flow variable. Indeed, the flow variable value needs to be adjusted as the workflow runs.

There are a number of nodes dedicated to modifying the values in flow variables. They all mirror existing nodes operating on data tables, like Java Snippet, Math Formula, Rule Engine, and String Manipulation. The only difference of the variable edition with respect to the data edition of these nodes consists in the input/output ports. The nodes operating on data get data as input and produce data as output. The corresponding nodes operating on variables get variables as input and produce variable as output. Besides that, the node configuration windows mimic the configuration windows of the corresponding nodes working on data.

To complete our example workflow “Flow_Vars”, we have added a few nodes for variable value manipulation.

A “Rule Engine Variable” and a “Java Edit Variable” both implement the task of creating the varying threshold x, that is equal to 500 if variable “country” value is “USA” and to 100 otherwise.

A “String Manipulation Variable” node transforms the content of flow variable “country” from just the country name to “Country: <country name>”. At the same time, a “Math Formula Variable” node doubles the value of flow variable “number items”.

Note. The type of the returned flow variable has to be consistent with the return type of the Java/Rule Engine/Math Formula/String Manipulation code.

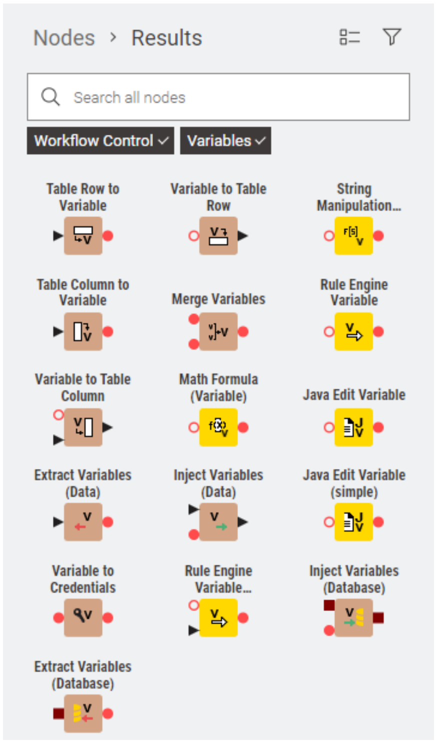
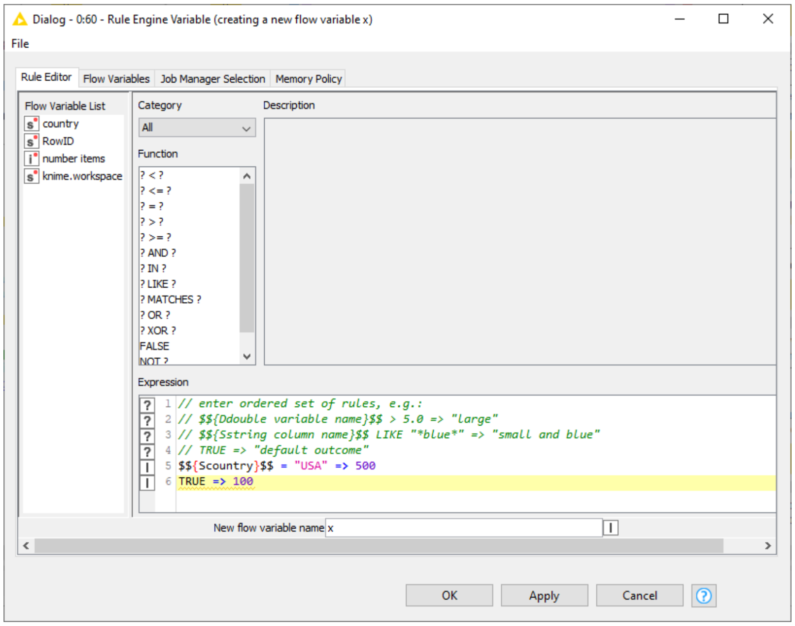


Figure 5.19. Nodes modifying values in flow variables. These nodes mimic the corresponding nodes working on data. That is, same configuration window, but variable input and output ports rather than data ports.


Dialog - 0:60 - Rule Engine Variable (creating a new flow variable x)

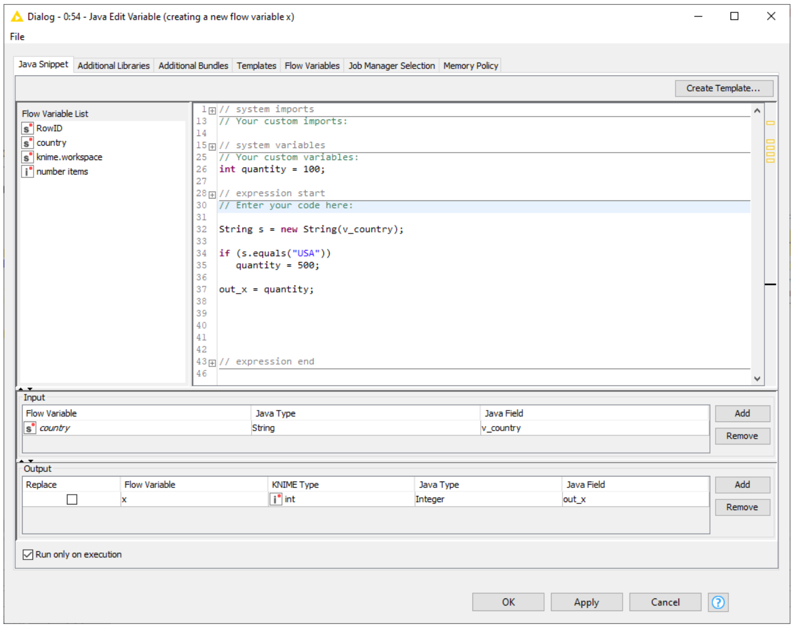
Flow Variable List: country, RowID, number items, knime.workspace

Expression:

```
// enter ordered set of rules, e.g.:
// $${Ddouble variable name}$$ > 5.0 => "large"
// $${Sstring column name}$$ LIKE "*blue*" => "small and blue"
// TRUE => "default outcome"
$${Scountry}$$ = "USA" => 500
TRUE => 100
```

New flow variable name: x

Figure 5.20. Configuration window of the "Rule Engine Variable" node to create a flow variable named "x" with value 500 if flow variable "country" is set to "USA" and 100 otherwise.


Dialog - 0:54 - Java Edit Variable (creating a new flow variable x)

Flow Variable List: RowID, country, knime.workspace, number items

```
// system imports
// Your custom imports:

// system variables
// Your custom variables:
int quantity = 100;

// expression start
// Enter your code here:

String s = new String(v_country);

if (s.equals("USA"))
   quantity = 500;

out_x = quantity;





// expression end
```

Input

Flow Variable	Java Type	Java Field
country	String	v_country

Output

Replace	Flow Variable	KNIME Type	Java Type	Java Field
☐	x	int	Integer	out_x

☑ Run only on execution

Figure 5.21. Configuration window of the "Java Edit Variable" node to create a flow variable named "x" with value 500 if flow variable "country" is set to "USA" and 100 otherwise.

Note. Even though all these "… Variable" nodes have an input port for flow variables, this port is often optional. That is, the nodes can also work without input, which makes them ideal nodes to start a workflow without reading data.

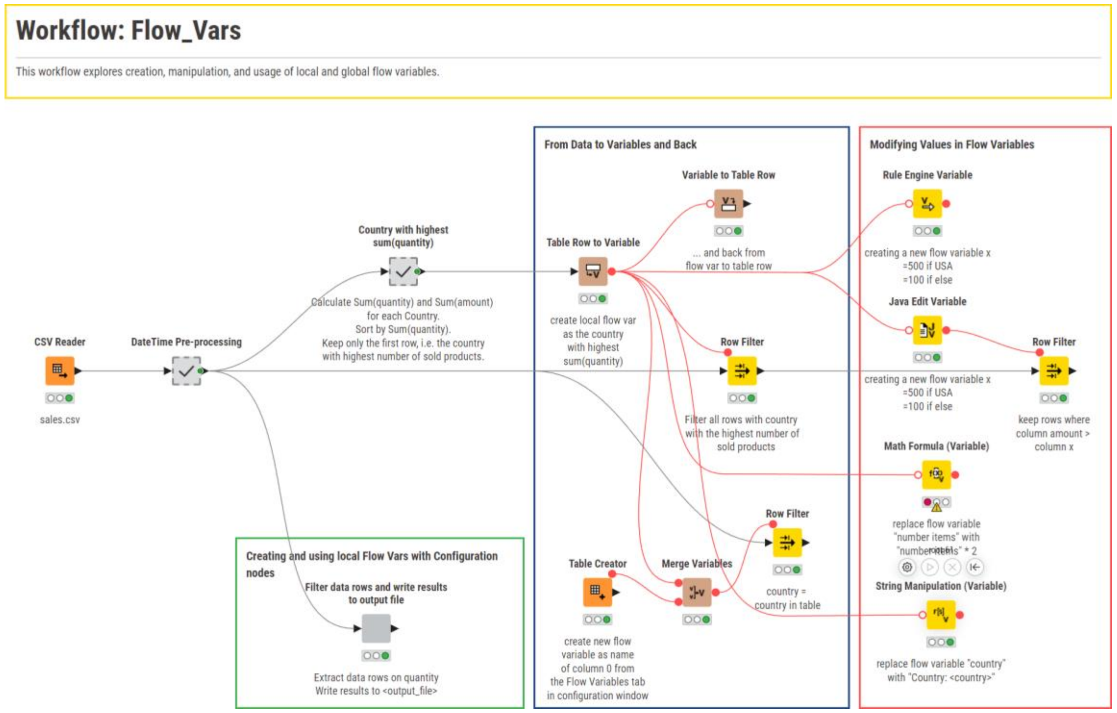

Workflow: Flow_Vars
This workflow explores creation, manipulation, and usage of local and global flow variables.
CSV Reader
sales.csv
DateTime Pre-processing
Country with highest sum(quantity)
Calculate Sum(quantity) and Sum(amount) for each Country. Sort by Sum(quantity). Keep only the first row, i.e. the country with highest number of sold products.
Creating and using local Flow Vars with Configuration nodes
Filter data rows and write results to output file
Extract data rows on quantity Write results to <output_file>
From Data to Variables and Back
Table Row to Variable
create local flow var as the country with highest sum(quantity)
Variable to Table Row
... and back from flow var to table row
Row Filter
Filter all rows with country with the highest number of sold products
Table Creator
create new flow variable as name of column 0 from the Flow Variables tab in configuration window
Merge Variables
Row Filter
country = country in table
Modifying Values in Flow Variables
Rule Engine Variable
creating a new flow variable x =500 if USA =100 if else
Java Edit Variable
creating a new flow variable x =500 if USA =100 if else
Row Filter
keep rows where column amount > column x
Math Formula (Variable)
replace flow variable "number items" with "number items" * 2
String Manipulation (Variable)
replace flow variable "country" with "Country: <country>"

Figure 5.22. The final workflow in "Chapter5/Flow_Vars".

5.7. More Configuration Nodes and Widget Nodes

In the remaining part of this chapter, we would like to explore some of the many options offered by the nodes in the "Workflow Abstraction" category and the possibility to build composite, complex, and organized views in components. In this section we concentrate on the nodes in the "Configuration and "Widgets" category.

In the previous sections of this chapter, we have encountered two Configuration nodes: the *Integer Configuration* node and the *String Configuration* node. Like all Configuration nodes, these two nodes provide an input form to enter the flow variable value.

A component containing Configuration nodes acquires a configuration window, including the input forms from the contained Configuration nodes. A similar, more complex, configuration window could be displayed as a web page using Widget nodes instead of the Configuration nodes.

The Widgets category offers a wide choice of nodes similar to the Configuration nodes, but with more complex and flexible input forms, producing items in web pages rather than in configuration windows. In particular, the *Integer Widget* and *String Widget* nodes are the corresponding nodes to the *Integer Configuration* and *String Configuration* nodes.

In this section, we will explore two interesting and widely used nodes from the Configuration and Widgets categories: the *Local File Browser Configuration* node (in Configuration) and the *Value Selection Widget* node (in Widgets).

The *Local File Browser Configuration* node is used to trigger the File Browser User Interface, to find, select, and import the file URL as a flow variable. The configuration window of the component therefore will contain a "Browse" button to search for file. The file URL variable should then be passed to a *File Reader* node that reads and imports the data set from the file.

The *Value Selection Widget* node operates a selection on the distinct values provided in the selected input column. This Widget node produces a web page item visible in the node view: a drop-down menu to select one of available values in the input column.

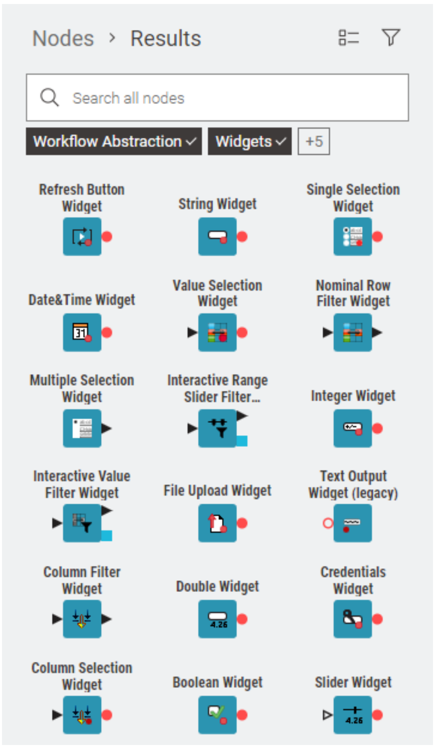


Figure 5.23. The "Workflow Abstraction" category with expanded "Widgets" sub-category.

Value Selection Widget

The *Value Selection Widget* node extracts a list of unique values from a column in the input data table and loads them in a menu or a group of radio buttons for selection. The menu or the radio buttons are displayed in the view of the node itself, in the view of the component where the node has been included, and in the stepwise execution on the KNIME Business Hub.

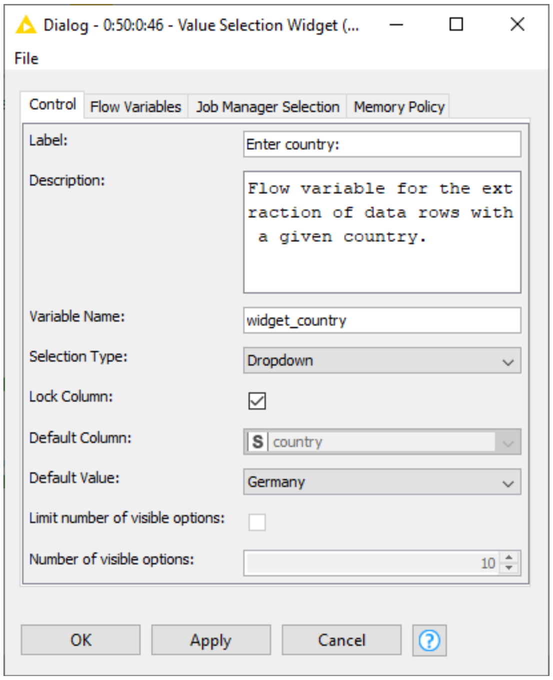
A value needs to be selected from the input form. Then a flow variable is created containing the selected value to be passed over to the subsequent part of the workflow. To configure the node, we need at least:

- The type of input form, whether menu or radio buttons, to display in the node view, component view, and the KNIME Business Hub from a web browser.

- The name of the flow variable to create
- The name of the input column with the list of unique values for the menu / radio buttons
- Since the selected column can be changed via the input form, the checked "Lock Column" flag avoids that
- The default value for the flow variable from the menu list; this value can be changed via the input form

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.

To display the node view, right-click the node and in the context menu select the "Interactive View: …" option. This visualizes a web page containing the input form with the list of values to choose from.



Label: Enter country:
Description: Flow variable for the extraction of data rows with a given country.
Variable Name: widget_country
Selection Type: Dropdown
Lock Column: ✓
Default Column: S country
Default Value: Germany
Limit number of visible options:
Number of visible options: 10

Figure 5.24. The configuration window of the Value Selection Widget node.

Local File Browser Configuration

The *Local File Browser Configuration* node explores the folder in the default file path, loads all found files into a menu, allowing to select one. The node produces a flow variable containing the URL of the selected file. The File Browser GUI is the input form produced by this node, visible in the configuration window of the component that contains it.

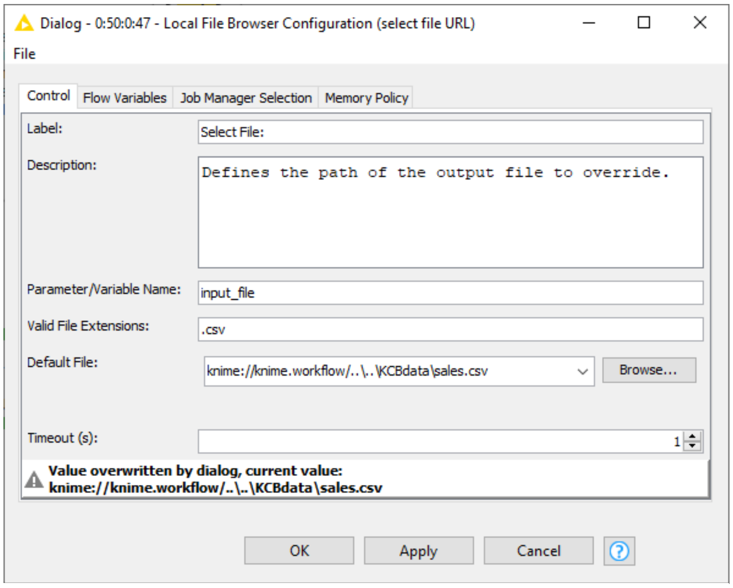
Usually, this node is connected to a reader node, like a *File Reader* node, or to a writer node, like a *CSV Writer* node, for importing/exporting a data table from/to a file. To configure the node, we need:

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

Optionally:

- The file extension to limit the number of files uploaded in the menu by their extension
- A description of the flow variable content

- A label that instructs on what to select



Dialog - 0:50:0:47 - Local File Browser Configuration (select file URL)

File

Control | Flow Variables | Job Manager Selection | Memory Policy

Label: Select File:

Description: Defines the path of the output file to override.

Parameter/Variable Name: input_file

Valid File Extensions: .csv

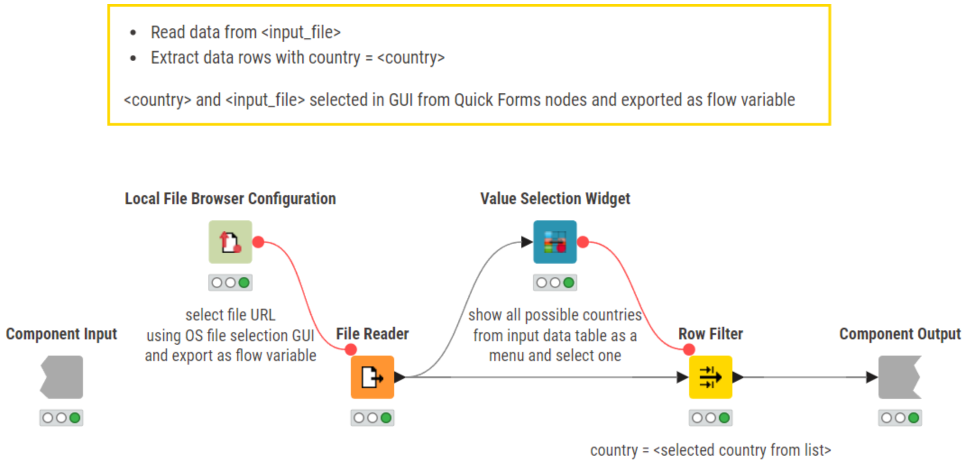
Default File: knime://knime.workflow/..\..\KCBdata\sales.csv

Timeout (s): 1

Value overwritten by dialog, current value: knime://knime.workflow/..\..\KCBdata\sales.csv

Figure 5.25. The configuration window of the Local File Browser Configuration node.

Summarizing, Configuration nodes produce items for configuration windows and Widget nodes items for web pages. When to use one or the other?



- Read data from <input_file>
- Extract data rows with country = <country>

<country> and <input_file> selected in GUI from Quick Forms nodes and exported as flow variable

Local File Browser Configuration
Value Selection Widget
Component Input
select file URL using OS file selection GUI and export as flow variable
File Reader
show all possible countries from input data table as a menu and select one
Row Filter
Component Output
country = <selected country from list>

Figure 5.26. Sub-workflow in component named “Select Country from list” with the Configuration and Widget nodes: a “Local File Browser Configuration” and “Value Selection Widget”.

Note. Input forms in web pages from Widget nodes are required when the workflow has to run on a web browser via the KNIME Business Hub. For a classic workflow execution within a KNIME Analytics Platform instance, input forms produced by Configuration nodes are sufficient.

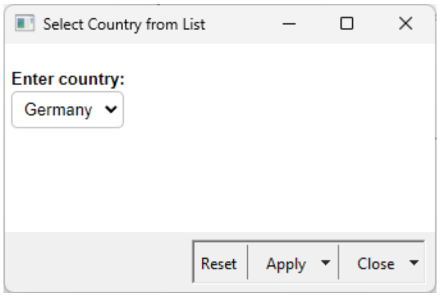


Figure 5.27. View of component "Select Country from list". The drop-down was created by the "Value Selection Widget".

In workflow "Composite_Views" under "Chapter5", we have introduced a *Value Selection Widget* node and a *Local File Browser Configuration* node inside the component named "Select Country from List". The first node adds a "Browse" button into the component configuration window, the latter adds a dropdown menu filled with country names in the component view.

Those two categories, Configuration and Widgets, contain many other useful nodes. For example, the *Multiple Selection Configuration/ Multiple Selection Widget* and *Single Selection Configuration/ Single Selection Widget* nodes in execution both present a list of values in a menu and allow for the selection of one or more of them.

The *Nominal Row Filter Configuration/ Nominal Row Filter Widget* nodes work similarly to the *Value Selection Configuration/ Value Selection Configuration Widget* nodes but return a table with the one or more selected values instead of a flow variable.

The *Column Selection Configuration/ Column Selection Configuration Widget* nodes let the user select a data column and transmit the name of the selected column to the following nodes by means of a flow variable.

5.8. Composite View in Components

As we have stated already, components include input forms from internal Configuration nodes in their configuration window. They also include views from internal JavaScript nodes in their view window. Both input forms and views get displayed on a web browser via the KNIME Business Hub during execution. So far, we have also seen that elements in the component's view and configuration window are arranged in a vertical sequence. It does not have to be just vertical!

If you are more artistically inclined and would like to place the component's items on a grid, you can do so using the layout editor. Access the layout editor

- by right-clicking the component and select Component > Open layout editor, or

- by opening the component (ctrl-double-click or right-click → "Component" → "Open") and clicking on the "Open layout editor" button in the toolbar (Figure 5.28).

The "Node Usage and Layout" window opens. In this window, items are organized on a grid. Here you can place the input form or the view in the desired position in the view or configuration window or web page.

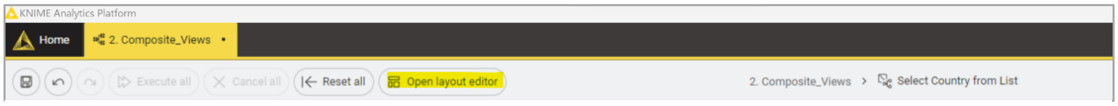
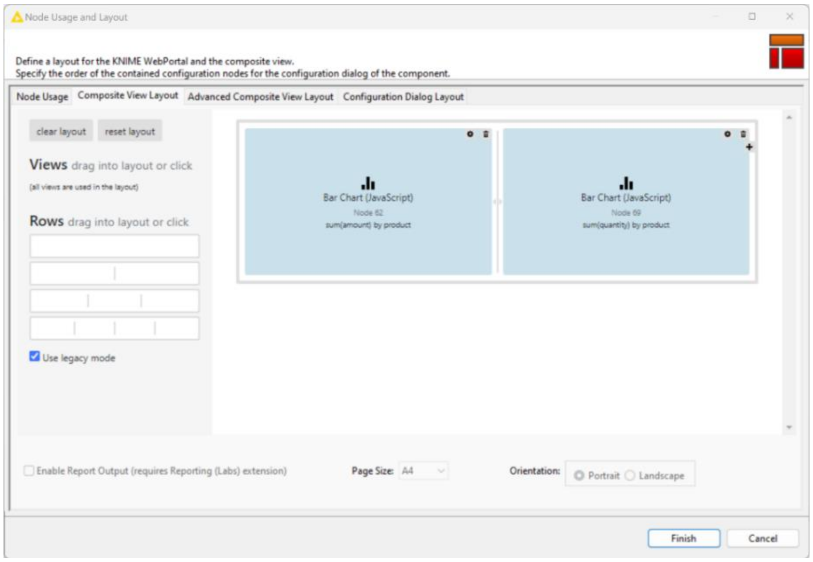


Figure 5.28. The Layout button in the tool bar of KNIME Analytics Platform. This button is enabled only when we are inside a component.

There are many ways to place items in the layout grid of a component via the four tabs of the "Node Usage and Layout" window.

- "Node Usage" sets whether the view or input form will be available in the component configuration window (dialog) and/or in the component View.
- "Composite View Layout" provides a drag and drop layout solution. On the left, layout sub-grids on a row are available: 1x1, 1x2, 1x3, or 1x4. Clicking on one of these sub-grids automatically adds them to the layout. Via drag&drop, you can then place the views of the *Widget* nodes and the forms of the *Component* nodes in any of the cells in the layout grid. Notice the trash bin button in the top right corner of each cell to empty it of its content.
- "Advanced Composite View Layout" lets you place the items on the same view grid, but via a JSON structure.


Node Usage and Layout

Define a layout for the KNIME WebPortal and the composite view.
Specify the order of the contained configuration nodes for the configuration dialog of the component.

Node Usage
Composite View Layout
Advanced Composite View Layout
Configuration Dialog Layout
clear layout
reset layout
Views drag into layout or click
(all views are used in the layout)
Rows drag into layout or click
Use legacy mode
Bar Chart (JavaScript)
Node 62
sum(amount) by product
Bar Chart (JavaScript)
Node 69
sum(quantity) by product
Enable Report Output (requires Reporting (Labs) extension)
Page Size: A4
Orientation: Portrait Landscape
Finish
Cancel

Figure 5.29. The "Node Usage and Layout" window to arrange views and input forms on a grid for a component.

In the workflow “Composite_Views”, the component named “Bar Charts side by side” contains two bar charts built on the data of the file sales.csv. The two bar charts are built with two “Bar Chart” nodes summing for each product once the number of sold items (“quantity”) and once the money amount (“amount”). We wanted to have the two bar charts displayed side by side in the component view. Thus, in the “Node Usage and Layout” window we placed the first chart in the left cell and second chart in the right cell of the first row, as shown in the figure above.

Notice that all view nodes are connected. Selecting groups/points/data rows in one view, automatically selects the same groups/points/data rows in the other view, if so set in the configuration window and/or in the interactive menu of the view. In the figure above, selecting the group of data for one product (“prod_1”) in one chart automatically selects the same group of data in the other chart.

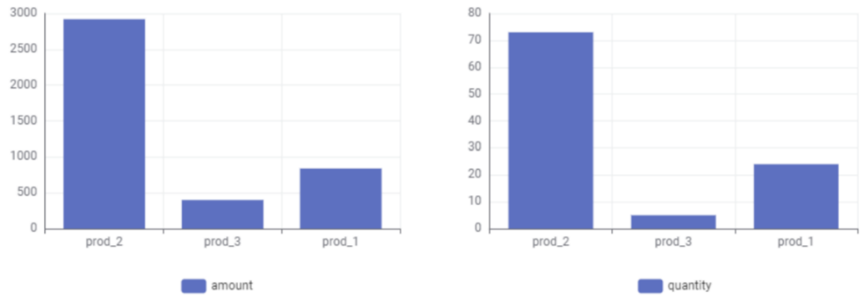


Figure 5.30. View of the component named “Bar Chart side by side” containing two bar charts generated by two “Bar Chart (JavaScript)” nodes. The two bar charts have been placed side by side through the “Node Usage and Layout” window.

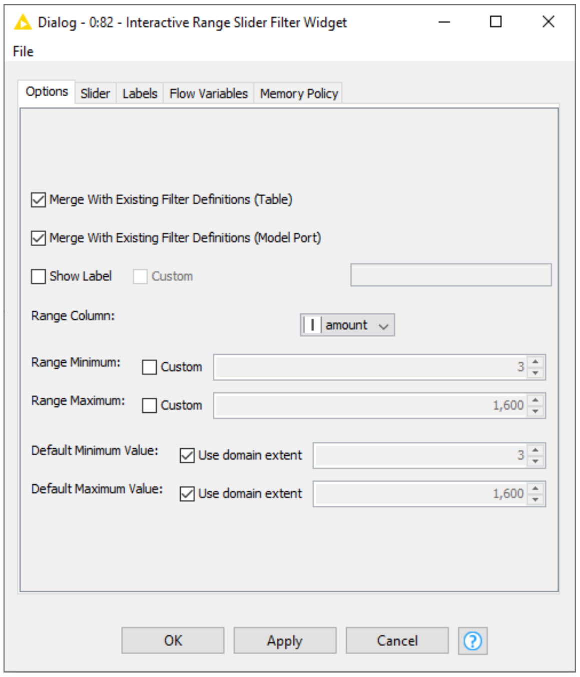
The last mention should go to a group of special nodes that offers control functions for charts and plots in a composite view. We will show here just one for all the nodes of this type: the “Interactive Range Slider Filter Widget” node.

The “Interactive Range Slider Filter Widget” node operates on numerical columns. It defines the range allowed for the numbers displayed in the view of the upcoming JavaScript nodes. It also produces a visual JavaScript item in the form of a slider to be used for the range definition. In the view, changing the numerical range set in the slider, affects the number of records displayed in the following charts and plots.

Interactive Range Slide Filter Widget

Interactive Range Slider Filter Widget” node defines the range of the numerical values to be plotted by a subsequent JavaScript node.

In the view it produces a slider item. Changing the range in this slider, changes the values displayed in the following chart or plot.


Dialog - 0:82 - Interactive Range Slider Filter Widget

File

Options | Slider | Labels | Flow Variables | Memory Policy

☑ Merge With Existing Filter Definitions (Table)

☑ Merge With Existing Filter Definitions (Model Port)

☐ Show Label ☐ Custom

Range Column: amount

Range Minimum: ☐ Custom 3

Range Maximum: ☐ Custom 1,600

Default Minimum Value: ☑ Use domain extent 3

Default Maximum Value: ☑ Use domain extent 1,600

OK Apply Cancel

Figure 5.31. The configuration window of the "Interactive Range Slider Filter Widget" node.

The configuration window requires:

- The numerical column to apply the range to.
- The minimum and maximum value for the full range. Default values are inherited from the column domain.
- The minimum and maximum value for the range defined by the slider handle. Again, default values are inherited from the column domain.
- If you are using a cascade of slider nodes, you can preserve the previous settings and build on top of them, by enabling the two merge checkboxes at the top. You can keep the previous range filters either in the result table or in the model or both.
- Optionally, an explanatory label can be added to clarify the range slider operation.

In our example workflow, named "Composite_Views", we created a component, named "Controlled Scatter Plot", to contain a "Scatter Plot" node and a "Interactive Range Slider Filter Widget" node. The scatter plot was set to display the sales input data in the ("amount", "quantity") space. The slider was hooked to column "amount" to filter the displayed data points in the plot. The first figure on the next page shows the view of the composite node, with all data points plotted in the scatter plot, the second figure shows the same component view, showing only those sales with "amount" lower than 482. This last range [3-482] has been manually defined by moving the slider handle in the component composite view.

The sub-workflow contained in component "Controlled Scatter Plot" is shown in Figure 5.34.

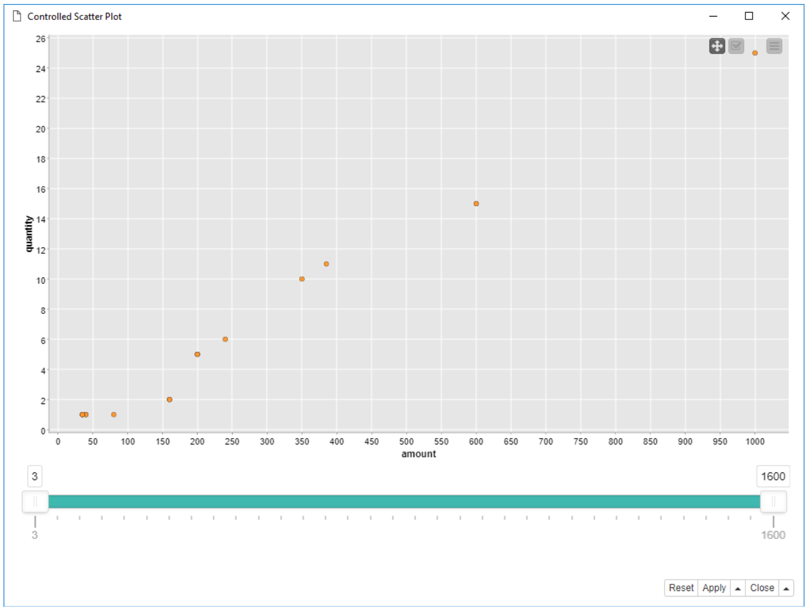


Figure 5.32. The view of the "Controlled Scatter Plot" component when the range for "amount" is left as the original column range.

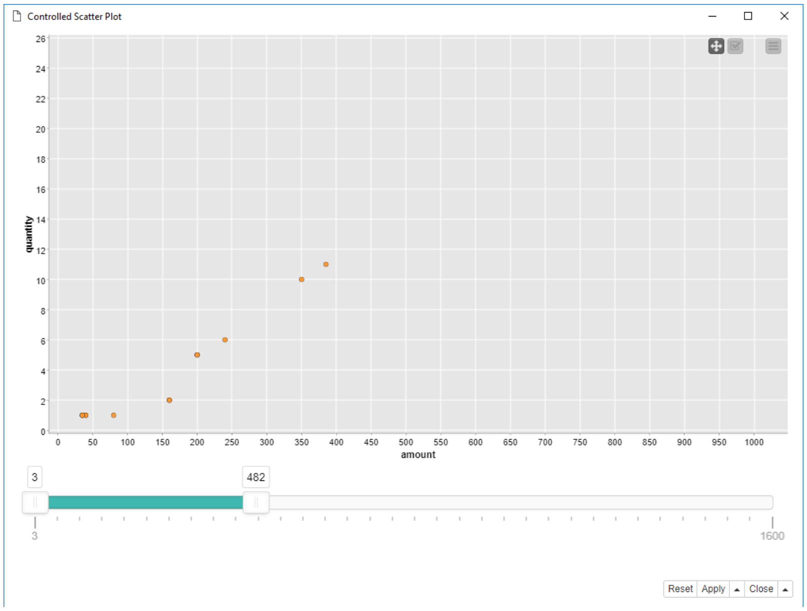


Figure 5.33. The view of the "Controlled Scatter Plot" component when the range for "amount" is set to [3, 482] by the slider handle.

We conclude here this chapter with a picture of the final workflow "Composite_Views", showing the component "Select Country from List" including "Local File Browser Configuration" and "Value Selection Widget" nodes; the component "Bar Charts side by side" with the two bar charts placed close to each other; the component "Controlled Scatter Plot" displaying a scatter plot controlled by a slider item.

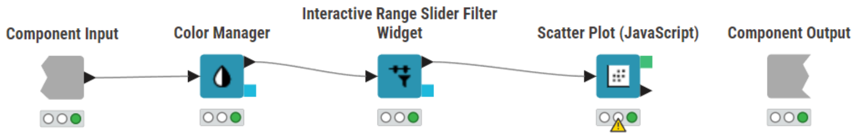
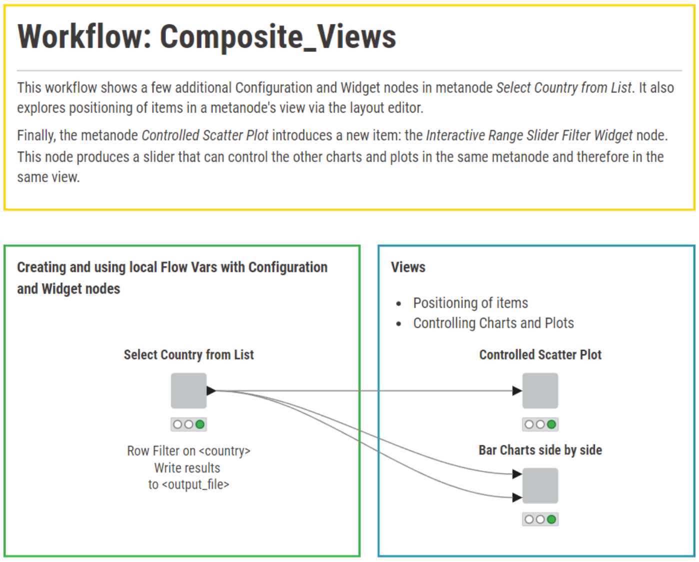

Component Input
Color Manager
Interactive Range Slider Filter Widget
Scatter Plot (JavaScript)
Component Output

Figure 5.34. The sub-workflow in component "Controlled Scatter Plot". Notice the "Interactive Range Slider Filter Widget" node feeding (and controlling) the data in the "Scatter Plot" node.


Workflow: Composite_Views

This workflow shows a few additional Configuration and Widget nodes in metanode *Select Country from List*. It also explores positioning of items in a metanode's view via the layout editor.

Finally, the metanode *Controlled Scatter Plot* introduces a new item: the *Interactive Range Slider Filter Widget* node. This node produces a slider that can control the other charts and plots in the same metanode and therefore in the same view.

Creating and using local Flow Vars with Configuration and Widget nodes

Views

- Positioning of items
- Controlling Charts and Plots

Figure 5.35. The final workflow "Composite_Views".

5.9. Components are for Sharing

If you have a particularly general component that encapsulates some logic or some data operations that can be useful to more people than just yourself, you can easily share it, with your future self, with colleagues, or with the whole KNIME Community.

Once you have a component, to make it into a sharable template, just right-click the component and in the context menu select "Component" → "Share". In the window that opens select the destination for the component template.

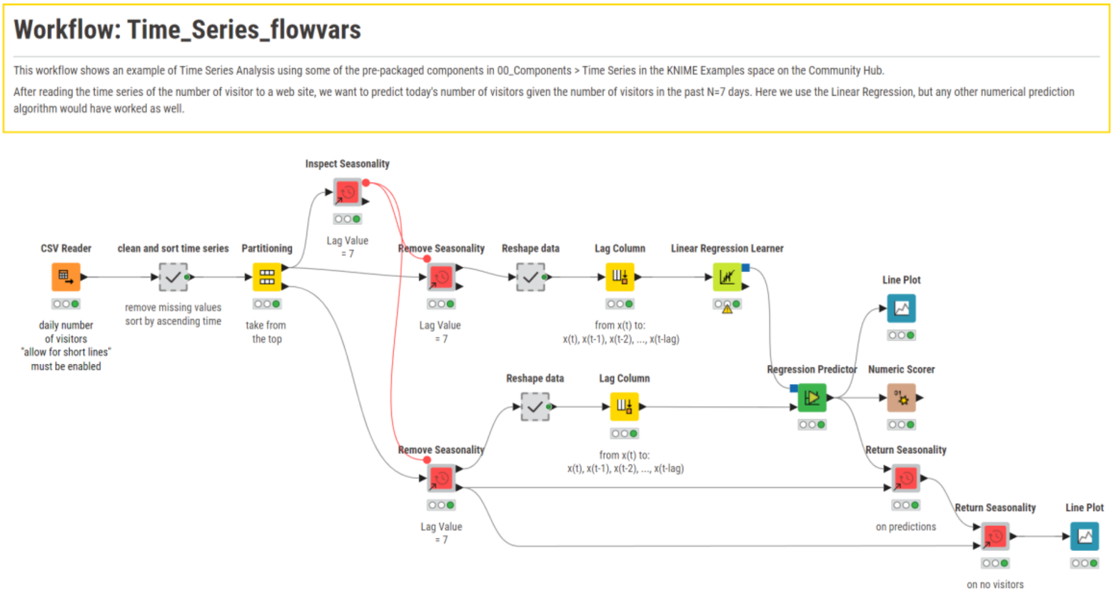
- In LOCAL workspace, the component template will only be visible to you and your future self;

- on a KNIME Business Hub it will be visible to you and your colleagues who have access to the same KNIME Business Hub;
- in the folder My-KNIME_Hub/Public in the KNIME Explorer it will be available for the whole KNIME community to use and accessible via the KNIME Community Hub.

To reuse a component template into your workflow, just drag&drop it from wherever it is into your workflow editor. A linked component will be automatically created, including the composite view and the composite configuration window of the original component. Every time you open the workflow, you will be prompted to check for updates in the component template and possibly import them into the linked instance.

Note. A linked component is created in read-only mode. You cannot change its content until you disconnect from the template and make your own local copy using the option "Component" -> "Disconnect Link" in the context (right-click) menu of the component.

We have used component templates in chapter 4, when using some of the time series components contained in EXAMPLES/00_Components/Time Series. There we made just use of the acquired configuration window of the components to set the right parameters for the task, but we made no use of flow variables to make the parameter passing from component to component easier. Indeed, the lag value, calculated in component "Inspect Seasonality", is presented at the top output port of the component, and could be easily passed to the next


Workflow: Time_Series_flowvars

This workflow shows an example of Time Series Analysis using some of the pre-packaged components in 00_Components > Time Series in the KNIME Examples space on the Community Hub.

After reading the time series of the number of visitor to a web site, we want to predict today's number of visitors given the number of visitors in the past N=7 days. Here we use the Linear Regression, but any other numerical prediction algorithm would have worked as well.

Figure 5.36. Workflow "Time_Series_flowvars". This is a revised version of workflow "Time_Series_noflowvars" developed in chapter 4 using flow variable connections to pass the lag value from the "Inspect Seasonality" component into the "Remove Seasonality" component.

component "Remove Seasonality" with a flow variable connection, avoiding the manual setting. The final workflow, taking advantage of flow variables, is shown in the figure above.

5.10. Exercises

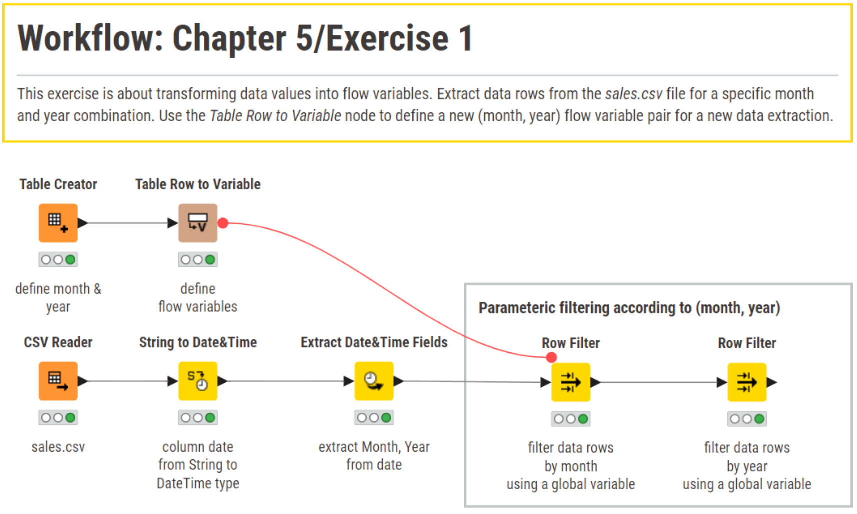
Exercise 1

Read the file *sales.csv* from the KALdata folder.

Select sale records for a specific month and year, like for example March 2009, March 2010, January 2009, January 2010 using flow variables for the month and the year values.

Solution to Exercise 1

To implement an arbitrary selection of data records based on month and year, we first applied an *Extract Date&Time Fields* node and defined two flow variables, one for the year and one for the month, using the *Table Creator* node followed by a *Table Row to Variable* node. 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.


Workflow: Chapter 5/Exercise 1

This exercise is about transforming data values into flow variables. Extract data rows from the *sales.csv* file for a specific month and year combination. Use the *Table Row to Variable* node to define a new (month, year) flow variable pair for a new data extraction.

Table Creator
define month & year
Table Row to Variable
define flow variables
CSV Reader
sales.csv
String to Date&Time
column date from String to DateTime type
Extract Date&Time Fields
extract Month, Year from date
Parameteric filtering according to (month, year)
Row Filter
filter data rows by month using a global variable
Row Filter
filter data rows by year using a global variable

Figure 5.37. Exercise 1: The workflow.

Exercise 2

Using the file *cars-85.csv* and build a data set with:

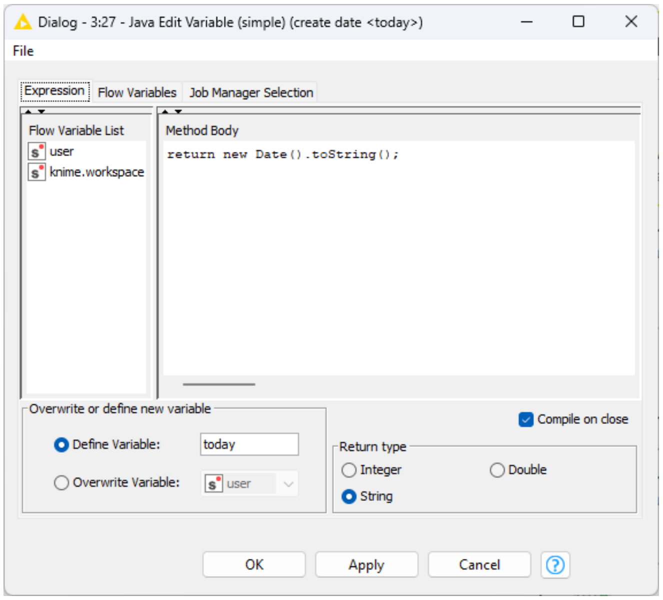
- The cars from the car manufacturer that is most present in the original data
- The cars from the car manufacturer that is least present in the original data

Put today's date on every row of the new data set to identify the 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 then sorted the aggregated rows by the count value in descending/ascending order and kept only the first row and only the "make" column. To create the flow variable, we used the *Table Row to Variable* node, creating a flow variable called "make". Finally, we used a *Row Filter* node to keep only the rows with that value in the "make" column.

To create today's date, we used a *Java Edit Variable (simple)* node and appended it to the data set in a column called "load_date" by means of a *Rule Engine* node.

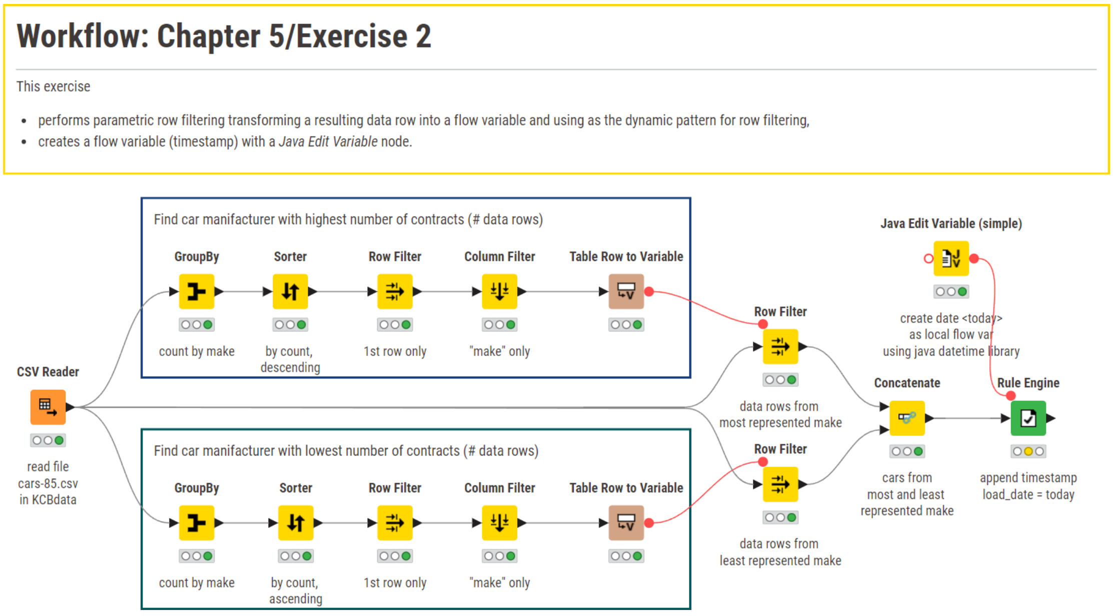


```
return new Date().toString();
```

Define Variable: today
Overwrite Variable: user
Compile on close: ✓
Return type: Integer [✓ String] Double

Figure 5.38. Exercise 2: Configuration window of the "Java Edit Variable (simple)" node.

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 Edit Variable (simple)* node rather than many *Java Snippet* nodes.



Workflow: Chapter 5/Exercise 2

This exercise

- performs parametric row filtering transforming a resulting data row into a flow variable and using as the dynamic pattern for row filtering,
- creates a flow variable (timestamp) with a *Java Edit Variable* node.

Find car manifacturer with highest number of contracts (# data rows)
GroupBy
count by make
Sorter
by count, descending
Row Filter
1st row only
Column Filter
"make" only
Table Row to Variable
CSV Reader
read file cars-85.csv in KCBdata
Find car manifacturer with lowest number of contracts (# data rows)
GroupBy
count by make
Sorter
by count, ascending
Row Filter
1st row only
Column Filter
"make" only
Table Row to Variable
Row Filter
data rows from most represented make
Row Filter
data rows from least represented make
Java Edit Variable (simple)
create date <today> as local flow var using java datetime library
Concatenate
cars from most and least represented make
Rule Engine
append timestamp load_date = today

Figure 5.39. Exercise 2: The workflow.

Exercise 3

Define a maximum engine size, for example 300, and put it into a flow variable named “max”.

Read the file *cars-85.csv* and build two data sets respectively with:

- All cars with “engine_size” between the maximum engine size defined (“max”) and its half
- All cars with “engine_size” between the half and a quarter of the original default value

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

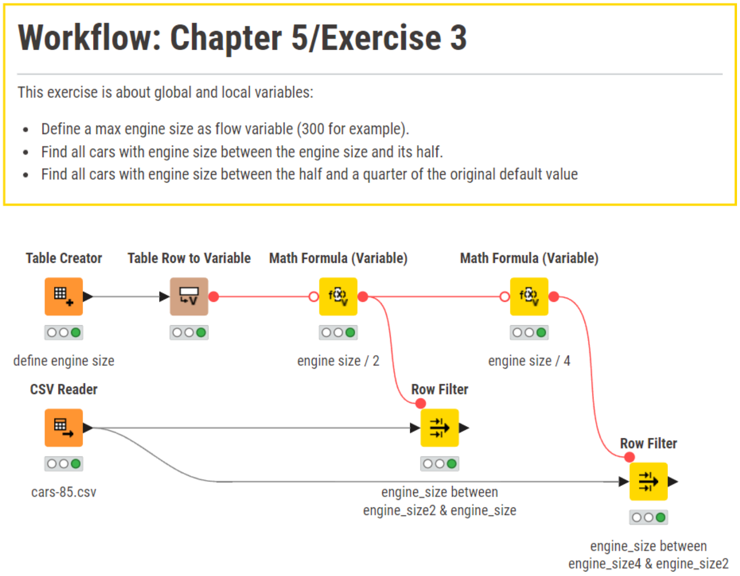
Solution to Exercise 3

Define “max” as an integer value using the *Table Creator* node and convert it into a flow variable using the *Table Row to Variable* node.

Create the flow variables “max2” with the value $\frac{max}{2}$ and “max4” with value $\frac{max}{4}$ with two *Math Formula (Variable)* nodes.

Configure two *Row Filter* nodes with “use range checking” as filter criterion and with the appropriate flow variables in lower bound and upper bound to build the required data tables.

Note. Remember to check the "Convert to Int" flag in the *Math Formula (Variable)* nodes. The column "engine_size" is of type integer and then the *Row Filter* node sees only flow variables of type integer.

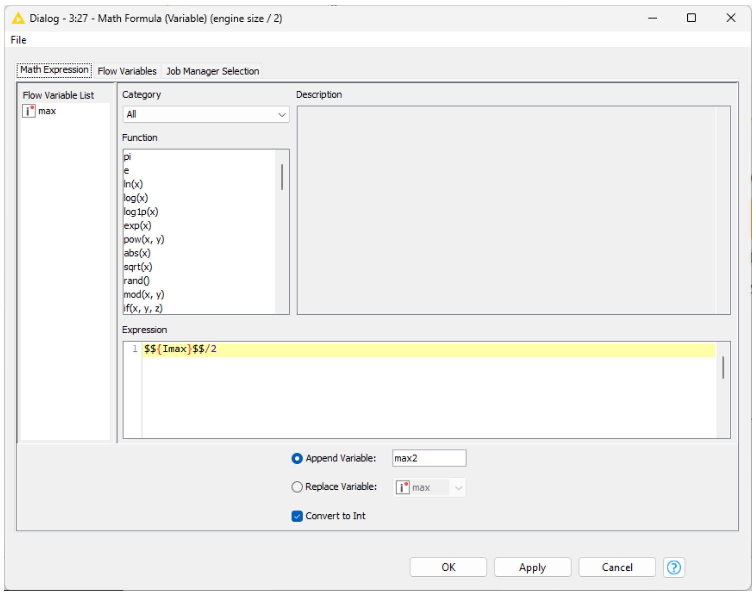


Workflow: Chapter 5/Exercise 3

This exercise is about global and local variables:

- Define a max engine size as flow variable (300 for example).
- Find all cars with engine size between the engine size and its half.
- Find all cars with engine size between the half and a quarter of the original default value

Figure 5.40. Exercise 3: The workflow.



```
$${Imax}$$/2
```

Append Variable: max2 [✓]

Replace Variable: max

Convert to Int [✓]

Figure 5.41. Exercise 3: Configuration window of the Math Formula (Variable) node.

Exercise 4

Using the *cars-85.csv* file, build a component with the following configuration window to:

- select data rows with a specific body style
- replace missing values in a column 1 with values in the same data row in a column 2.

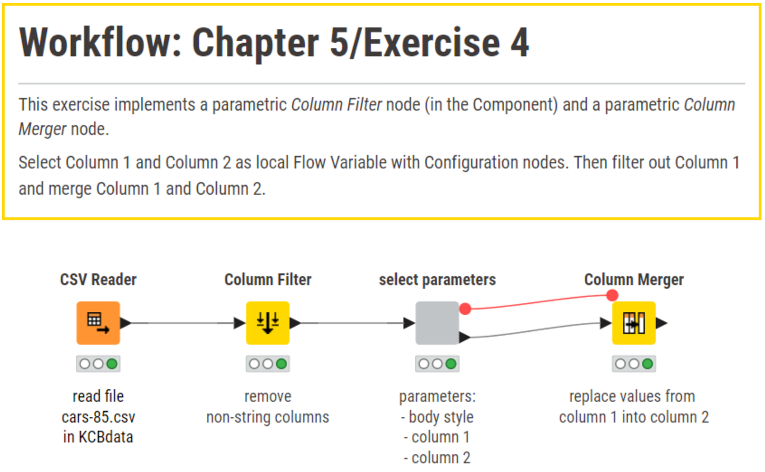
Limit the choice of columns to only String columns. For the missing value operation use the *Column Merger* node.

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 parameters for the configuration window of the component with Configuration nodes:

- A *Value Selection Configuration* node applied to the "body_style" column
- A *Column Selection Configuration* node to select column 1 to feed to the *Column Merger* node
- A second *Column Selection Configuration* node to select the column 2 to feed to the *Column Merger* node

Then, we configure a *Column Merger* node using the variables created for column 1 and column 2.

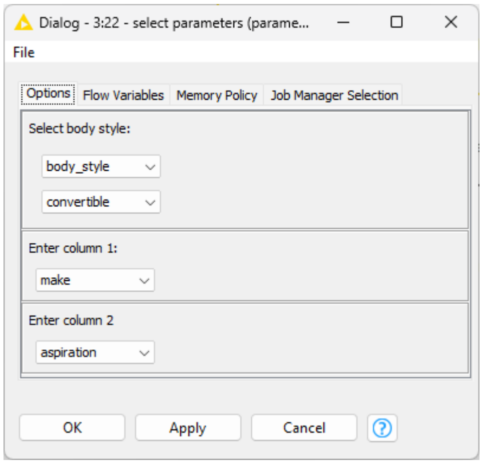


Workflow: Chapter 5/Exercise 4

This exercise implements a parametric *Column Filter* node (in the Component) and a parametric *Column Merger* node.

Select Column 1 and Column 2 as local Flow Variable with Configuration nodes. Then filter out Column 1 and merge Column 1 and Column 2.

Figure 5.42. Exercise 4: The workflow.



Dialog - 3:22 - select parameters (parame...

File

Options | Flow Variables | Memory Policy | Job Manager Selection

Select body style:

body_style

convertible

Enter column 1:

make

Enter column 2

aspiration

OK | Apply | Cancel

Figure 5.43. Exercise 4: The component configuration window.

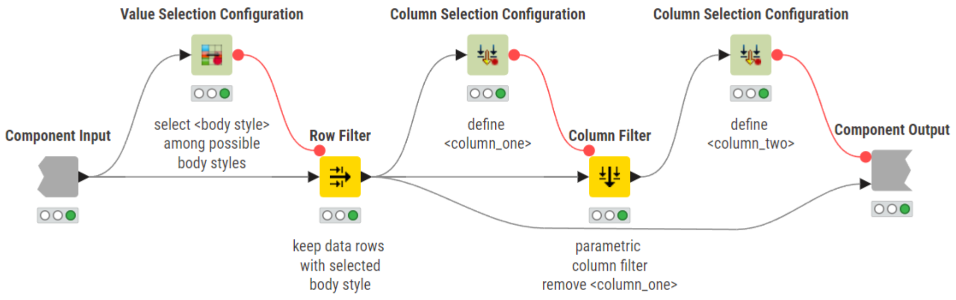
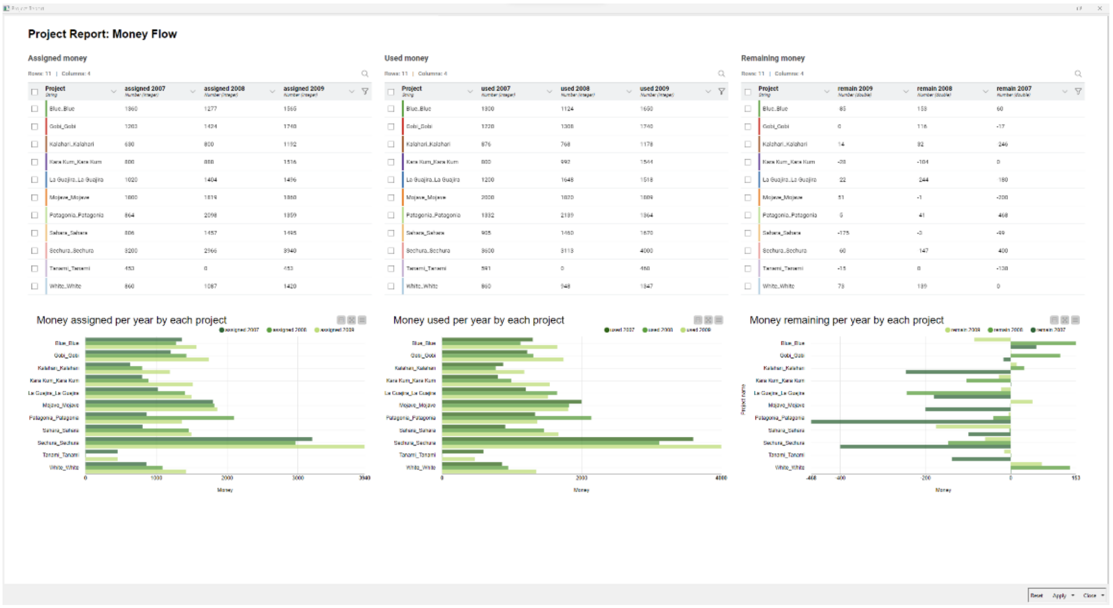

Value Selection Configuration
Column Selection Configuration
Column Selection Configuration
Component Input
select <body style> among possible body styles
Row Filter
define <column_one>
Column Filter
define <column_two>
Component Output
keep data rows with selected body style
parametric column filter remove <column_one>

Figure 5.44. Exercise 4: The sub-workflow in the component.

Chapter 6: Advanced Dashboards with Composite Views

6.1. A few examples of Advanced Dashboards

In KNIME Beginner's Luck (Chapter 6: Dashboards with Composite View) we built a dashboard including a table for assigned money, a table for used money, and a table for remaining money and the corresponding bar charts across all projects/years (see figure below).



Project Report: Money Flow

Assigned money

Rows: 11 | Columns: 4

Project (String)	assigned 2007 (Number (integer))	assigned 2008 (Number (integer))	assigned 2009 (Number (integer))
Blue_Blue	1360	1277	1566
Gobi_Gobi	1203	1424	1740
Kalahari_Kalahari	630	800	1192
Kara Kum_Kara Kum	800	888	1516
La Guajira_La Guajira	1020	1404	1496
Mojave_Mojave	1800	1819	1860
Patagonia_Patagonia	864	2098	1359
Sahara_Sahara	806	1457	1495
Sechura_Sechura	3200	2966	3940
Tanami_Tanami	453	0	453
White_White	860	1087	1420

Used money

Rows: 11 | Columns: 4

Project (String)	used 2007 (Number (integer))	used 2008 (Number (integer))	used 2009 (Number (integer))
Blue_Blue	1300	1124	1650
Gobi_Gobi	1220	1308	1740
Kalahari_Kalahari	876	768	1178
Kara Kum_Kara Kum	800	992	1544
La Guajira_La Guajira	1200	1648	1518
Mojave_Mojave	2000	1820	1809
Patagonia_Patagonia	1332	2139	1364
Sahara_Sahara	905	1460	1670
Sechura_Sechura	3600	3113	4000
Tanami_Tanami	591	0	468
White_White	860	948	1347

Remaining money

Rows: 11 | Columns: 4

Project (String)	remain 2009 (Number (double))	remain 2008 (Number (double))	remain 2007 (Number (double))
Blue_Blue	-85	153	60
Gobi_Gobi	0	116	-17
Kalahari_Kalahari	14	32	-246
Kara Kum_Kara Kum	-28	-104	0
La Guajira_La Guajira	-22	-244	-180
Mojave_Mojave	51	-1	-200
Patagonia_Patagonia	-5	-41	-468
Sahara_Sahara	-175	-3	-99
Sechura_Sechura	-60	-147	-400
Tanami_Tanami	-15	0	-138
White_White	73	139	0

Figure 6.1. The bar charts from the original simple dashboard in KNIME Beginner's Luck (Ch. 6).

This dashboard was good enough for at-a-glance observations and simple enough to build. Years and projects were statistically represented in all three bar charts in shade of green.

However, with your newly acquired knowledge of flow variables from this book, in this chapter we will change this static dashboard into a dynamic one or two. The goal of this chapter is to show how to introduce advanced dashboard elements in the composite view to

- interactively select one or more attributes to draw in the charts
- make dynamic, real-time updates

- autocomplete text queries
- introduce custom option selection
- implement an animated bar chart via a community component

In particular, we will build two interactive dashboards and an animated bar chart.

After completing this chapter, you will have learned how to create 2 interactive dashboards and an animated bar chart.

The first bar chart, named "Project Report: Money Flow" (figure below), introduces the concept of interactive selection, by allowing us to select the attributes (money used, assigned, and remaining) and the specific project for the bar chart. The workflow implementing this dashboard is "Project Report 1: Filtering by Year and Project" found in the "Advanced Dashboards" KNIME file.

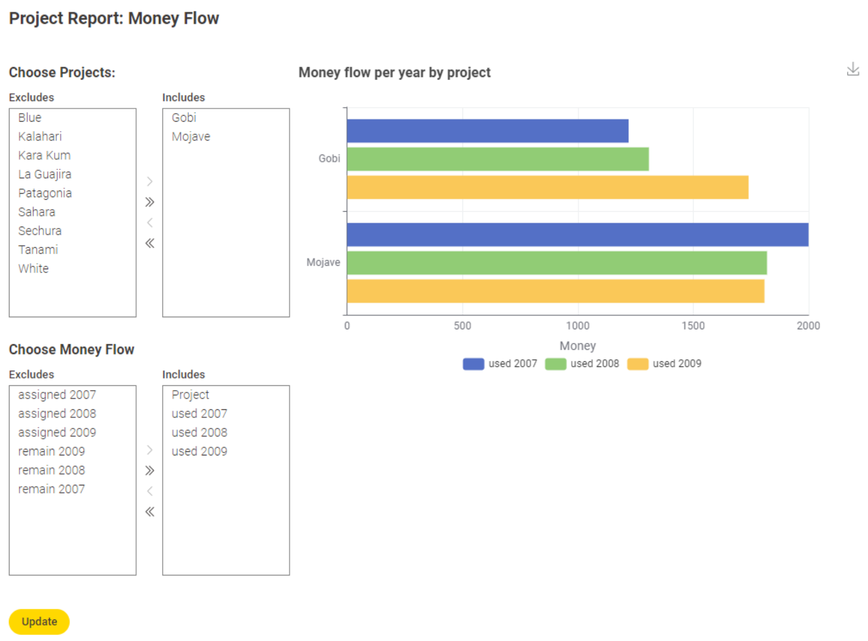


Figure 6.2. The first interactive dashboard of this chapter "Project Report: Money Flow".#

The second dashboard, named "Advanced Money Flow with Text Autocompletion" (figure below), builds upon the first dashboard by introducing real time update, text autocomplete, and custom option selection. The workflow implementing this dashboard is called "Advanced Dashboards".

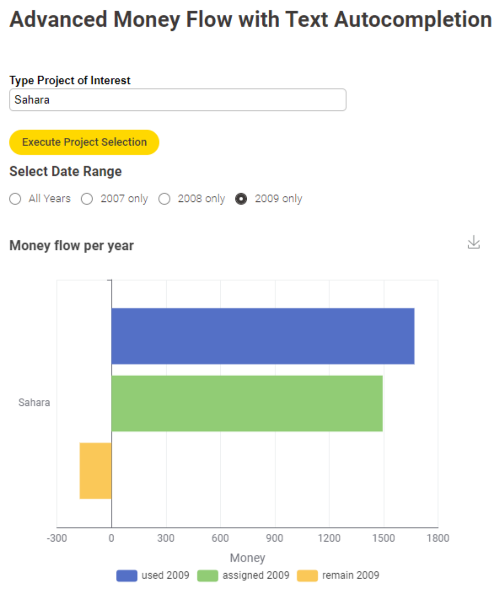


Figure 6.3. The second interactive dashboard of this chapter "Advanced Money Flow with Text Autocompletion" found in the "Advanced Dashboards" KNIME file.

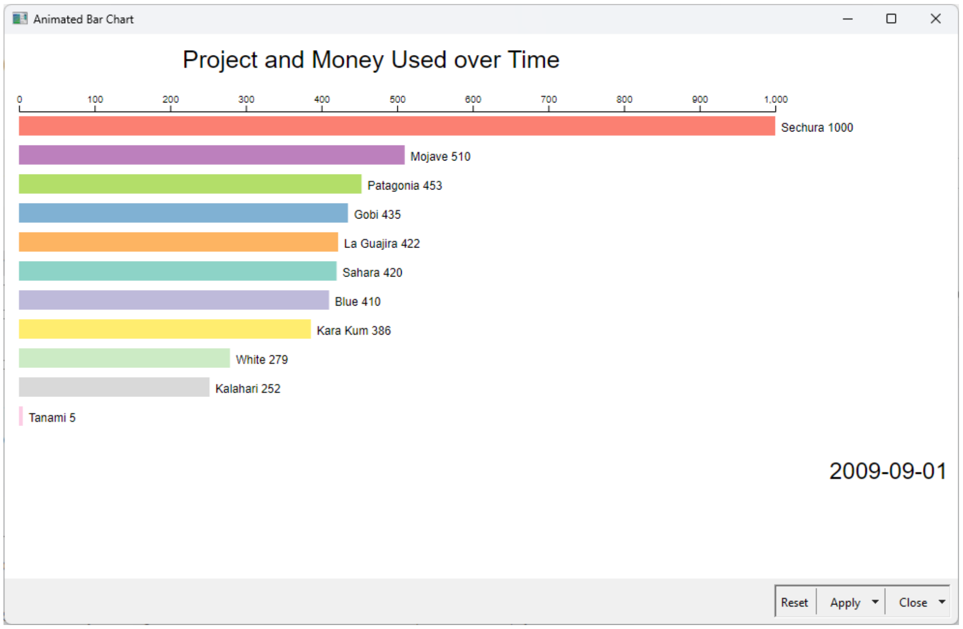


Figure 6.4. The animated bar chart component creates a visualization where bars race over time, increasing in range. This figure only captures a static image of the final chart.

Finally, the last part of this chapter shows how to build animated bar charts with a component template available on the KNIME Community Hub. The animated bar chart adds a splash of movement meant to capture an audience's attention. The workflow implementing this dashboard is called "Animated Bar Chart".

So which nodes allow us to create such interactive dashboards? Widget nodes. All Widget nodes are available under Workflow Abstraction in the Node Repository panel. Widget nodes implement an interactive item in the composite view of a component or in the web page.

You have previously used the Value Selection Widget, the Text Output Widget and the Interactive Range Slider Filter Widget in either KNIME Beginner's Luck or in previous chapters of this text.

In the following sections we will introduce you to more widgets which allow for customization and dynamic interactivity in composite views - especially for advanced dashboards.

In each section we will introduce key widgets and gradually build the objective interactive dashboards as well as several simpler dashboards. Here are the widgets we will cover in each section:

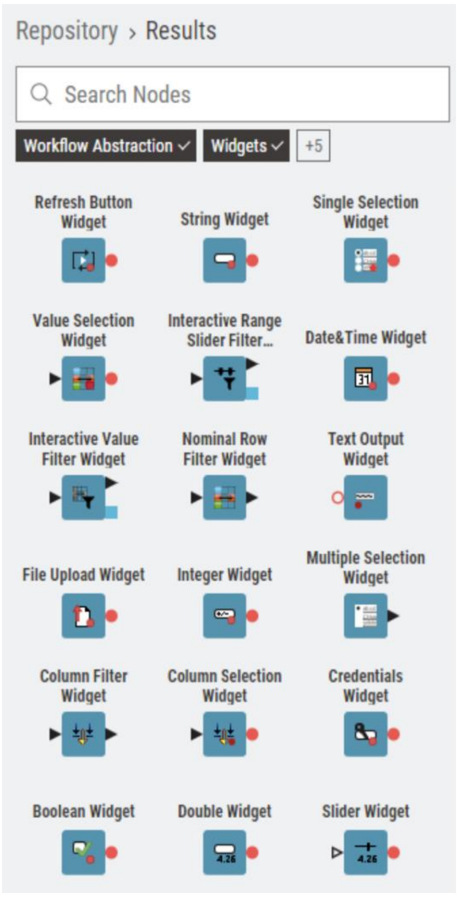


Figure 6.5. A list of Widget nodes found within Workflow Abstraction in the Node Repository.

- 6.2 - Column Filter Widget
- 6.3 - Refresh Button Widget
- 6.4 - Nominal Row Filter Widget - at this point we can make dashboard 1
- 6.5 - Autocomplete Text Widget
- 6.6 - Single Selection Widget - at this point we can make dashboard 2

In section 6.7 we will introduce how to animate bar graphs by using a community component. In section 6.8 we will touch the topic of Geospatial Analysis and introduce selected nodes of the Geospatial Analytics Extension for KNIME, and finally, section 6.9 will propose two exercises, one dealing with color palettes and the other with stacked area charts, to aid you in building your own dashboards.

6.2. Interactively selecting one or more attributes to represent in a chart

Let’s start from the composite view implemented in KNIME Beginner’s Luck (Figure 6.1). This composite view was static, i.e., it was not possible to select the input columns to be displayed. This made for a view with many elements and the inability to have the audience pay attention to any one particular aspect of the graphics. In this section we want to add one control item to allow for custom selection of the input attributes.

As shown in Figure 6.1, in the original dashboard the three bar charts were always shown. But, by adding the ability to control which columns to represent, we can create simpler, easier-to-parse graphics. We will do this by selecting our columns interactively using the Column Filter Widget node.

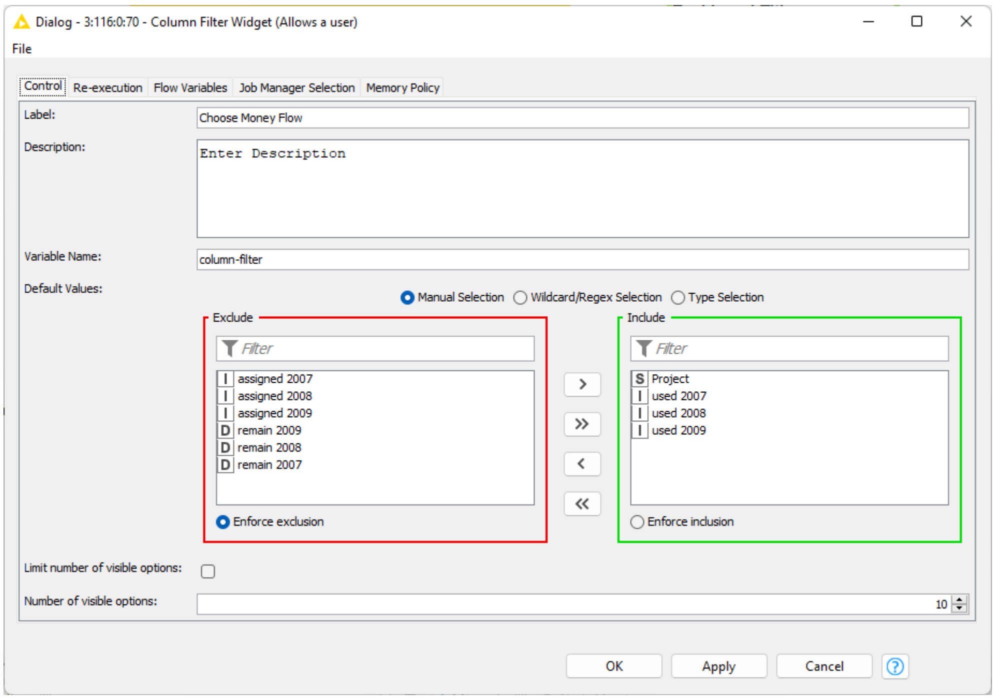
Column Filter Widget

The “Column Filter Widget” node creates an interactive column filter selection menu for use in component views. It takes a data table and returns a filtered data table with only the selected columns.

The configuration window of the node does not require any special settings to be configured/selected, besides the default filtering to show when opening the interactive view for the first time.

Check the “Re-execution on widget value change” box in the Re-execution tab, if you would like re-execution to happen automatically upon choice selection.

The Column Filter Widget node then produces a framework to allow us to interactively choose the columns we want displayed. For example, let’s build a simple component connecting a Column Filter Widget node, with the configuration window shown in the figure below, to a Bar Chart node.


Dialog - 3:116:0:70 - Column Filter Widget (Allows a user)

File

Control | Re-execution | Flow Variables | Job Manager Selection | Memory Policy

Label: Choose Money Flow

Description: Enter Description

Variable Name: column-filter

Default Values: [✓ Manual Selection] Wildcard/Regex Selection Type Selection

Exclude:
- I assigned 2007
- I assigned 2008
- I assigned 2009
- D remain 2009
- D remain 2008
- D remain 2007

[✓ Enforce exclusion]

Include:
- S Project
- I used 2007
- I used 2008
- I used 2009

Enforce inclusion

Limit number of visible options: ☐

Number of visible options: 10

OK Apply Cancel

Figure 6.6. The configuration window of the Column Filter Widget node.

If we were to right-click on this component and inspect its interactive view, we would get the output depicted in Figure 6.8. Notice the Excludes/Includes framework at the top generated by the Column Filter Widget node. Notice also that the bar chart, implemented by the Bar Chart node, has operated only on the selected input attributes (assigned 2007, assigned 2008, assigned 2009).

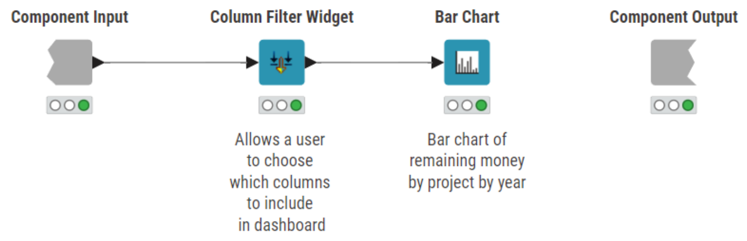

Component Input
Column Filter Widget
Bar Chart
Component Output
Allows a user to choose which columns to include in dashboard
Bar chart of remaining money by project by year

Figure 6.7. Simple component with a Column Filter Widget node and a Bar Chart node producing the dashboard shown in the next figure.

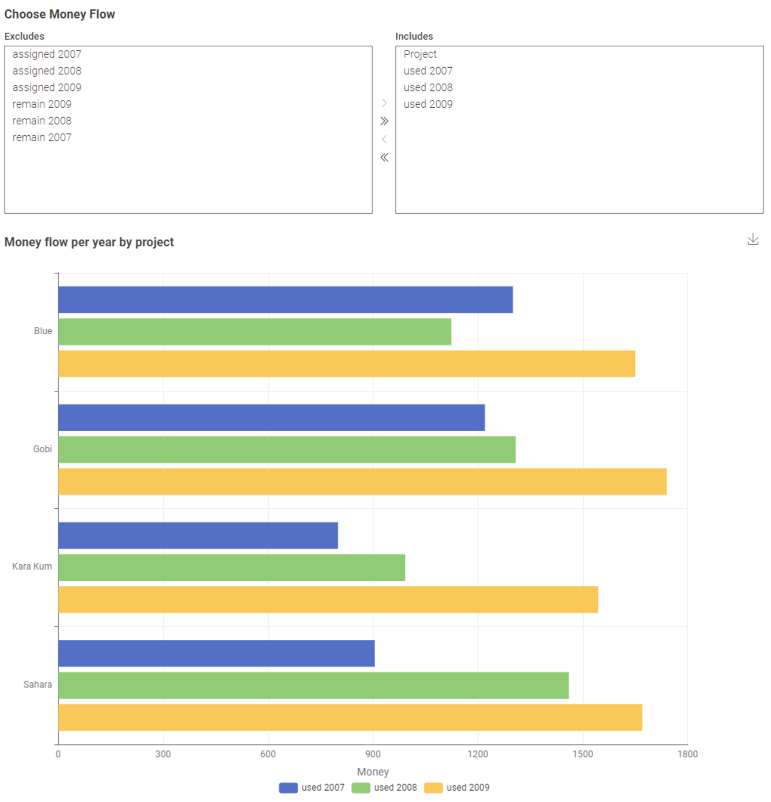


Figure 6.8. Output of the simple component.

But there is one problem. At the moment, to apply a new selection of columns to the bar chart in this dashboard, the composite view must be manually re-executed. That is, this dashboard is not yet dynamically malleable. In order to avoid the manual re-execution of the composite view, we'll need to learn about a feature within some widget nodes called "Re-execution", and about the Refresh Button Widget node. This is the subject of the next section.

6.3. Dynamically update the dashboard

In this section, we will learn how to dynamically update a dashboard upon the selected columns. That is, we will select the column(s) to use - money assigned, used, remaining and year - and dynamically update the dashboard. To do this we will learn about a concept called re-execution.

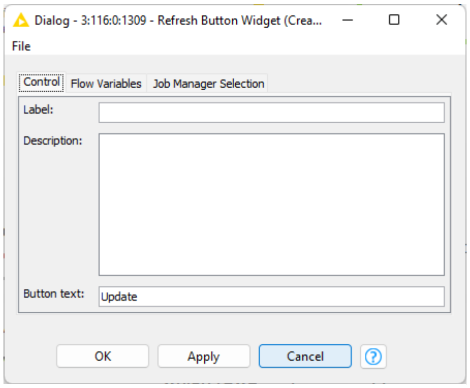
Re-execution allows us to dynamically apply any selection we have made within a composite view. Some widgets have a "Re-execution" tab while others do not. In addition, the Refresh

Button Widget node gives the re-execution capability to any widget node, therefore let's discuss the Refresh Button Widget node first and then later we will talk about the dedicated tab.

Refresh Button Widget

Button Widget allows us to (re-)execute downstream nodes in a component. It creates a yellow button in the composite view of the component. Simply clicking the yellow button triggers the re-execution of the downstream nodes in the component.

The dialog options of the Refresh Button Widget are straightforward and leaving most fields blank is acceptable. The only important setting is the name of the yellow button.


Dialog - 3:116:0:1309 - Refresh Button Widget (Crea...

File

Control | Flow Variables | Job Manager Selection

Label:

Description:

Button text: Update

Figure 6.9. The dialog options of the Refresh Button Widget.

The Refresh Button Widget node must be connected via a flow variable connection to the widget node whose options you would like to refresh (re-execute). You can decide which workflow segments should be (re-)executed by where you place the Refresh Button Widget node in the workflow.

Let's consider the component shown in Figure 6.6, including a Column Filter Widget to select the columns to use in the following Bar Chart node. After selecting new columns to display in the bar chart, the execution of the Column Filter Widget node and of the Bar Chart node must be retriggered. Thus, the Refresh Button Widget node must be connected via flow variable to the Column Filter Widget node. The new component is shown in the figure below.

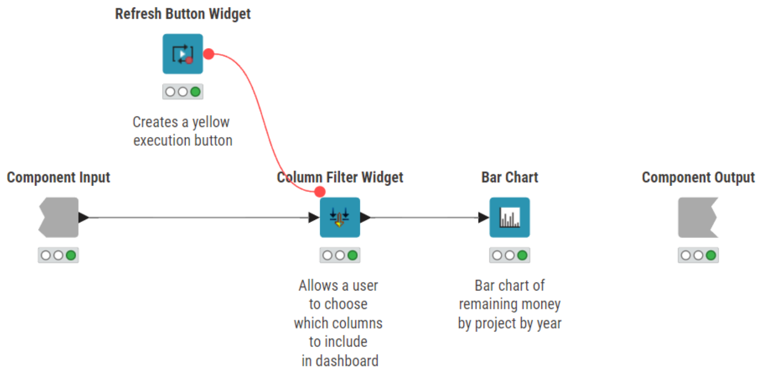

Refresh Button Widget
Creates a yellow execution button
Component Input
Column Filter Widget
Allows a user to choose which columns to include in dashboard
Bar Chart
Bar chart of remaining money by project by year
Component Output

Figure 6.10. Adding the Refresh Button Widget node to control re-execution of the component shown above.

Note. Unlike most flow variable connections, you do not set any flow variables in the dialog of the flow-variable-receiving node. In this case, the Refresh Button Widget is sending a flow variable connection to the Column Filter Widget and the Column Filter Widget node is receiving the flow variable connection. You do not need to specify anything in the Flow Variables tab in the configuration dialog of the Column Filter Widget node.

So, in the composite view of the component, once we change the selection of the columns to display, we click the yellow button, named in this case "Update", and the selection frame as well as the bar chart get updated.

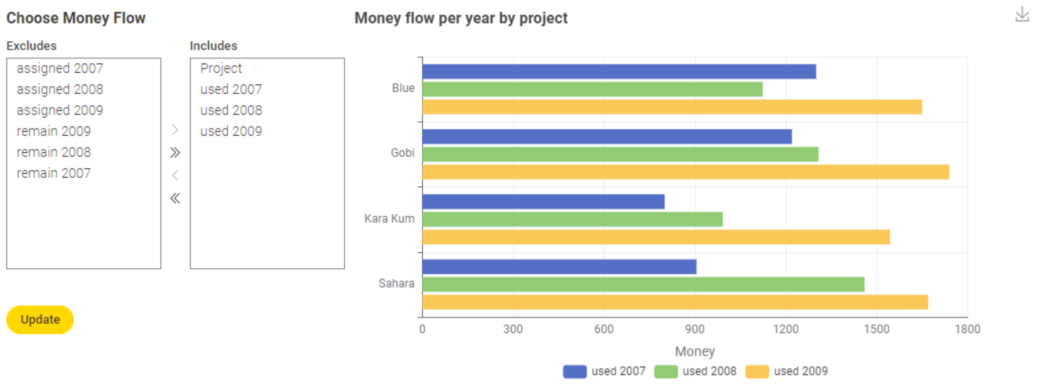
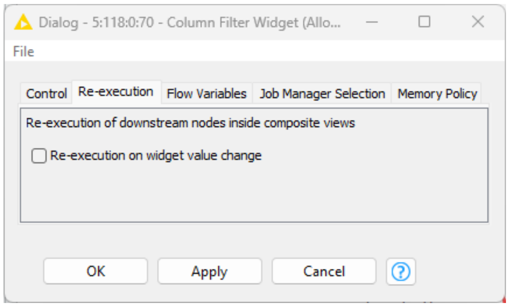


Figure 6.11. Selecting columns in the Include/Exclude framework and clicking the yellow button produces an update of the dashboard based on the new column selection.

But what if you don't want to click that yellow button every time you make a new choice? In KNIME Analytics Platform v4.5 and above, there is an option to automatically reload the interactive view whenever a setting of the node is changed.

To enable this option, simply right-click on the widget node, select "Configure…", and look for the "Re-execution" tab in the configuration window. Select that tab and then check the box next to "Re-execution on widget value change".



Re-execution of downstream nodes inside composite views

☐ Re-execution on widget value change

Figure 6.12. The automatic re-execution functionality option in a Widget node.

Note. A small symbol appears on the upper right corner of a widget node that has the re-execution option enabled.

In general, it is up to the user to decide what to use, whether the re-execution tab or the Refresh Button Widget node. Most users will use the dedicated Refresh Button Widget node if they have a very large dataset and re-executing each time a selection is made takes too long or if the widget they are using does not have the Re-execution tab.

Now that we can select columns and dynamically refresh all connected nodes and composite views, either automatically or by clicking the yellow refresh button, we will turn our attention to interactively selecting rows in a dataset.

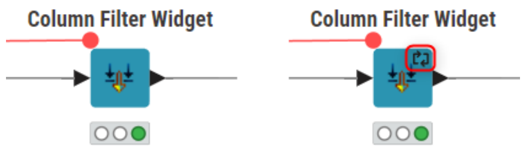


Figure 6.13. The widget icon without (left) and with (right) the refresh symbol in the upper right corner indicating that re-execution has been enabled.

6.4. Interactively selecting rows by column values

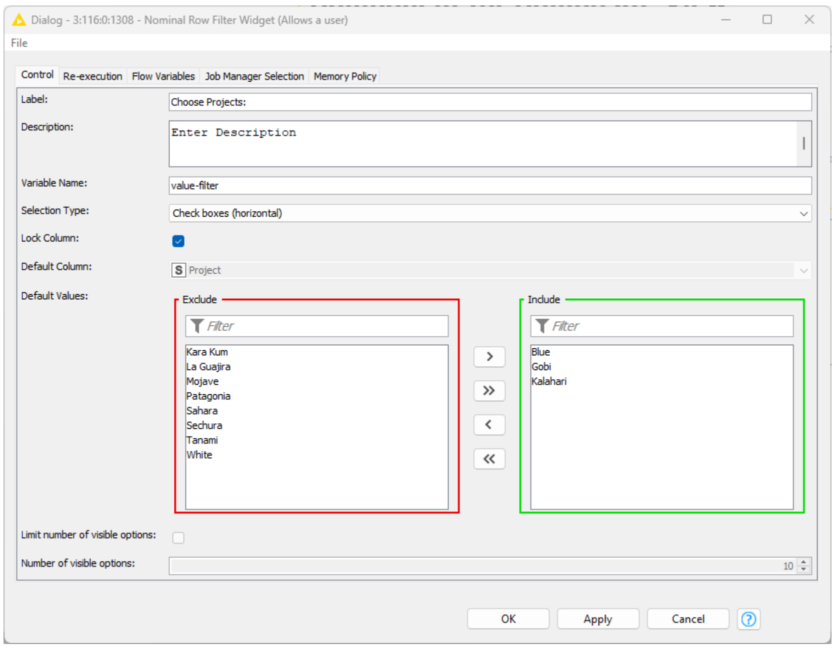
In the last section we learned how to filter (add and remove) columns of interest inside the dashboard. But what if we would like to filter rows from values within a column? That is, we don't want to remove a column entirely as we did before, but rather hone in on certain elements within the column. In this section, we will select specific projects, by selecting their names in the Projects column. If we only wanted to focus our attention on Blue and/or some other projects, we would need to filter the rows to only include the data for those projects.

Now that we have seen how to filter columns of interest dynamically, let's learn how to filter rows of interest dynamically. As well, if those rows are nominal (a.k.a. categorical), we can easily filter those data points using the Nominal Row Filter Widget node. The Nominal Row Filter Widget node receives a data table at the input port and produces the filtered rows at the output port. The filtering rule is defined via and Exclude/Include framework in the node's interactive view.

Nominal Row Filter Widget

The "Nominal Row Filter Widget" node creates an Exclude/Include framework for categorical row filter selection. The configuration window of the node does not require any special settings to be configured/selected, besides the default filtering to show as starting point in the interactive view. Check the "Re-execution on widget value change" box in the Re-execution tab, if you would like to trigger re-execution automatically upon any setting change. "Selection Type" allows the framework for the selection of rows to vary from twinlists, to boxes, and even lists.

In our example, we have set the Selection Type to “Check boxes (horizontal)”, selected projects Blue, Gobi, and Kalahari, and enabled re-execution. The dashboard then updated automatically to show the data of the three selected projects.


Dialog - 3:116:0:1308 - Nominal Row Filter Widget (Allows a user)

File

Control | Re-execution | Flow Variables | Job Manager Selection | Memory Policy

Label: Choose Projects:

Description: Enter Description

Variable Name: value-filter

Selection Type: Check boxes (horizontal)

Lock Column: ✓

Default Column: S Project

Default Values:

Exclude	Include
Kara Kum	Blue
La Guajira	Gobi
Mojave	Kalahari
Patagonia	
Sahara	
Sechura	
Tanami	
White	

Limit number of visible options:

Number of visible options: 10

OK | Apply | Cancel

Figure 6.14. The configuration window of the Nominal Row Filter Widget node.

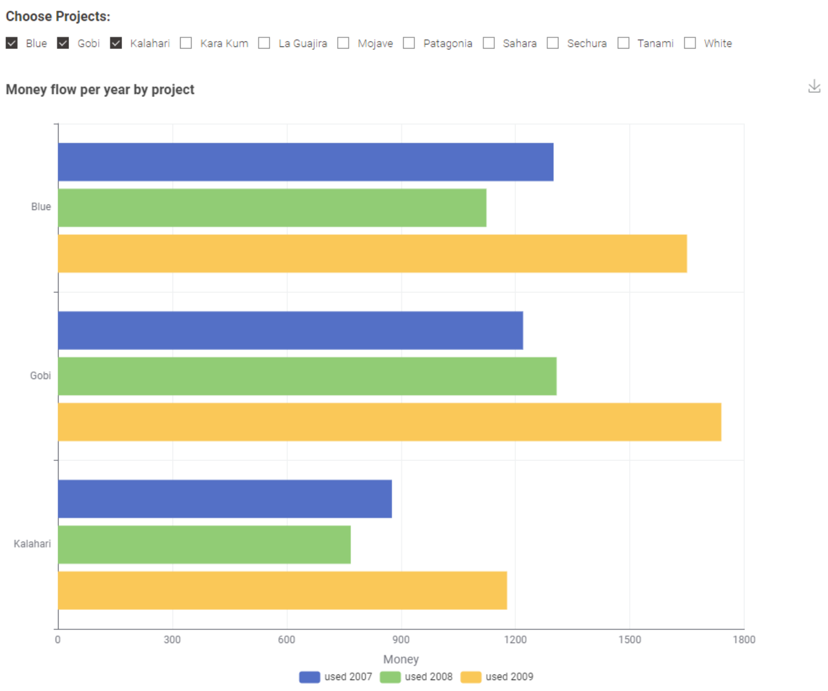


Figure 6.15. Dashboard showing used money amounts for selected projects.

Unlike the Column Filter Widget which only uses the “Selection Type” called Twinlist, above we used the Selection Type called “Check boxes (horizontal)” to give some variety. Below we give a direct comparison between these two selection types.

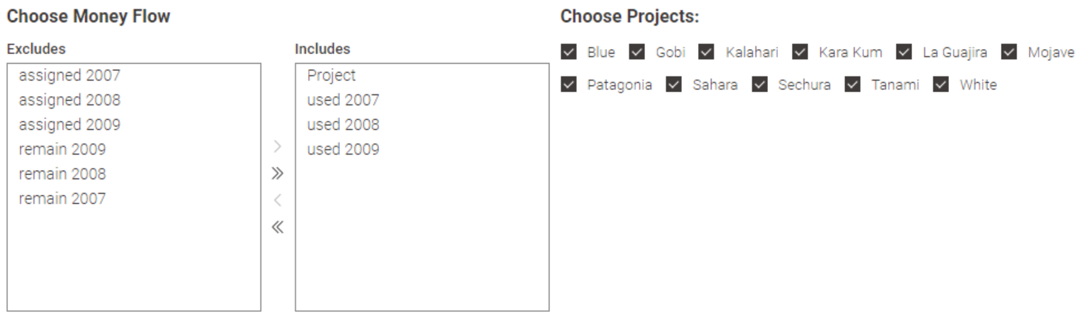


Figure 6.16. Ways to visualize column and row filter widgets - the left is called “Twinlist”, and the right is called “Check boxes (horizontal)”.

Now that we have learned about the Column Filter Widget, the Refresh Button Widget, the Re-execution tab, and the Nominal Row Filter Widget, we are ready to make the component “Project Report: Money Flow” implementing the first complete dashboard, as it was described in section 6.1.

Summarizing, the component contains:

- A Column Filter Widget node to select the data (money flow and year)
- A Nominal Row Filter Widget node to select one or more projects
- A Bar Chart node to display the selected data
- A Text Output Widget to provide a title for the dashboard
- And finally a Refresh Button Widget node to trigger re-execution at any change in the dashboard settings.
- As a consequence, the composite view of this component shows a dashboard with two filtering elements, one for the columns and one for the project/rows, and a bar chart.

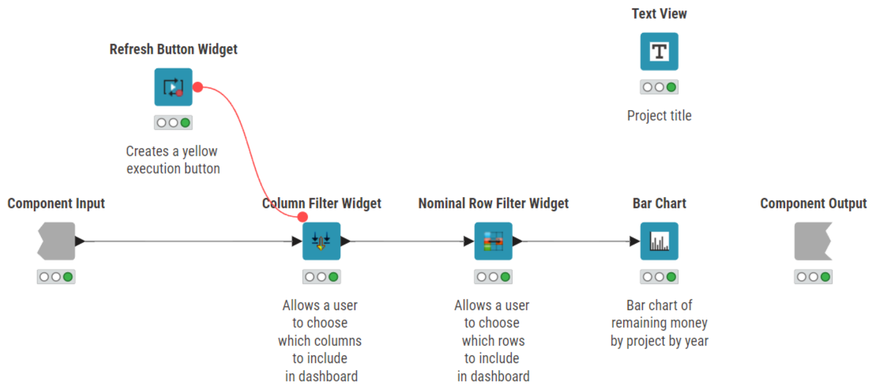
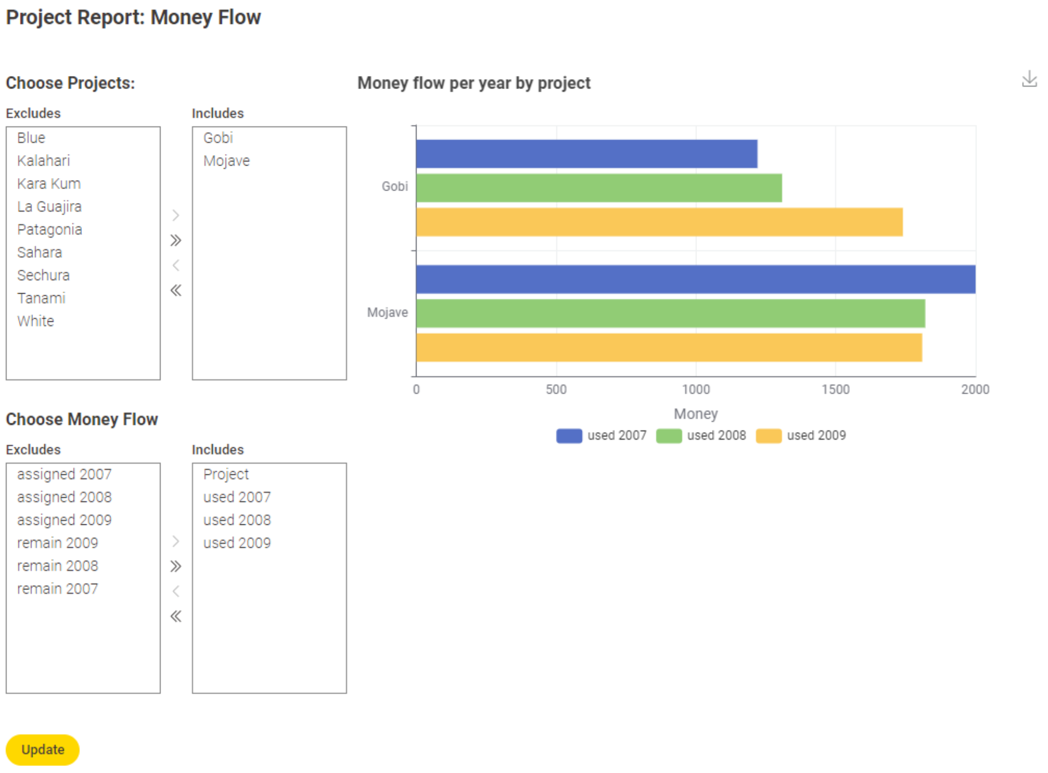

Text View
Project title
Refresh Button Widget
Creates a yellow execution button
Component Input
Column Filter Widget
Nominal Row Filter Widget
Bar Chart
Component Output
Allows a user to choose which columns to include in dashboard
Allows a user to choose which rows to include in dashboard
Bar chart of remaining money by project by year

Figure 6.17. Component Project Report: Money Flow to implement the first Dashboard.

Notice that the Refresh Button Widget only applies to the Column Filter Widget and allows us to manually trigger re-execution of the dashboard upon a new column selection. However, we use the Re-execution tab in the Nominal Row Filter Widget for an automatic re-execution every time a new project is selected in the dashboard. In certain cases, it may be easier to manually re-execute, while in other cases you may want re-execution to happen automatically.

With this dashboard, we have the ability to focus on any particular set of years (2007, 2008, 2009) or categories of money flow (used, assigned, remaining) as well as on any particular project or set of projects (such as Gobi, Kalahari, etc.).

To advance our dashboard-making skills even further, we should now learn how to improve the user experience. This is the topic of the next two sections.


Project Report: Money Flow

Choose Projects:

Excludes	Includes
Blue	Gobi
Kalahari	Mojave
Kara Kum	
La Guajira	
Patagonia	
Sahara	
Sechura	
Tanami	
White	

Choose Money Flow

Excludes	Includes
assigned 2007	Project
assigned 2008	used 2007
assigned 2009	used 2008
remain 2009	used 2009
remain 2008	
remain 2007	

Figure 6.18. The Project Report: Money Flow Dashboard.

6.5. Text autocompletion

In the Project Report: Money Flow dashboard, you may have noticed that the twinlists take up quite a bit of space to expose all possible combinations to the end-user. This may not be ideal from a user experience point of view. In this section we will look at a way to save space, add a level of sophistication to the user experience, and restrict what data we allow the user to access with the Autocomplete Text Widget node.

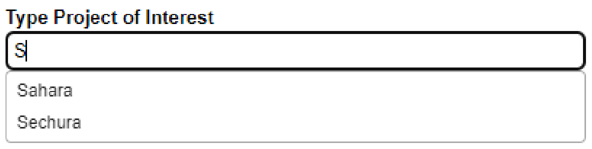


Figure 6.19. Autocomplete textbox from the Autocomplete Text Widget node in an interactive view.

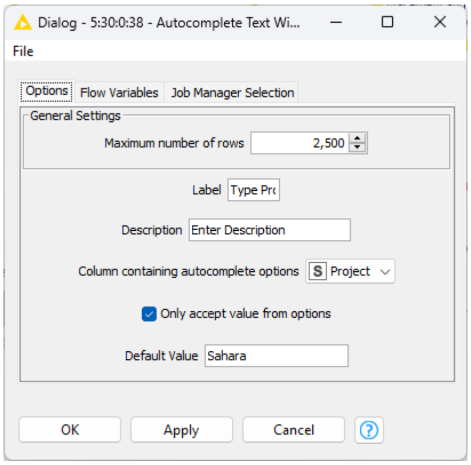
As the name suggests, a user can enter the first few letters of a word into a search box and this node will offer a selection of values from a selected input column, to complete the word. The node also contains a degree of protection by hiding specific column values.

Autocomplete Text Widget

The "Autocomplete Text Widget" node creates an interactive view displaying a text input field with autocomplete functionality.

The configuration window of the node does not require any settings to be configured/selected, besides the "Column containing autocomplete options" dropdown menu in which you select the column containing the wor ds you would like to be automatically completed.

Notice that the list of available words to complete will be from the "Maximum number of rows" contents, so if you have unique values beyond the default maximum, increase the maximum value to capture all possible words you would like autocompleted.


Dialog - 5:30:0:38 - Autocomplete Text Wi...

General Settings

Maximum number of rows: 2,500

Label: Type Pro

Description: Enter Description

Column containing autocomplete options: S Project

☑ Only accept value from options

Default Value: Sahara

Figure 6.20. The configuration window of the Autocomplete Text Widget node.

Let us now look at a simple example component which makes use of the Autocomplete Text Widget node. In this example, we use this node to select a specific project. The specific project then controls a Row Filter node to select the corresponding data rows. A Refresh Button Widget node is then introduced for re-execution after a selection of a new project. The composite view of this component is shown in the figure below.

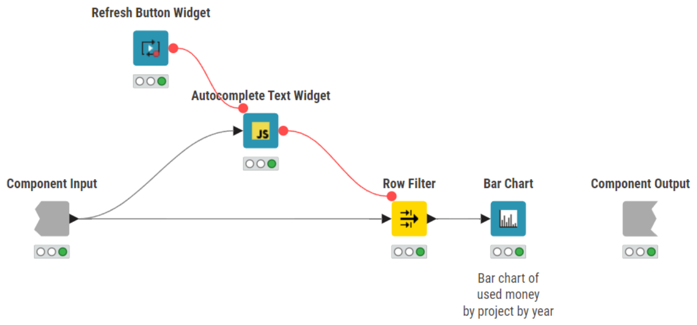


Figure 6.21. Simple component using the Autocomplete Text Widget node for project selection.

Notice that we require the Refresh Button Widget node since the Autocomplete Text Widget does not have a Re-execution tab.

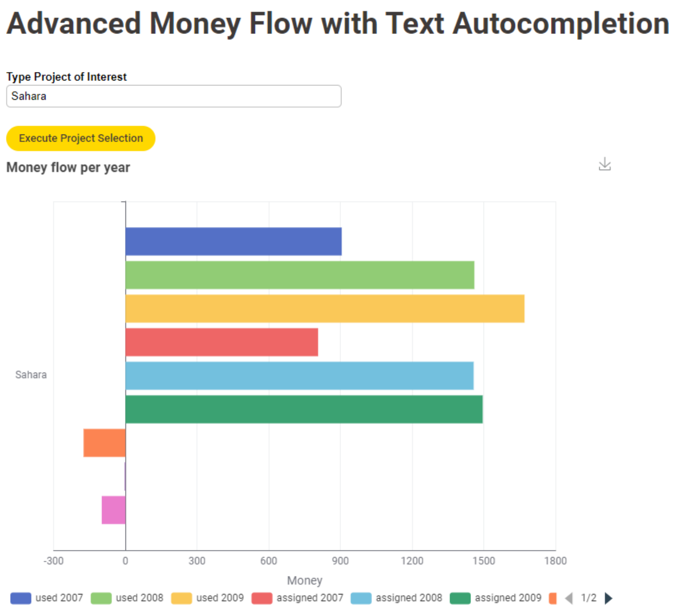


Figure 6.22. Output of the simple Text Autocompletion component using the Autocomplete Text Widget.

However, when we inspect the Interactive View of this simple component, we get a dashboard that is a bit too colorful for our taste. Due to the variety of colors, the dashboard viewer has no clear focus. In the next section, we will create custom color filters to focus better on certain groups of years. We can then extend what we learned to create custom filters for data.

6.6. Custom Filtering

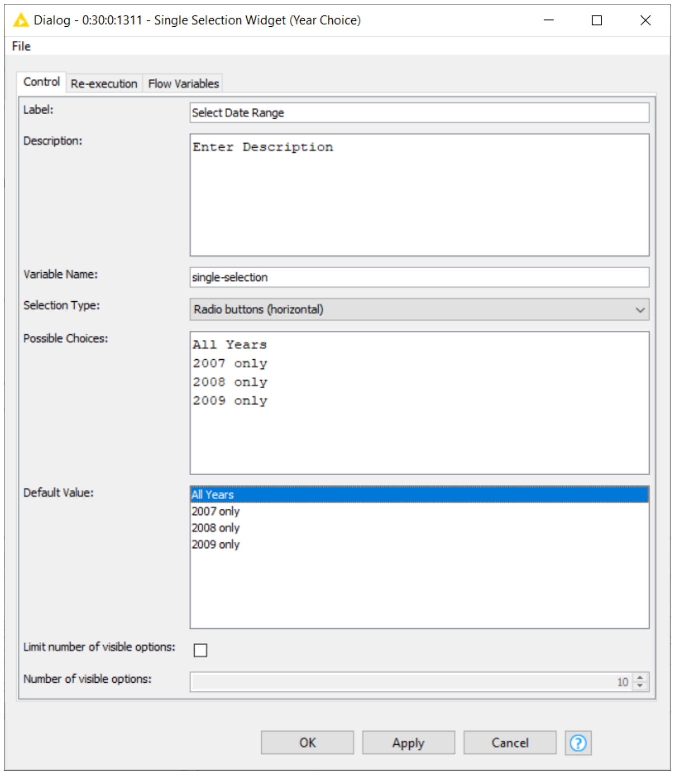
In the previous section we allowed a user to see all columns from the original dataset and then to select a subset of rows to view in their dashboard. Sometimes, however, it is required to allow users to see only some of the columns and/or some of the rows. Some other times, we want to expose columns with more user-friendly and meaningful names. In such cases, the Single Selection Widget node can be useful.

Single Selection Widget

Single Selection Widget node allows selecting a single value from a list of strings in an encapsulating component's view.

In “Possible Choices” we simply type out the options we would like displayed to a user who accesses the Configure menu from the component encapsulating this widget.

Notice we can also set a label, description, and flow variable name. Additionally, we can change the type of button displayed in the view via “Selection Type” and we can limit the number of visible options.


Dialog - 0:30:0:1311 - Single Selection Widget (Year Choice)
File
Control
Re-execution
Flow Variables
OK
Apply
Cancel

Label: Select Date Range
Description: Enter Description
Variable Name: single-selection
Selection Type: Radio buttons (horizontal)
Possible Choices:
All Years
2007 only
2008 only
2009 only
Default Value: All Years
2007 only
2008 only
2009 only
Limit number of visible options: ☐
Number of visible options: 10

Figure 6.23. The configuration window of the Single Selection Widget node.

Let’s give a practical example. Let’s suppose that we are interested in visualizing the cash flow for each single year; we would like this to enable a filter to select the one year of cash flow to visualize.

We can manually type the years as the menu options into the Default Value pane of the configuration window of the node. Since the menu options do not match one to one the column names, we use a Rule Engine Variable to force the match and the Column Filter node to select the column to visualize. The Rule Engine Variable, discussed in detail in Chapter 5, will transform the menu options from the Single Selection Widget node into column names that the Column Filter Node recognizes (see figure below).

Now that we have a general understanding of how these nodes are connected, let us concretely

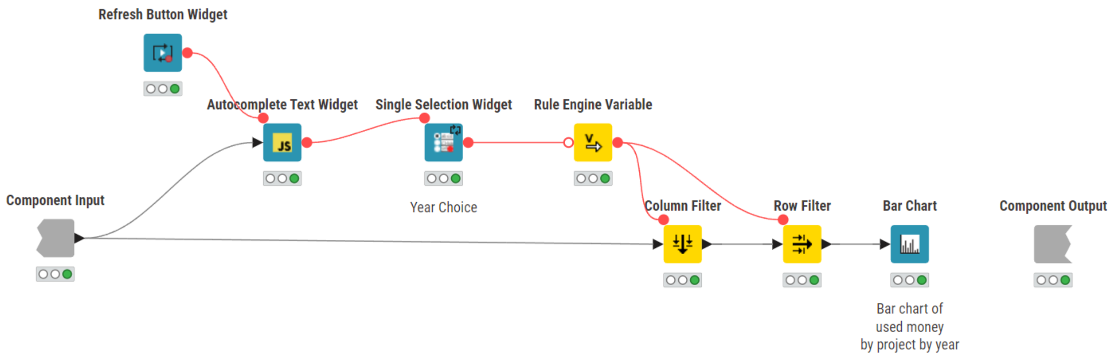


Figure 6.24. Combining a Single Selection Widget node with the Rule Engine Variable node to filter columns.

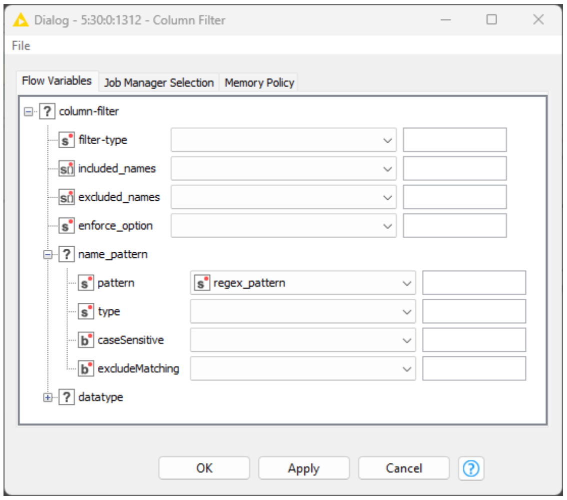
state what is happening here:

1. Create custom menu options ("All Years", "2007 only", "2008 only", "2009 only") by manually typing them in the Single Selection Widget configuration window (i.e., within Possible Choices pane).

2. Translate those option names into column names that the Column Filter node understands via the Rule Engine Variable node. In this case we use pattern matching to look for "200X".

3. Within the Column Filter node's Flow Variables setting (image on the right), set the Rule Engine Variable node's flow variable (this flow variable is called regex_pattern in our example below).

With that knowledge, we are finally able to build the Advanced Money Flow with Text Autocompletion dashboard. We will



column-filter
- filter-type
- included_names
- excluded_names
- enforce_option
- name_pattern
  - pattern: regex_pattern
  - type
  - caseSensitive
  - excludeMatching
- datatype

Figure 6.25. The configuration window of the Column Filter node.

combine the Autocomplete Text Widget node and the Single Selection Widget node to build a sleek, easy to use dashboard. The content of the component is shown in Figure 6.24.

And remember, if we would like a large title, we can simply add the Text Output Widget node as well.

Did you notice we didn't include the usual title? What do you think? Do we need a title? How about the spacing or positions of the widgets? Remember that if you would like to change the layout of the dashboard, you can either

- right-click the component and select *Component* > *Open layout editor*, or
- open the component and click on the "Open layout editor" button in the toolbar.

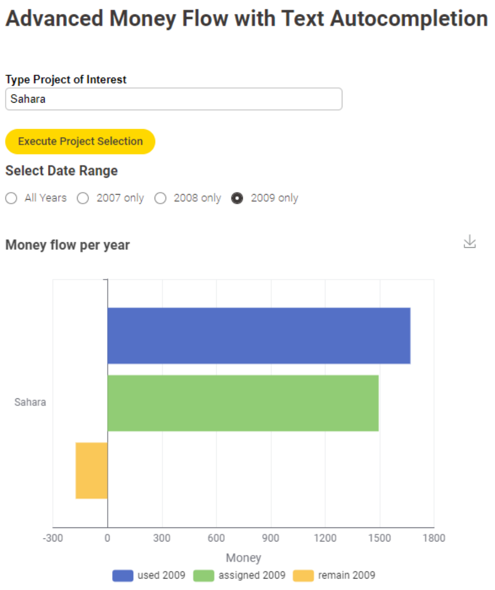


Figure 6.26. Advanced Money flow with Text Autocompletion Dashboard.

Having completed both of the Objective Dashboards, next we will look at how to build an Animated Bar Chart which we can use to capture the attention of our audience.

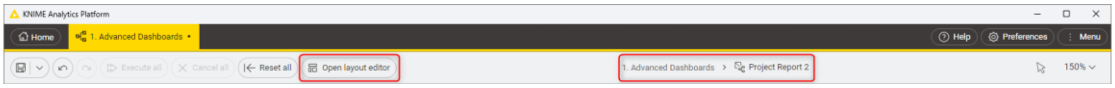


Figure 6.27. The layout button in the toolbar of the KNIME Workbench.

6.7. Animating a vanilla bar chart with community components

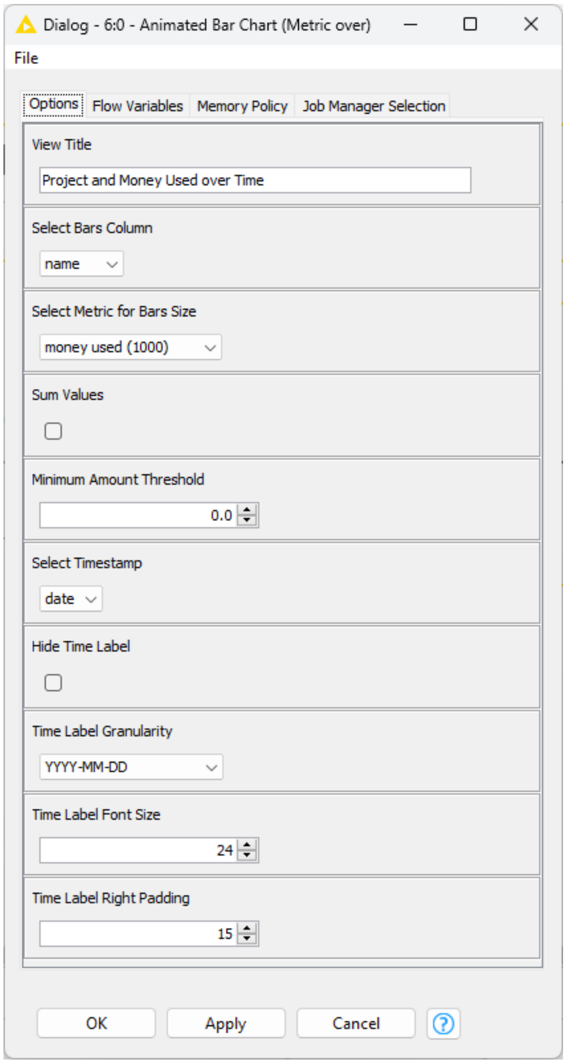
As you know, components are a sequence of nodes packaged together that you create within a KNIME workflow. They encapsulate and abstract functionality, can have their own dialog, and can have their own sophisticated, interactive views. Components can be reused in your own workflows but also shared with others: via KNIME Business Hub or the KNIME Community Hub. They can also represent web pages in a guided analytics application deployed via KNIME Business Hub. That is to say, anyone can create and share their components with the community.

Verified components are a set of components that behave like KNIME nodes, including error handling capabilities, which are developed by KNIME and regularly released on the KNIME Community Hub. Examples include the AutoML component (great for Data Scientists), the Measure Fractional Years component (great for Excel users) and the Animated Bar Chart component (great for everyone). In this section we will introduce the Animated Bar Chart component.

The Animated Bar Chart component requires three input columns:

- a nominal column with category values (such as Project name),
- a numerical column with some metric values (such as money used),
- and a date column.

In the case of the Project.txt file, we have: money related columns, the project column, and the year column. We need to transform the year column into a date column. From there, we simply populate "Select Bar Column" (nominal value) with the project column, "Select Metric for Bars Size" (numeric value) with the money column, and "Select Timestamp" (datetime value) with the year column now in form of a date&time column.



View Title: Project and Money Used over Time
Select Bars Column: name
Select Metric for Bars Size: money used (1000)
Sum Values: ☐
Minimum Amount Threshold: 0.0
Select Timestamp: date
Hide Time Label: ☐
Time Label Granularity: YYYY-MM-DD
Time Label Font Size: 24
Time Label Right Padding: 15

Figure 6.28. The configuration window of the Animated Bar Chart component.

Animated Bar Chart

"Animated Bar Chart" verified component creates a bar chart that visually updates starting with the earliest data until reaching the latest data.

It requires a nominal column, a numeric column, and a datetime column. The configuration window requires:

- “Select Bar Column” (nominal value) for the category,
- “Select Metric for Bars Size” (numeric value) for the measure changing over time,
- “Select Timestamp” (datetime value) for the time values

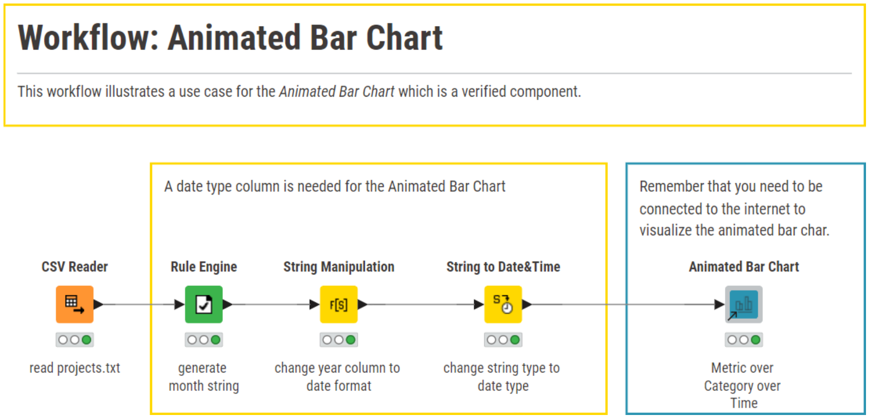
You can tick the Sum Values option to add an aggregation if needed.

As well, if you set a Minimal Amount Threshold, an aggregation sum lower than this value will be discarded. Raising this threshold increases the smoothness of the visualization and reduces the number of animations.

We will use the *Projects.txt* data and transform the year column which is typically read as an Integer into a date format, via a *String Manipulation* node to zero pad the year and then via a String to Date&Time node to convert it to a Date&Time object.

Once a date column is ready, we will set the following options in the Animated Bar Chart component using columns from Projects.txt:

1. “Select Bar Column”: “name” which is a nominal value, i.e., the name of the project
2. “Select Metric for Bars Size”: “money used (1000)” which is a numeric value
3. “Select Timestamp”: “date” which is the year as a date value


Workflow: Animated Bar Chart

This workflow illustrates a use case for the *Animated Bar Chart* which is a verified component.

A date type column is needed for the Animated Bar Chart

Remember that you need to be connected to the internet to visualize the animated bar char.

Figure 6.29. Workflow for animating a bar chart.

This allows us to produce a bar chart that shows us the change in money used through time for each project.

Congratulations for finishing these seven advanced sections. You may have noticed that we did not touch upon the topic of color which is important when creating dashboards. This was on purpose. In the final section of this chapter, one of your exercises will ask to create a color palette option using the Single Selection Widget node. Once you’ve completed and reviewed the answers provided for each exercise, you will be able to create interactive dashboards with a customized experience for internal or external use.

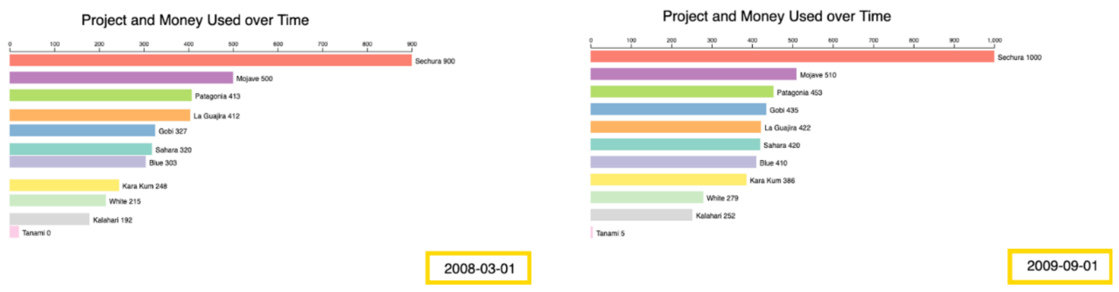
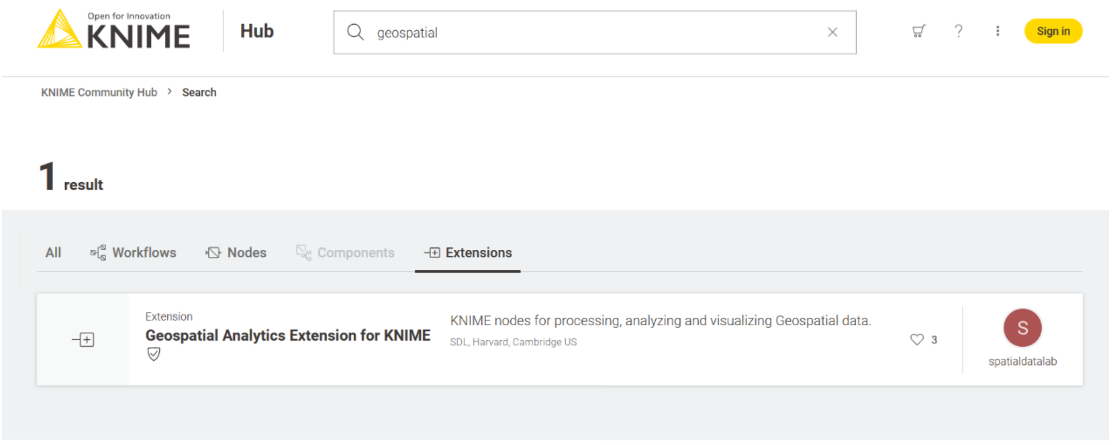


Figure 6.30. Time elapsing in an Animated Bar Chart interactive view.

6.8. Geospatial Analytics Extension

KNIME software was not born as a data visualization tool, but it is starting to become one. You can already produce advanced plots and charts and assemble them together to form fancy interactive dashboards accessible via web browser. The latest addition to the data visualization extension are the geospatial analytics nodes. Developed by the group of Prof. Guan at Harvard University, these nodes wrap Python GeoPandas libraries to export geospatial functionality into the familiar easy to use KNIME GUI. By the way, this is one of the first examples of successful usage of the latest KNME Python integration, which allows users to create KNIME nodes from Python code.



1 result

Extension
Geospatial Analytics Extension for KNIME

KNIME nodes for processing, analyzing and visualizing Geospatial data.
SDL, Harvard, Cambridge US

3

spatialdatalab

Figure 6.31. The extension with geolocation nodes on the KNIMECommunity Hub.

To create geolocation maps in your dashboard, first of all, you need to download and install the KNIME Geospatial Extension. The geospatial analytics extension has been included into KNIME Analytics Platform as of December 2020 with version 4.7. On the KNIME Community Hub search for "geospatial" under the tab "Extensions", then drag & drop the row "Geospatial Analytics Extension for KNIME” onto your KNIME Analytics Platform to install it.

Once installed, you should see a folder named “Geospatial Analytics” under “Community Nodes” in the Node Repository of your KNIME Analytics Platform. This folder contains a very large number of nodes offering a variety of geolocation functions and we will see just a few of these nodes in this section.

Note. The visualization nodes of the geospatial Analytics Extension require an internet connection to work properly, since they need to connect to the underlying dataset to display the world map.

We will implement two very simple geolocation tasks, just to show how to use these nodes:

- Task 1: Display a place (like a city) on a map
- Task 2: Display a country on a map and overlay a representative point to it.

Besides these very simple tasks, more could be achieved, like for example calculating distances, analyzing data, and producing models. For the sake of this introduction, however, we will just show how to display polygons and points on a map.

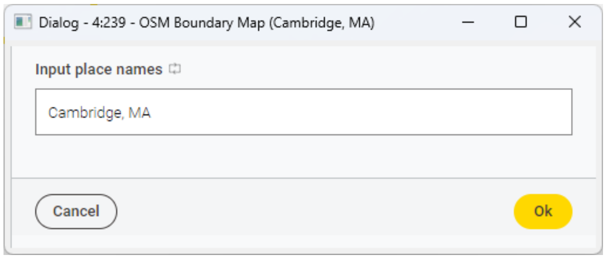
Task 1: Display a Place on a Map

Let's start with a simple task: the visualization of a place, any place, on a world map. Let’s take the city of Cambridge (MA) as an example. Geolocation is based on two basic shapes: polygons and dots. Polygons enclose a city or a country within their geographical boundaries; dots indicate just the location without any information on boundaries.

Let’s try to display the city of Cambridge (MA) with its boundaries on the world map. What we need for this simple task are two nodes: the OSM Boundary Map node and the Geospatial View node.

OSM Boundary Map

The OSM Boundary Map node relies on an open map dataset, for example the Open Street Map (OSM) dataset, for boundary information. The boundary of a place is expressed via a polygon. The node receives as input the name of the place – either country, city, or village – retrieves the required information from the dataset, and produces the


Dialog - 4:239 - OSM Boundary Map (Cambridge, MA)

Input place names

Cambridge, MA

Figure 6.32. The configuration window of the OSM Boundary Map node.

corresponding boundary polygon at the output form. Data such as polygons are stored in a new data type: the Geometry type. The only setting required in the node configuration window is the place we want to show in the map.

Another interesting node from the OSM dataset is the OSM POIs node. This node extracts all points of interest for a selected category in the region(s) enclosed in the polygon(s) in a Geometry type column.

Let's add the OSM Boundary Map node to the workflow and let's configure it to export the polygon around the city of Cambridge (MA). The second node we need is a visualization node: the Geospatial View node.

Geospatial View

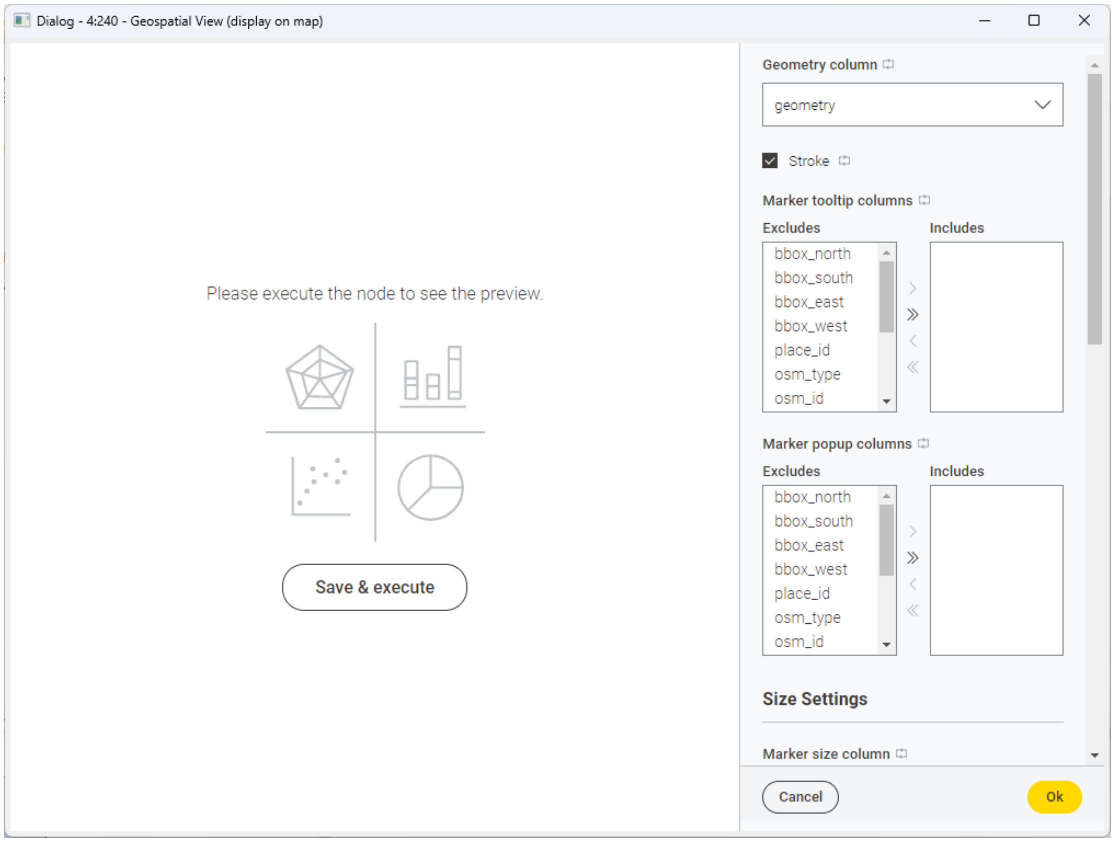
This node displays polygons - via polygon objects in a Geometry column - and points - via latitude and longitude coordinates - on a world map. The first setting in the configuration window is the Geometry-type column to use as input for the display.

Besides that, a number of visualization settings are also required, such as tooltip content, size, color, classification, base map, and legend. The base map is an interesting parameter, since different types of maps display different representations of the same polygon. For this example, we chose the Open Street Map for this example.

This node has a preview on the left of the configuration window. This preview can be refreshed after each setting modification to see the final view. To refresh the preview, just click the button "Save & Execute".

To confirm the setting update, click the button "OK" in the lower right corner. If you have refreshed the preview, the node will then appear as already executed, and the final view would already be available.

The node has a View, like all visualization nodes. Hover over the node and click the "Open view" button in the node's Action Bar. You will get the map with the selection of Cambridge in blue. Notice the zoom in /zoom out button in the top left corner of the view.


Dialog - 4:240 - Geospatial View (display on map)

Please execute the node to see the preview.

Save & execute

Geometry column: geometry

☑ Stroke

Marker tooltip columns

Excludes	Includes
bbox_north	
bbox_south	
bbox_east	
bbox_west	
place_id	
osm_type	
osm_id	

Marker popup columns

Excludes	Includes
bbox_north	
bbox_south	
bbox_east	
bbox_west	
place_id	
osm_type	
osm_id	

Size Settings

Marker size column

Cancel Ok

Figure 6.33. The configuration window of the Geospatial View node.

After execution, the view of the *Geospatial View* node will show the polygon enclosing the city of Cambridge (MA) on a world map. The final workflow, named “3. Visualize a place on a map” and consisting of just two nodes, is reported in the figure above and is available in the Chapter 6 folder.

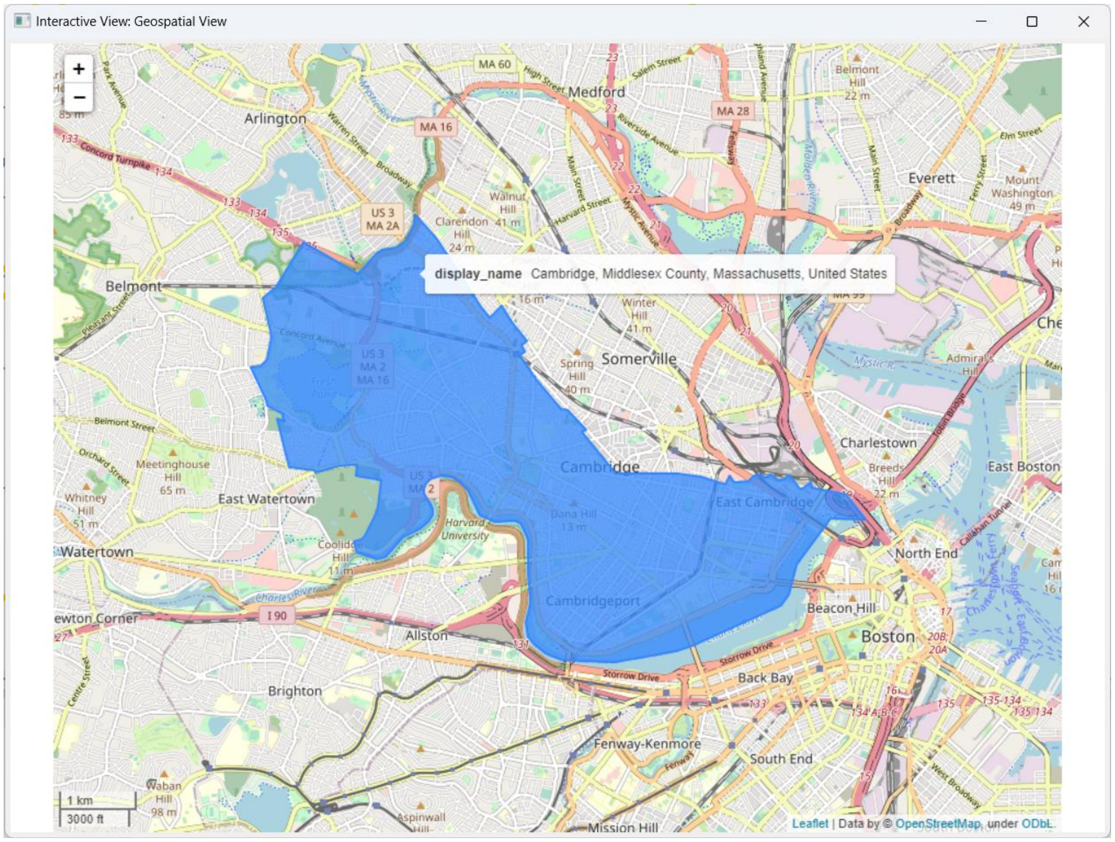
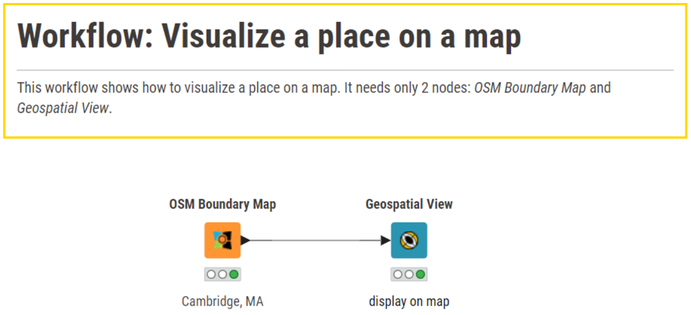

Interactive View: Geospatial View
display_name Cambridge, Middlesex County, Massachusetts, United States
Leaflet | Data by © OpenStreetMap, under ODbL

Figure 6.34. Cambridge (MA) on a map.



Workflow: Visualize a place on a map

This workflow shows how to visualize a place on a map. It needs only 2 nodes: *OSM Boundary Map* and *Geospatial View*.

Figure 6.35. The minimum workflow to locate and display a place within its boundaries.

A final note on these new nodes. The tabs at the top of the configuration window are missing. To configure your settings via Flow Variables, you need to right-click the node and select "Configure Flow Variables…". This will then lead you to the window that overwrites the configuration settings via Flow Variables. Once a configuration setting has been set via a flow variable, a little barred square with a red dot at the top appears close to that setting in the configuration window. This means the node operates based on the value of a Flow Variable for that setting.

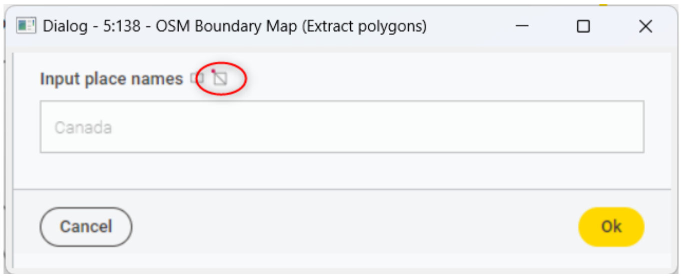


Figure 6.36. The circled icon shows that this setting is operated by a Flow Variable.

Task 2: Visualization of a Country on a World Map

Let's take on a more complex challenge. Let's visualize a country and a representative point on it. As an example, let's visualize Italy. In the *OSM Boundary Map* node, now we set "Italy" as setting.

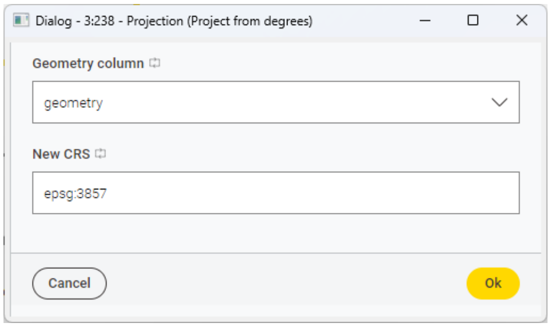
The polygon around Italy is then produced and stored in a Geometry type column. Coordinates in the polygon are expressed in degrees. We want to pass from the degree system to the metric system for an easier and more interpretable calculation of distances, for example. The node that does that is the *Projection* node.

After that, we want to draw a representative point for the region that we have previously selected. The node to do that is the *Geometry To Point* node.

Projection

The Projection node transforms the Coordinate Reference System (CRS) of the Geometry column into a new system compatible with the metric system, all points into objects, and all connections into lines.

The configuration window requires the Geometry type column that we want to transform and the new coordinate system. The output data table has the same content as the input data table, but the Geometry type column now uses the new reference system.


Dialog - 3:238 - Projection (Project from degrees)

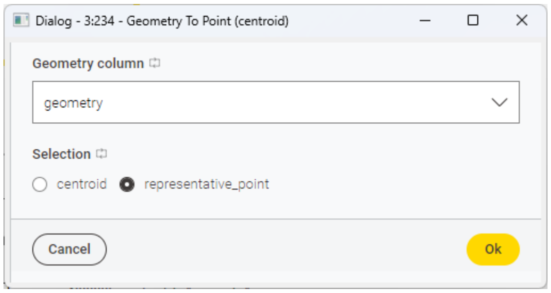
Geometry column: geometry

New CRS: epsg:3857

Figure 6.37. The configuration window of the Projection node.

Geometry To Point

Geometry To Point returns a point representing each geometry. There are two types of points: centroids and representative points. The centroids are calculated and depending on the shape of the polygon can happen to be located outside of the region; the representative points on the opposite are fixed and are guaranteed to be within each geometry. All the configuration window needs here is:


Dialog - 3:234 - Geometry To Point (centroid)

Geometry column: geometry

Selection: centroid [✓ representative_point]

Figure 6.38. The configuration window of the Geometry To Point node.

- the Geometry type column containing the polygons
- the type of point we want to draw, whether the centroid or the representative point

After execution this node produces a description of the point, including its latitude and longitude and the point stored in a new Geometry-type column.

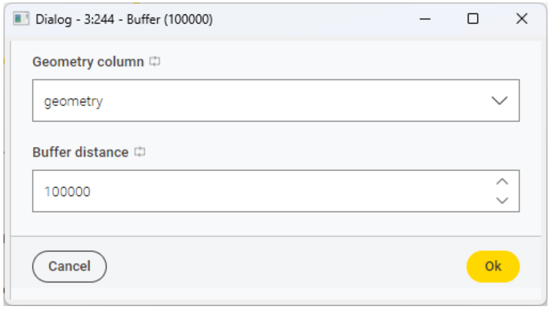
The visualization of a point is very tiny and hard to distinguish especially if compared to the size of the country. So, we use the Buffer node to pad the dot with some extra space on the map.

Buffer

The Buffer node pads a point on the map with some extra space all around. The only settings required by the Buffer node are:

- the Geometry type column containing the point
- the buffer size (aka distance)

The Buffer node then transforms the point into a polygon containing the padding space. This polygon is then saved in yet another Geometry type column.


Dialog - 3:244 - Buffer (100000)

Geometry column: geometry

Buffer distance: 100000

Figure 6.39. The configuration window of the Buffer node.

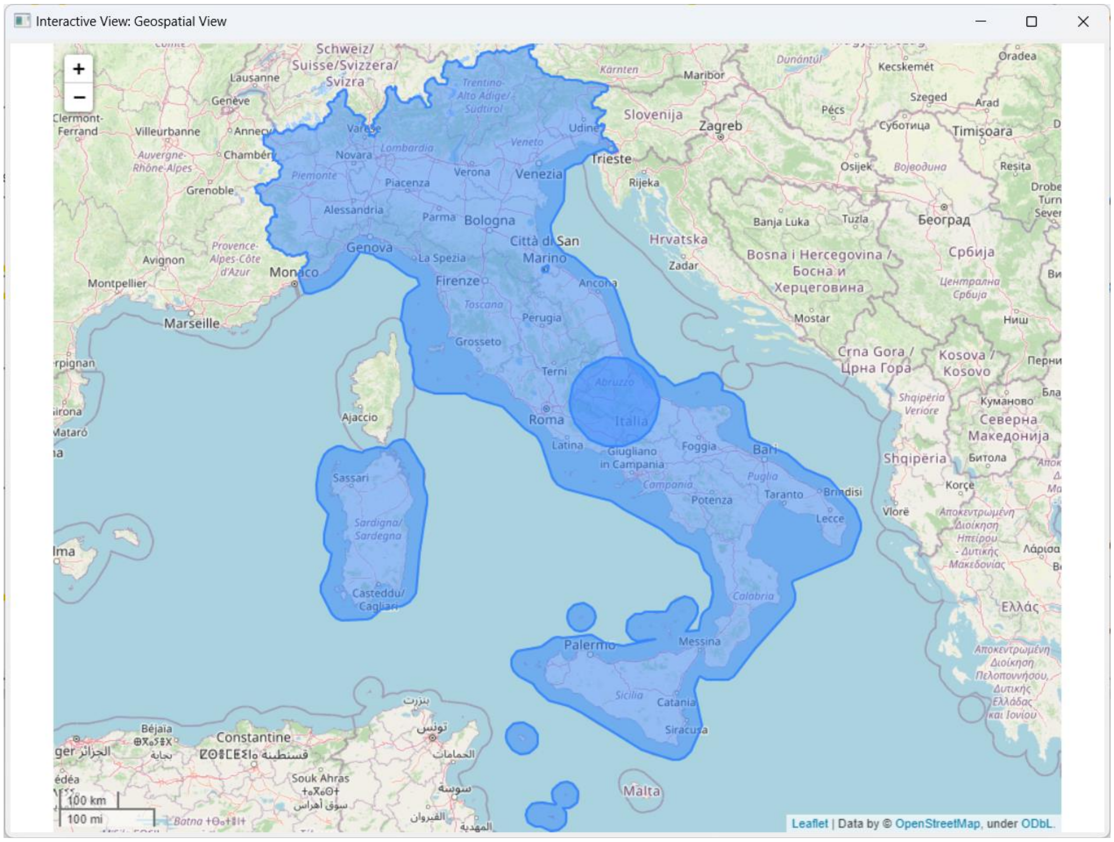

Interactive View: Geospatial View
Italia
Roma
Leaflet | Data by © OpenStreetMap, under ODbL.
100 km
100 mi

Figure 6.40. Italy on a world map with its buffered representative point in the middle.

We used a buffer size of 100 000. The output data table thus contains a larger point stored as a polygon in the new Geometry type column. At this point, we keep all polygons – country and representative point – to be visualized, via a Concatenate node, and we feed with those a Geospatial View node for the visualization. The final visualization is shown in the figure above, the final workflow named "4. Visualize a country on a map", is displayed below.

We will stop here for now, knowing that what we have shown is just a little taste of what the geospatial analytics extension can offer.

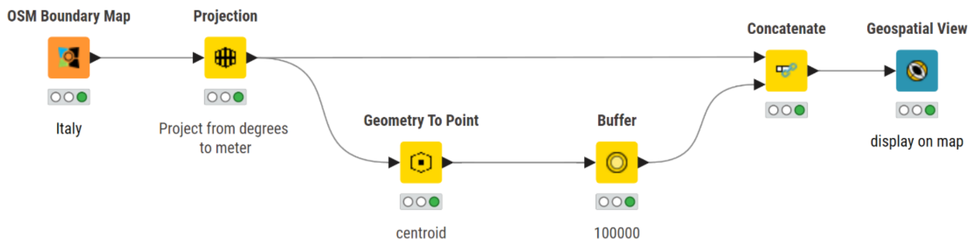


Figure 6.41. Workflow that visualizes a country and its representative point.

6.9. Exercises

Exercise 1

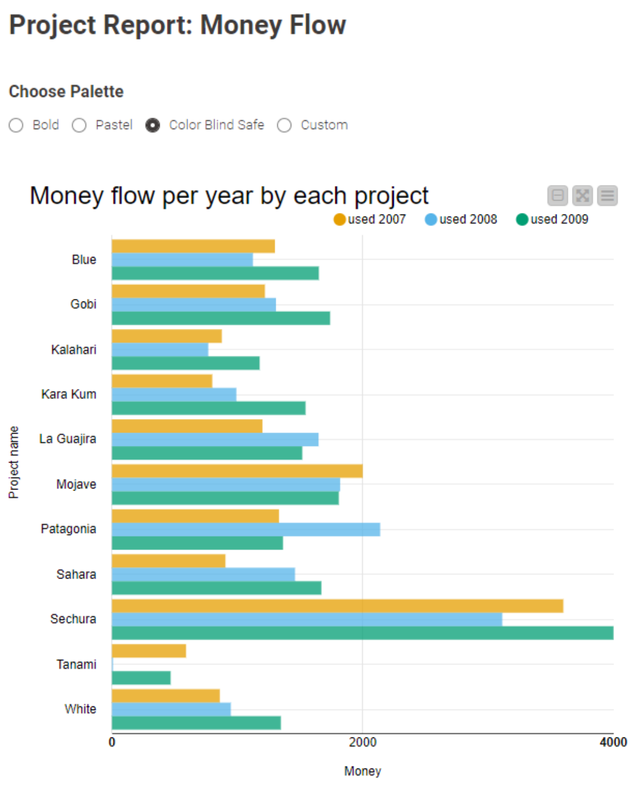
In this chapter, we have learned to use the Single Selection Widget node to create custom menu options. For this exercise, you will create radio buttons which allow the user to change the color palette of a simple dashboard. For the dashboard, create a bar graph using the data from the "used" year columns in the Projects.txt file.

Solution to Exercise 1

We require several nodes contained within a component to achieve this task. Specifically, we need a:

- Single Selection Widget node to create the selection menu
- Rule Engine Variable node to transform the menu selection into a viable setting
- Color Manager node using the selected setting to implement the desired color palette
- Bar Chart (JavaScript) node to display the bar chart in the desired color palette
- Extract Table Spec node allows us to put the columns in a row format which the Color Manager will accept.

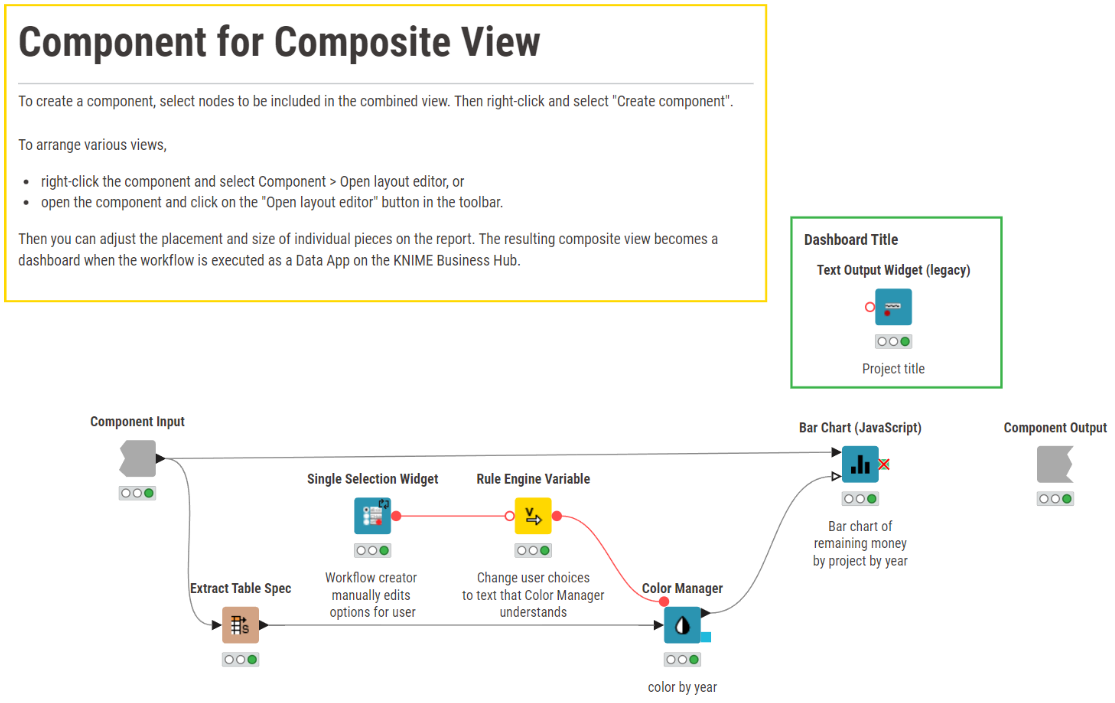
Note. The Single Selection Widget must have re-execution enabled.


Project Report: Money Flow

Choose Palette

○ Bold ○ Pastel [✓ Color Blind Safe] ○ Custom

Figure 6.42. The interactive view produced by the Single Selection Widget node to be created for Exercise 1.



Component for Composite View

To create a component, select nodes to be included in the combined view. Then right-click and select "Create component".

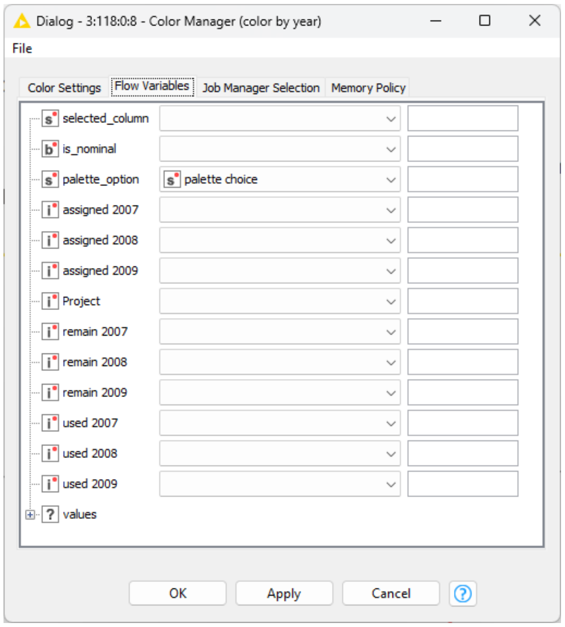
To arrange various views,

- right-click the component and select Component > Open layout editor, or
- open the component and click on the "Open layout editor" button in the toolbar.

Then you can adjust the placement and size of individual pieces on the report. The resulting composite view becomes a dashboard when the workflow is executed as a Data App on the KNIME Business Hub.

Dashboard Title
Text Output Widget (legacy)
Project title
Component Input
Bar Chart (JavaScript)
Component Output
Single Selection Widget
Rule Engine Variable
Bar chart of remaining money by project by year
Extract Table Spec
Workflow creator manually edits options for user
Change user choices to text that Color Manager understands
Color Manager
color by year

Figure 6.43. Exercise 1: Contents of the component implementing the dashboard.



selected_column	
is_nominal	
palette_option	palette choice
assigned 2007	
assigned 2008	
assigned 2009	
Project	
remain 2007	
remain 2008	
remain 2009	
used 2007	
used 2008	
used 2009	
values	

Figure 6.44. Configuration of the Color Manager's "Flow Variables" tab.

The Rule Engine Variable node implements the following rules:

```
$${Isingle-selection-navigation}$$ = "Bold" => "set_1"
$${Isingle-selection-navigation}$$ = "Pastel" => "set_2"
$${Isingle-selection-navigation}$$ = "Color Blind Safe" => "set_3"
$${Isingle-selection-navigation}$$ = "Custom" => "custom_set"
```

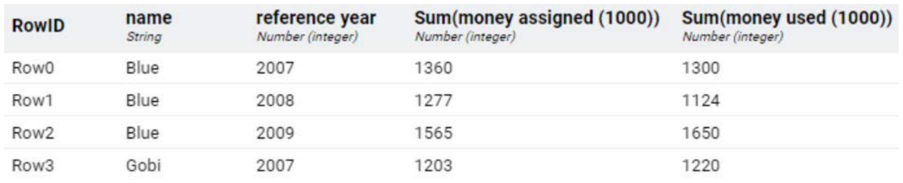
If the user chooses "Bold", then the "set_1" option is chosen for the "palette_option" flow variable within the Color Manager dialog. If the user chooses "Pastel", then "set_2" is used as the flow variable, so on and so forth. The flow variable named "palette choice" is what we defined as the "New flow variable name" within the Rule Engine Variable node.

Exercise 2

The goal of this exercise is to create a dashboard, based on file Projects.txt, with a stacked area chart to visually show the difference between money used and money assigned per year and per project. The dashboard must also include the ability to dynamically filter to a distinct project using text autocompletion. Finally, allow range slider filtering over the amount of money used.

If that seems difficult then here are a few hints. This exercise has 3 primary subtasks:

- Transform the raw Projects.txt data such that your table contains the sum of money used and money assigned for each project, per year. The first 7 rows of expected data are shown below for your reference.
- With the transformed data, add a Refresh Button Widget node and the Autocomplete Text Widget node. The user should have the ability to refresh the text using the yellow button and the text chosen by the user must filter the data from raw Projects.txt.
- Finally, add the Interactive Range Slider Widget node.



RowID	name String	reference year Number (integer)	Sum(money assigned (1000)) Number (integer)	Sum(money used (1000)) Number (integer)
Row0	Blue	2007	1360	1300
Row1	Blue	2008	1277	1124
Row2	Blue	2009	1565	1650
Row3	Gobi	2007	1203	1220

Figure 6.45. Transformation of the Projects.txt data.

Bonus: For which years was the money used above the value 1000?

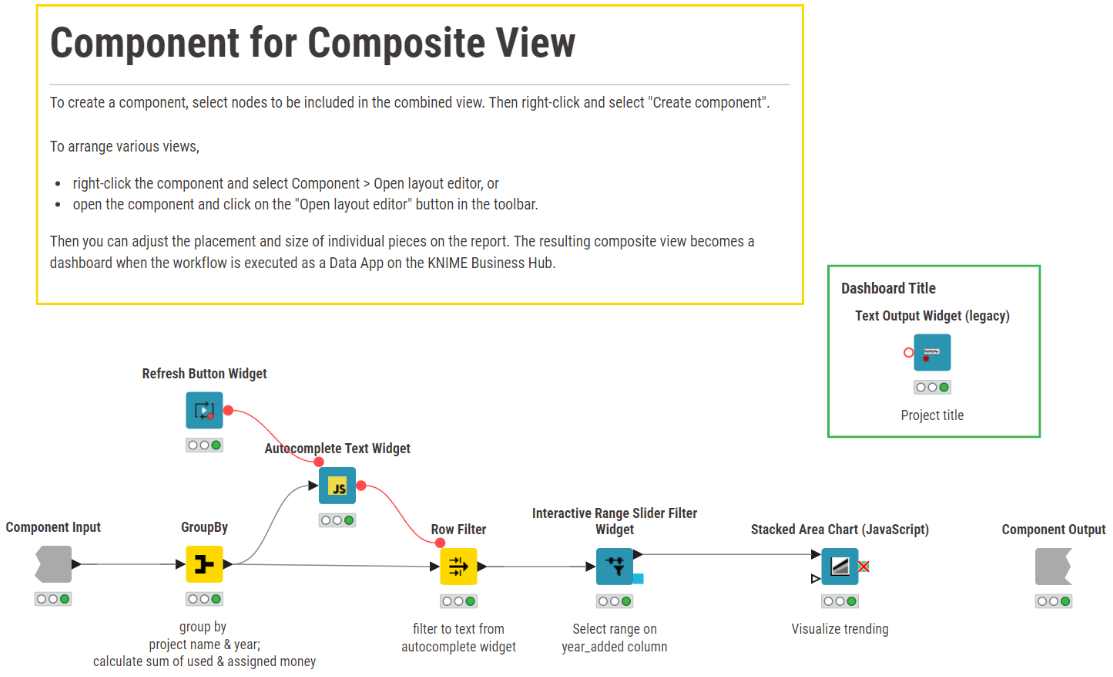
Solution to Exercise 2

First, transform the raw data using the GroupBy node. In the Groups tab, we will group using the project name and reference year columns. In the Manual Aggregation tab, we will sum the money assigned and the money used.

Next, we will filter our data, using a Row Filter node, to the project of interest using a Autocomplete Text Widget node. This node must be connected to a Refresh Button Widget node so the user can change the selected project at any time.

Next, we will connect to an Interactive Range Slider Filter Widget node which allows us to slide over money used.

Finally, we will connect our aggregated data to the Stacked Area Chart (JavaScript).



Component for Composite View

To create a component, select nodes to be included in the combined view. Then right-click and select "Create component".

To arrange various views,

- right-click the component and select Component > Open layout editor, or
- open the component and click on the "Open layout editor" button in the toolbar.

Then you can adjust the placement and size of individual pieces on the report. The resulting composite view becomes a dashboard when the workflow is executed as a Data App on the KNIME Business Hub.

Dashboard Title
Text Output Widget (legacy)
Project title
Refresh Button Widget
Autocomplete Text Widget
Component Input
GroupBy
group by project name & year; calculate sum of used & assigned money
Row Filter
filter to text from autocomplete widget
Interactive Range Slider Filter Widget
Select range on year_added column
Stacked Area Chart (JavaScript)
Visualize trending
Component Output

Figure 6.46. Exercise 2: Contents of the component solution to implement the required dashboard.

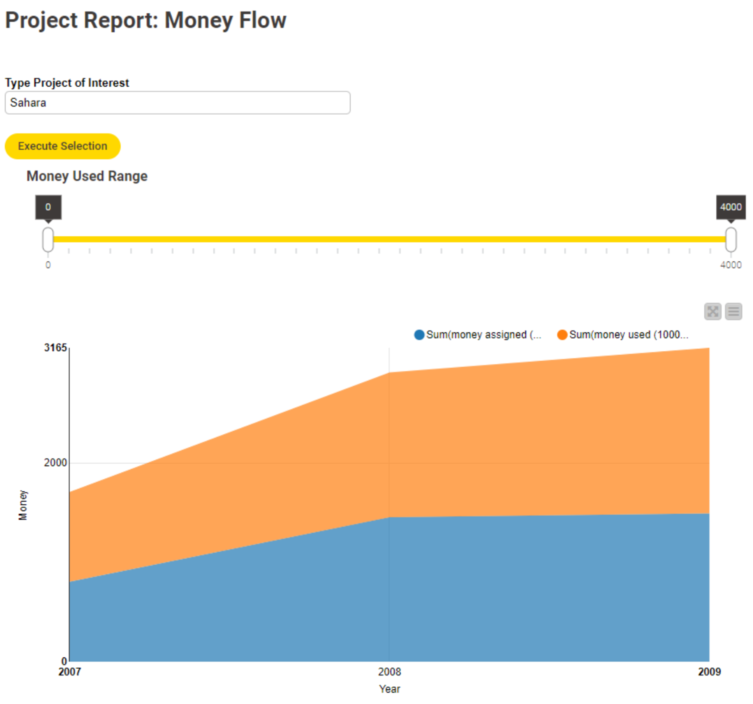


Figure 6.47. The dashboard for exercise 2.

Chapter 7: Loops

7.1. What is a Loop

In the “Node Repository” there is a full category called “Workflow Control” for logical and control operations on the data flow. The “Workflow Control” category contains a number of sub-categories, like:

- “Automation” contains a number of utility nodes to improve the controlled execution and orchestration of workflows;
- “Variables” contains all those nodes that create, delete, and update flow variables (Chapter 4).
- “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;
- “Error Handling” nodes switch to an alternative workflow branch in case of error
- “Meta Nodes” contains a few pre-packaged commonly used loops, for example to inject variables or to read all files in a folder.

In this section we will work with the nodes in the “Loop Support” 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 adds the results to the output 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 of operations that is repeated. Each execution of the loop body is called a loop iteration. Loops do not work only on processing data, but also on flow variables. Similar loops can be constructed to export data or flow variables. The output data table at each iteration can appended as a column set or concatenated as a row set to the previous results.

A very commonly used loop iterates across a list of files - for example all files found in a folder with the “List Files” node - reads the content of each file, and piles up the results into the loop output data table. This of course works only if all files have a compatible data structure.

Another example could be to grow an initial numerical value by the current iteration value for N iterations. If the initial numerical value is named “inp” and set to 5, at each loop iteration we add the value of the current iteration “i” - where “i”=1,2,3,4,5,…,N - till “i” = “N”. If “N” is set to 10, the output data table will contain values 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14. In this example, the initial value “inp”=5 can be contained in a flow variable and the output values can be stacked or appended in a data table.

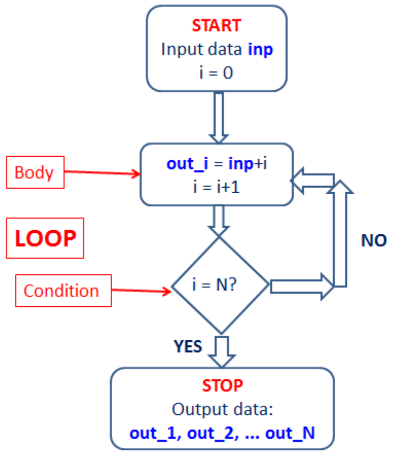


Figure 7.1. Loop Example.

Finally, the classic loop example adds one unit to an integer variable N times. Starting from the variable initial value, we add 1 at each iteration to the result of the previous iteration. The loop stops when we have reached the maximum number of iterations N. The loop output in this case is just one value, i.e., the final sum and not a data table. This one output value could then be exported as a flow variable rather than as a data table. Notice that in this loop each iteration works with the result of the previous iteration. While this is common in programming languages it is less common in data analytics. In data analytics the goal is usually to transform the entire data set and less often to micro-transform the variables beneath. The KNIME Analytics Platform has only one loop for this kind of task: the recursive loop.

In KNIME Analytics Platform there are 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

Note. Not all loops are in the “Workflow Control/Loop Support” category. You will find a few specialized loops here and there in other sub-categories, such as “Ensemble Learning”, “IO”, “Optimization”, and more. Just type “loop” in the search box on top of the “Node Repository” panel and see how many nodes show up that are implementing some kind of loop.

7.2. Loop with a pre-defined number of iterations

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 (see Scatter Plot below). The goal is to produce a copy of this dataset shifted one unit to the right on the x-axis. Let's repeat this operation 4 times to obtain 4 copies of the original three clusters, each copy shifted one more unit to the right than the previous copy. To build the required 4 copies of the original data set, we introduce a loop with 4 iterations. Each iteration moves the original data set of "i" units to the right, where "i" is the number of the current iteration and concatenates the resulting data with the data processed so far.

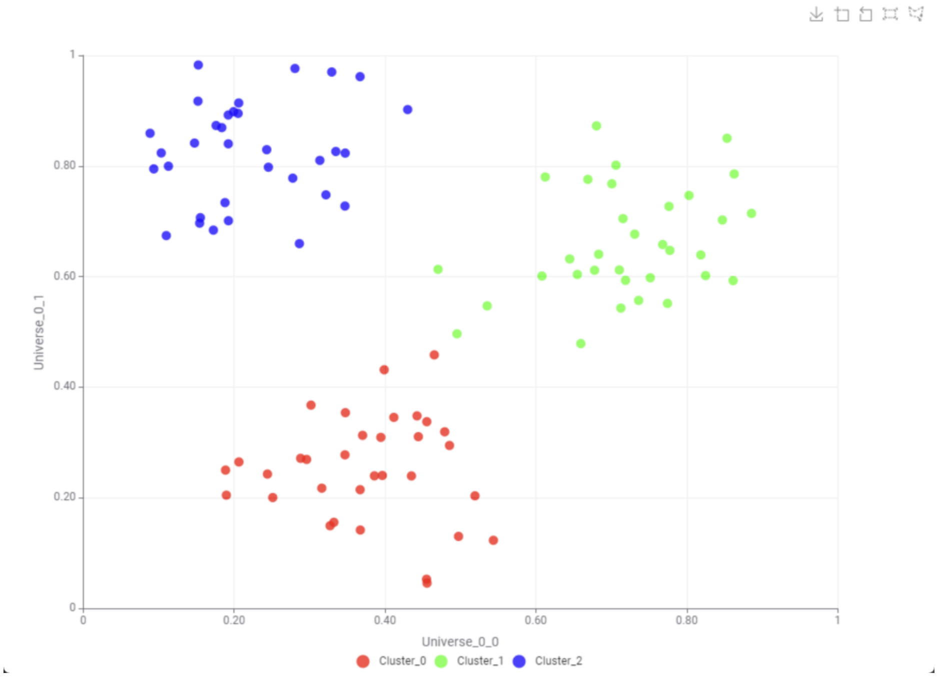


Figure 7.2. Dataset with 3 clusters across two variables.

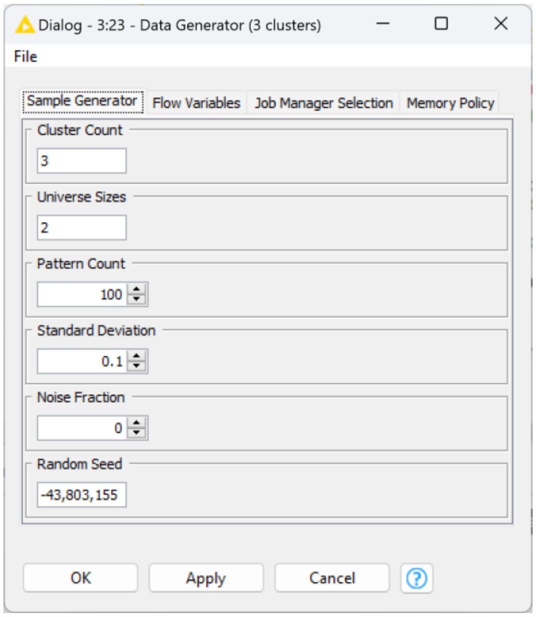
In a new workflow group named "Chapter 7", we created a new empty workflow called "Counting Loop 1". First of all, the workflow needed to read in the original data set. Since we did not have such a data set available, we artificially created it by using the "Data Generator" node.

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 the following settings:

- The number of clusters to be created
- The number of space coordinates (i.e., “Universe Sizes”)
- The total number of data rows (i.e., “Pattern Count”). An equal number of patterns (= number of data rows/number of clusters) is subsequently generated for each cluster.
- The standard deviation is used to generate data in 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 pattern generation reproducible



Cluster Count: 3
Universe Sizes: 2
Pattern Count: 100
Standard Deviation: 0.1
Noise Fraction: 0
Random Seed: -43,803,155

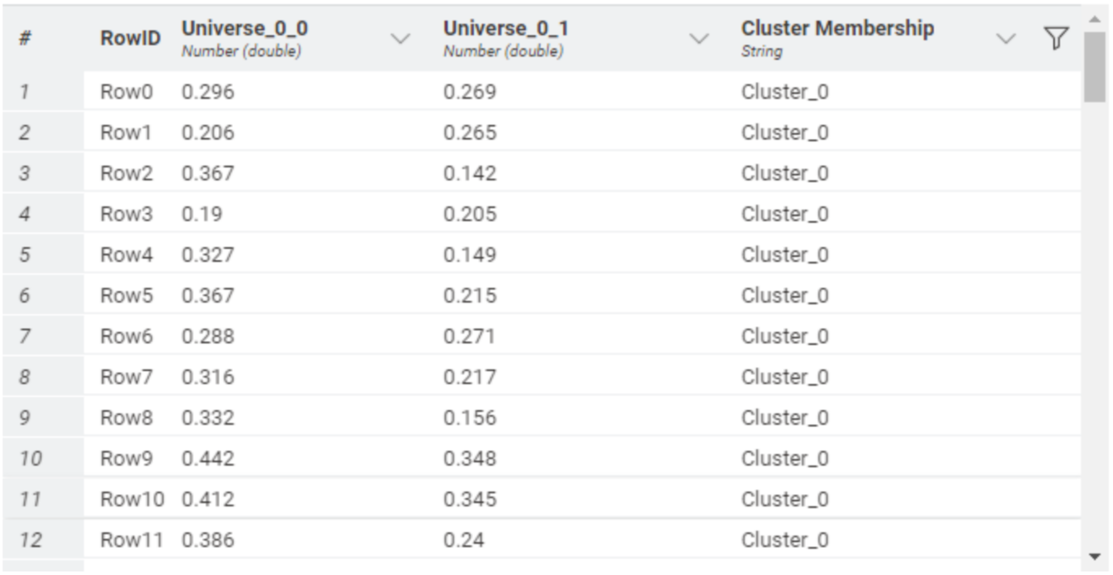
Figure 7.3. Configuration window of the Data Generator node.

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 in the space coordinates

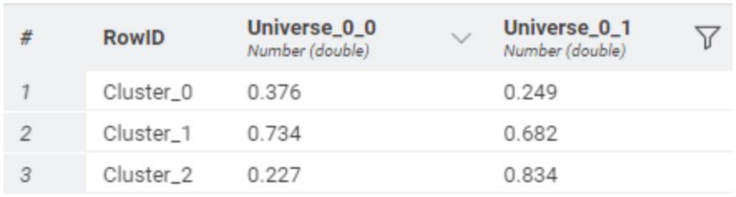
Note. Data can be generated in more than one universe, each universe with a different number of clusters and a different number of coordinates.

In case of more than one universe to be created, the cluster counts and the universe sizes are set as a sequence of comma separated numbers. For example, 2 universes with 3 and 4 clusters and 2 and 3 coordinates each are described by "Cluster Count" = "3,4" and "Universe Sizes"="2,3".



#	RowID	Universe_0_0 Number (double)	Universe_0_1 Number (double)	Cluster Membership String
1	Row0	0.296	0.269	Cluster_0
2	Row1	0.206	0.265	Cluster_0
3	Row2	0.367	0.142	Cluster_0
4	Row3	0.19	0.205	Cluster_0
5	Row4	0.327	0.149	Cluster_0
6	Row5	0.367	0.215	Cluster_0
7	Row6	0.288	0.271	Cluster_0
8	Row7	0.316	0.217	Cluster_0
9	Row8	0.332	0.156	Cluster_0
10	Row9	0.442	0.348	Cluster_0
11	Row10	0.412	0.345	Cluster_0
12	Row11	0.386	0.24	Cluster_0

Figure 7.4. Generated random data for workflow "Counting Loop 1".



#	RowID	Universe_0_0 Number (double)	Universe_0_1 Number (double)
1	Cluster_0	0.376	0.249
2	Cluster_1	0.734	0.682
3	Cluster_2	0.227	0.834

Figure 7.5. Cluster centers of the generated random data in workflow "Counting Loop 1".

We then started the new workflow "Counting Loop 1" with a "Data Generator" node to generate one single universe with 100 data rows equally distributed across 3 clusters, on a two attribute space, with a standard deviation 0.1 for each cluster, and no noise added. The node was commented with "3 clusters, 2 coordinates". The generated random data and their cluster centers are reported in the figures below. In order to visualize the data and their clusters, we added a "Color Manager" node and a "Scatter Plot" node into the workflow. 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.

The goal was to generate 4 copies of the original data set and shift each copy one unit to the right. That is, the first copied data set should be identical to the original data set inside [0,1] x [0,1]; the second copied data set should be like to the original data set but inside [1,2] x [0,1], and so on. We then needed a loop cycle to shift the data 4 times, each time to a different region.

In particular, since we already knew how many times the copy had to be made, we used a loop with a pre-defined number of iterations, i.e., 4. To move the original data set one unit to the right 4 times, the loop was to 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 starts with a "Counting Loop Start" node. The most generic loop ending node is the one named "Loop End". Notice that the Counting Loop Start node is created with no input ports and no output ports, while the Loop End node is created with just one input port and one output port. This can be changed by clicking on the plus sign which appears when you hover mouse pointer over the node.

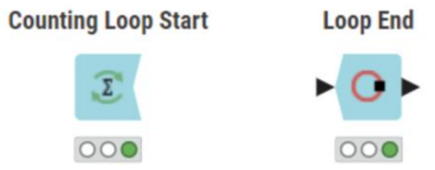


Figure 7.6. Counting Loop Start and Loop End node.

Clicking on the plus sign opens a window to add one input and one output port of a predefined type, i.e., data, database, connections, and so on. Clicking again allows you to add one more port at the input and at the output and so on, as to obtain a loop start node as complex as needed. Clicking on the added port(s) allows you to remove input/output ports. The same is true for the Loop End node.

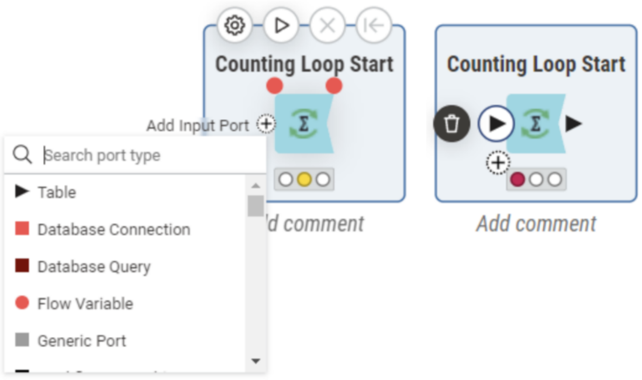


Figure 7.7. Clicking on the plus sign allows to add input ports while clicking on the added port(s) allows you to remove the port.

Let's add one more data port at the input and output of the Counting Loop Start node and let's use the Loop End node as it is after being created.

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 the number of pre-defined iterations ("Number of loops").

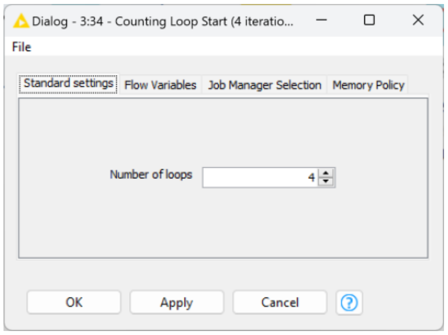
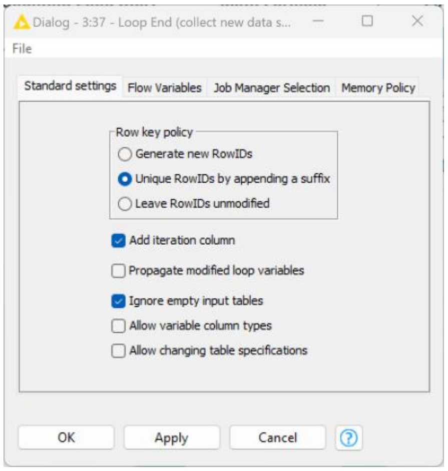

Dialog - 3:34 - Counting Loop Start (4 iteratio...
File
Standard settings | Flow Variables | Job Manager Selection | Memory Policy
Number of loops: 4
OK | Apply | Cancel

Figure 7.8. Configuration window of the Counting Loop Start node.

Loop End

The "Loop End" node closes a loop and concatenates the resulting data tables from all loop iterations. It requires the following configuration settings:

- The row key policy; that is: keeping the old RowIDs, make the old RowIDs unique through suffix, or just generate new unique RowIDs.
- The flag specifying whether to append a column to the output data set with the iteration number. Iteration numbers start with 0.
- The flags needed for possible incongruent data across iterations: empty input tables, different output column types, and different table specs in general.


Dialog - 3:37 - Loop End (collect new data s...
File
Standard settings | Flow Variables | Job Manager Selection | Memory Policy
Row key policy
- ○ Generate new RowIDs
- ● Unique RowIDs by appending a suffix
- ○ Leave RowIDs unmodified

- ☑ Add iteration column
- ☐ Propagate modified loop variables
- ☑ Ignore empty input tables
- ☐ Allow variable column types
- ☐ Allow changing table specifications

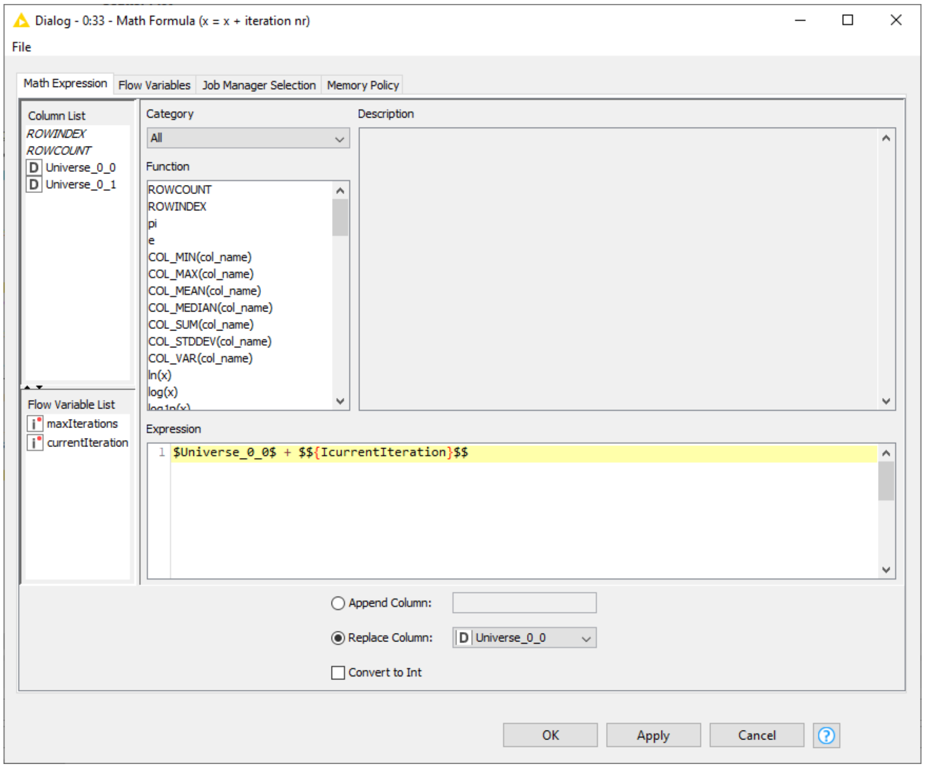
OK | Apply | Cancel

Figure 7.9. Configuration window of the Loop End node.

Note. The "Loop End" node concatenates the resulting data tables produced at each iteration. However, other loop end nodes can append the resulting data tables as additional columns, one iteration after the other.

The "Counting Loop Start" node, while starting a loop, also creates two new flow variables: one, named "currentIteration", contains the current iteration number and the other, named "maxIterations", contains the total number of iterations.

So far, we have started and closed a loop with a pre-defined number of iterations, without any operations in between. 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 values in the "currentIteration" flow variable created by the "Counting Loop Start" node. Indeed, we used a "Math Formula" node where we added the $${IcurrentIteration}$$ flow variable to the $Universe_0_0$ attribute (the x-axis) (see configuration window above).



```
$Universe_0_0$ + $${IcurrentIteration}$$
```

Replace Column: Universe_0_0

Figure 7.10. Configuration window of the Math Formula node used to implement the loop body.

In the "Flow Variable List" panel on the bottom left, you can see the two new flow variables introduced by the "Counting Loop Start" node: "currentIteration" and "maxIterations". Double-clicking one of them automatically introduces it in the formula editor with the right syntax. The same for any column listed in the panel above, named "Column List".

The math formula $Universe_0_0$ + $${IcurrentIteration}$$ overwrites the Universe_0_0 coordinate, effectively moving each copy of the data to the right on the x-axis as many units as the iteration number.

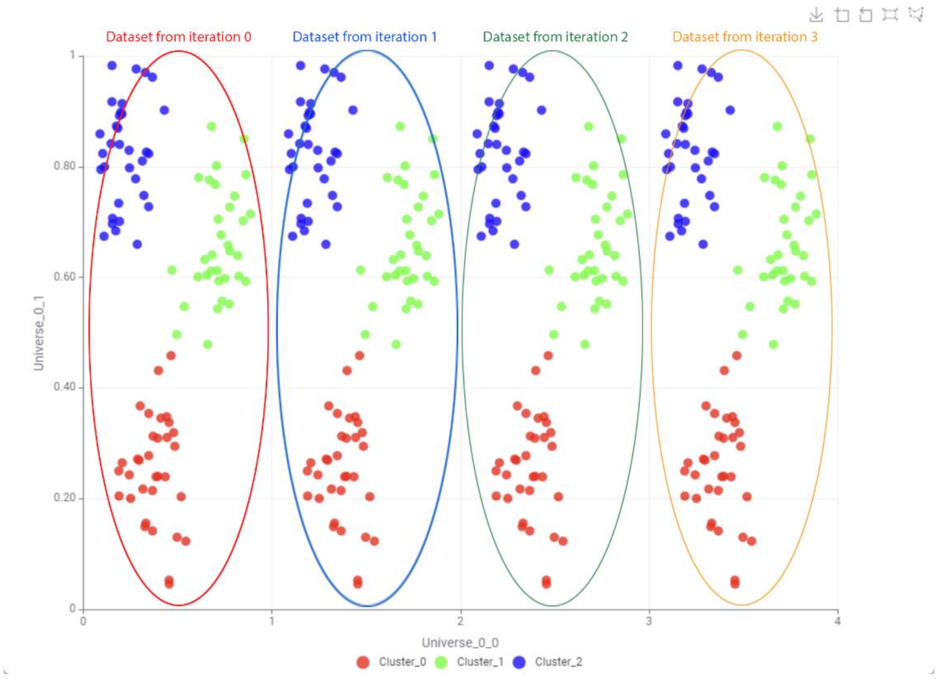
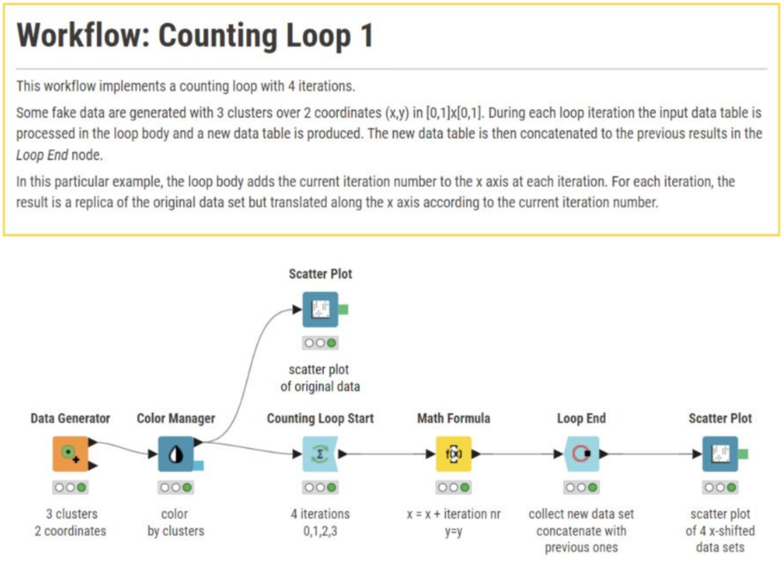


Figure 7.11. Visualization of the data set resulting from the loop implemented by in the "Counting Loop 1" workflow.

The final data set then consists of 4 identical vertical stripes of data equally spaced in [0, 4] on the x-axis. We used a "Scatter Plot" node again to verify the new data placement in the final data table. Finally, the figure below shows the "Counting Loop 1" workflow.



Workflow: Counting Loop 1

This workflow implements a counting loop with 4 iterations.

Some fake data are generated with 3 clusters over 2 coordinates (x,y) in [0,1]x[0,1]. During each loop iteration the input data table is processed in the loop body and a new data table is produced. The new data table is then concatenated to the previous results in the *Loop End* node.

In this particular example, the loop body adds the current iteration number to the x axis at each iteration. For each iteration, the result is a replica of the original data set but translated along the x axis according to the current iteration number.

Figure 7.12. The "Counting Loop 1" workflow.

7.3. Dedicated Commands for Loop Execution

Loops have a number of dedicated execution options. The list below shows most commands available from the Node Action Bar of loop end and loop start nodes and their functionalities.

Loop End nodes

- "Execute" (second from the left) runs the whole loop with all required iterations. To execute the whole loop you need to select "Execute" in the loop end node
- "Cancel" (fourth from the left) stops the loop execution and resets the loop
- "Reset" (fifth from the left) resets all nodes in the loop
- "Step Loop Execution" (third from the left) executes the next loop iteration and pauses at the end, allowing the visualization of the intermediate results in the loop body.

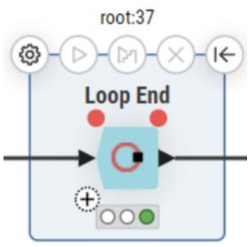


Figure 7.13. Node Action Bar of the Loop End node.

Loop Start nodes

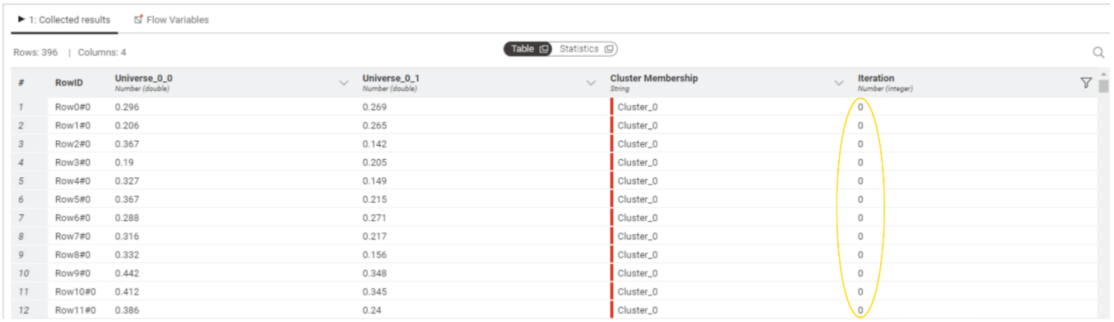
- "Execute" only executes the loop start node, i.e. it only transfers the input data into the loop
- "Cancel" stops the loop execution and resets the loop
- "Reset" resets all nodes in the loop

To reset the loop, it is enough to select the "Reset" option in the context menu of any node in the loop, including the loop start, the loop end node, and any node in the loop body. The last three options in the loop end nodes – *Pause*, *Resume*, and *Step Loop Execution* – 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 together the data tables produced at each loop iteration. A part of the output table of the “Loop End” node in the “Counting Loop 1” workflow is shown below. The last column of the data table, named “Iteration”, is created by the “Loop End” node (if specified so in the configuration window), and shows the iteration number during which each data row was created.

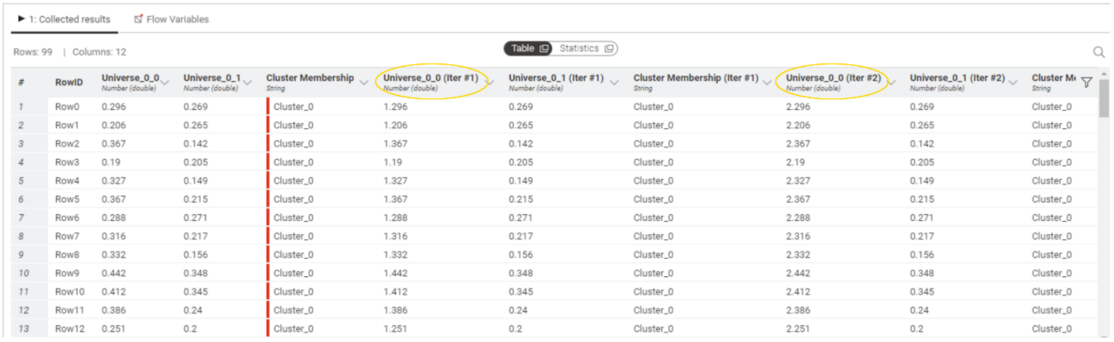


Rows: 396 | Columns: 4

#	RowID	Universe_0_0 Number (double)	Universe_0_1 Number (double)	Cluster Membership String	Iteration Number (integer)
1	Row0#0	0.296	0.269	Cluster_0	0
2	Row1#0	0.206	0.265	Cluster_0	0
3	Row2#0	0.367	0.142	Cluster_0	0
4	Row3#0	0.19	0.205	Cluster_0	0
5	Row4#0	0.327	0.149	Cluster_0	0
6	Row5#0	0.367	0.215	Cluster_0	0
7	Row6#0	0.288	0.271	Cluster_0	0
8	Row7#0	0.316	0.217	Cluster_0	0
9	Row8#0	0.332	0.156	Cluster_0	0
10	Row9#0	0.442	0.348	Cluster_0	0
11	Row10#0	0.412	0.345	Cluster_0	0
12	Row11#0	0.386	0.24	Cluster_0	0

Figure 7.14. Output table of the Loop End node in the "Counting Loop 1" workflow.

However, sometimes we might like to append the data table resulting from each iteration as new columns to the previous results. That is, at each iteration new data columns are generated and appended to the current output data table. The figure below shows an output data table where column “Cluster Membership (Iter #1)” contains the data generated from column “Cluster Membership” at the end of loop iteration number 1, and column “Universe_0_0 (Iter #2)” contains the data generated from column “Universe_0_0” at the end of loop iteration number 2, and so on.



Rows: 99 | Columns: 12

#	RowID	Universe_0_0 Number (double)	Universe_0_1 Number (double)	Cluster Membership String	Universe_0_0 (Iter #1) Number (double)	Universe_0_1 (Iter #1) Number (double)	Cluster Membership (Iter #1) String	Universe_0_0 (Iter #2) Number (double)	Universe_0_1 (Iter #2) Number (double)	Cluster Mı String
1	Row0	0.296	0.269	Cluster_0	1.296	0.269	Cluster_0	2.296	0.269	Cluster_0
2	Row1	0.206	0.265	Cluster_0	1.206	0.265	Cluster_0	2.206	0.265	Cluster_0
3	Row2	0.367	0.142	Cluster_0	1.367	0.142	Cluster_0	2.367	0.142	Cluster_0
4	Row3	0.19	0.205	Cluster_0	1.19	0.205	Cluster_0	2.19	0.205	Cluster_0
5	Row4	0.327	0.149	Cluster_0	1.327	0.149	Cluster_0	2.327	0.149	Cluster_0
6	Row5	0.367	0.215	Cluster_0	1.367	0.215	Cluster_0	2.367	0.215	Cluster_0
7	Row6	0.288	0.271	Cluster_0	1.288	0.271	Cluster_0	2.288	0.271	Cluster_0
8	Row7	0.316	0.217	Cluster_0	1.316	0.217	Cluster_0	2.316	0.217	Cluster_0
9	Row8	0.332	0.156	Cluster_0	1.332	0.156	Cluster_0	2.332	0.156	Cluster_0
10	Row9	0.442	0.348	Cluster_0	1.442	0.348	Cluster_0	2.442	0.348	Cluster_0
11	Row10	0.412	0.345	Cluster_0	1.412	0.345	Cluster_0	2.412	0.345	Cluster_0
12	Row11	0.386	0.24	Cluster_0	1.386	0.24	Cluster_0	2.386	0.24	Cluster_0
13	Row12	0.251	0.2	Cluster_0	1.251	0.2	Cluster_0	2.251	0.2	Cluster_0

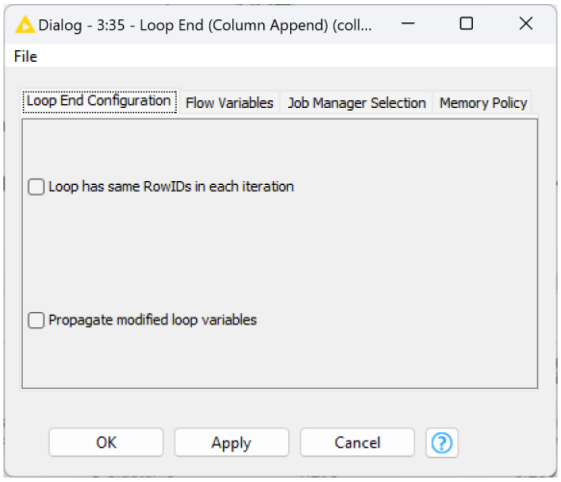
Figure 7.15. Output table of a Loop End node where the data resulting from each iteration are appended as new columns.

In order to demonstrate how such a loop output data table has been produced, we slightly modified the workflow used in the previous section 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.

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 columns are appended as a set of new columns to the output data table collected so far.

This node requires minimal configuration, just a flag on whether to keep the same RowIDs at each iteration. This flag, when enabled, speeds up the joining process of the new data columns to the table of the so far collected results.



☐ Loop has same RowIDs in each iteration

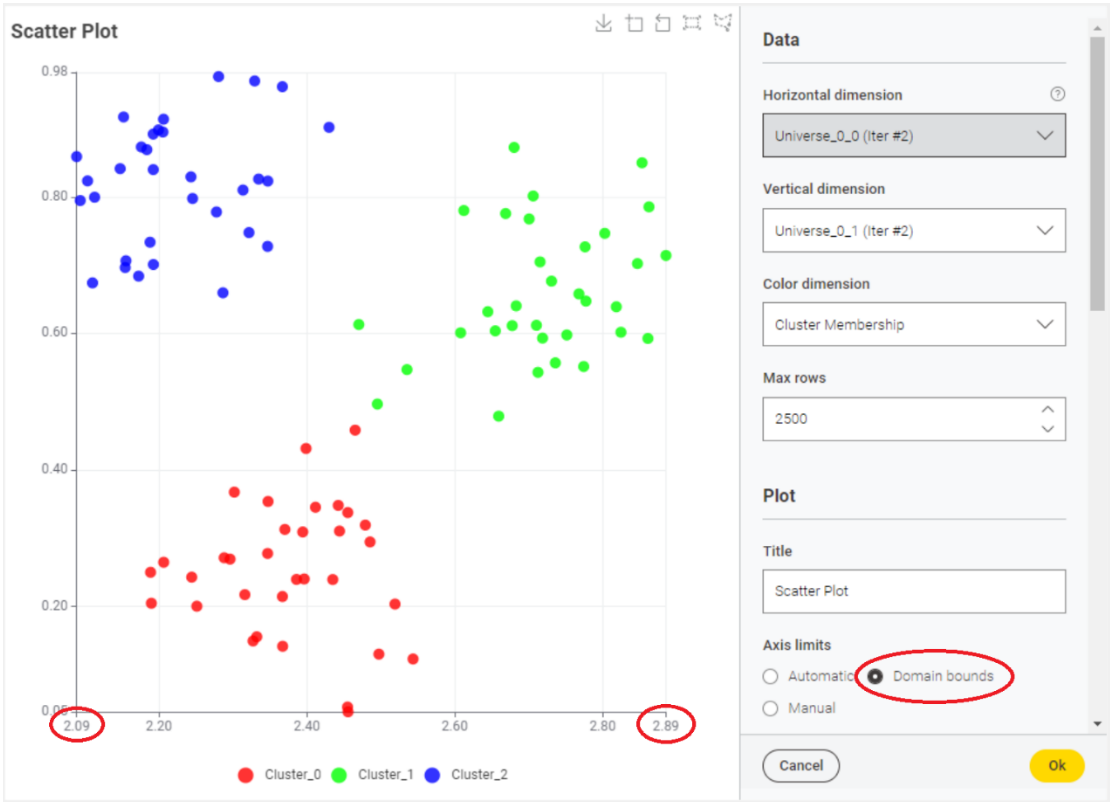
☐ Propagate modified loop variables

Figure 7.16. Configuration window of the Loop End (Column Append) node.

The “Loop End (Column Append)” node is the appropriate choice to close a loop, if the columns produced by the loop body are supposed to be appended to the output table.

By using the “Loop End (Column Append)” node to close the loop in the “Counting Loop 2” workflow, the data table in Figure 7.15 is generated after the loop execution. 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.

Notice that the “Scatter Plot” node at the end of the “Counting Loop 2” workflow can only visualize two coordinates at a time and therefore we cannot have the full visualization of the results of all loop iterations as in Figure 7.11. Thus, in the Scatter Plot below we visualized “Universe_0_0 (iter #2)” vs. “Universe_0_1 (iter #2)” with a scatter plot. There you can see that the x-range falls in [2,3] and not in [0,1] like for the original data set.



Data

Horizontal dimension: Universe_0_0 (Iter #2)

Vertical dimension: Universe_0_1 (Iter #2)

Color dimension: Cluster Membership

Max rows: 2500

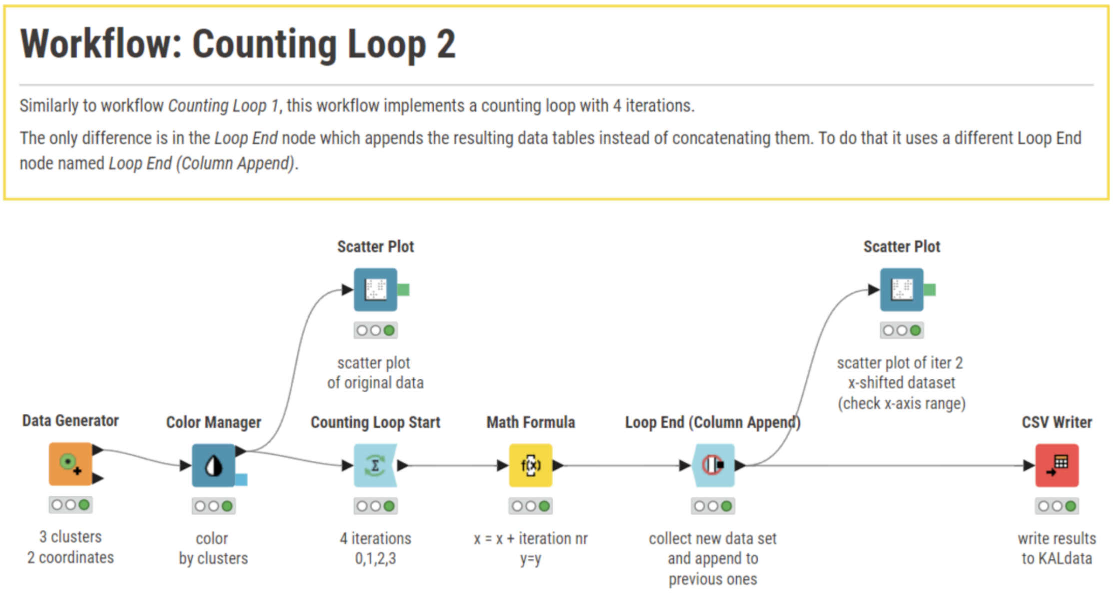
Plot

Title: Scatter Plot

Axis limits: Automatic [✓ Domain bounds] Manual

Figure 7.17. Scatter Plot of the data table generated by loop iteration #2 results.

Note. The "Loop End (Column Append)" node appends all columns produced in the output table, not only the processed ones. Therefore, the input columns, if not removed with a "Column Filter" node, will appear multiple times (as many times as many iterations) unaltered in the final output table.



Workflow: Counting Loop 2

Similarly to workflow *Counting Loop 1*, this workflow implements a counting loop with 4 iterations.

The only difference is in the *Loop End* node which appends the resulting data tables instead of concatenating them. To do that it uses a different Loop End node named *Loop End (Column Append)*.

Figure 7.18. Workflow "Counting Loop 2".

7.5. Loop on a List of Columns

Let's suppose now that we want to create two more shifted data sets but shifted differently on the x-axis and on the y-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 custom x- and y-shifts to the data set. We could also use a loop that iterates on a list of selected columns.

A loop that iterates on a list of columns starts with a "Column List Loop Start" node. This kind of loop iterates across all columns, one by one, and runs the group of operations specified in the body loop for each column. The loop then can end with a "Loop End (Column Append)" node or just with a "Loop End" node, depending on how we would like to collect the results in our final data table.

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 reads the data set produced by the "Counting Loop 2" workflow, including the original data set and 3 copies of the same data set differently shifted across the x-axis. The goal here is to multiply the x-axis of each iteration set by a different numerical factor. How do we identify each iteration set? If you remember, the "Loop End (Column Append)" node was renaming columns with duplicate names by appending a "(Iter #n)" suffix to the column name.

Thus, data set columns from iteration n are identified by the suffix "(Iter #n)" in the column names.

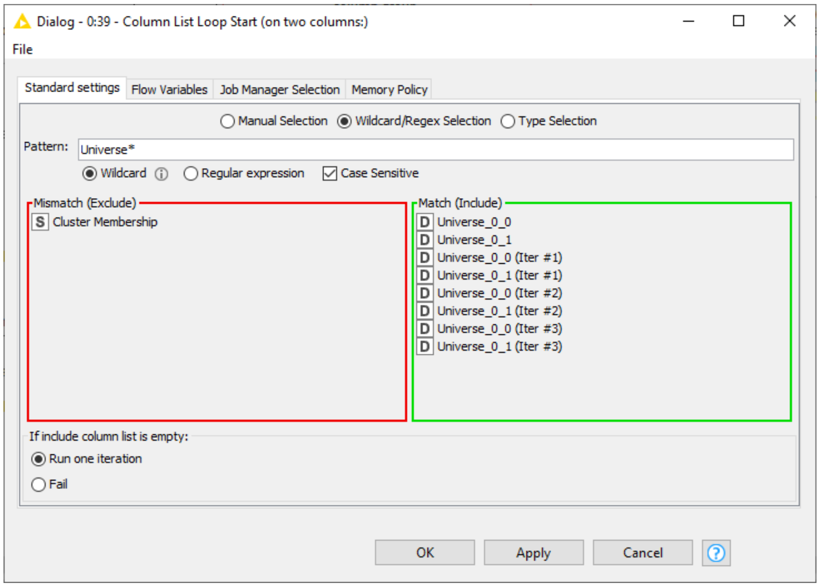
The goal is to multiply the x ("Universe_0_0") and y ("Universe_0_1") value of each iteration set by the iteration number in the column name plus one. So, "Universe_0_0 (Iter #2)" and "Universe_0_1 (Iter #2)" should both be multiplied by 3, while "Universe_0_0" and "Universe_0_1" by 1 and so on. To define the correct multiplying factor, we need to work on each column name String and extract the number in the "Iter #" suffix. Once we have it, we need to multiply all values in that column by the associated numerical factor (= iteration number in column name +1).

Column List Loop Start

The "Column List Loop Start" node iterates over a list of columns in the input table, one by one.

The columns on which to iterate are selected in the configuration window by means of an "Exclude/Include" framework. Columns can be included manually, through wildcard and regex selection, or based on type.

The "Column List Loop Start" node separates the columns in the input data table into iteration columns ("Include" panel) and non-iteration columns ("Exclude" panel).


Dialog - 0:39 - Column List Loop Start (on two columns:)

Manual Selection [✓ Wildcard/Regex Selection] Type Selection

Pattern: Universe*

[✓ Wildcard] Regular expression [✓ Case Sensitive]

Mismatch (Exclude):
S Cluster Membership

Match (Include):
D Universe_0_0
D Universe_0_1
D Universe_0_0 (Iter #1)
D Universe_0_1 (Iter #1)
D Universe_0_0 (Iter #2)
D Universe_0_1 (Iter #2)
D Universe_0_0 (Iter #3)
D Universe_0_1 (Iter #3)

If include column list is empty:
[✓ Run one iteration] Fail

Figure 7.19. Configuration window of the Column List Loop Start node.

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.

Additionally, a strategy can be defined in case the iteration column is empty.

We used a “Column List Loop Start” node to iterate on the different columns of the input data set. The configuration window includes all “Universe*” columns in the iteration group and excludes only the column named “Cluster_Membership”. Such inclusion could be performed manually (those are not that many columns) or using the wild card-based selection. In the case of the wildcard-based selection, “Universe*” should be used as pattern with wildcard to match the column names. After execution, the “Column List Loop Start” node produces two flow variables: “currentIteration” with the current iteration number and “currentColumnName” with the name of the current column in the loop.

Our goal included the structure of the final table to be identical to the structure of the input table, only with different values in each column. In order to append the data columns into the same position as in the original table, the loop was closed with a “Loop End (Column Append)” node.

The loop was built with two parallel branches.

- One branch defines the multiplying factor. That is, it checks the flow variable “currentColumnName” for the presence of “#”; it then extracts the character following the position of “#”; if “#” was not found, default multiplying factor is 0, otherwise is the number following “#”; the last step is to add +1 to the multiplying factor. All of this is obtained with three “String Manipulation” nodes, one “Math Formula (Variable)” node, and one “Rule Engine Variable” node. This whole part has been grouped under a metanode named “define factor”.
- One branch eliminates “Cluster_Membership” column from the iteration set, renames the current column to an anonymous “temp_column”, applies the multiplying factor from the parallel metanode, and changes the column name back to its original one.

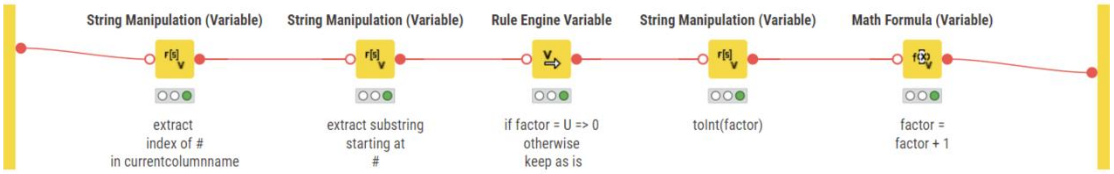
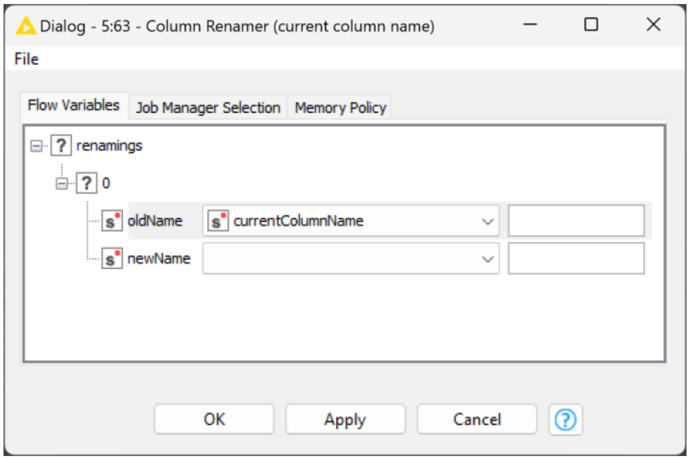

String Manipulation (Variable)
extract index of # in currentcolumnname
String Manipulation (Variable)
extract substring starting at #
Rule Engine Variable
if factor = U => 0 otherwise keep as is
String Manipulation (Variable)
toInt(factor)
Math Formula (Variable)
factor = factor + 1

Figure 7.20. Content of metanode “define factor”. “String Manipulation (Variable)” node detects “#” in column name via the “indexOf()” function, it then extracts the number following “#” if any; if no number, “factor” is 0 otherwise is the detected number; finally the “Math Formula (Variable)” node adds +1 to flow variable “factor”.

Renaming the current iteration column to "temp_column" is necessary to allow the "Math Formula" node to work on any input column independently of its name. The renaming process is carried out through a "Column Rename" node using the value in the flow variable "currentColumnName" to overwrite the "old_column_name" setting. The inverse renaming process is carried out again through a "Column Rename" node using the value in the flow variable "currentColumnName" to overwrite this time "new_column_name" in the configuration settings.



renamings
0
oldName currentColumnName
newName

Figure 7. 21. Value of flow variable "currentColumnName" overwrites configuration setting "new_column_name" in the last "Column Rename" node.

After the column is renamed, it is presented to the "Loop End (Column Append)" closing node, to be appended to the other already processed data columns. Finally, a "Joiner" node rescues the "Cluster_Membership" data column from the original data table. This will be used for coloring in the "Scatter Plot" node.

The two following Scatter Plots use the newly generated columns "* (Iter #2)" and "*(Iter #3)" multiplied by different factors. The shape looks similar but the x- and y-range are different.

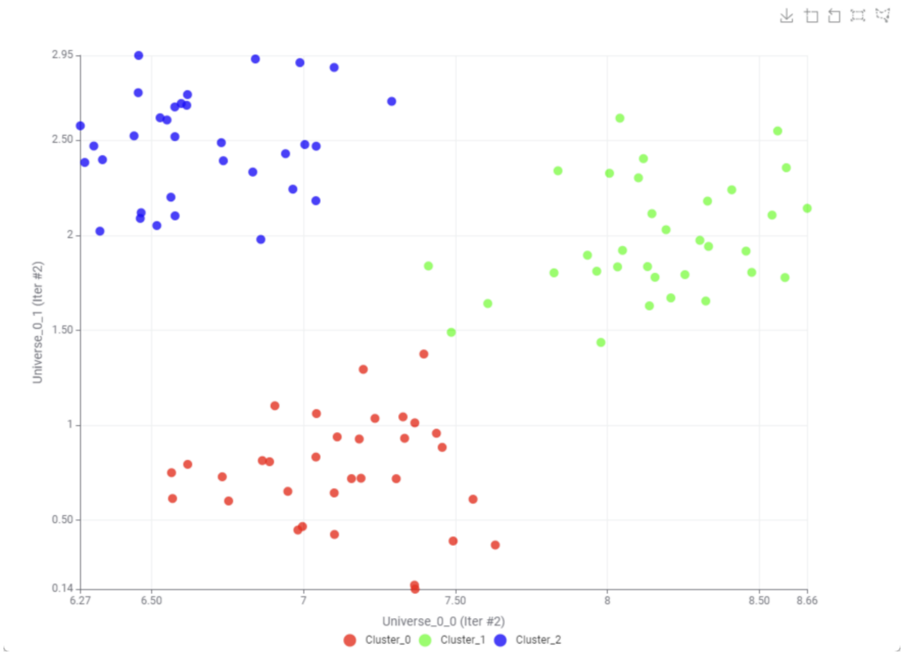


Figure 7.22. Data points from altered "Iter #2" columns. Notice the range of the axis and compare with plot below.

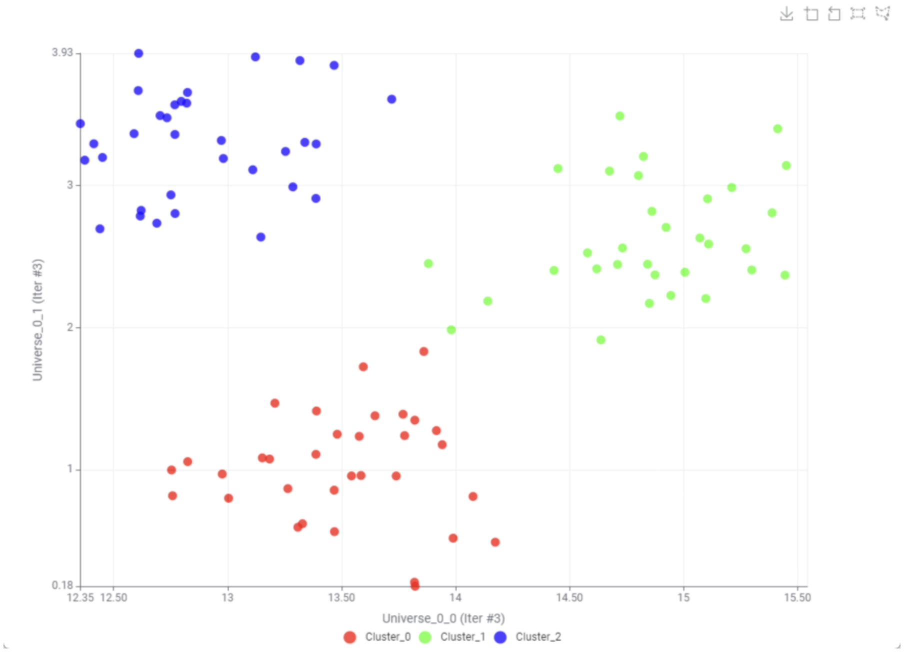
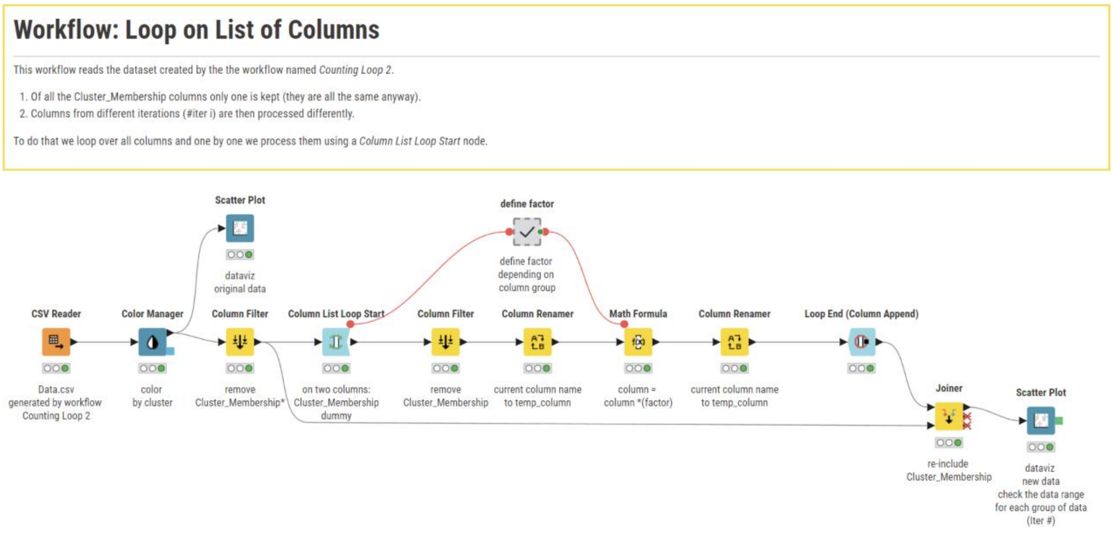


Figure 7.23. Data points from altered "Iter #3" columns. Notice the range of the axis and compare with plot above.

7.6. Loop on a List of Values

Let’s now use the “sales.csv” file from the KALdata folder that we have used in some previous sections. The goal of this section is to calculate some statistical measures about the sales in each country.

- The statistical measures for both numerical and nominal columns are calculated through a “Statistics” node.
- Data about each country are isolated through a “Row Filter” node.
- The list of unique countries included in the data column “country” is extracted with a “GroupBy” node using the “country” column as the group column and no aggregation columns. A “GroupBy” node with only one group column and no aggregation columns produces the list of unique values available in the group column.



Workflow: Loop on List of Columns

This workflow reads the dataset created by the the workflow named *Counting Loop 2*.

1. Of all the Cluster_Membership columns only one is kept (they are all the same anyway).
2. Columns from different iterations (#iter i) are then processed differently.

To do that we loop over all columns and one by one we process them using a *Column List Loop Start* node.

Figure 7.24. Workflow "Loop on List of Columns".

What we are still missing is a loop that allows us to iterate over all countries in the list and to calculate the sales statistics for each country at each iteration.

In the `“Workflow Control” -> “Loop Support”` category, the node “Table Row To Variable Loop Start” starts exactly the kind of loop that iterates across a list of values.

Table Row To Variable Loop Start

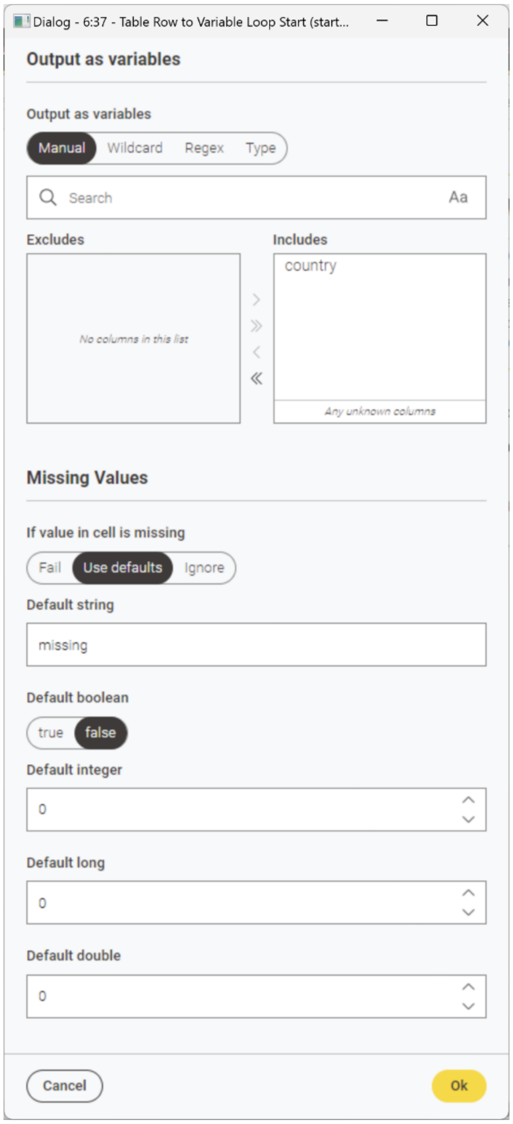
The “Table Row To Variable Loop Start” starts a loop that iterates on a list of values.

This node takes a data table at the input port and, row by row, transforms the data cells into flow variables with the same name as their column name. At the next iteration, the flow variables update their value with the next value from the corresponding column.

The Table Row To Variable Loop Start node requires the following settings:

- a handling strategy if the field to be transformed into a variable has a missing value. Such strategy can be drastic, such as:
    - fail the node execution or
    - omit the creation of the variable

  or can be more tolerant and keep the node running and still create the desired flow variable. It just fills it with a fictitious value, such as:
    - a fixed string value
    - or a number fixed value
- the selection of the input fields to be transformed into flow variables. This is achieved as usual via an exclude/include frame.


Dialog - 6:37 - Table Row to Variable Loop Start (start...

Output as variables

Output as variables: [✓ Manual] Wildcard Regex Type

Excludes: No columns in this list

Includes: country

Any unknown columns

Missing Values

If value in cell is missing: Fail [✓ Use defaults] Ignore

Default string: missing

Default boolean: true [✓ false]

Default integer: 0

Default long: 0

Default double: 0

Figure 7.25. Configuration window of the Table Row to Variable Loop Start node.

The loop performs as many iterations as the number of rows in the input table at the loop start node.

In a new workflow named “Loop on List of Values”, we started by reading the data from the “sales.csv” file with a “File Reader” node. After that, we isolated the data column “country” with a “Column Filter” node and we built the list of countries with a “GroupBy” node using “country” as group column and no aggregation columns. The output data table of the “GroupBy” node contains just one column named “country” with a list of unique country values. This is the input data table of the “Table Row To Variable Loop Start” node.

If we execute the “Table Row To Variable Loop Start” node and look at the output port view (right-click the node to open its context menu and select the option “Variable Connection”), we can see that there is a new flow 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 with two input and two output ports.

Note. Remember that you can add input and output ports to the “Loop End” node by clicking the three dots in the lower left corner of the node and selecting “Add Collector Port”.

If we now run “Step Loop Execution” from the context menu of the “Loop End” node, at the second iteration we see that the flow variable named “country” takes the next value in the input list: “China”, and so on.

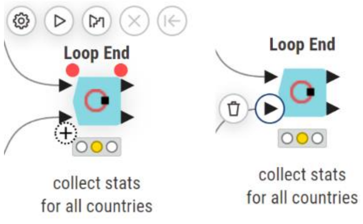
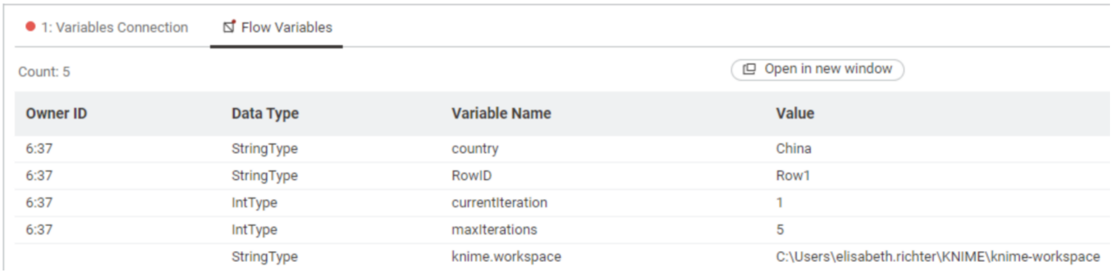

Loop End
collect stats for all countries
Loop End
collect stats for all countries

Figure 7.26. Adding one more port to the Loop End node.

We still need to build the body of the loop. The goal was to isolate the data rows for each country, to calculate the statistical measures through a “Statistics” node, to add the country to the measures, and to export the final results. After the “Table Row To Variable Loop Start” node and before the “Loop End” node, we introduced a “Row Filter” node to select only the data rows for the current value of the flow variable “country”; a “Statistics” node to calculate mean and variance and other statistical measures on the selected data group; and a “Variable to Table Column” node to add the current value of flow variable “country”.

The execution of a loop on a list of values can become quite slow, if the data set to loop on is



Count: 5

Owner ID	Data Type	Variable Name	Value
6:37	StringType	country	China
6:37	StringType	RowID	Row1
6:37	IntType	currentIteration	1
6:37	IntType	maxIterations	5
	StringType	knime.workspace	C:\Users\elisabeth.richter\KNIME\knime-workspace

Figure 7.27. Output port view from the context menu of the Table Row To Variable Loop Start node.

very large and the list of values is quite long. One way to make the workflow execution faster is to cache the data group. There is a node that is used to cache data: the “Cache” node.

Cache

The “Cache” node caches all input data onto disk.

This is particularly useful when the preceding node performs a column transformation. In fact, many column transformation nodes only hide the unwanted part of the input data, showing what the result should be but really keeping in memory the whole input data table. For example, a “Filter Column” node just hides the unwanted columns from the output.

A “Cache” node instead caches only the visible data from the input table. Therefore, a “Cache” node after a “Column Filter” node loads only the visible input data and this 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.

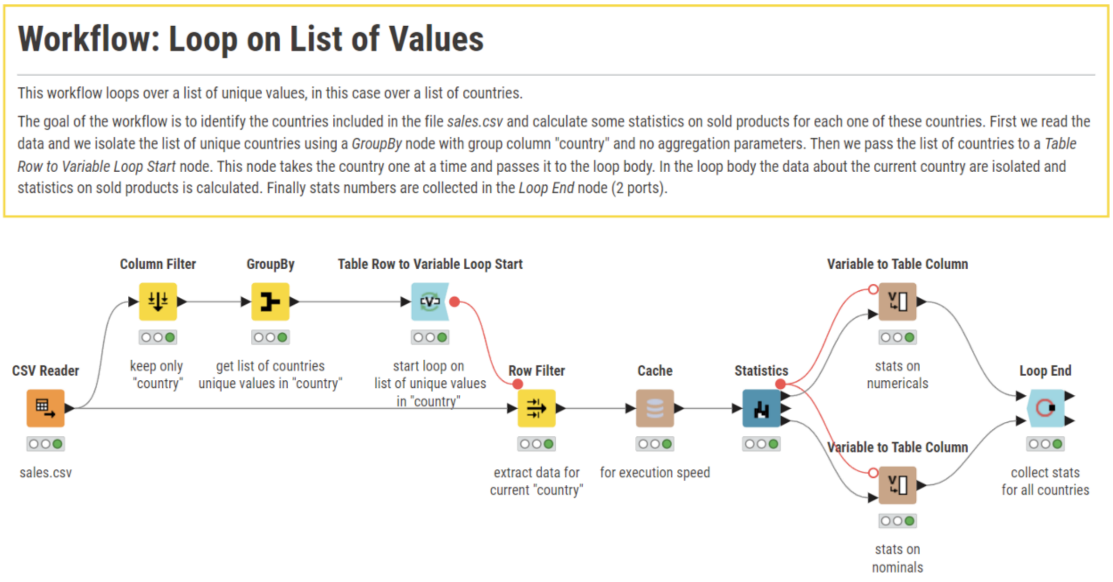
A “Cache” node was inserted inside the loop after the “Row Filter” node to speed the loop execution at each iteration.

The “Statistics” node has three output ports:

- the top port for the statistics on numerical columns,
- the lowest port for the statistics on nominal columns,
- the port in the middle for the histogram tables.

Our goal is to collect the statistical measures for both nominal and numerical columns. We need then a loop end node capable to collect two data flows. The “Loop End” node can do exactly that. It has two input ports and two output ports. It collects the loop results from two different branches at its input ports and produces the concatenated data tables at the output ports.

The final version of the “Loop on List of Values” workflow, named “filter by country”, is shown in the figure below.



Workflow: Loop on List of Values

This workflow loops over a list of unique values, in this case over a list of countries.

The goal of the workflow is to identify the countries included in the file *sales.csv* and calculate some statistics on sold products for each one of these countries. First we read the data and we isolate the list of unique countries using a *GroupBy* node with group column "country" and no aggregation parameters. Then we pass the list of countries to a *Table Row to Variable Loop Start* node. This node takes the country one at a time and passes it to the loop body. In the loop body the data about the current country are isolated and statistics on sold products is calculated. Finally stats numbers are collected in the *Loop End* node (2 ports).

Figure 7.28. Workflow "Loop on List of Values".

7.7. Loop on Data Groups and Data Chunks

In the previous section we have isolated the list of countries and calculated a few sales statistics for each one of them. Now, would not that be much easier to loop on the groups directly? The "Group Loop Start" node identifies unique values in a selected data column, extracts the associated groups of data rows in the input data table, and loops over them. This makes the task described in the previous section even easier!

Instead of having a "Table Row to Variable Loop Start" node followed by a "Row Filter" node and a "Cache" node like in the previous section, we just introduced a "Group Loop Start" node. The "Group Loop Start" node was set to loop on unique values in the "country" column, i.e., on the groups of data defined by each country value.

The loop body was then again made by the "Statistics" node and the two "Variable to Table Column" nodes to append the current country name to the resulting data tables. The loop was then closed with a "Loop End" node with two ports.

Group Loop Start

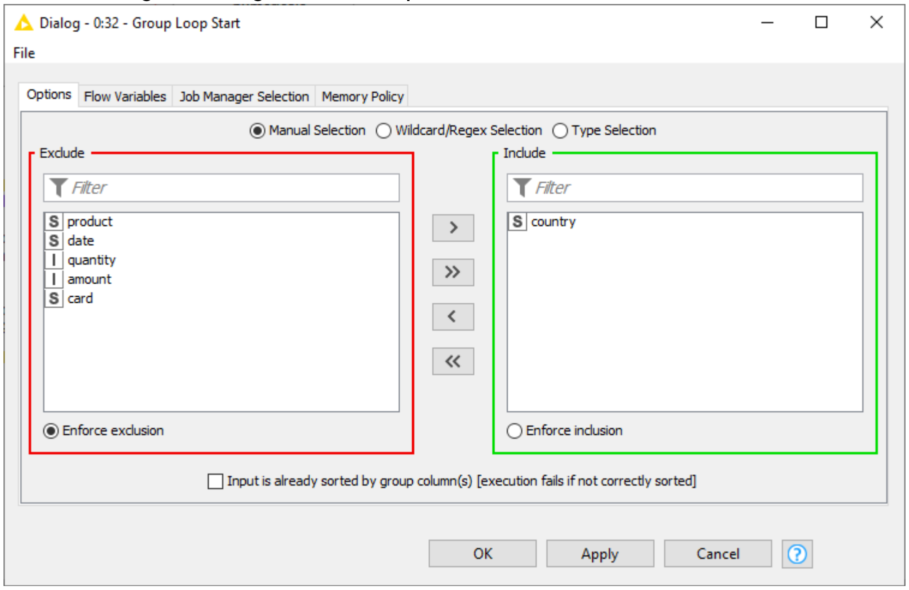
The "Group Loop Start" node starts a group loop. It identifies a list of unique values in one or more of the input data columns, detects the groups in the input data table associated with each one of these values, and iterates over those groups. The configuration window of the "Group Loop Start" node requires the data column(s) from whose values to build the data groups.

The column selection is obtained by means of an Exclude/Include framework. Columns can be included manually, based on a wildcard or regex expression, or based on type. Data columns in the "Include" frame will be used to group the data rows; those in the "Exclude" frame will not. In case of manual selection, "Enforce Inclusion" and "Enforce Exclusion" add possible new data columns to the "Exclude" frame or to the "Include" frame respectively.

Since the "Group Loop Start" node is equivalent to the "Table Row to Variable Loop Start" followed by a "Row Filter" node, the output data tables produced by the "Loop End" node should be identical in this new workflow – named "Loop on Groups of Data" – as in the workflow generated in the previous section – named "Loop on List of Values".

The "Workflow Control" -> "Loop Support" category includes a type of loop similar to the loop on groups: the loop on chunks. This loop divides the input data table into smaller pieces (chunks) and iterates from the first piece to the last one till the end of the data table has been reached. So, for example, a data set with 99 data rows can have 3 chunks of 33 data rows each.

The biggest advantage in using a chunk loop is not in the number of iterations on the input data



[✓ Manual Selection] Wildcard/Regex Selection Type Selection

Exclude: product (S), date (S), quantity (I), amount (I), card (S)

Include: country (S)

[✓ Enforce exclusion] Enforce inclusion

☐ Input is already sorted by group column(s) [execution fails if not correctly sorted]

Figure 7.29. Configuration window of the Group Loop Start node.

set, since it only covers the input data set once, but the speed. Indeed, processing smaller chunks of data at a time speeds up execution. A second advantage in using a chunk loop is that different processing can be applied to different chunks of data. A chunk loop starts with a "Chunk Loop Start" node and ends with any of the loop end nodes.

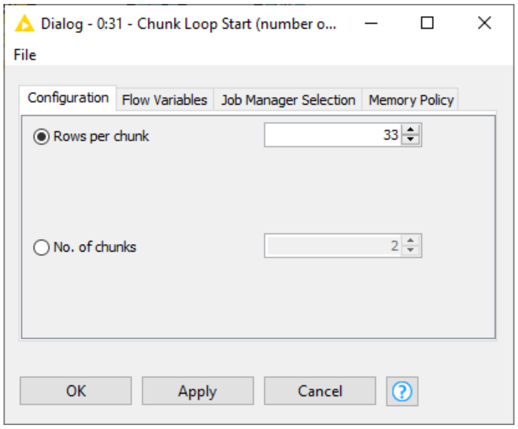
Chunk Loop Start

The “Chunk Loop Start” node starts a chunk loop.

It divides the input data table into a number of pieces (chunks) and iterates over those chunks.

The configuration window of the “Chunk Loop Start” node requires:

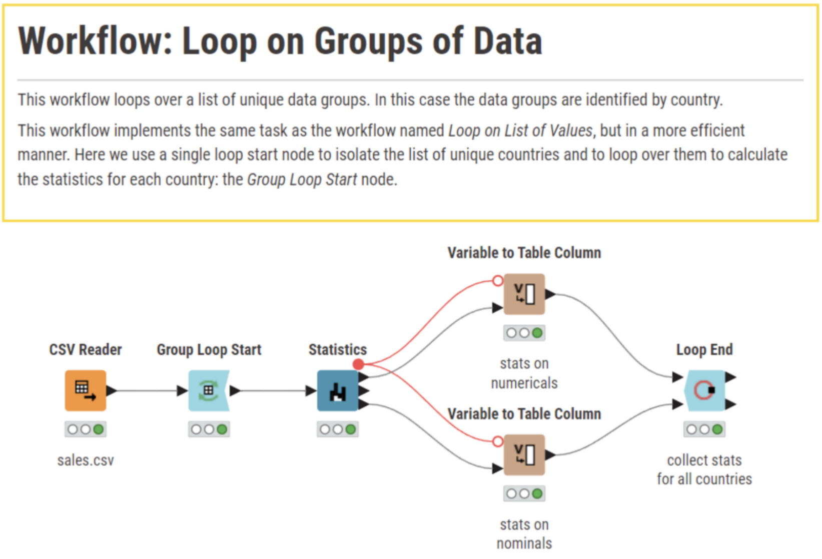
- Either the (maximum) number of data rows in each chunk
- Or the number of chunks



Rows per chunk [✓] 33
No. of chunks 2

Figure 7.30. Configuration window of the Chunk Loop Start node.

We created a new workflow named “Chunk


Workflow: Loop on Groups of Data

This workflow loops over a list of unique data groups. In this case the data groups are identified by country.

This workflow implements the same task as the workflow named *Loop on List of Values*, but in a more efficient manner. Here we use a single loop start node to isolate the list of unique countries and to loop over them to calculate the statistics for each country: the *Group Loop Start* node.

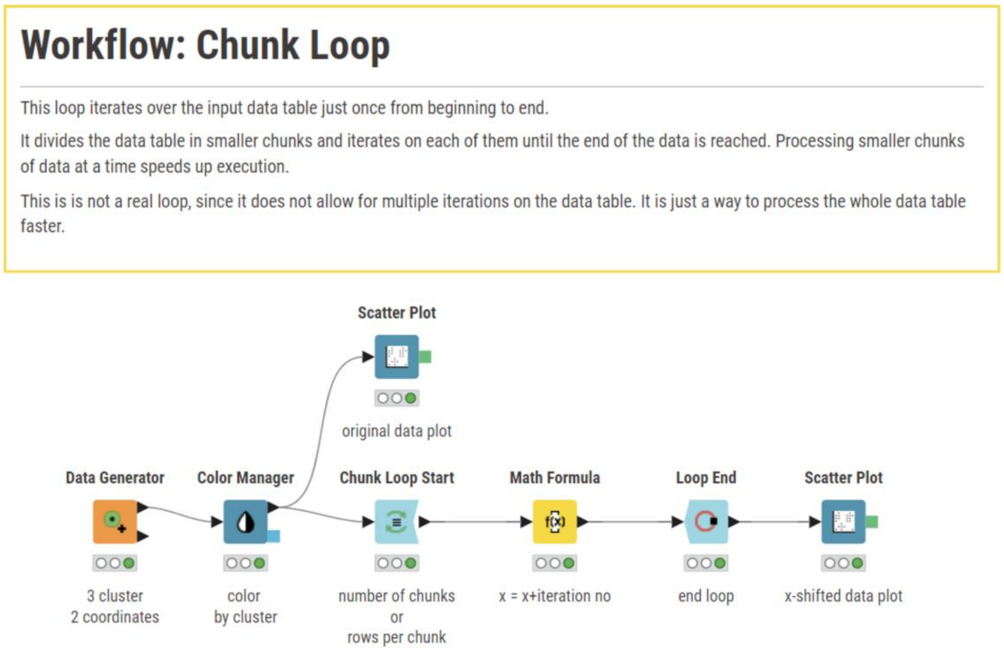
Figure 7.31. Workflow "Loop on Groups of Data”. This workflow performs the same task as the workflow named “Loop on List of Values”.

Loop”. This workflow uses the same data generated by the “Data Generator” node named “3 clusters, 2 coordinates" described in section 7.2. We set the “Data Generator” node to produce 99 data rows distributed across 3 clusters along 2 coordinates. The original data set then contains 33 data rows for cluster 1, 33 data rows for cluster 2, and 33 data rows for cluster 3 and defining chunks of 33 rows fits the clusters size perfectly.

We implemented a chunk loop on chunks of 33 data rows. For each chunk, we shift the data along the x-axis of as many units as the iteration number. This translated into:

- A “Chunk Loop Start” node to start the chunk loop, with “Rows per chunk” set to 33;
- A “Math Formula” node with “$Universe_0_0$ + $${IcurrentIteration}$$” where $Universe_0_0$ is a data column and {IcurrentIteration}$$ the loop flow variable
- A generic “Loop End” node to close the loop and concatenate the results of each iteration.
- We then inserted a “Scatter Plot” node to visualize the final data table.

The “Chunk Loop” workflow as well as the dataset before and after the progressive shifting is shown below.



Workflow: Chunk Loop

This loop iterates over the input data table just once from beginning to end.

It divides the data table in smaller chunks and iterates on each of them until the end of the data is reached. Processing smaller chunks of data at a time speeds up execution.

This is is not a real loop, since it does not allow for multiple iterations on the data table. It is just a way to process the whole data table faster.

Figure 7.32. Workflow "Chunk Loop".

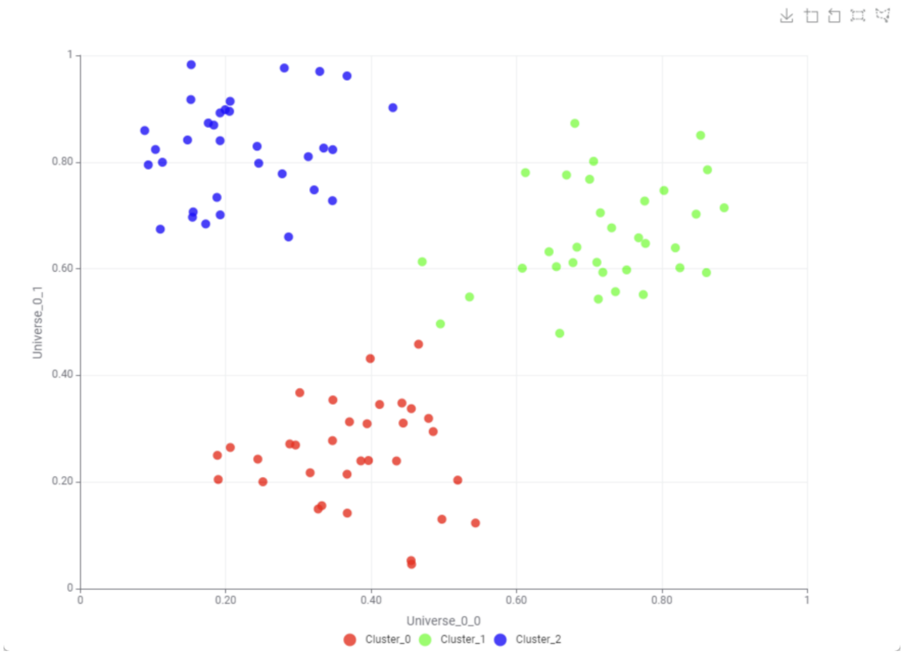


Figure 7.33. Original dataset.

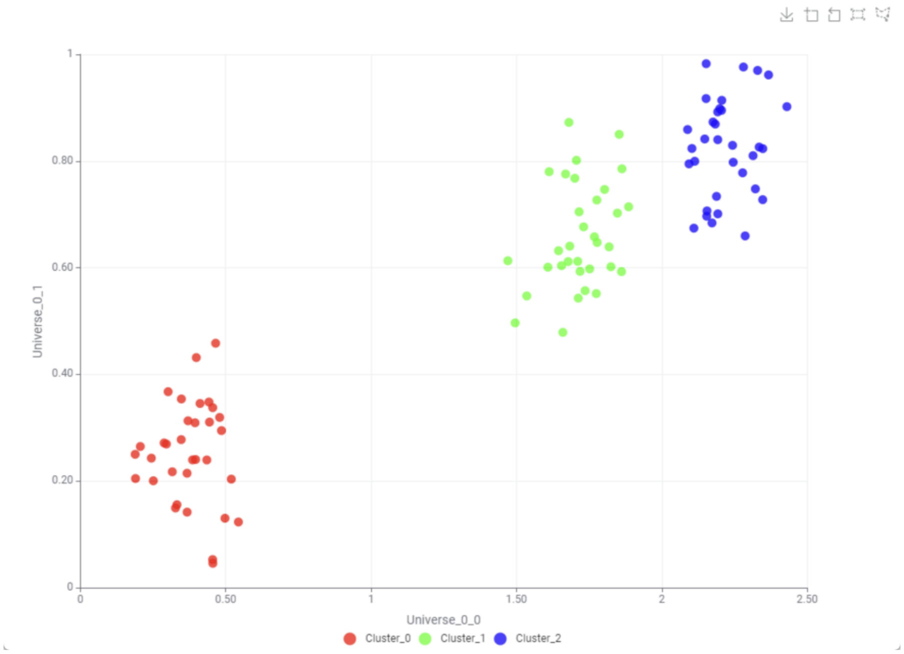


Figure 7.34. Progressively x-Shifted Data Set after executing the workflow "Chunk Loop".

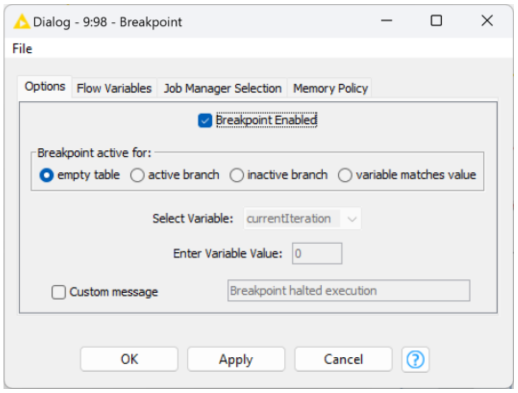
Another node in the “Loops” category that might be worth mentioning is the “Breakpoint” node. The “Breakpoint” node can be inserted in a loop body and, while it does not process data, it might give some control on the loop execution.

Breakpoint

The “Breakpoint” node prevents the loop execution when the input table fulfills a user-specified condition, like for example the input data table is empty or a variable matches some set value.

The configuration window then needs:

- The flag enabling the breakpoint
- The condition for which the breakpoint has to become active and 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.


Dialog - 9:98 - Breakpoint

File

Options | Flow Variables | Job Manager Selection | Memory Policy

☑ Breakpoint Enabled

Breakpoint active for: [✓ empty table] ○ active branch ○ inactive branch ○ variable matches value

Select Variable: currentIteration

Enter Variable Value: 0

☐ Custom message: Breakpoint halted execution

Figure 7.35. Configuration window of the Chunk Loop Start node.

7.8. Keep Looping till a Condition is verified

In the first sections of this chapter, 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.

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

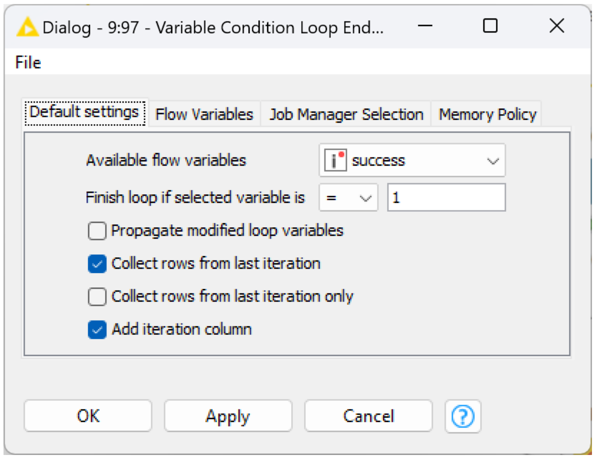
The “Generic Loop Start” node, located in the “Workflow Control” -> “Loop Support” category, starts a generic loop on all data rows without any previous assumptions and, because of that, it needs no configuration settings.

The “Generic Loop Start” node starts a generic loop on all data rows without any assumptions. The “Variable Condition Loop End” implements the stop condition and checks it at the end of each iteration. If the loop condition is verified the “Variable Condition Loop End” ends the loop and makes the results, collected till now, available at the output port. Otherwise, it passes the control back to the loop start node and proceeds with the next iteration.

Variable Condition Loop End

The “Variable Condition Loop End” implements a condition to terminate a loop. At the end of each 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:



Available flow variables: success
Finish loop if selected variable is: = 1
☐ Propagate modified loop variables
✓ Collect rows from last iteration
☐ Collect rows from last iteration only
✓ Add iteration column

Figure 7.36. Configuration window of the Variable Condition Loop end node.

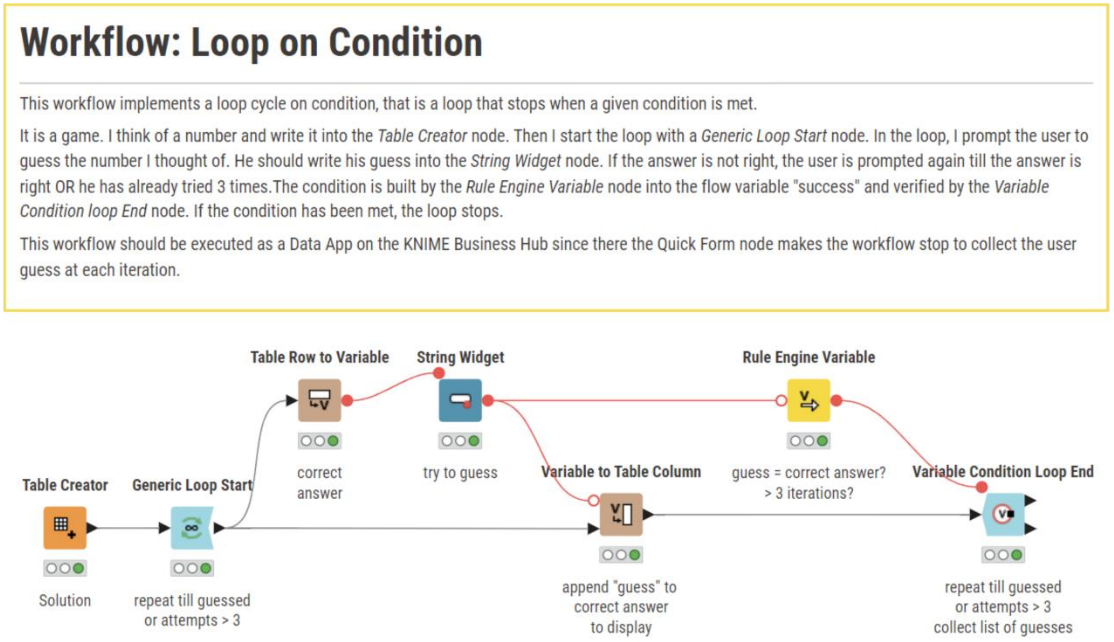
- the flow variable on which the condition is evaluated
- the condition (i.e., comparison operator + value)
- a few flags to exclude or only include the last iteration results and/or to append the iteration column

A second output port shows the progressive values of the flow variable used to implement the loop condition throughout the loop iterations.

Note. The 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 end node is reached. It can of course be updated at each iteration.

In order to show how such a loop can be implemented, we created a new workflow under the "Chapter7" workflow group, and we named it "Loop on Condition". This workflow implements a game. I think of a number and write it into the "Table Creator" node. Then I prompt the user to guess it. The user can try a new guess three times, after which the game just ends either way.

This workflow starts with a *Table Creator* node containing my mystery number. Then a *Generic Loop Start* node starts the guessing loop. In the loop the user is prompted to guess the number through a *String Widget* node. Guess and number are compared in a *Rule Engine Variable* node that produces 1 if the guess was correct OR the number of iterations was > 2, 0 otherwise. The result is stored in a flow variable named "success". The loop should terminate when success = 1. This condition is checked in the "Variable Condition loop End" node that closes the loop and the workflow.



Workflow: Loop on Condition

This workflow implements a loop cycle on condition, that is a loop that stops when a given condition is met.

It is a game. I think of a number and write it into the *Table Creator* node. Then I start the loop with a *Generic Loop Start* node. In the loop, I prompt the user to guess the number I thought of. He should write his guess into the *String Widget* node. If the answer is not right, the user is prompted again till the answer is right OR he has already tried 3 times.The condition is built by the *Rule Engine Variable* node into the flow variable "success" and verified by the *Variable Condition loop End* node. If the condition has been met, the loop stops.

This workflow should be executed as a Data App on the KNIME Business Hub since there the Quick Form node makes the workflow stop to collect the user guess at each iteration.

Table Creator
Solution
Generic Loop Start
repeat till guessed or attempts > 3
Table Row to Variable
correct answer
String Widget
try to guess
Variable to Table Column
append "guess" to correct answer to display
Rule Engine Variable
guess = correct answer? > 3 iterations?
Variable Condition Loop End
repeat till guessed or attempts > 3 collect list of guesses

Figure 7.37. Workflow "Loop on 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.

Both the *Variable Condition Loop End* node and the *Generic Loop Start* node can add and remove ports through their three dots placed in the lower left corner.

7.9. Recursive Loop

All loops described in the previous sections do not change the content of the loop input data table across iterations. At each new iteration the input data table is unchanged. Each iteration might extract a different part of the input data table, but the data table itself is always the same. For example, the counting loop loops a number N of times on the same input data table. That is, all so far seen loops do not have memory.

In the workflow named “Counting Loop 1” and implemented in section 7.2 the iteration number is added to the initial x values. In this section, we want to re-elaborate that workflow as to increment the x values in the input data one more unit at each new iteration. Therefore, we need a loop that can pass the incremented x-values back to the loop start: that is we need a loop with memory. There is only one such a loop in the KNIME Analytics Platform: the recursive loop.

A recursive loop starts with a “Recursive Loop Start” node and ends with a “Recursive Loop End” node. The particularity of this loop lies in the loop end node. The “Recursive Loop End” node has two input ports: one to collect the iteration output data table (like all other loop end nodes) and one to pass the processed data back to the loop start node as the new input data for the next iteration. The recursive loop node pair enables the passing of a data table from the “Recursive Loop End” node back to the “Recursive Loop Start” node. The output data table collected at the current iteration may or may not be the input data table for the next iteration. Hence the two distinct input ports of the “Recursive Loop End” node.

We copied the old “Counting Loop 1” workflow and renamed it “Recursive Loop”. As in the original workflow, the “Data Generator” node generates 100 data rows distributed across 3 clusters along two coordinates (“Universe_0_0” and “Universe_0_1”). However, instead of the counting loop we introduced a recursive loop, with a “Recursive Loop Start” node and a “Recursive Loop End” node with maximum number of iterations set to 4.

The goal of this loop is to increment the initial x-values (“Universe_0_0”) 1 unit at each iteration. So, the loop body was implemented through a “Math Formula” node with mathematical expression “$Universe_0_0$ + 1”, that is adding one unit to all x-values at each iteration. As usual two “Scatter Plot” nodes show the data cluster distribution before and after the loop.

Recursive Loop Start

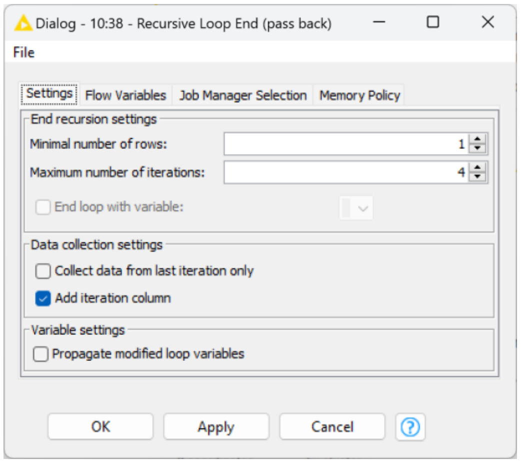
The “Recursive Loop Start” node starts a recursive loop and must be used together with a “Recursive Loop End” node.

The “Recursive Loop Start” node and the “Recursive Loop End” node communicate with each other at the end of each iteration. At iteration 0 the “Recursive Loop Start” node uses the input data able to feed the loop body. After iteration 0, the “Recursive Loop Start” node feeds the loop body with the data table received from the second input port of the “Recursive End Loop” node. No configuration settings needed.

Recursive Loop End

The “Recursive Loop End” node closes a recursive loop, started by a “Recursive Loop Start” node. The “Recursive Loop End” node has two input ports.

- The first input port collects the results of the current iteration and concatenates them with the results from the previous iterations. At the end of the loop the final data table will be passed to the only output port.
- The second input port receives the data table to pass back to the “Recursive Loop Start” node to feed the next loop iteration.


Dialog - 10:38 - Recursive Loop End (pass back)
File
Settings | Flow Variables | Job Manager Selection | Memory Policy

End recursion settings
Minimal number of rows: 1
Maximum number of iterations: 4
[ ] End loop with variable:

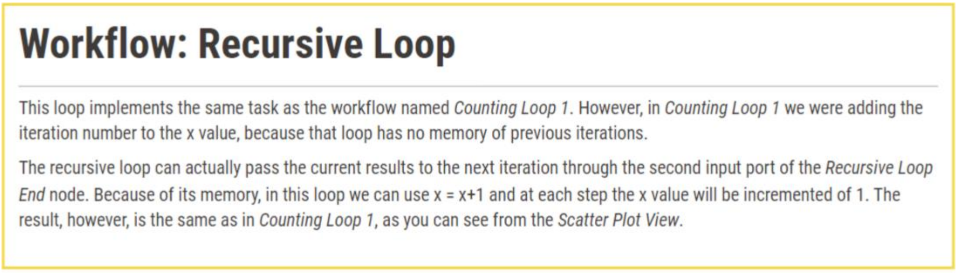
Data collection settings
[ ] Collect data from last iteration only
[✓] Add iteration column

Variable settings
[ ] Propagate modified loop variables

Figure 7.38. Configuration of the Recursive Loop End node.

The first three settings in the configuration window are stop settings, to avoid infinite loops. The loop will stop if:

- The data table passed back to the loop start node contains less than the required minimum number of rows
- A maximum number of iterations has been reached
- A variable takes value “true”
- The last two settings regulate the data structure.
- Collect the output data only from the last iteration or from all iterations.
- Add a data column containing the iteration number to the final output data table
- And finally pass on all the loop variables that have been created and modified.



Workflow: Recursive Loop

This loop implements the same task as the workflow named *Counting Loop 1*. However, in *Counting Loop 1* we were adding the iteration number to the x value, because that loop has no memory of previous iterations.

The recursive loop can actually pass the current results to the next iteration through the second input port of the *Recursive Loop End* node. Because of its memory, in this loop we can use x = x+1 and at each step the x value will be incremented of 1. The result, however, is the same as in *Counting Loop 1*, as you can see from the *Scatter Plot View*.

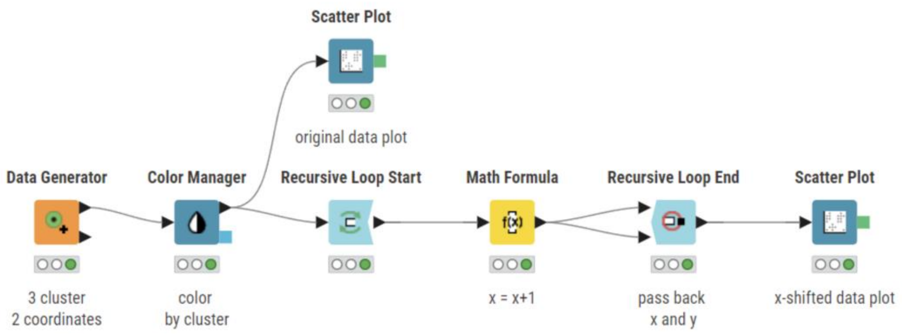


Figure 7.39. Workflow "Recursive Loop".

7.10. Exercises

Exercise 1

Generate 5400 data rows 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.

Then 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 and the scatter plot of the processed data are shown in the figures below.

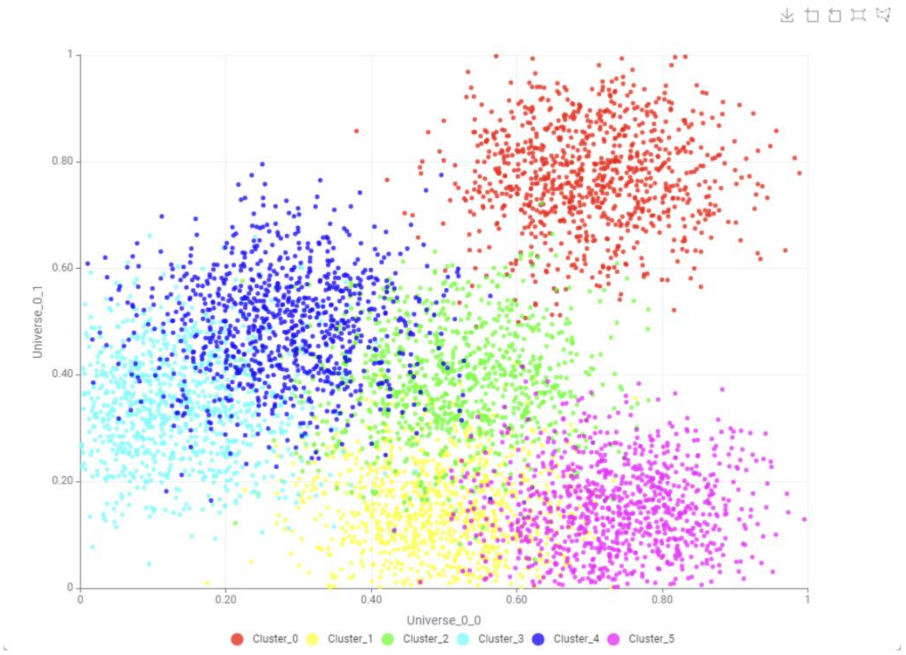


Figure 7.40. Scatter Plot of the original dataset.

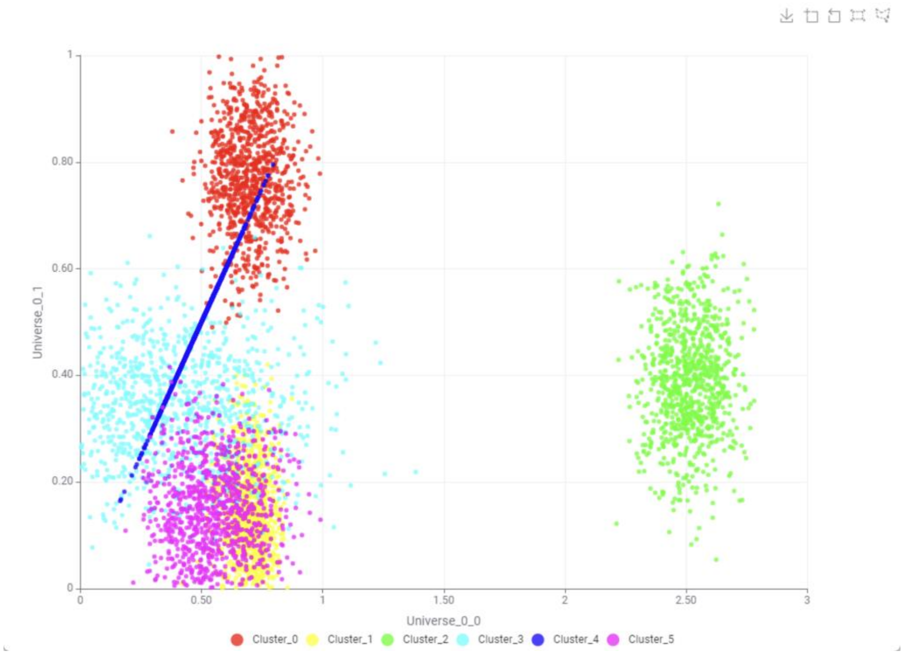


Figure 7.41. Scatter Plot of the resulting data.

Workflow: Chapter 7/Exercise 1

Apply the following transformations to different clusters in the data:

- cluster 0: x=x
- cluster 1: x=sqrt(x)
- cluster 2: x=x+iter#
- cluster 3: x=x*iter#
- cluster 4: x=y
- cluster 5: x=x*x

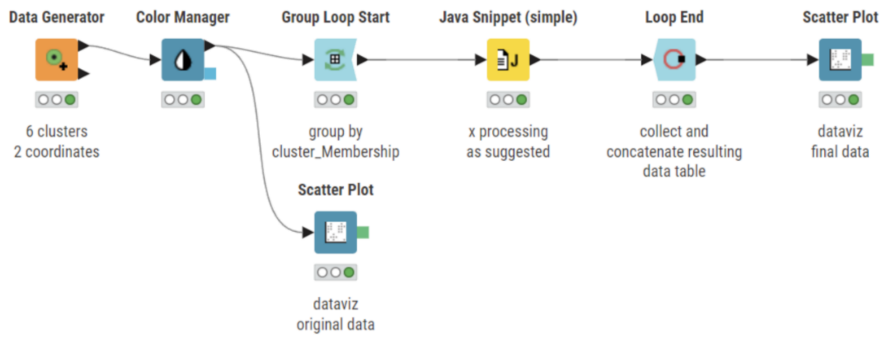


Figure 7.42. Exercise 1: The workflow.

Exercise 2

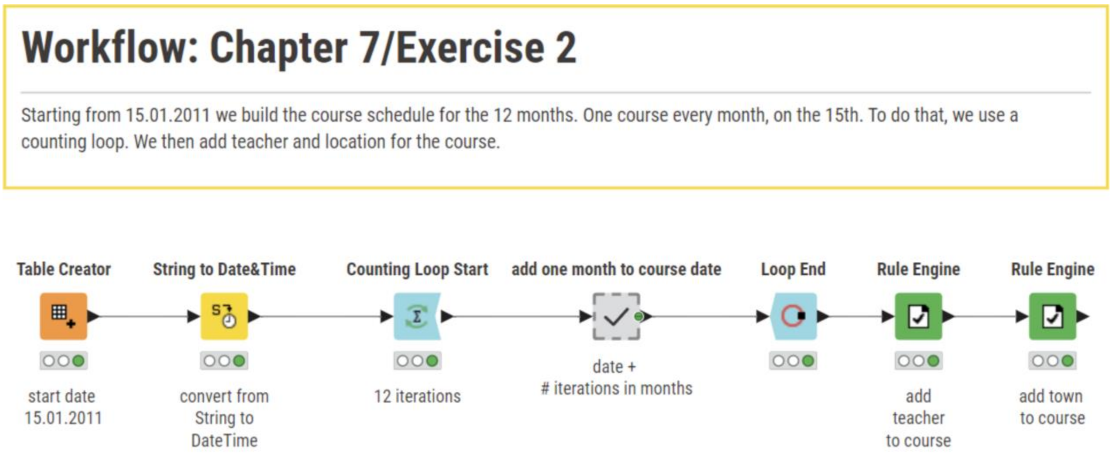
On the 15th of each month a course takes place, starting from 15.01.2011. Teacher "Maria" teaches till the end of March, teacher "Jay" till the end of September, and teacher "Michael" till the end of the year. The course is held in San Francisco from May to September and in New York the other months. 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.



Workflow: Chapter 7/Exercise 2

Starting from 15.01.2011 we build the course schedule for the 12 months. One course every month, on the 15th. To do that, we use a counting loop. We then add teacher and location for the course.

Figure 7.43. Exercise 2: The workflow.

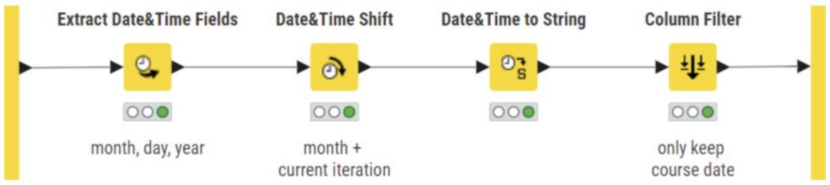


Figure 7.44. Exercise 2: The "add one month to course date" metanode.

Exercise 3

This exercise gets rid of duplicated unnecessary data rows.

In the folder KALdata there is a file called “wrong_sales_file.txt” which contains sales records, each one with a contract number, as well as 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 date of record creation. 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

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 “Table Row 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 anymore.

The same result could have been obtained with just a GroupBy node, grouping by contract number and taking just the first (or last depending on sorting) of all other values.

Note. Since KNIME has been thought in terms of data tables, loops are rarely needed. Before using a loop make sure that a dedicated node for what you have in mind does not exist!

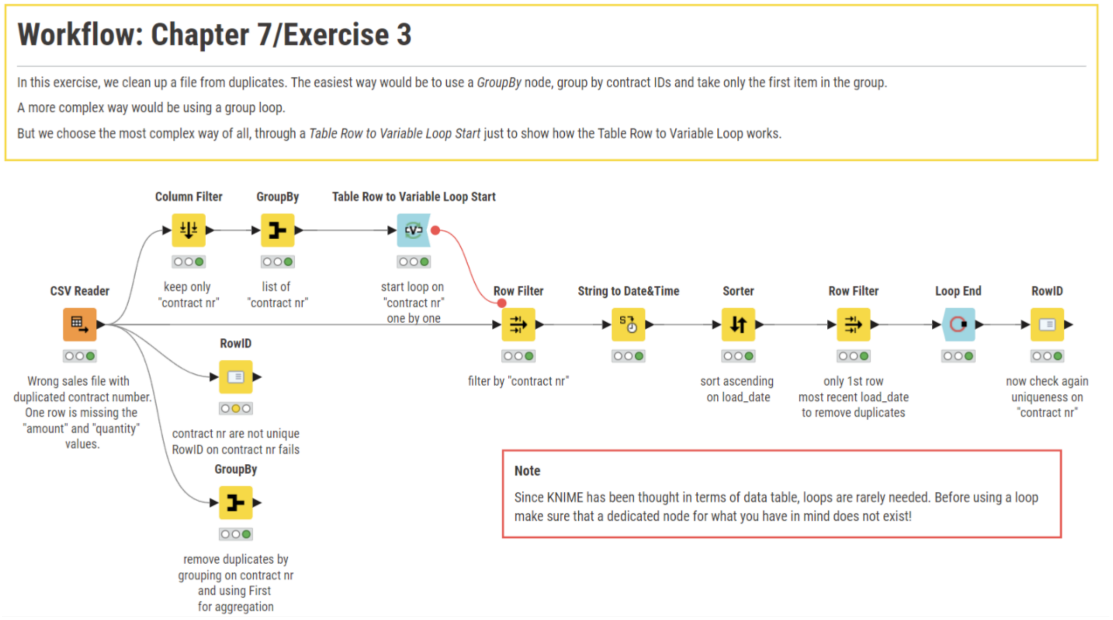

Workflow: Chapter 7/Exercise 3
In this exercise, we clean up a file from duplicates. The easiest way would be to use a GroupBy node, group by contract IDs and take only the first item in the group.
A more complex way would be using a group loop.
But we choose the most complex way of all, through a Table Row to Variable Loop Start just to show how the Table Row to Variable Loop works.
CSV Reader
Wrong sales file with duplicated contract number. One row is missing the "amount" and "quantity" values.
Column Filter
keep only "contract nr"
GroupBy
list of "contract nr"
Table Row to Variable Loop Start
start loop on "contract nr" one by one
RowID
contract nr are not unique RowID on contract nr fails
GroupBy
remove duplicates by grouping on contract nr and using First for aggregation
Row Filter
filter by "contract nr"
String to Date&Time
Sorter
sort ascending on load_date
Row Filter
only 1st row most recent load_date to remove duplicates
Loop End
RowID
now check again uniqueness on "contract nr"
Note
Since KNIME has been thought in terms of data table, loops are rarely needed. Before using a loop make sure that a dedicated node for what you have in mind does not exist!

Figure 7.45. Exercise 3: The workflow.

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 "KALdata/sales".

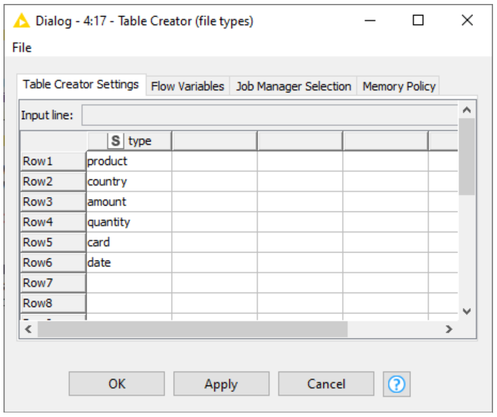
This exercise tries to find all the pieces and to collect them together to form the original file "sales.txt". In "KALdata/sales" we find files like "sales_<column name>.txt" containing the "contract nr" and the "<column name>" columns of the sale records. There are 6 files for 6 columns: "card", "amount", "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 “Table Row to Variable Loop Start” starts a loop that:

- loops over the list of column names in “type”,
- builds the file path with a “String Manipulation (Variable)” node with: join($${Sfile-path}$$, "sales_", $${Stype}$$, ".csv")
- passes the file path as a workflow variable to a “CSV Reader” node,
- reads the file via the “CSV Reader” node,
- collects the final results with a “Loop End (Column Append)” node.

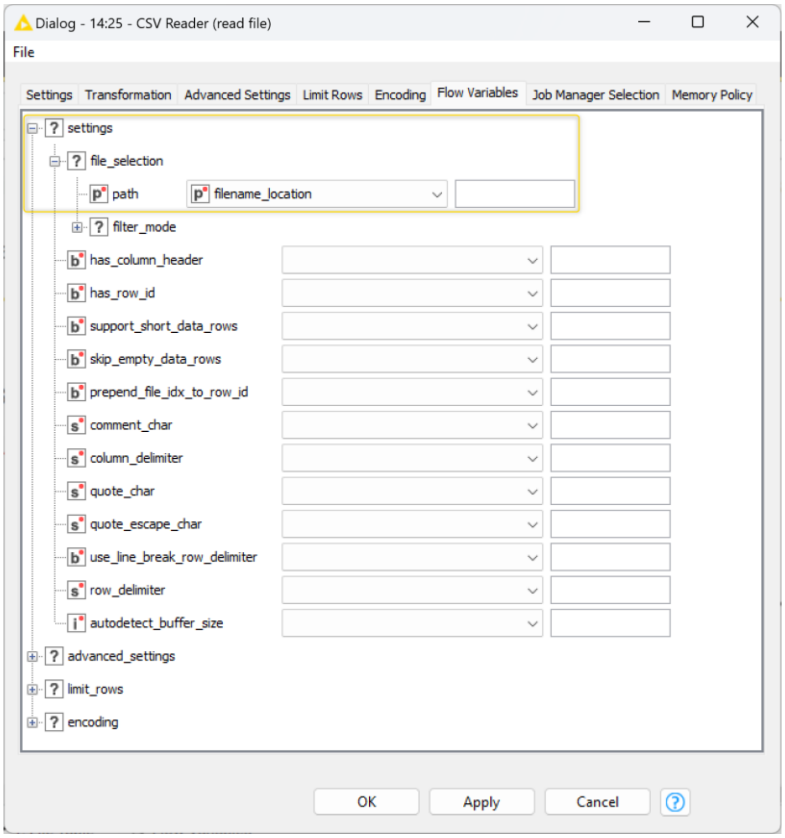


	S type
Row1	product
Row2	country
Row3	amount
Row4	quantity
Row5	card
Row6	date
Row7	
Row8	

Figure 7.46. The configuration window of the Table Creator 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 “CSV Reader” node needs to use the flow variable called “filename_location” to read the file, which was created according to the variable “type” by the “Table Row To Variable Loop Start”.


Dialog - 14:25 - CSV Reader (read file)

File

Settings | Transformation | Advanced Settings | Limit Rows | Encoding | Flow Variables | Job Manager Selection | Memory Policy

- settings
  - file_selection
    - path: filename_location
    - filter_mode
  - has_column_header
  - has_row_id
  - support_short_data_rows
  - skip_empty_data_rows
  - prepend_file_idx_to_row_id
  - comment_char
  - column_delimiter
  - quote_char
  - quote_escape_char
  - use_line_break_row_delimiter
  - row_delimiter
  - autodetect_buffer_size
- advanced_settings
- limit_rows
- encoding

OK | Apply | Cancel

Figure 7.47. The "ColumnName" setting in the "Workflow Variables" Tab of the configuration window of the "File Reader" node to name the data column read at each iteration.

Workflow: Chapter 7/Exercise 4

Here we have single files, each one containing a data column of a bigger data table. Let's list all the file names, build the correct path for each filename, and read the data from the file.

At the end all data columns read from files are joined together in the *Loop End (Column Append)* node.

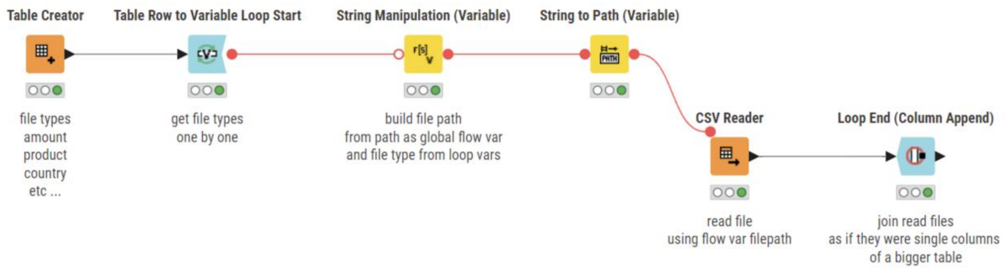


Figure 7.48. Exercise 4: The workflow.

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 just 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 Start" node determines the flow of data via one or more workflow branches. Data flows into a "Switch Start" node and is then directed down one or more routes where a number of specific operations are performed, while all other routes remain inactive. All routes originating from a "Switch Start" node are finally collected by a "Switch End" node. The KNIME switch concept is illustrated below, where a switch with three alternative parallel routes has been implemented.

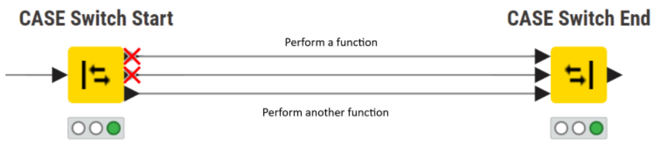


Figure 8.1. Generic illustration of data flow control via Switches.

Depending on the type of "Switch Start" node, the data can flow out through one or more ports. In the example above, the "CASE Switch Start" node has three output ports, of which only one is active at a time. In this particular configuration, the data has been enabled to flow out from the third output port of the "CASE Switch Start" node only. The top output ports, in fact, are blocked, as depicted by a red cross. The data then flows through a number of nodes implementing "another function" till the "CASE Switch End" node, thus completing the switch process. Which output port of the "Switch Start" node is active, and therefore which route the data takes, can be controlled in the configuration window.

All "Switch Start" and "Switch End" nodes are located in the "Workfow Control" -> "Switches" category in the "Node Repository" panel. There are two main types of "Switch" nodes: the "IF Switch" nodes and the "CASE Switch" nodes. The "IF Switch" node starts a two branch switch closed by the “End IF” node. The “CASE Switch Start” node starts three parallel branches and results from the three branches are collected by a “CASE Switch End” node.

Notice the plus sign in the lower corner of the “CASE Switch” nodes. Indeed, while the “IF Switch” nodes only operates on data through their data ports, the “CASE Switch” nodes can operate on data, models, variables, and so on just by dynamically adding ports of the desired type through the plus sign feature. You can also mix and match, i.e. you can start with a “CASE Switch Start” node with data ports, passing the data to the switch branches, and close with a “CASE Switch End” node with variable ports, collecting results of type flow variable to pass on to the next nodes, or vice versa.

In the “Workflow 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.

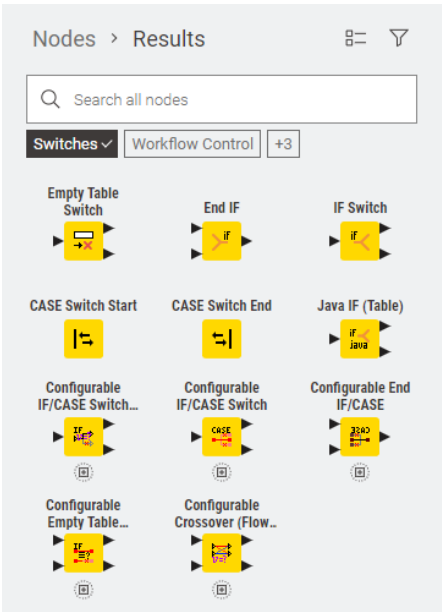


Figure 8.2. The nodes contained in the "Switches" category.

Finally, the last node in this category is the “Empty Table Switch” node. The output of an “Empty Table Switch” 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 “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 block.

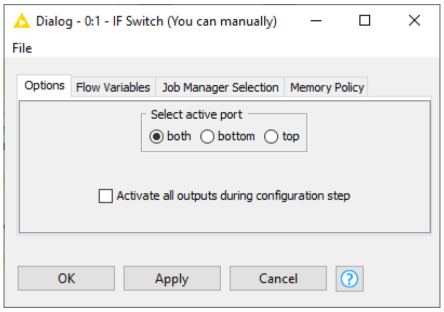
Let’s start with the simplest of the two switch nodes: the “IF Switch” node. In the workflows that you have imported for this book (see chapter 1), two workflows are available in folder “Chapter8” that demonstrate how to set the “IF Switch” node manually or automatically: the “Manual IF Switch” workflow and the “Automatic IF Switch” workflow.

The “Manual IF Switch” workflow reads “cars-85.csv” file from the folder KALdata, 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 branch to bin the data set on “city mpg” data column and the other branch 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 enable the top branch by default. If the data is supposed to flow through the bottom branch, a manual change has to be made to the setting in its configuration window.

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 flow variable.



Select active port
[✓ both] bottom top

Activate all outputs during configuration step

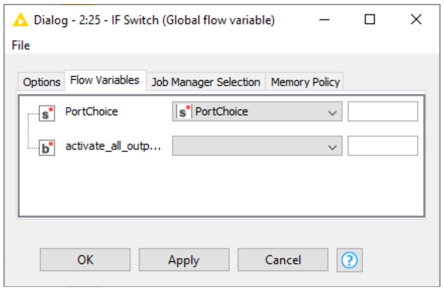
Figure 8.3. Configuration window of the IF Switch node: the "Options" tab.

In the configuration window:

- The “Options” tab contains the options for the manual setting of the active output port; option “Activate all outputs during configuration step” enables all outputs during configuration of other subsequent nodes.
- The “Flow Variables” tab allows to overwrite 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 to overwrite 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”.



PortChoice: PortChoice
activate_all_outp...:

Figure 8.4. Configuration window of the IF Switch node: the "Flow Variables" tab.

The goal of this workflow is to calculate some statistics for columns “curb weight”, “engine size” and “horse power”, sometimes on the quantiles of column “city_mpg” and sometimes on the quantiles of column “highway_mpg”. So, in both branches following the “IF Switch” node, an “Auto-Binner” node was used to bin the data set based on sample quartiles of “city_mpg” on one branch and of “highway_mpg” on the other branch of the IF block. 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 switch block is then closed by an “END IF” node.

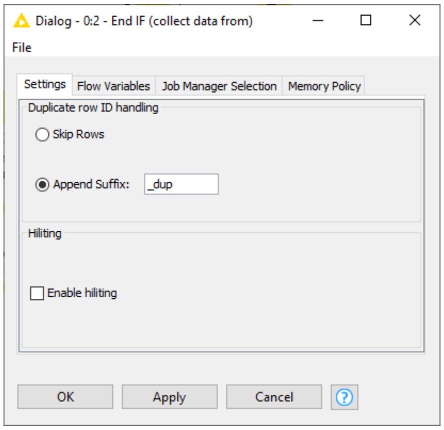
End IF

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 the RowIDs unique.

Check the “Enable hiliting” box if you wish to preserve any hiliting after this node.



Duplicate row ID handling
Skip Rows [✓ Append Suffix: _dup]

Hiliting
Enable hiliting: ☐

Figure 8.5. The configuration window of the End IF node.

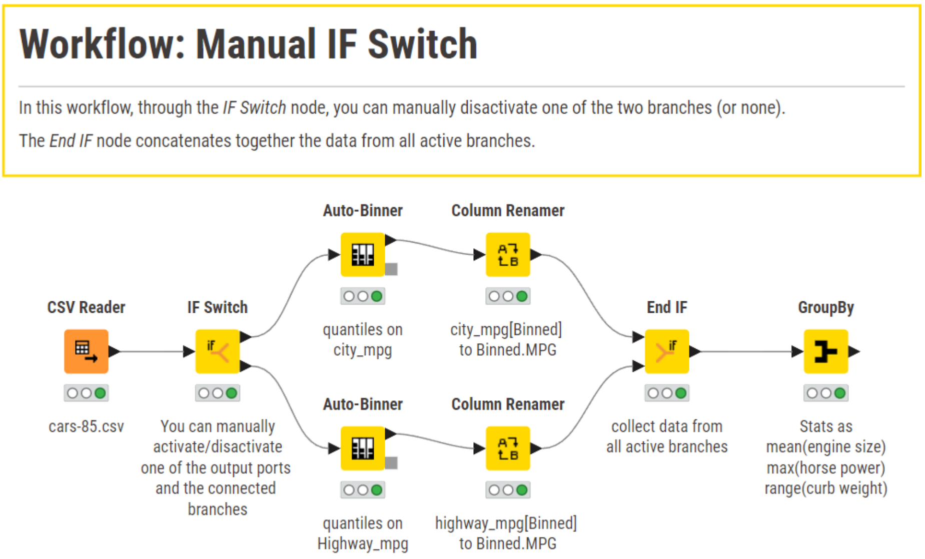
A “GroupBy” node was appended at the end of the switch block to calculate the statistics of “curb weight”, “engine size”, and “horse power” over the defined bins produced by the active workflow branch. Since it would be cumbersome to manually change the configuration settings of the “GroupBy” node every 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, i.e. “Binned.MPG”. The “Manual IF Switch” workflow is shown below.

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. 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 column to be binned is selected via an “Exclude/Include” framework, manually or via Regular / wildcard expression.

The manual selection of the column to be binned 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.

You can move either selected or all visible columns between the “Exclude” and “Include” lists by using four buttons between the lists. A “Filter” box in each frame allows searching for specific columns, in case an excessive number of columns impedes an easy overview of the data.

The selection based on a regular expression or a wildcard requires the regex or wildcard matching pattern as configuration setting.


Workflow: Manual IF Switch

In this workflow, through the *IF Switch* node, you can manually disactivate one of the two branches (or none).

The *End IF* node concatenates together the data from all active branches.

CSV Reader
cars-85.csv
IF Switch
You can manually activate/disactivate one of the output ports and the connected branches
Auto-Binner
quantiles on city_mpg
Column Renamer
city_mpg[Binned] to Binned.MPG
Auto-Binner
quantiles on Highway_mpg
Column Renamer
highway_mpg[Binned] to Binned.MPG
End IF
collect data from all active branches
GroupBy
Stats as mean(engine size) max(horse power) range(curb weight)

Figure 8.6. The "Manual IF Switch" node.

Auto-Binner

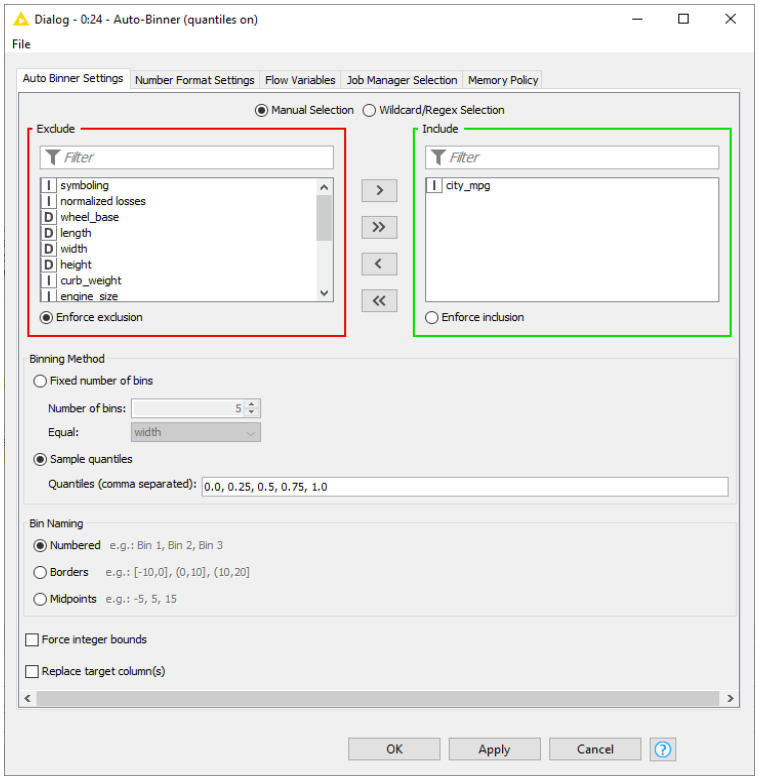
The configuration window offers the choice between two methods to build the bins:

- As a fixed number of equally spaced bins;
- As pre-defined sample quantiles.

The configuration window also offers the choice between three naming options for the bins:

- "Bin1", "Bin2", …, "BinN"
- By using the bin borders, e.g. [0.5674, 0.7899]
- By using midpoints, e.g., -5, 5, 15

There is then a flag to avoid these non-round boundaries of the bin intervals. You can also choose to overwrite the original column by means of another flag. The "Number Format Settings" tab allows for custom formatting of double and string bin labels.


Dialog - 0:24 - Auto-Binner (quantiles on)

Manual Selection [✓] Wildcard/Regex Selection

Exclude: symboling, normalized losses, wheel_base, length, width, height, curb_weight, engine_size

Enforce exclusion [✓]

Include: city_mpg

Binning Method: Sample quantiles [✓]

Number of bins: 5

Equal: width

Quantiles (comma separated): 0.0, 0.25, 0.5, 0.75, 1.0

Bin Naming: Numbered [✓] e.g.: Bin 1, Bin 2, Bin 3

Borders e.g.: [-10,0], (0,10], (10,20]

Midpoints e.g.: -5, 5, 15

Force integer bounds [ ]

Replace target column(s) [ ]

Figure 8.7. Configuration window of the Auto-Binner node: Manual Selection.

Pre-defined sample quantiles produce bins corresponding to the given list of frequencies. The smallest element corresponds to a frequency of 0 and the largest to a frequency 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.

Alternative 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, for example named “PortChoice”, and we could use that variable to overwrite the manual settings in the “IF Switch” node’s configuration window. If we activate the output port of the “IF Switch” node via a flow variable, we do not need to manually change the node configuration settings to enable the other branch of the switch block. In this case, it is enough to change the value of the flow 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 again in folder “Chapter8”. This workflow is identical to the “IF Manual Switch” workflow, except for the flow variable “PortChoice” and the configuration settings of the “IF Switch” node. In fact, the “IF Switch” node is configured so as to overwrite the active port choice with the value of the flow variable “PortChoice”.

8.3. The “Java IF (Table)” node

A similar node to the “IF Switch” node is the “Java IF (Table)” node. In the “Java IF (Table)” node the active port, and therefore the data flow direction, is controlled through some Java code. The return value of the Java code determines which port is activated and which branch of the workflow is active.

A switch block, then, can start with a “Java IF (Table)” node, branch off into two different data flow paths, and collect the results of the two branches with an “End IF” node. The “JAVA IF (Table)” node can be found like all other switch nodes under the “Workflow Control” -> “Switches” category.

To demonstrate the use of the “Java IF (Table)” node, we created a new workflow, named “Java IF & Tables”, in the workflow group called “Chapter8”. The goal of the workflow was to produce a box plot on car price, engine size, and horsepower for either two- or four-door cars for data from the “cars-85.csv” file.

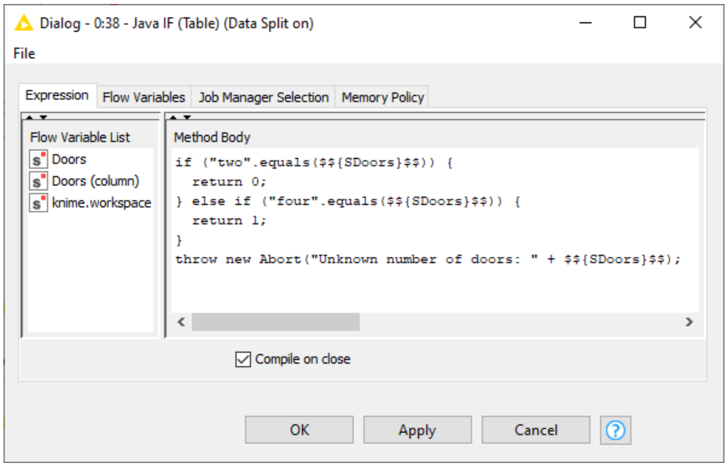
The selection of either two-door cars or four-door cars was implemented by means of a “Java IF (Table)” switch node governed by a flow variable called “Doors”. This flow variable can take two string values, either “two” or “four”, which determines the behaviour of the “Java IF (Table)” switch node. Finally, an “Box Plot” node produces the box plot on the data coming out of the switch block. Depending on the value of the “Doors” flow variable and therefore on the active branch in the switch block, a box plot for engine size, horsepower, and scaled price of two- or four-door cars is produced.

Java IF (Table)

The “Java IF (Table)” node acts like an “IF Switch” node. It takes one input data table and redirects it to one of two output ports.

However, it differs from the “IF Switch” node, because it can activate only one output port at a time.

The main 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 lower output port.



```
if ("two".equals($${SDoors}$$)) {
  return 0;
} else if ("four".equals($${SDoors}$$)) {
  return 1;
}
throw new Abort("Unknown number of doors: " + $${SDoors}$$);
```

Figure 8.8. The configuration window of the Java IF (Table) node.

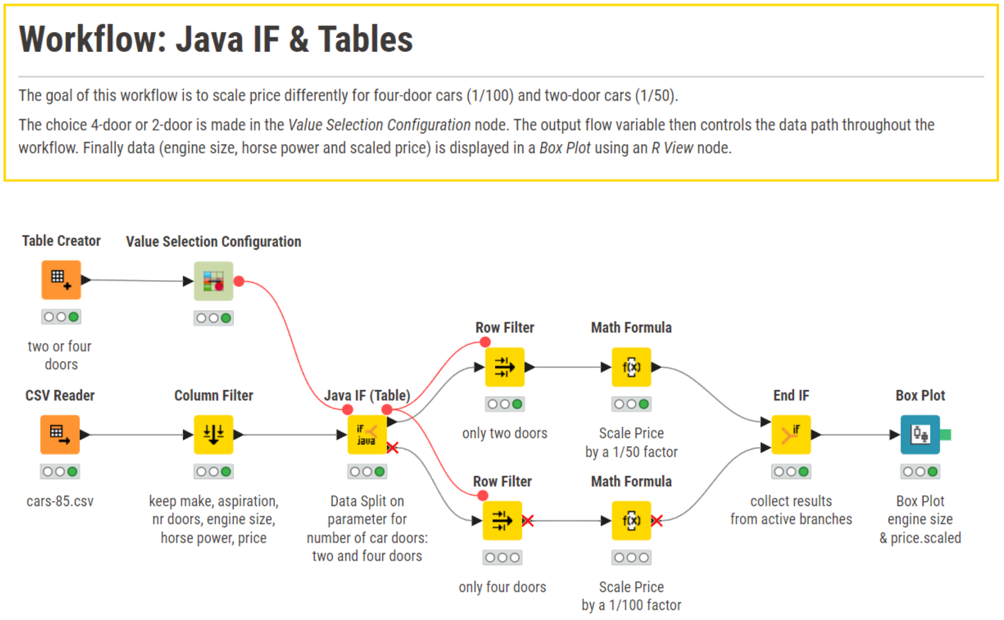
The configuration window of the “Java IF (Table)” node resembles the configuration window of the “Java Edit Variable” node. It contains:

- A “Method Body” panel for the piece of Java code to be executed at execution time
- A “Flow Variable List” panel where all flow variables available to this node are listed

Like the “Java Edit Variable” node, the Java code in the “Java IF (Table)” node operates only on flow 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 cannot output any value for the return variable, just 0 or 1. Any other return value will result in an error at execution time.

Note. Inside the “Java IF (Table)” node configuration window, an exception handling statement “`throw new Abort();`” is required at the end of the Java code, to handle results that do not produce a 0/1 return value.



Workflow: Java IF & Tables

The goal of this workflow is to scale price differently for four-door cars (1/100) and two-door cars (1/50).

The choice 4-door or 2-door is made in the *Value Selection Configuration* node. The output flow variable then controls the data path throughout the workflow. Finally data (engine size, horse power and scaled price) is displayed in a *Box Plot* using an *R View* node.

Table Creator
two or four doors
Value Selection Configuration
CSV Reader
cars-85.csv
Column Filter
keep make, aspiration, nr doors, engine size, horse power, price
Java IF (Table)
Data Split on parameter for number of car doors: two and four doors
Row Filter
only two doors
Math Formula
Scale Price by a 1/50 factor
Row Filter
only four doors
Math Formula
Scale Price by a 1/100 factor
End IF
collect results from active branches
Box Plot
Box Plot engine size & price.scaled

Figure 8.9. The "Java IF & Tables" workflow.

8.4. The CASE Switch Block

We have finally reached the most commonly used switch block in the KNIME workflows: 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 Start” node opens a switch block which is then completed by inserting a “CASE Switch End” node.

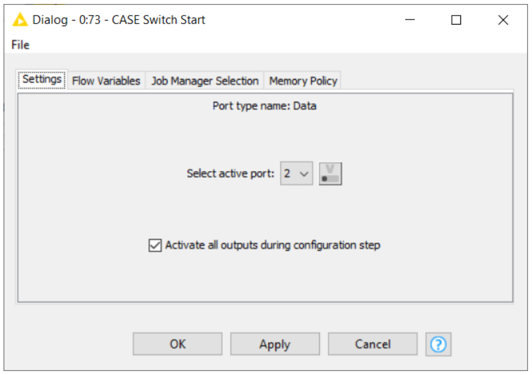
The “CASE Switch Start” node offers the choice of a few possible mutually exclusive data flow paths. The “CASE Switch End” node collects the results from the active path(s) of the switch block. The “CASE Switch Start” node and the “CASE Switch End” node are created with no input and no output ports. Input and output ports of the desired tape can be added via the dynamic insertion feature reachable through the three dots on the lower left corner of the node.

CASE Switch Start

The "Case Switch Start" node starts a switch block with multiple data flow branches. Of the output ports available, only one can be active at a time. The output ports are indexed by an integer number, such as 0, 1, and 2.

The configuration window requires only the index of the active output port.

Like the "IF Switch" node, the active port index can be provided manually or automatically via the Flow Variable button in the configuration window.



Port type name: Data

Select active port: 2

☑ Activate all outputs during configuration step

Figure 8.10. The configuration window of the CASE Switch Start node.

In "Chapter8/CASE Switch" workflow, there is an example of a CASE switch block. This workflow was implemented to work on the "cars-85.csv" data. It was implemented to build alternatively a few data classification models on the input data set: either a Decision Tree, a Probabilistic Neural Network (PNN), or a rule system.

The decision of which model to implement is supposed to be arbitrarily made by the workflow end user. The workflow then required three separate branches: one to implement the decision tree, one to implement the PNN, and one to implement a rule system. An "IF Switch" node here was not enough: "CASE Switch" block with three output data ports was needed.

To toggle between the models, a new flow variable was created and named "ClassificationModel". This flow variable is used to indicate the type of analysis to do and can take three values only: "rule system", "decision tree", and "pnn". However, a "CASE Switch Start" node cannot understand string values, like "decision tree" or "pnn". It only takes integer values to define which output port is active. So, the string value of the "Classification Model" workflow variable had to be converted to an integer value. We used a "Rule Engine (Variable)" node, to transform "rule system" into 0, "decision tree" into 1, and "pnn" into 2 and we assigned the final value to a new flow variable "classification_int". This last flow variable was used to control the active port in the "CASE Switch Start" node.

CASE Switch End

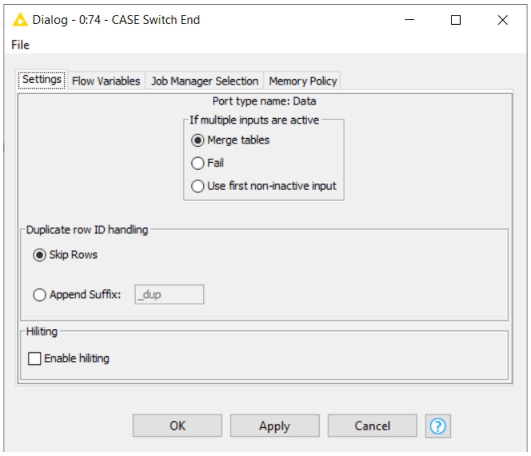
The "CASE Switch End" node closes a CASE switch block with multiple branches and collects the resulting data table(s).

If more than one branch is active at the same time, the "CASE Switch End" node offers a few options:

- merge the resulting data tables together
- fail
- or use only the output of the first inactive branch

Another problem could be the possible existence of rows with duplicate RowIDs. In this case, two options are offered:

- Skip rows with duplicate IDs
- Make those RowIDs unique by appending a suffix to the duplicate values.



Port type name: Data

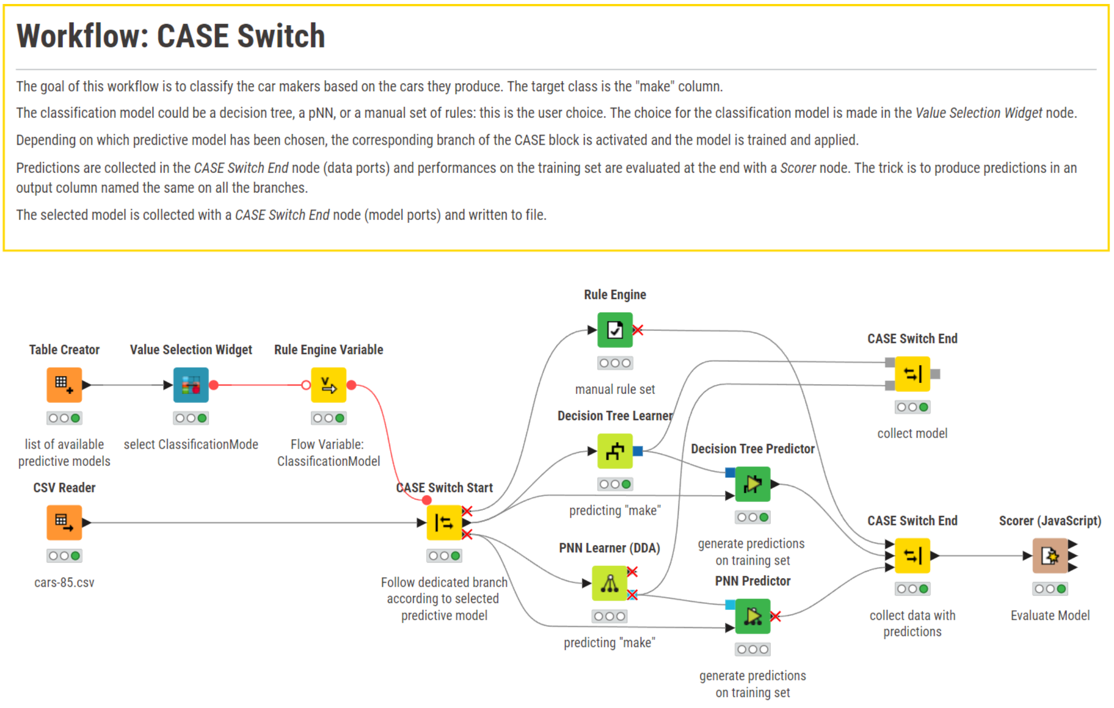
If multiple inputs are active
[✓ Merge tables]
Fail
Use first non-inactive input

Duplicate row ID handling
[✓ Skip Rows]
Append Suffix: _dup

Hiliting
Enable hiliting

Figure 8.11. The configuration window of the CASE Switch End node.

Next, we implemented a decision tree, a pnn, and a rule system respectively on each one of the three branches coming out of the "CASE Switch Start" node. Each model produces two kinds of results: the model itself and the data tables with the predictions. In order to collect both results, we used two End nodes to close the switch block: a "CASE Switch End" node with generic ports to collect the models, and a "CASE Switch End" node to collect the predicted data. The models are then saved to a file and the predictions are evaluated by a "Scorer" node.


Workflow: CASE Switch

The goal of this workflow is to classify the car makers based on the cars they produce. The target class is the "make" column.

The classification model could be a decision tree, a pNN, or a manual set of rules: this is the user choice. The choice for the classification model is made in the *Value Selection Widget* node.

Depending on which predictive model has been chosen, the corresponding branch of the CASE block is activated and the model is trained and applied.

Predictions are collected in the *CASE Switch End* node (data ports) and performances on the training set are evaluated at the end with a *Scorer* node. The trick is to produce predictions in an output column named the same on all the branches.

The selected model is collected with a *CASE Switch End* node (model ports) and written to file.

Figure 8.12. The "CASE Switch" workflow.

8.5. Transforming an Empty Data Table Result into an Inactive Branch

In “Chapter8” folder, 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 type and sends the resulting data table to a “CASE Switch End” node. Finally, a “GroupBy” node counts the number of cars by fuel-type for that “make” with that type of wheel drive.

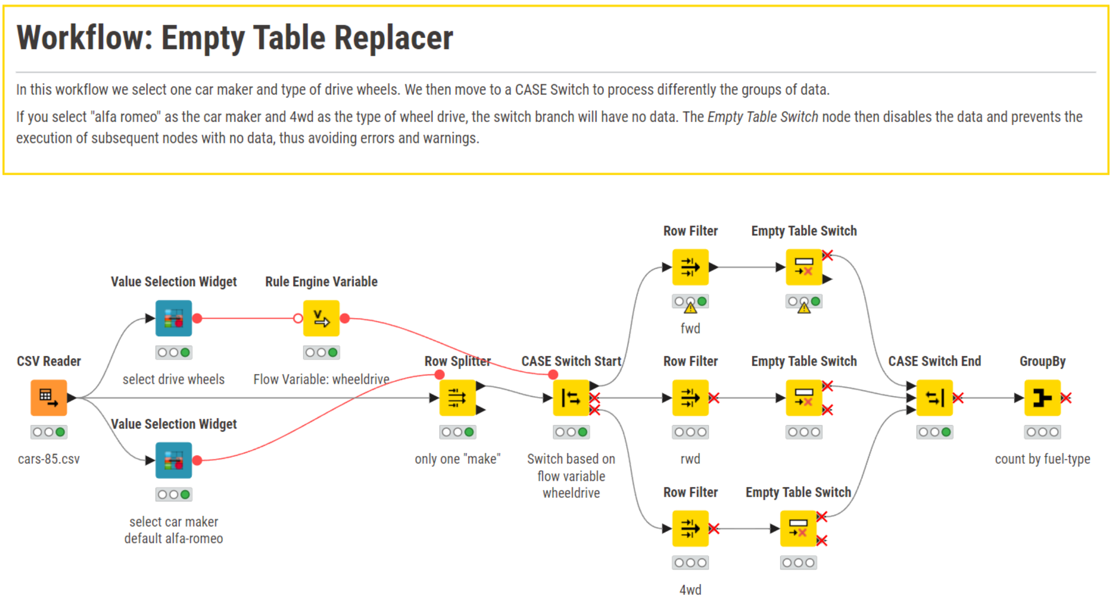
In the “Empty Table Replacer” workflow, the “CASE Switch Start” node is controlled by a flow variable, named “wheeldrive”. The flow variable “wheeldrive” can only take “rwd”, “fwd”, and “4wd” string values, which are then transformed into output port indices by a “Rule Engine (Variable)” node to control the active port in the “CASE Switch Start” node.

The data selection in the switch branches is implemented by “Row Filter” nodes on the wheel drive type. The “Row Filter” nodes are controlled by the same flow variable “wheeldrive” that controls the active branch in the switch block.

What happens, though, if one of the “Row Filter” nodes returns an empty data table? For example, if you assign the value “rwd” to the flow variable “wheeldrive”, the “Row Filter” node in the corresponding active branch 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, like the “GroupBy” node, but they could also 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: one is active when an empty data table is created, and one is active when a value-filled data table is created.

Empty Table Switch

The “Empty Table Switch” node has two output ports. It activates the lower output port for an empty input data table. Otherwise, it activates the top output port. This allows for the creation of an alternate branch in case of an empty input data table and avoids the execution of subsequent nodes on tables with no actual content.



Workflow: Empty Table Replacer

In this workflow we select one car maker and type of drive wheels. We then move to a CASE Switch to process differently the groups of data.

If you select "alfa romeo" as the car maker and 4wd as the type of wheel drive, the switch branch will have no data. The *Empty Table Switch* node then disables the data and prevents the execution of subsequent nodes with no data, thus avoiding errors and warnings.

Figure 8.13. The "Empty Table Replacer" workflow.

The configuration window of the “Empty Table Switch” node does not require any setting.

In each branch of the CASE switch block, an “Empty Table Switch” node was introduced, 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 flow variable “wheeldrive” = “fwd”, the “Row Filter” node of the second branch produces an empty data table, the output of the “CASE Switch End" node collects no results, the “GroupBy” node becomes inactive and is not executed.

8.6. Exercises

Exercise 1

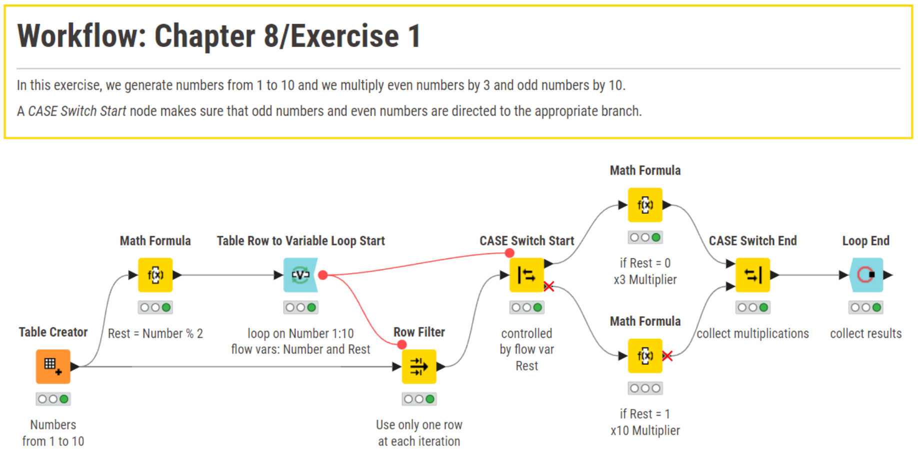
Create a table containing the numbers 1-10. Then, by using an “CASE Switch” node, create a workflow to generate an additional integer column produced by multiplying the even numbers by a factor of 3 and the odd numbers by a factor of 10.

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. The rest from a division by 2 is calculated for each number by means of a “Math Formula” node. The value of this rest defines the multiplying factor, 3 or 10. An easy solution to solve this problem could have been to apply a “Rule Engine” node and transform the rest value into the corresponding multiplying factor. A “Math Formula” node then would have finished 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 “Table Row To Variable Loop Start” node, turning each number and each rest value into flow variables.

A “CASE Switch Start” node implemented the switch for different multipliers. At each iteration of the loop, as input data for the “CASE Switch Start” we use the row of initial table that corresponds to the current iteration. We filter it with the “Row Filter” node by matching to the flow variable “Number”. Numbers are multiplied by 3 on one of the two branches and by 10 on the other. The active port of the “CASE Switch Start” node is controlled by the rest value as calculated in the Math Formula node: 0 enables the upper port; 1 enables the second output port.



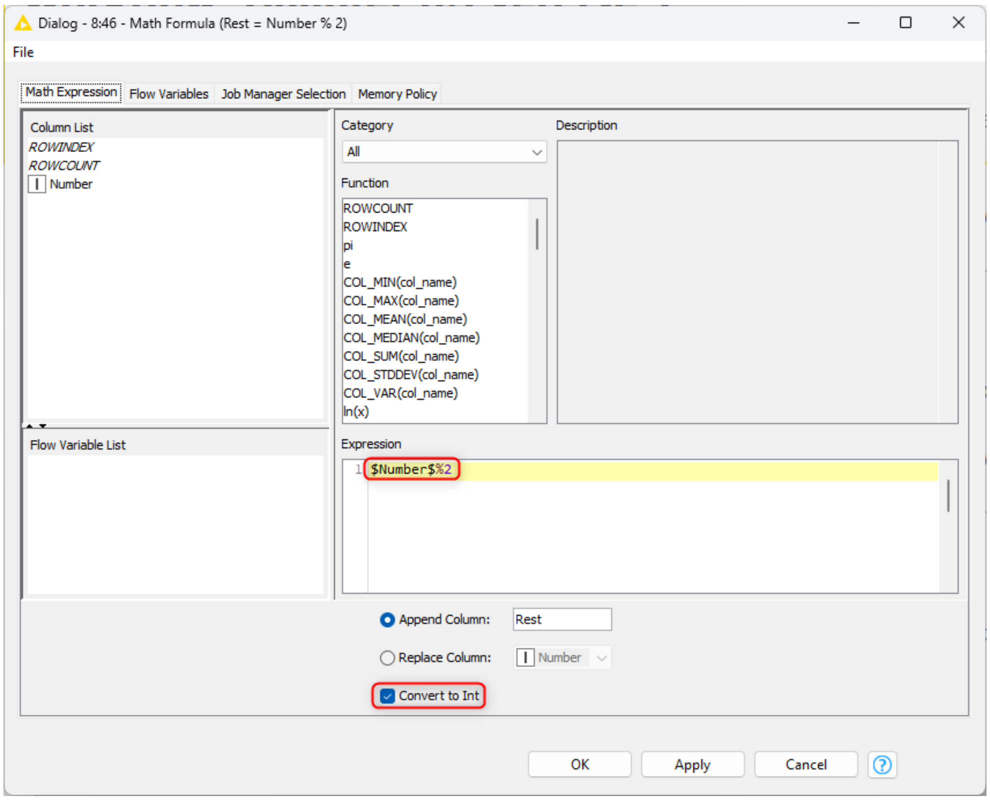
Workflow: Chapter 8/Exercise 1

In this exercise, we generate numbers from 1 to 10 and we multiply even numbers by 3 and odd numbers by 10.

A *CASE Switch Start* node makes sure that odd numbers and even numbers are directed to the appropriate branch.

Table Creator
Numbers from 1 to 10
Math Formula
Rest = Number % 2
Table Row to Variable Loop Start
loop on Number 1:10 flow vars: Number and Rest
Row Filter
Use only one row at each iteration
CASE Switch Start
controlled by flow var Rest
Math Formula
if Rest = 0 x3 Multiplier
Math Formula
if Rest = 1 x10 Multiplier
CASE Switch End
collect multiplications
Loop End
collect results

Figure 8.14. Exercise 1: The workflow.



```
$Number$%2
```

Append Column: Rest

Replace Column: Number

Convert to Int [✓]

Figure 8.15. 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

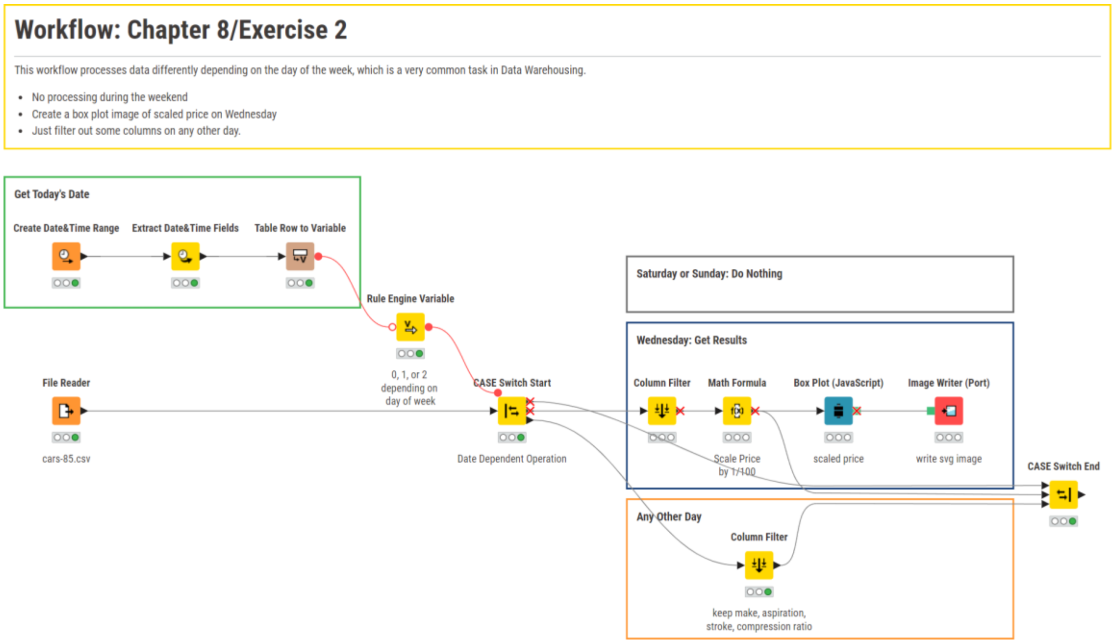
First of all, we need to extract the current date and time information. We did that with a “Create Date&Time Range” node, followed by an “Extract Date&Time Fields” node and a “TableRow to Variable” node. We are interested in the DAY_OF_WEEK information, expressed as an integer 1-7, 1 being Sunday.

A subsequent “Rule Engine (Variable)” node implements the following rule:

```
$${IDay of week (number)}$$ = 1 OR $${IDay of week (number)}$$ = 7 => 0
$${IDay of week (number)}$$ = 4 => 1
TRUE =>  2;
```

This 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 sprouting from the “CASE Switch” node and the final results are collected by an “End CASE” node.



Workflow: Chapter 8/Exercise 2

This workflow processes data differently depending on the day of the week, which is a very common task in Data Warehousing.

- No processing during the weekend
- Create a box plot image of scaled price on Wednesday
- Just filter out some columns on any other day.

Figure 8.16. Exercise 2: The workflow.

Node & Topic Index

KNIME Advanced's Luck

This book presents some more advanced features like looping, workflow variables, reading and writing data from and to a database, dealing with Date&Time objects, and more. The goal is to elevate your data analysis from a basic exploratory level to a more professionally organized and complex structure.

Dr. Rosaria Silipo has been mining data since her master's degree in 1992. She kept mining data throughout all her doctoral program, her postdoctoral program, and most of her following job positions. She has many years of experience in data analysis, reporting, business intelligence, training, and writing. In the last few years she has been using KNIME for all her data science work, becoming a KNIME trainer and evangelist.

Sanket Joshi works as a Data Analyst in the Evangelism team. After completing a bachelor's degree in Computer Science, he moved to Germany to pursue his master's Degree in Data and Knowledge Engineering at OVGU, Magdeburg where he worked with different data tools. Sanket handles the internal reporting of the team and loves building workflows.

ISBN: **978-3-9523926-6-9**