Open for Innovation
KNIME

KNIME Database Extension Guide

KNIME AG, Zurich, Switzerland
Version 5.4 (last updated on)

Table of Contents

Introduction

The KNIME Database Extension provides a set of KNIME nodes that allow connecting to JDBC-compliant databases. These nodes reside in the *DB* category in the Node Repository, where you can find a number of database access, manipulation and writing nodes.

The database nodes are part of every KNIME Analytics Platform installation. It is not necessary to install any additional KNIME Extensions.

This guide describes the KNIME Database extension, and shows, among other things, how to connect to a database, and how to perform data manipulation inside the database.

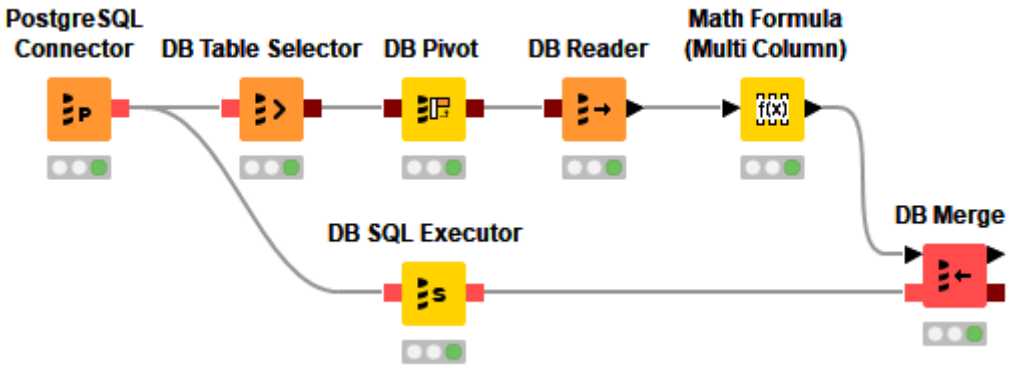

Figure 1. Example workflow using DB nodes

Port Types

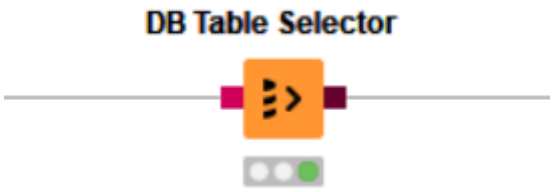

Figure 2. Two types of Database port

There are two types of ports in the Database extension, the DB Connection port (red) and the DB Data port (dark red).

The *DB Connection* port stores information about the current DB Session, e.g data types, connection properties, JDBC properties, driver information, etc.

The *DB Data* port gives you access to a preview of the data.

Outport views

After executing a DB node, you can inspect the result in the outport view by right clicking the node and selecting the outport to inspect at the bottom of the menu. For more information on how to execute a node, please refer to the Quickstart Guide.

DB Connection outport view

The outport view of a DB Connection has the *DB Session* tab, which contains the information about the current database session, such as database type, and connection URL.

DB Data outport view

When executing a database manipulation node that has a *DB Data* outport, for example a *DB GroupBy* node, what the node does is to build the necessary SQL query to perform the GroupBy operation selected by the user and forward it to the next node in the workflow. It does not actually execute the query. However, it is possible to inspect a preview of a subset of the intermediate result and its specification.

To do so, select the node and click *Fetch 100 table rows* in the node monitor at the bottom of the UI.

By default only the first 100 rows are cached, but you can select also other options by opening the dropdown menu of the *Fetch* button. Hoewever, be aware that, depending on the complexity of the SQL query, already caching only the first 100 rows might take a long time.

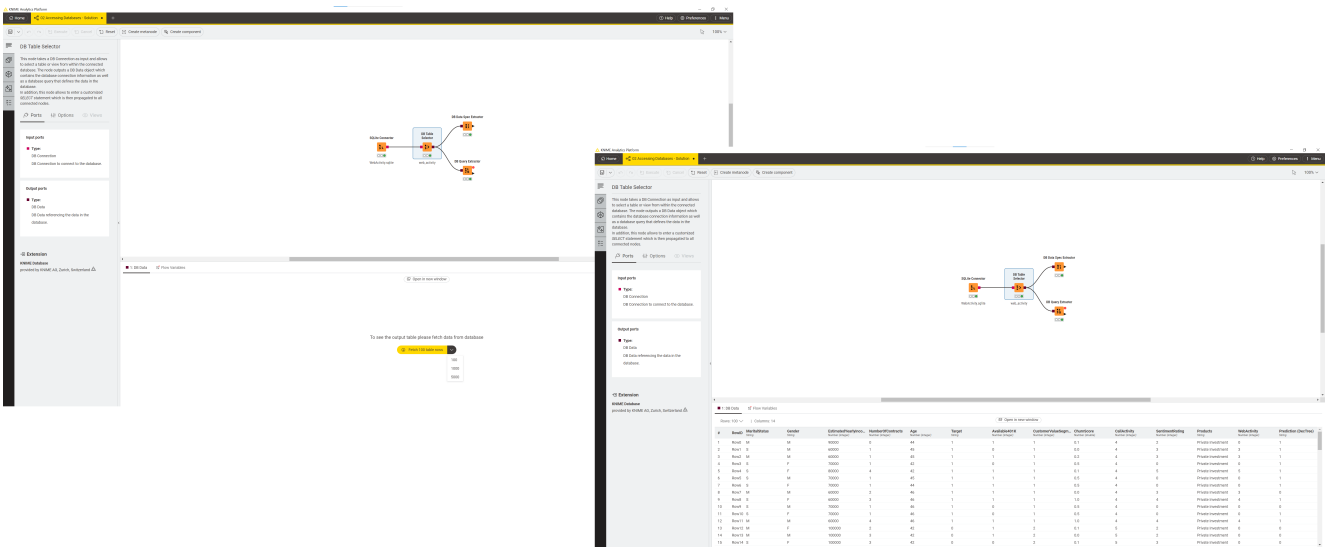

Figure 3. DB Outport View with retrieved rows

The table specification can be inspected by using the DB Data Spec Extractor node The

output of this node will show the list of columns in the table, with their database types and the corresponding KNIME data types.

For more information on the type mapping between database types and KNIME types, please refer to the Type Mapping section.

The generated SQL query can be extracted by using the DB Query Extractor node.

Session Handling

The DB Session life cycle is managed by the Connector nodes. Executing a Connector node will create a DB Session, and resetting the node or closing the workflow will destroy the corresponding DB Session and with it the connection to the database.

To close a DB Session during workflow execution the DB Connection Closer node can be used. This is also the preferred way to free up database resources as soon as they are no longer needed by the workflow. To close a DB Session simply connect it to the DB Connection Closer node which destroys the DB Session and with it the connection to the database as soon as it is executed. Use the input flow variable port of the DB Connection Closer node to executed it once it is save to destroy the DB Session.

Connecting to a database

The *DB → Connection* subcategory in the Node Repository contains

- a set of database-specific connector nodes for commonly used databases such as Microsoft SQL Server, MySQL, PostgreSQL, H2, etc.
- as well as the generic *Database Connector* node.

A Connector node creates a connection to a database via its JDBC driver. In the configuration dialog of a Connector node you need to provide information such as the database type, the driver, the location of the database, and the authentication method if available.

Most of the database-specific connector nodes already contain the necessary JDBC drivers and provide a configuration dialog that is tailored to the specific database. It is recommended to use these nodes over the generic *DB Connector* node, if possible.

Connecting to predefined databases

The following are some databases that have their own dedicated Connector node:

- Amazon Redshift
- Amazon Athena
- Google BigQuery
- H2
- Microsoft Access
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- Snowflake
- SQLite
- Vertica

Some dedicated Connector nodes, such as Google BigQuery or Amazon Redshift, come without a JDBC driver due to licensing restriction. If you want to use these nodes, you need to register the corresponding JDBC driver first. Please refer to the Register your own JDBC drivers section on how to register your own driver. For Amazon Redshift, please refer to the Third-party Database Driver Plug-in section.

If no dedicated connector node exists for your database, you can use the generic *DB Connector* node. For more information on this please refer to the Connecting to other databases section.

After you find the right Connector node for your database, double-click on the node to open the configuration dialog. In the *Connection Settings* window you can provide the basic parameters for your database, such as the database driver, location, or authentication. Then click *Ok* and execute the node to establish a connection.

If you select "Use latest driver version available" upon execution the node will automatically use the driver with the latest (highest) driver version that is available for the current database type. This has the advantage that you do not need to touch the workflow after a driver update. However, the workflow might break in the rare case that the behavior of the driver, e.g. type mapping, changes with the newer version.

KNIME Analytics Platform in general provides three different types of connector nodes the File-based Connector node , the Server-based Connector node and the generic Connector nodes which are explained in the following sections.

File-based Connector node

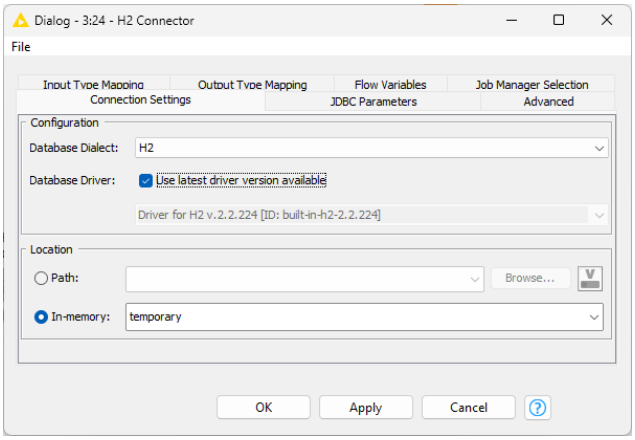

Figure 4. H2 Connector configuration dialog

The figure on the left side shows an example of the node dialog for a file-based database, such as SQLite, H2, or MS Access. The most important node settings are described below:

Configuration: In the configuration window you can choose the registered database dialect and driver.

Location: The location to the database. You can provide either the *path* to an existing database, or choose *in-memory* to create a temporary database that is kept in memory if the database supports this feature.

Server-based Connector node

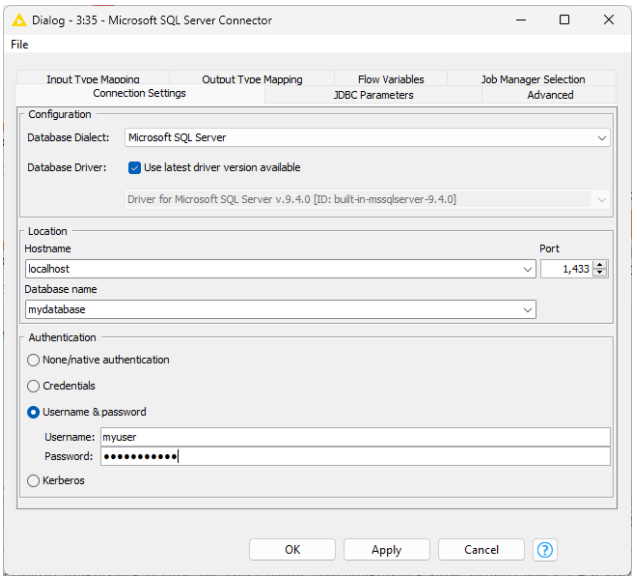

Figure 5. MS SQL Server Connector configuration dialog

The figure on the left side shows an example of the node dialog for a server-based database, such as MySQL, Oracle, or PostgreSQL. The most important node settings are described below.

Configuration: In the configuration window you can choose the registered database dialect and driver.

Location: The location to the database. You should provide the hostname and the port of the machine that hosts the database, and also the name of the database which might be optional depending on the database.

Authentication: Login credentials 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. For more information on Kerberos authentication, please refer to the Kerberos User Guide.

For more information on the JDBC Parameters and Advanced tab, please refer to the JDBC Parameters and Advanced Tab section. The Type Mapping tabs are explained in the Type Mapping section.

Third-party Database Driver Plug-in

As previously mentioned, the dedicated database-specific connector nodes already contain the necessary JDBC drivers. However, some databases require special licensing that prevents us from automatically installing or even bundling the necessary JDBC drivers with the corresponding connector nodes. For example, KNIME provides additional plug-ins to install the Oracle Database driver, official Microsoft SQL Server driver or the Amazon Redshift driver which require special licenses.

To install the plug-ins, go to *File* → *Install KNIME Extensions…*. In the *Install* window, search

for the driver that you need (Oracle, MS SQL Server or Redshift), and you will see something similar to the figure below. Then select the plug-in to install it. If you don't see the plug-in in this window then it is already installed. After installing the plug-in, restart KNIME. After that, when you open the configuration dialog of the dedicated Connector node, you should see that the installed driver of the respective database is available in the database driver list.

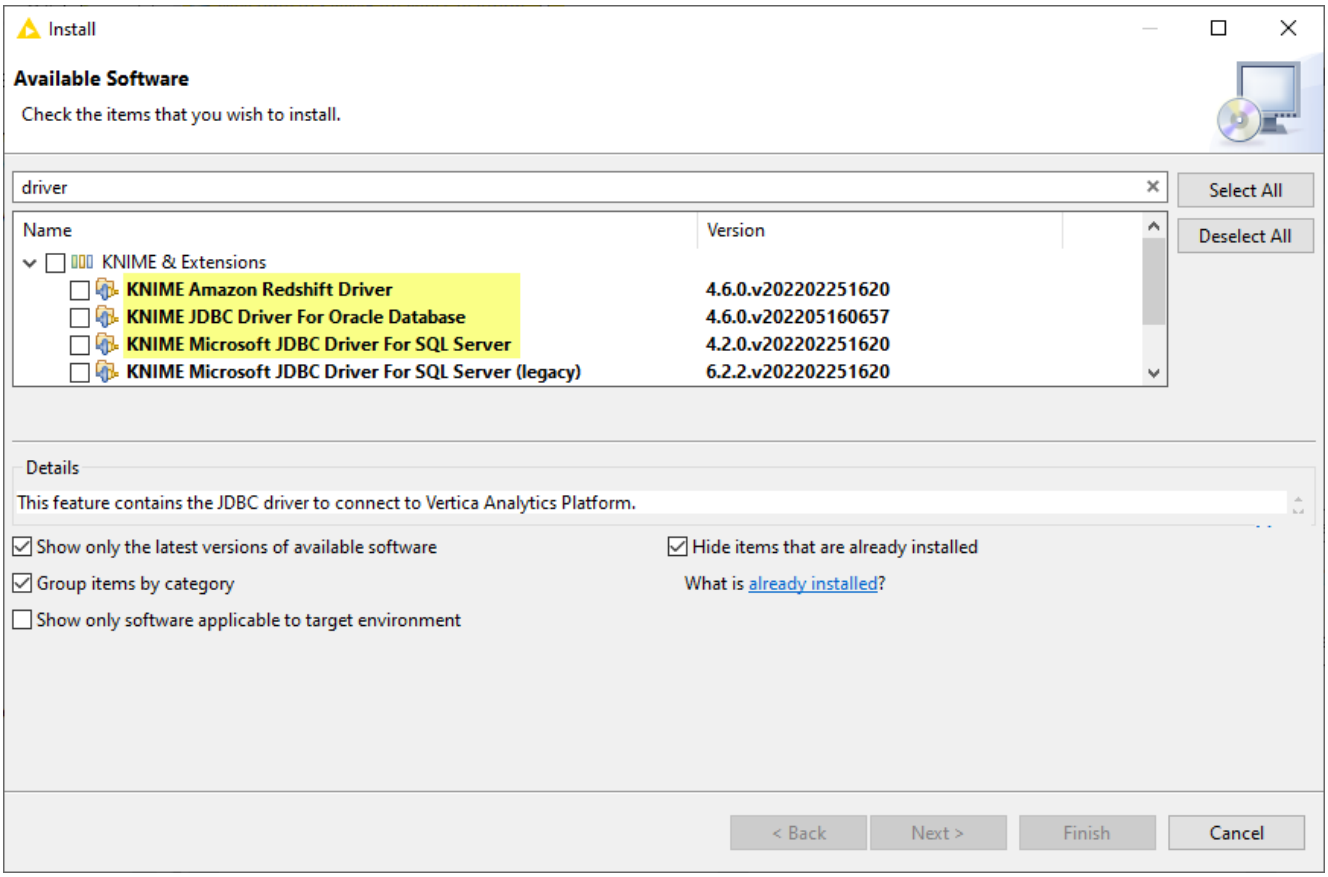

Figure 6. Install Window

Connecting to other databases

The generic *DB Connector* node can connect to arbitrary JDBC compliant databases. The most important node settings are described below.

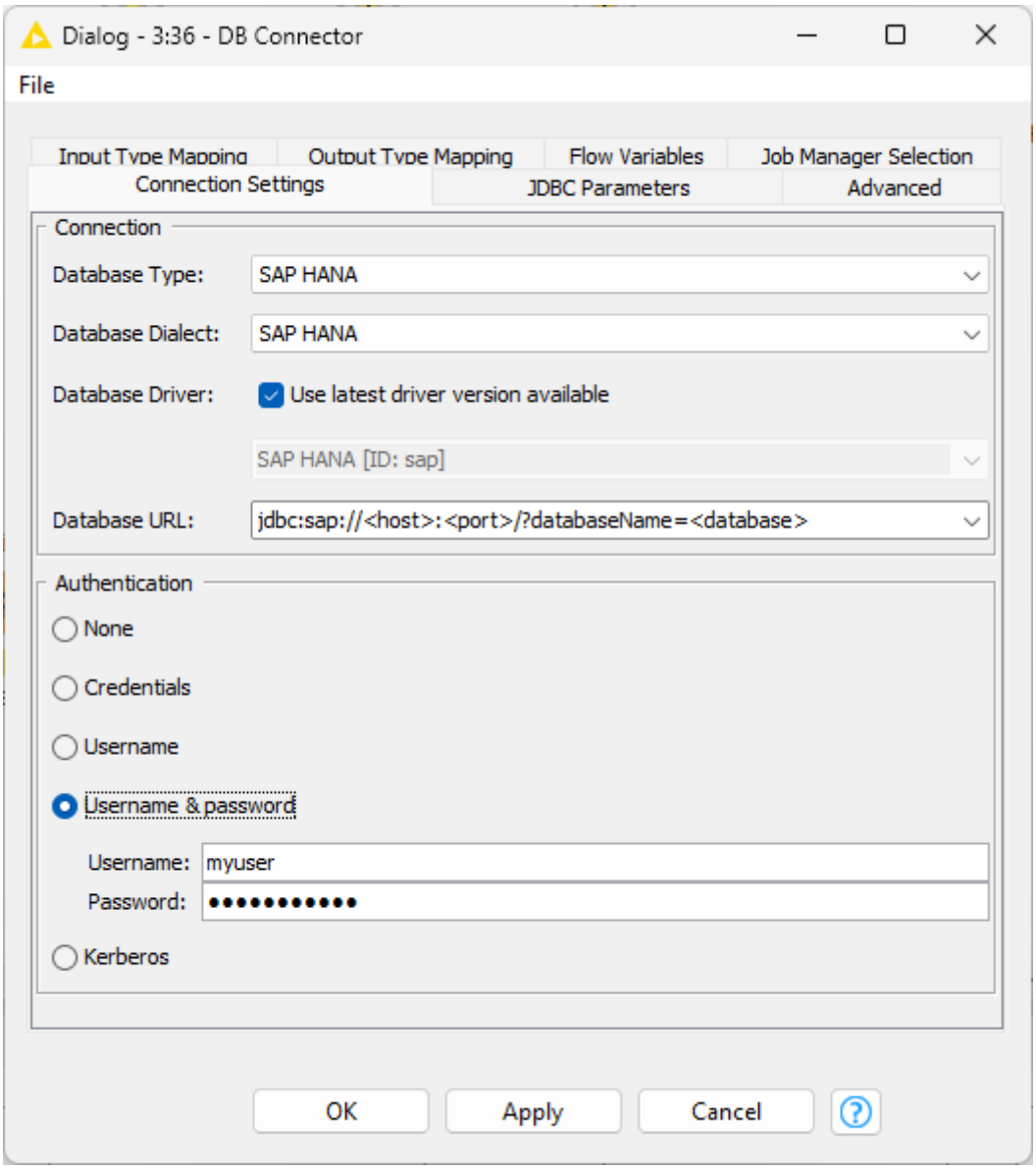

Figure 7. Database Connector configuration dialog

Database Type: Select the type of the 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 which defines how the SQL statements are generated.

Database Driver: Select an appropriate driver for your specific database or select the "Use latest driver version available" option in which case upon execution the node will automatically use the driver with the latest (highest) driver version that is available for the current database type. If there is no matching JDBC driver it first needs to be registered, see Register your own JDBC drivers. Only drivers that have been registered for the selected database type will be available for selection.

Database URL: A driver-specific JDBC URL. Enter the database information in the placeholder,

such as the host, port, and database name.

Authentication: Login credentials 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. For more information on Kerberos authentication, please refer to the Kerberos User Guide.

The selected database type and dialect determine which data types, statements such as insert, update, and aggregation functions are supported.

If you encounter an error while connecting to a third-party database, you can enable the *JDBC logger* option in the Advanced Tab. If this option is enabled all JDBC operations are written into the KNIME log which might help you to identify the problems. In order to tweak how KNIME interacts with your database e.g. quotes identifiers you can change the default settings under the Advanced Tab according to the settings of your database. For example, KNIME uses " as the default identifier quoting, which is not supported by default by some databases (e.g Informix). To solve this, simply change or remove the value of the *identifier delimiter* setting in the Advanced Tab.

Register your own JDBC drivers

For some databases KNIME Analytics Platform does not contain a ready-to-use JDBC driver. In these cases, it is necessary to first register a vendor-specific JDBC driver in KNIME Analytics Platform. Please consult your database vendor to obtain the JDBC driver. A list of some of the most popular JDBC drivers can be found below.

The JDBC driver has to be JDBC 4.1 or above compliant.

To set up JDBC drivers on KNIME Server, please refer to the section JDBC drivers on KNIME Hub and KNIME Server.

To register your vendor-specific JDBC driver, go to *File* → *Preferences* → *KNIME* → *Databases*.

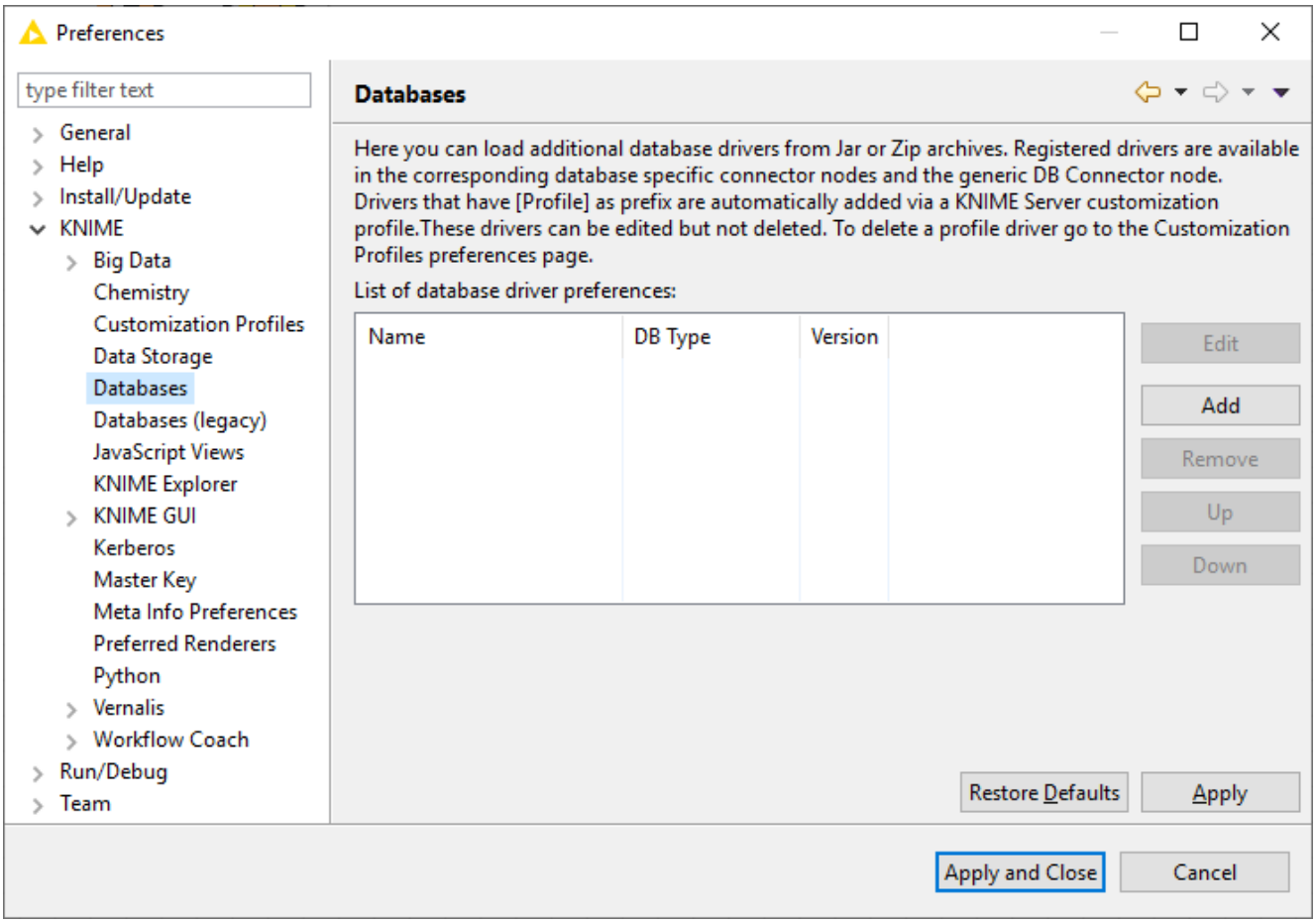

Figure 8. DB Preference page

Clicking *Add* will open 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*: The database type. If you select a specific database type e.g. MySQL the driver will be available for selection in the dedicated connector node e.g. MySQL Connector. However if your database is not on the list, you can choose *default*, which will provide you with all available parameters in the Advanced Tab. Drivers that are registered for the default type are only available in the generic DB Connector node.
- *Description*: Optional description of the JDBC driver.
- *URL template*: The JDBC driver connection URL format which is used in the dedicated connector nodes. If you select a database other than *default* in the *Database type*, the URL template will be preset with the default template for the selected database. Please refer to the URL Template syntax information below or the JDBC URL Template section for more information.

- *URL Template syntax information*: Clicking on the question mark will open an infobox which provides information about the URL template syntax in general. Additionally, if you select a database other than *default* in the *Database type*, one or more possible URL template examples will be provided for the selected database type, which you can copy and paste in the *URL template* field.
- *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.

If the JDBC driver requires native libraries e.g DLLs you need to put all of them into a single folder and then register this folder via the *Add directory* button in addition to the JDBC driver .jar file.

- *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.

If your database is available in the *Database type* drop down list, it is better to select it instead of setting it to *default*. Setting the *Database type* to *default* will allow you to only use the generic *DB Connector* node to connect to the database, even if there is a dedicated Connector node for that database.

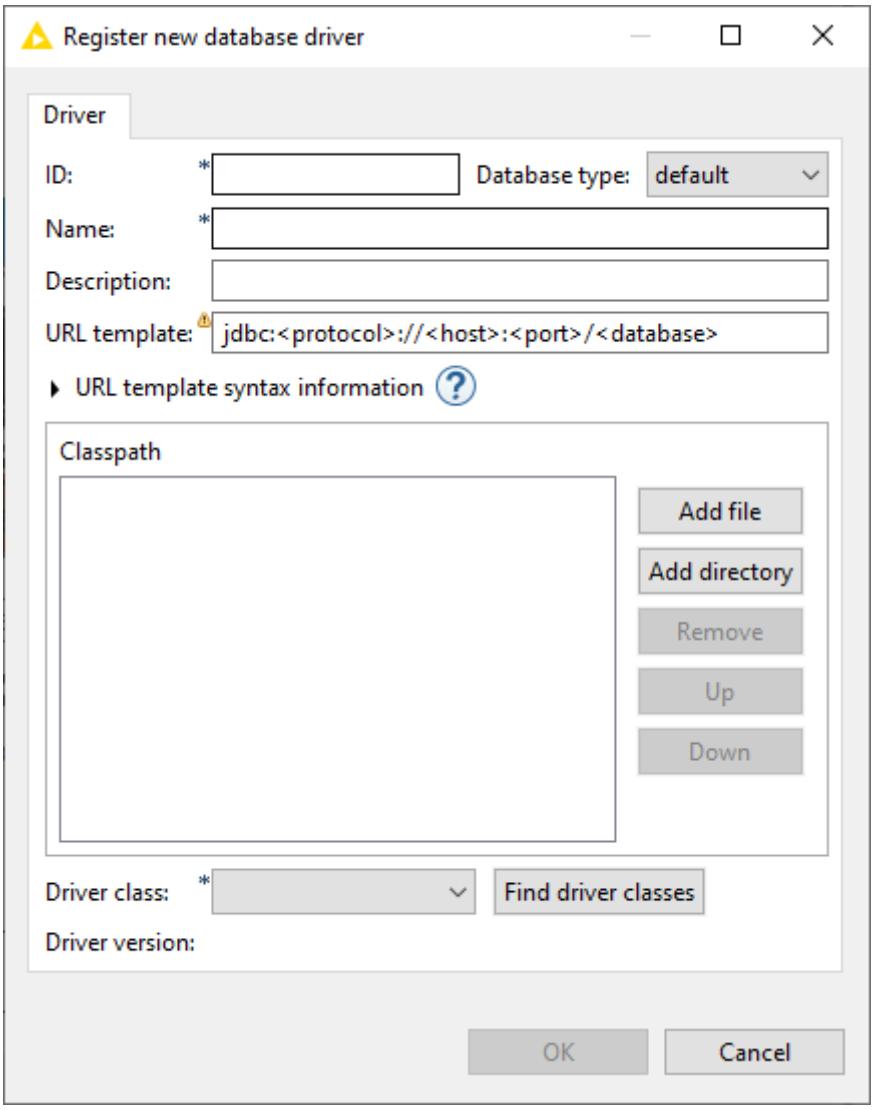

Figure 9. Register new database driver window

KNIME Server can distribute JDBC drivers automatically to all connected KNIME Analytics Platform clients (see JDBC drivers on KNIME Hub and KNIME Server).

JDBC URL Template

When registering a JDBC driver, you need to specify its JDBC URL template, which will be used by the dedicated Connector node to create the final database URL. For example, `jdbc:oracle:thin:@<host>:<port>/<database>` is a valid driver URL template for the Oracle thin driver. For most databases you don't have to find the suitable URL template by yourself, because the URL Template syntax information provides at least one URL template example for a database.

The values of the variables in the URL template, e.g `<host>`, `<port>`, or `<database>` can be specified in the configuration dialog of the corresponding Connector node.

Tokens:

- *Mandatory value (e.g. `<database>`)*: The referenced token must have a non-blank value. The name between the brackets must be a valid token name (see below for a list of supported tokens).
- *Optional value (e.g. `[database]`)*: The referenced token may have a blank value. The name between the brackets must be a valid token name (see below for a list of supported tokens).
- *Conditions (e.g. `[location=in-memory?mem:<database>]`)*: This is applicable for **file-based databases**, such as H2, or SQLite. The first `?` character separates the condition from the content that will only be included in the URL if the condition is true. The only explicit operator available currently is `=`, to test the exact value of a variable. The left operand must be a valid variable name, and the right operand the value the variable is required to have for the content to be included. The content may include mandatory and/or optional tokens (`<database>/[database]`), but no conditional parts. It is also possible to test if a variable is present. In order to do so, specifying the variable name e.g. database as the condition. E.g. `jdbc:mydb://<host>:<port>[database?/databaseName=<database>]` will result in `jdbc:mydb://localhost:10000/databaseName=db1` if the database name is specified in the node dialog otherwise it would be `jdbc:mydb://localhost:10000`.

For **server-based** databases, the following tokens are expected:

- *host*: The value of the Hostname field on the *Connection Settings* tab of a Connector node.
- *port*: The value of the Port field on the *Connection Settings* tab of a Connector node.
- *database*: The value of the Database name field on the *Connection Settings* tab of a Connector node.

For **file-based** databases, the following tokens are expected:

- *location*: The Location choice on the *Connection Settings* tab of a Connector node. The file value corresponds to the radio button next to *Path* being selected, and in-memory to the radio button next to *In-memory*. This variable can only be used in conditions.
- *file*: The value of the *Path* field on the *Connection Settings* tab of a Connector node. This variable is only valid if the value of the location is `file`.
- *database*: The value of the *In-memory* field on the *Connection Settings* tab of a Connector node. This variable is only valid if the value of the location is `in-memory`.

Field validation in the configuration dialog of a Connector node depends on whether the (included) tokens referencing them are mandatory or optional (see above).

List of common JDBC drivers

Below is a selected list of common database drivers you can add among others to KNIME Analytics Platform:

- Apache Derby
- Exasol
- Google BigQuery
- IBM DB2 / Informix
- SAP HANA

The list above only shows some example of database drivers that you can add. If your driver is not in the list above, it is still possible to add it to KNIME Analytics Platform.

Deprecated JDBC Drivers

With version 5.3.0, KNIME Analytics Platform introduced the concept of deprecating JDBC drivers. Older version of JDBC drivers become deprecated if they are not supported by the database vendor anymore or if they contain security vulnerabilities. Deprecated drivers are clearly marked in the node dialog and if selected will display a warning in the workflow editor.

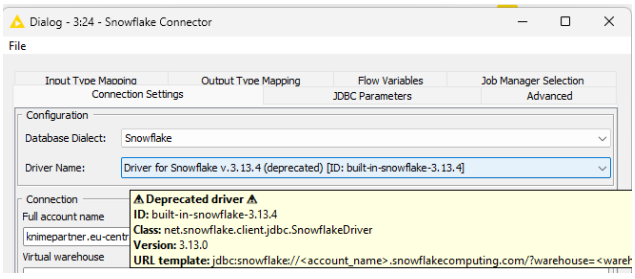

Figure 10. Tooltip of a deprecated driver in the driver list

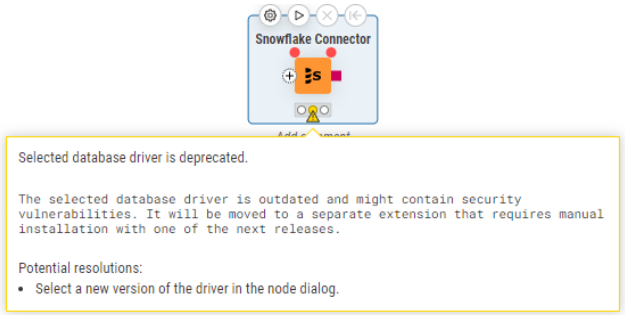

Figure 11. Node warning if a deprecated driver is selected

After a certain time, deprecated drivers will be removed from the platform. If such a driver is still referenced in a workflow, KNIME Analytics Platform will display an exception indicating

which extension needs to be installed to make the driver available again.

To install a deprecated driver extension you need to open the installation dialog as described https://docs.knime.com/latest/analytics_platform_installation_guide/index.html#*installing_extensions_and_integrations[here.] In the installation dialog deselect the _Group items by category* option, search for the mentioned extension and install it. After restarting KNIME Analytics Platform the deprecated driver will be available again.

In the example below the deprecated Snowflake driver is installed.

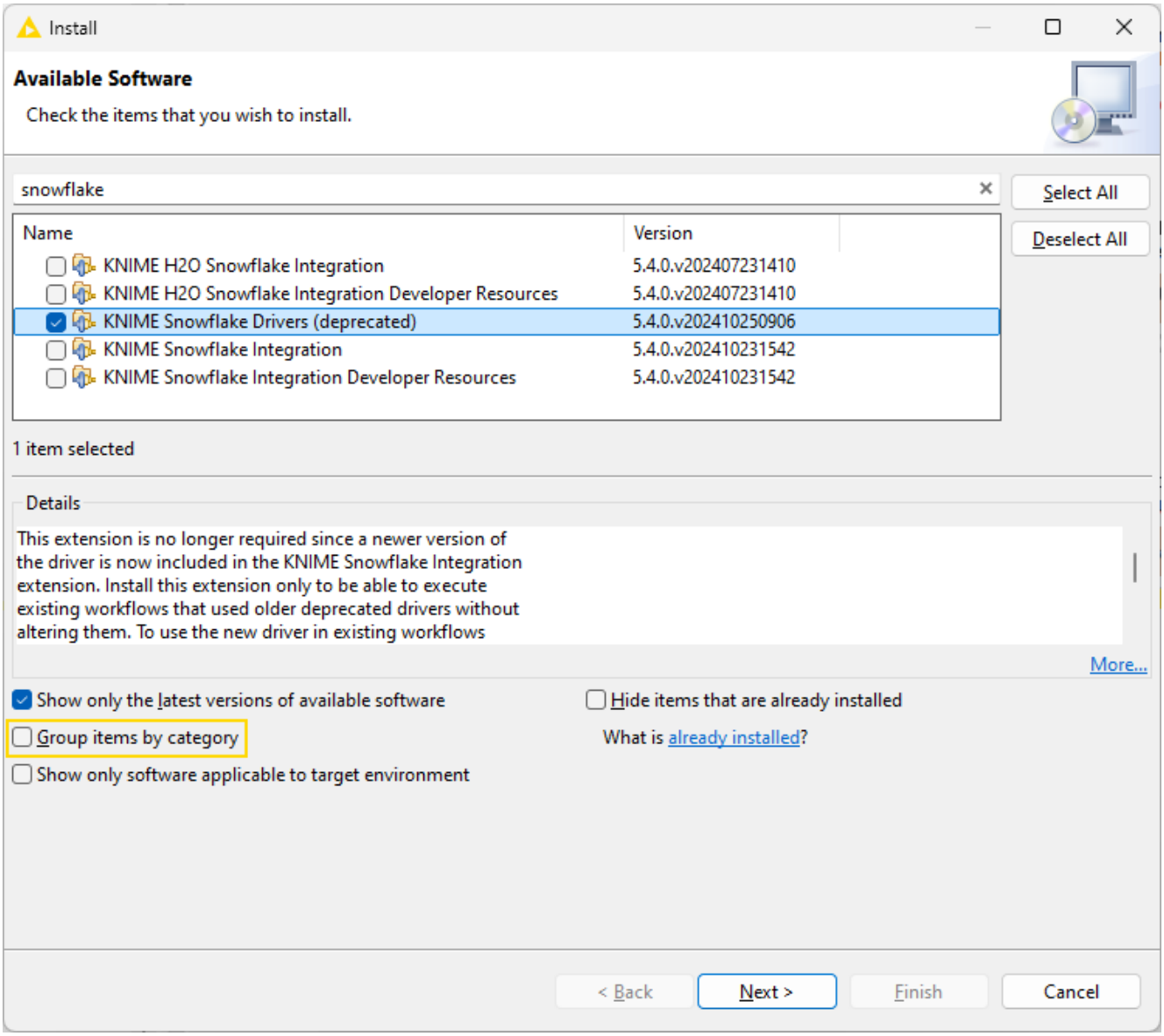

Figure 12. Install deprecated Snowflake drivers

Advanced Database Options

JDBC Parameters

The JDBC parameters allow you to define custom JDBC driver connection parameter. The value of a parameter can be a constant, variable, credential user, credential password, KNIME URL or path flow variable. In case of a path flow variable only standard file systems are supported but no connected file systems. For more information about the supported connection parameter please refer to your database vendor.

The figure below shows an example of SSL JDBC parameters with different variable types. You can set a boolean value to enable or disable *SSL*, you can also use a KNIME relative URL to point to the *SSLTrustStore* location, or use a credential input for the *trustStorePassword* parameter.

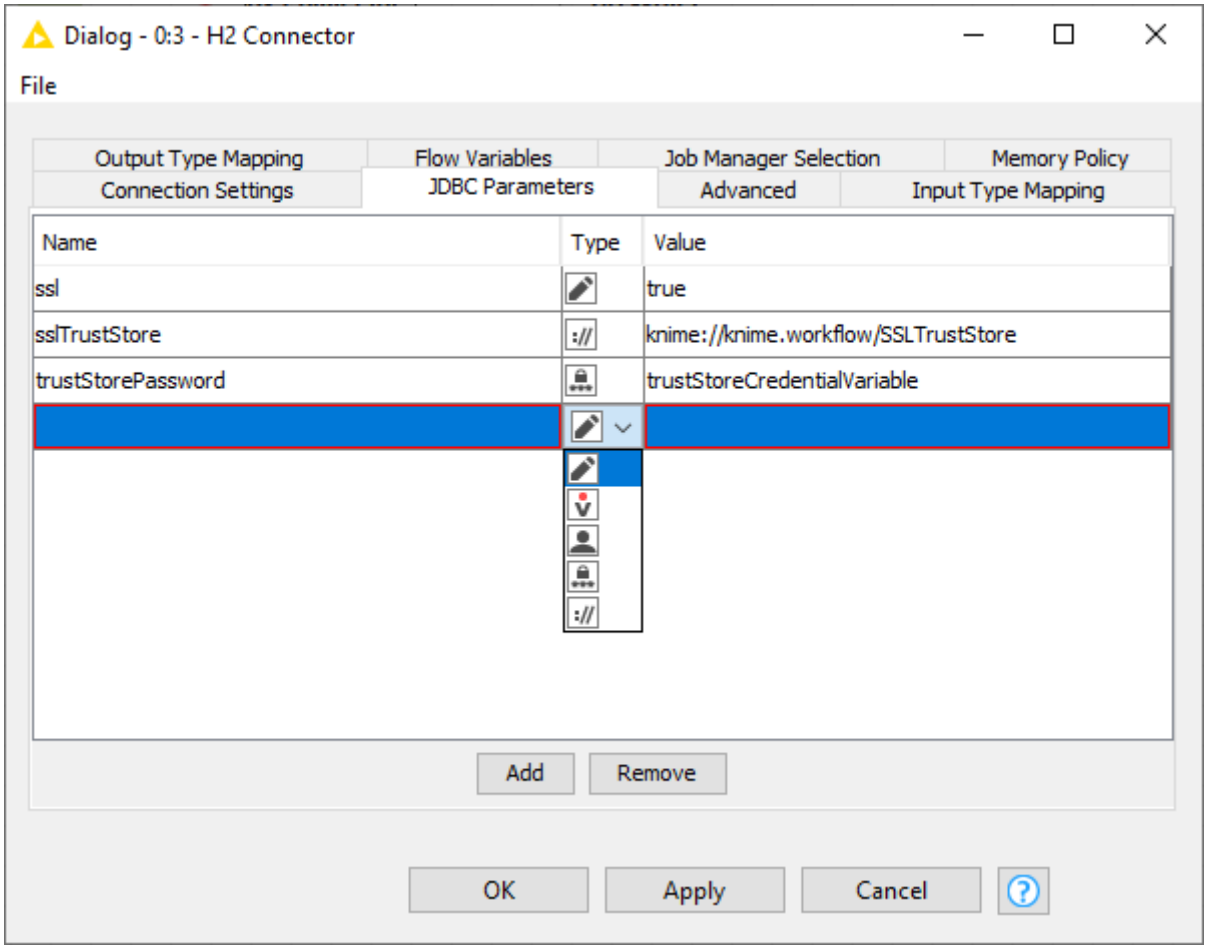

Figure 13. JDBC Parameters Tab

Please be aware that when connecting to PostgreSQL with SSL the key file has to first be converted to either `pkcs12` or `pkcs8` format. For more information about the supported driver properties see the PostgreSQL documentation.

Advanced Tab

The settings in the *Advanced* tab allow you to define KNIME framework properties such as connection handling, advanced SQL dialect settings or query logging options. This is the place where you can tweak how KNIME interacts with the database e.g. how the queries should be created that are send to the database. In the *Metadata* section you can also disable the metadata fetching during configuration of a node or alter the timeout when doing so which might be necessary if you are connected to a database that needs more time to compute the metadata of a created query or you are connected to it via a slow network.

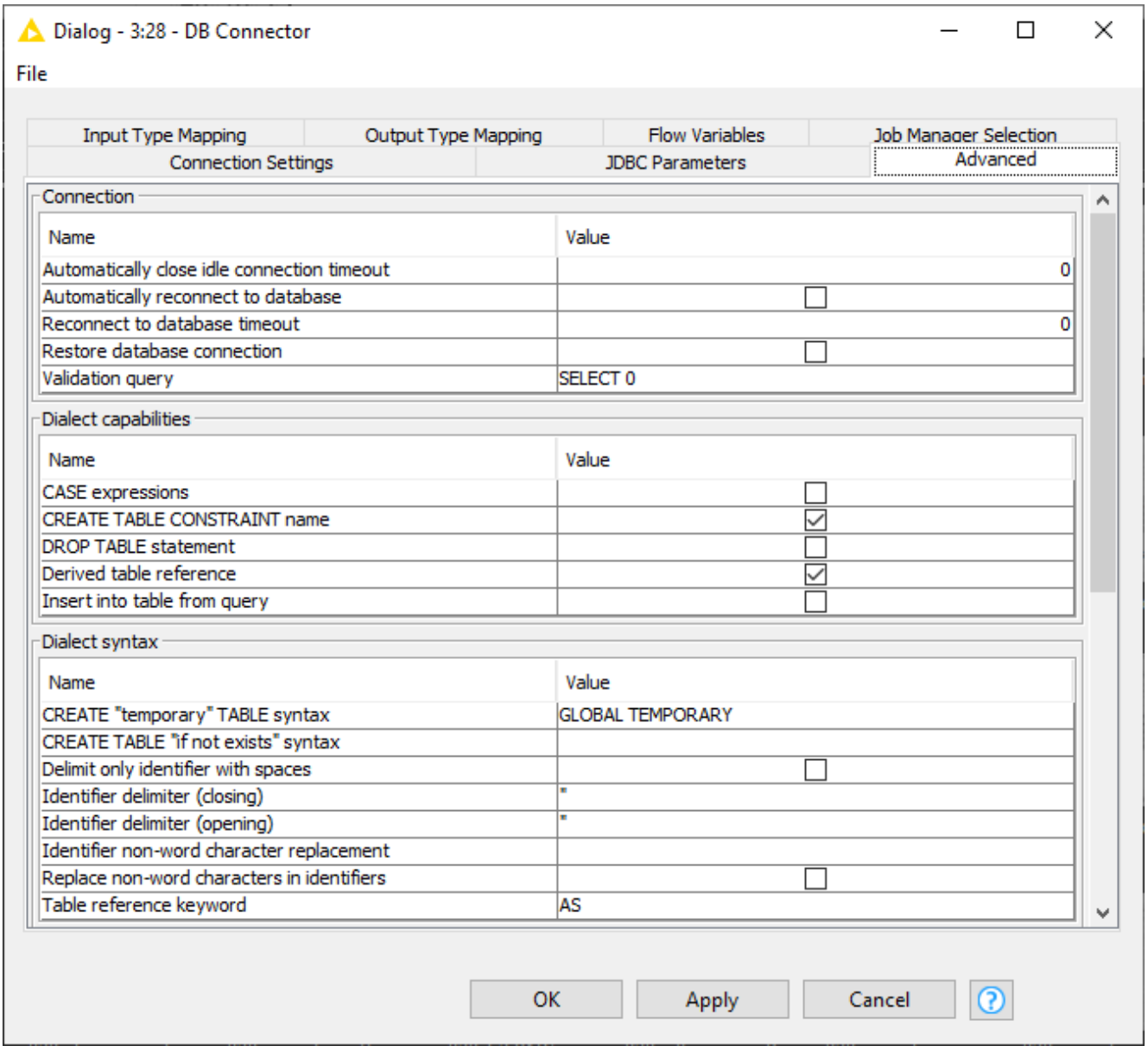

Figure 14. Advanced Tab

The full available options are described as follow:

Connection

- *Automatically close idle connection timeout*: Time interval in seconds that a database connection can remain idle before it gets closed automatically. A value of **0** disables the automatic closing of idle connections.
- *Automatically reconnect to database*: Enables or disables the reconnection to the database if the connection is invalid. Connection depending object will no longer exist after reconnection.
- *Reconnect to database timeout*: Time interval in seconds to wait before canceling the reconnection to the database. A value of **0** indicates the standard connection timeout.
- *Restore database connection*: Enables or disables the restoration of the database connection when an executed connector node is loaded.
- *Validation query*: The query to be executed for validating that a connection is ready for use. If no query is specified KNIME calls the Connection.isValid() method to validate the connection. Only errors are checked, no result is required.

Dialect capabilities

- *CASE expressions*: Whether CASE expressions are allowed in generated statements.
- *CREATE TABLE CONSTRAINT name*: Whether names can be defined for CONSTRAINT definitions in CREATE TABLE statements.
- *DROP TABLE statement*: Whether DROP TABLE statements are part of the language.
- *Derived table reference*: Whether table references can be derived tables.
- *Insert into table from query*: Whether insertion into a table via a select statement is supported, e.g. INSERT INTO T1 (C1) (SELECT C1 FROM T2).

Dialect syntax

- *CREATE "temporary" TABLE syntax*: The keyword or keywords for creating temporary tables.
- *CREATE TABLE "if not exists" syntax*: The syntax for the table creation statement condition "if not exists". If empty, no such statement will automatically be created, though the same behavior may still be non-atomically achieved by nodes.
- *Delimit only identifier with spaces*: If selected, only identifiers, e.g. columns or table names, with spaces are delimited.
- *Identifier delimiter (closing)*: Closing delimiter for identifier such as column and table name.
- *Identifier delimiter (opening)*: Opening delimiter for identifier such as column and table name.

- *Identifier non-word character replacement*: The replacement for non-word characters in identifiers when their replacement is enabled. An empty value results in the removal of non-word characters.
- *Replace non-word characters in identifiers*: Whether to replace non-word characters in identifiers, e.g. table or column names. Non-word characters include all characters other than alphanumeric characters (a-z, A-Z, 0-9) and underscore (_).
- *Table reference keyword*: The keyword before correlation names in table references.

JDBC logger

- *Enable*: Enables or disables logger for JDBC operations.

JDBC parameter

- *Append JDBC parameter to URL*: Enables or disables appending of parameter to the JDBC URL instead of passing them as properties.
- *Append user name and password to URL*: Enables or disables appending of the user name and password to the JDBC URL instead of passing them as properties.
- *JDBC URL initial parameter separator*: The character that indicates the start of the parameters in the JDBC URL.
- *JDBC URL parameter separator*: The character that separates two JDBC parameter in the JDBC URL.

JDBC statement cancellation

- *Enable*: Enables or disables JDBC statement cancellation attempts when node execution is canceled.
- *Node cancellation polling interval*: The amount of milliseconds to wait between two checking of whether the node execution has been canceled. Valid range: [100, 5000].

Metadata

- *Flatten sub-queries where possible*: Enables or disables sub-query flattening. If enabled sub-queries e.g. `SELECT * FROM (SELECT * FROM table) WHERE COL1 > 1` will become `SELECT * FROM table WHERE COL1 > 1`. By default this option is disabled since query flattening is usually the job of the database query optimizer. However some database either have performance problems when executing sub-queries or do not support sub-queries at all. In this case enabling the option might help. However not all queries are flatten so even enabled sub-queries might be send to the database.
- *List of table types to show in metadata browser*: Comma separated list of table types to show in metadata browser. Some databases e.g. SAP HANA support more than the

standard `TABLE` and `VIEW` type such as `CALC VIEW`, `HIERARCHY VIEW` and `JOIN VIEW`.

- *Retrieve in configure*: Enables or disables retrieving metadata in configure method for database nodes.
- *Retrieve in configure timeout*: Time interval in seconds to wait before canceling a metadata retrieval in configure method. Valid range: [1,).

Transaction

- *Enabled*: Enables or disables JDBC transaction operations.

Misc

- *Fail if WHERE clause contains any missing value*: Check every value of a WHERE clause (e.g., update, delete or merge) and fail if one is missing.
- *Fetch size*: Hint for the JDBC driver about the number of rows that should be fetched from the database when more rows are needed. Valid range: [0,).
- *Support multiple databases*: Enables or disables support for multiple databases in a single statement.

Dedicated DB connectors (e.g. Microsoft SQL Server Connector) and built-in drivers usually show only a subset of the above mentioned options since most options are predefined, such as whether the database supports CASE statements, etc.

Examples

In this section we will provide examples on how to connect to some widely-known databases.

Connecting to Oracle

The first step is to install the Oracle Database JDBC driver which is provided as a separate plug-in due to license restrictions. Please refer to Third-party Database Driver Plug-in for more information about the plug-in and how to install it.

It is also possible to use your own Oracle Database driver if required. For more details refer to the Register your own JDBC drivers section.

Once the driver is installed you can use the dedicated Oracle Connector node. Please refer to Connecting to predefined databases on how to connect using dedicated Connector nodes.

Kerberos authentication

To use this mode, you need to select *Kerberos* as authentication method in the *Connection Settings* tab of the Oracle Connector. For more information on Kerberos authentication, please refer to the Kerberos User Guide. In addition, you need to specify the following entry in the JDBC Parameters tab: *oracle.net.authentication_services* with value *(KERBEROS5)*. Please do not forget to put the value in brackets. For more details see the Oracle documentation.

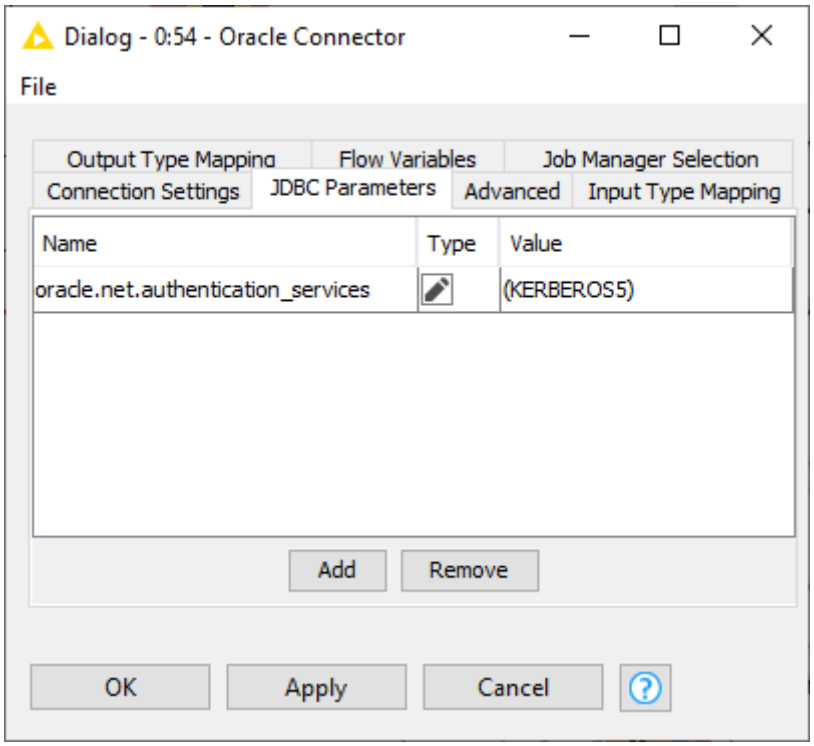

Figure 15. JDBC Parameters tab with Kerberos settings

Connecting to Databricks

To connect to Databricks, you need to install the KNIME Databricks Integration.

The next step is to download the Databricks Simba JDBC driver from the official website to your machine. Then go to *File* → *Preferences* → *KNIME* → *Databases*, and click *Add*.

> KNIME provides Apache Hive JDBC driver for Databricks which you can use as a fallback driver. But it is strongly recommended to use the official JDBC driver provided in the link above.

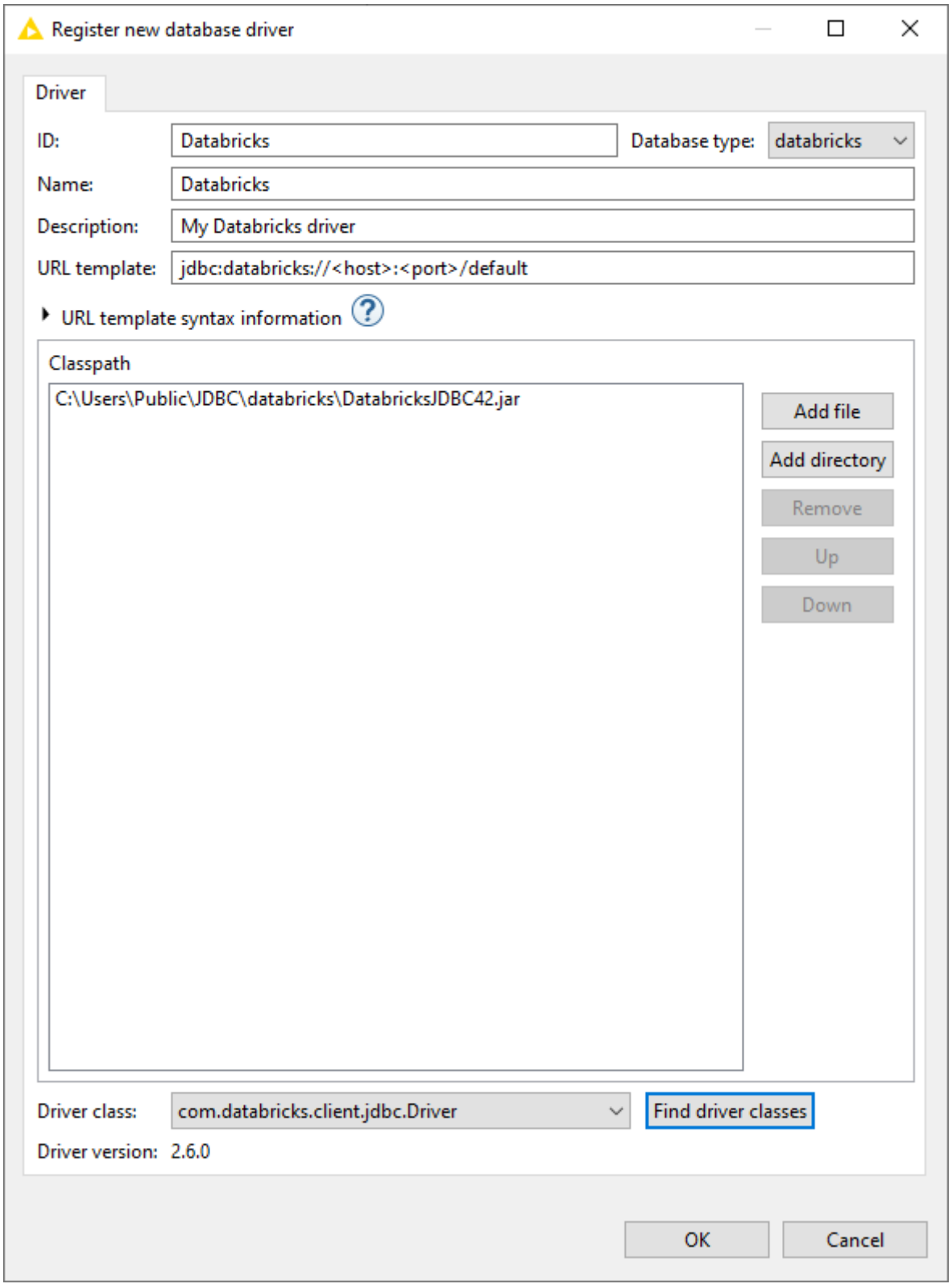

Figure 16. Register new Databricks driver

In the new database driver window, provide the following information:

- *ID*: Databricks, but you can enter your own driver ID as long as it only contains alphanumeric characters and underscores.

- *Name*: Databricks, but you can enter your own driver name.
- *Database type*: Databricks is available in the drop down list, so the database type is set to *databricks*.
- *Description*: My Databricks driver, for example.
- *URL template*: By selecting *databricks* in the *Database type*, the URL template is automatically preset to the default JDBC URL template for Databricks, i.e `jdbc:spark://<host>:<port>/default`. For more possible templates, simply click on the URL Template syntax information directly below. Please refer to the JDBC URL Template section for more information on the supported tokens e.g. host, port and database.
- *Classpath*: Click *Add file* to add the Databricks JDBC driver file. The path to the driver file will then appear in the Classpath area.
- *Driver class*: clicking *Find driver classes* will automatically detect all available JDBC driver classes and versions, which in this case is *com.simba.spark.jdbc4.Driver* in version 2.6.0.

After filling all the information, click *Ok*, and the newly added driver will appear in the database driver preferences table. Click *Apply and Close* to apply the changes.

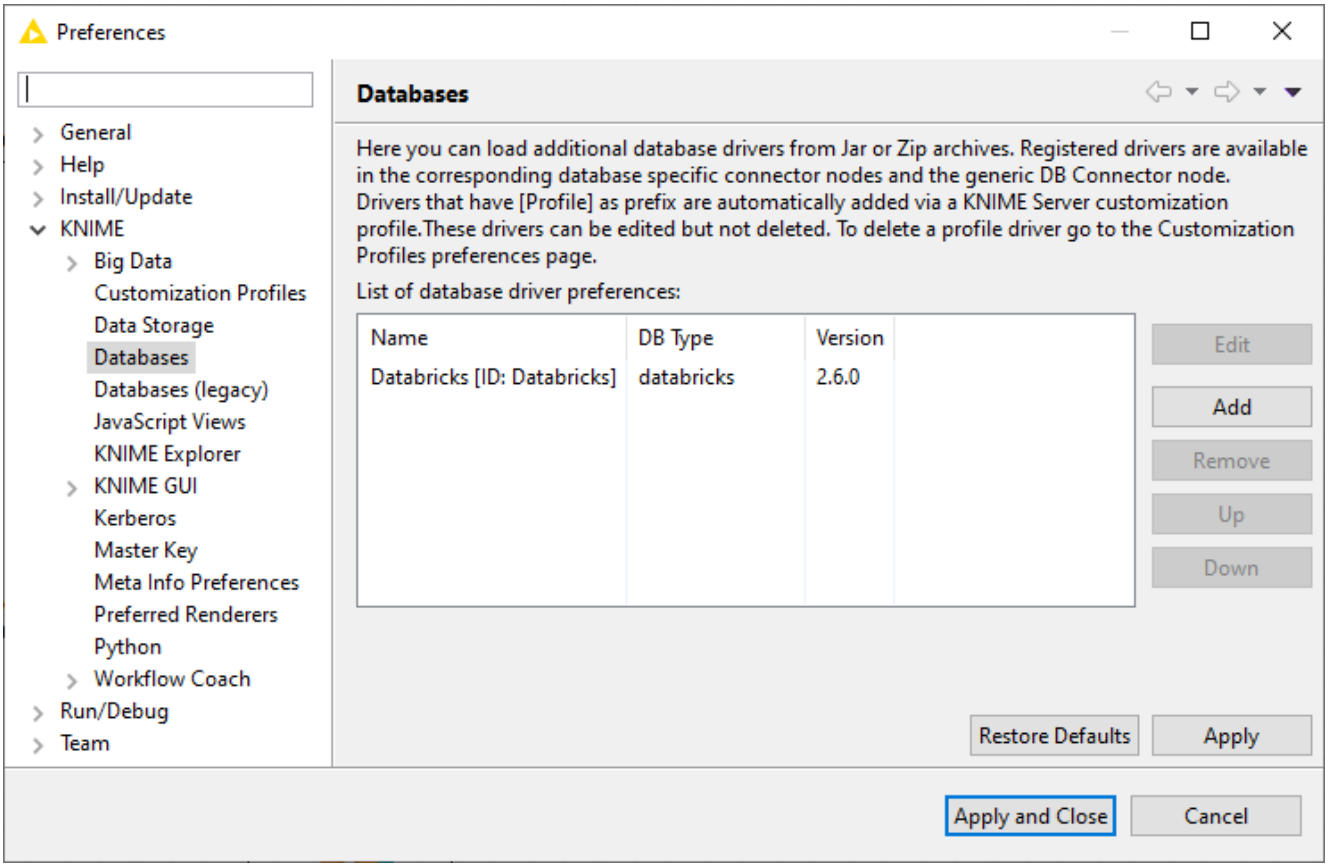

Figure 17. Database Preference page

To connect to Databricks, you need to first create a Databricks environment that is connected

to an existing Databricks cluster. To do this, use the *Create Databricks Environment* node. In the configuration window, you need to provide:

- *Databricks URL*: Full URL of the Databricks deployment, which is either *https://<account>.cloud.databricks.com* on AWS or *https://<region>.azuredatabricks.net* on Azure.
- *Cluster ID*: Unique identifier of a cluster in the Databricks workspace.
- *Workspace ID*: Workspace ID for Databricks on Azure, leave blank on AWS.

In the *DB Port* → *Driver* tab, you can select the database driver, which in this case is the Databricks Simba JDBC driver we have registered previously.

For the authentication, Databricks strongly recommends using tokens. Please refer to the authentication in Databricks AWS or Azure documentation for more information about personal access token.

Connecting to Google BigQuery

To connect to BigQuery, you need to install the KNIME BigQuery Extension.

Due to license restrictions the BigQuery JDBC driver is not part of KNIME Analytics Platform and needs to be downloaded and registered separately. To download the BigQuery JDBC driver please visit the official website. Then go to *File* → *Preferences* → *KNIME* → *Databases*, and click *Add*.

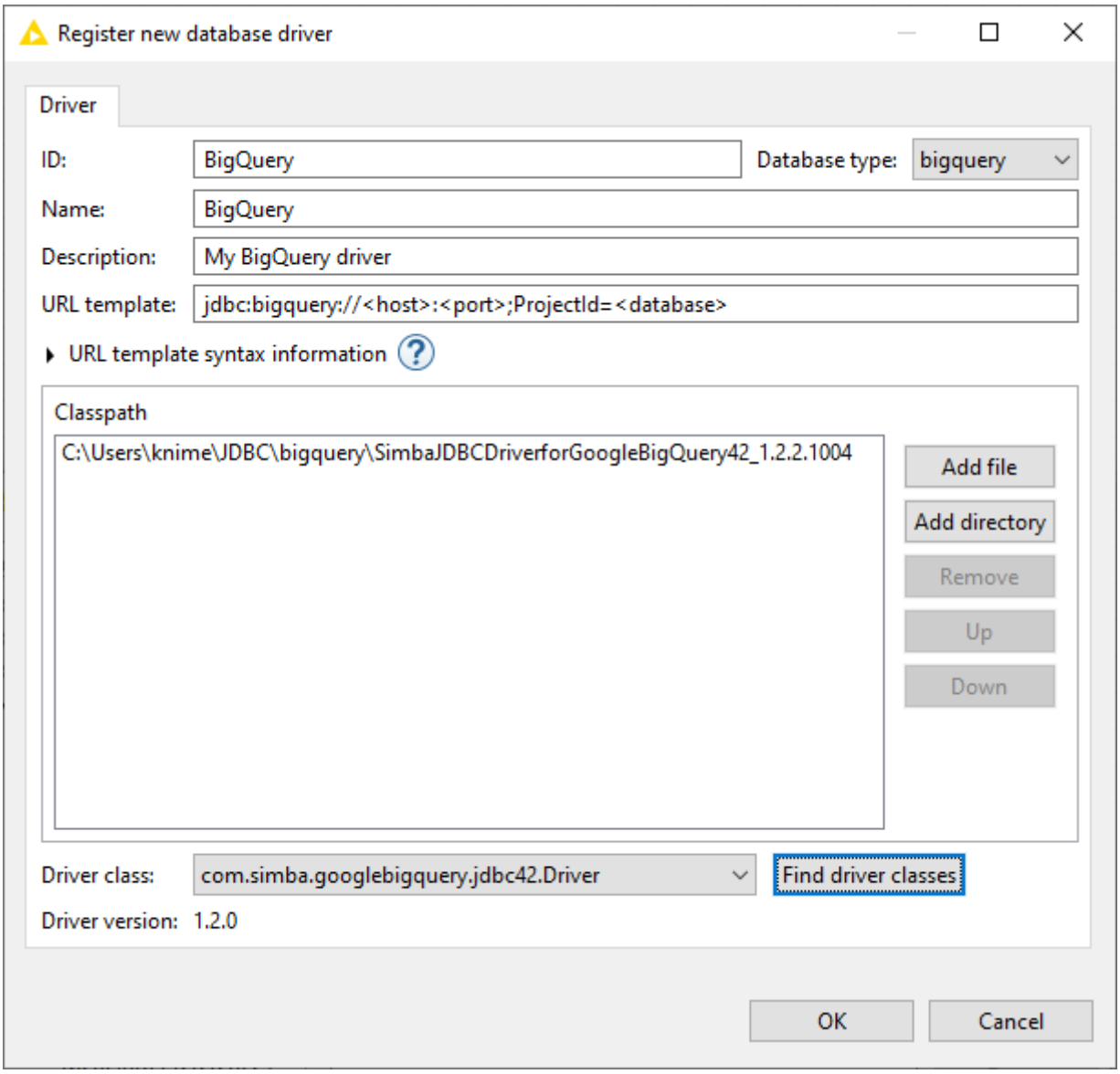

Figure 18. Register new BigQuery driver

In the new database driver window, provide the following information:

- *ID*: BigQuery, but you can enter your own driver ID as long as it only contains alphanumeric characters and underscores.
- *Name*: BigQuery, but you can enter your own driver name.
- *Database type*: BigQuery is available in the drop down list, so the database type is set to *bigquery*.
- *Description*: My Google BigQuery driver, for example.
- *URL template*: By selecting *bigquery* in the *Database type*, the URL template is automatically preset to the default JDBC URL template for BigQuery, i.e `jdbc:bigquery://<host>:<port>;ProjectId=<database>`. Please refer to the URL Template syntax information directly below or the JDBC URL Template section for more information on the supported tokens e.g. host, port and database.

- *Classpath*: Click *Add directory* to add the BigQuery JDBC driver directory that contains the GoogleBigQueryJDBC42.jar file and all required companion libraries. The path to the driver directory will then appear in the Classpath area.
- *Driver class*: clicking *Find driver classes* will automatically detect all available JDBC driver classes and versions, which in this case is *com.simba.googlebigquery.jdbc42.Driver* in version 1.2.0.

Make sure to include the directory with the GoogleBigQueryJDBC42.jar file and all required companion libraries.

After filling all the information, click *Ok*, and the newly added driver will appear in the database driver preferences table. Click *Apply and Close* to apply the changes.

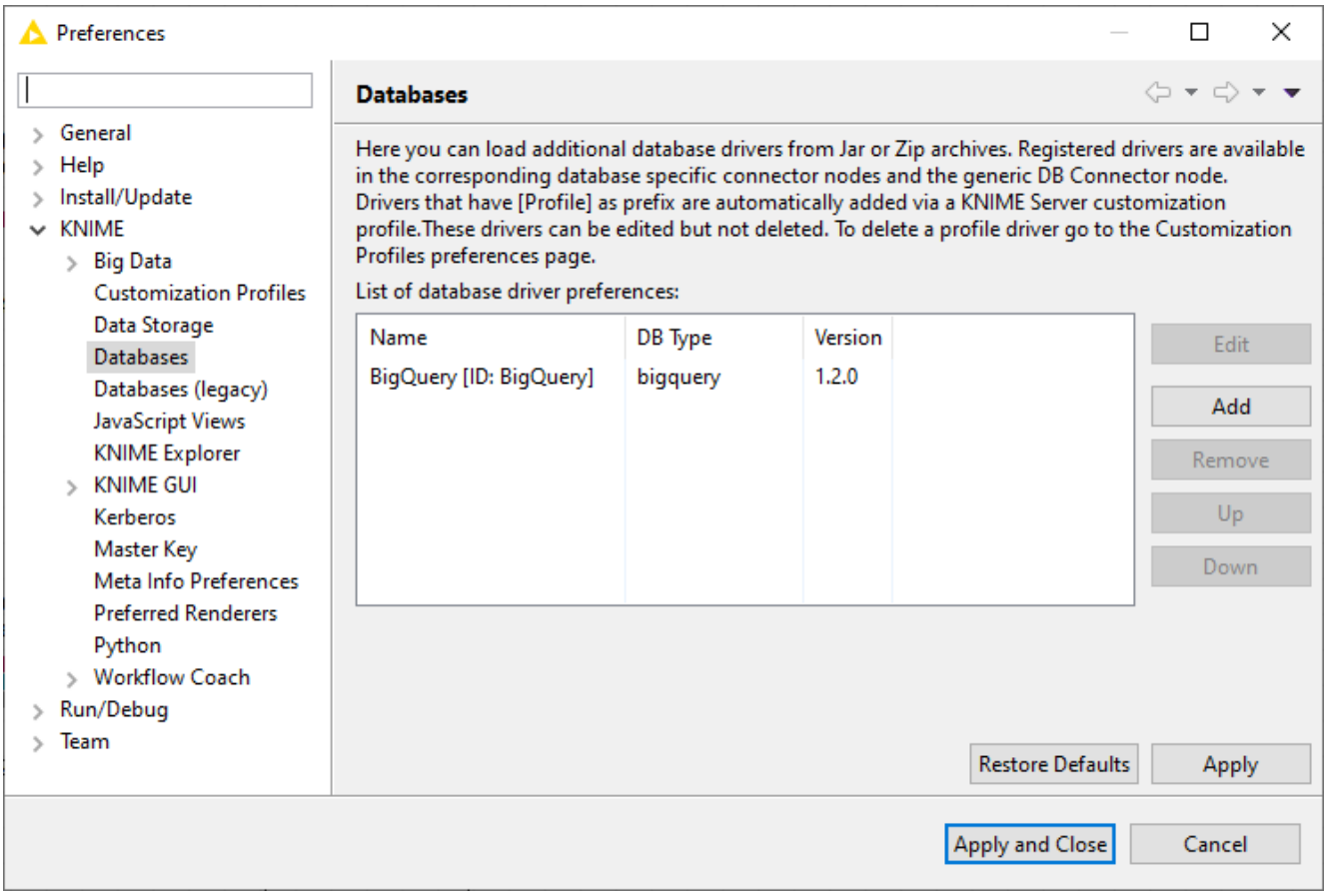

Figure 19. Database Preference page

To connect to the BigQuery server, we suggest to use the *Google Authentication (API Key)* node to authenticate and create a connection to the Google APIs, and then the *Google BigQuery Connector* node to connect to BigQuery. As an alternative you can also specify the driver specific authentication information via the JDBC Parameters tab.

Figure 20. Connecting to BigQuery

In the configuration window of the *Google Authentication (API Key)* node, you need to provide:

- *Service account email*: Email address of the service account. Please see the BigQuery documentation for more information on how to create a service account.
- *P12 key file location*: Path to the private P12 key file. While creating a service account email, you can create a service account key in the form of either JSON or P12 key/credentials. Note that only P12 keys are supported here.
- *Scopes*: The scopes that will be granted for this connection. Please see the Google documentation on the list of available BigQuery API scopes. For example, selecting *Google BigQuery Connection* allows you to view and manage your data in Google BigQuery.

Only P12 keys are supported in the *Google Authentication (API Key)* node!

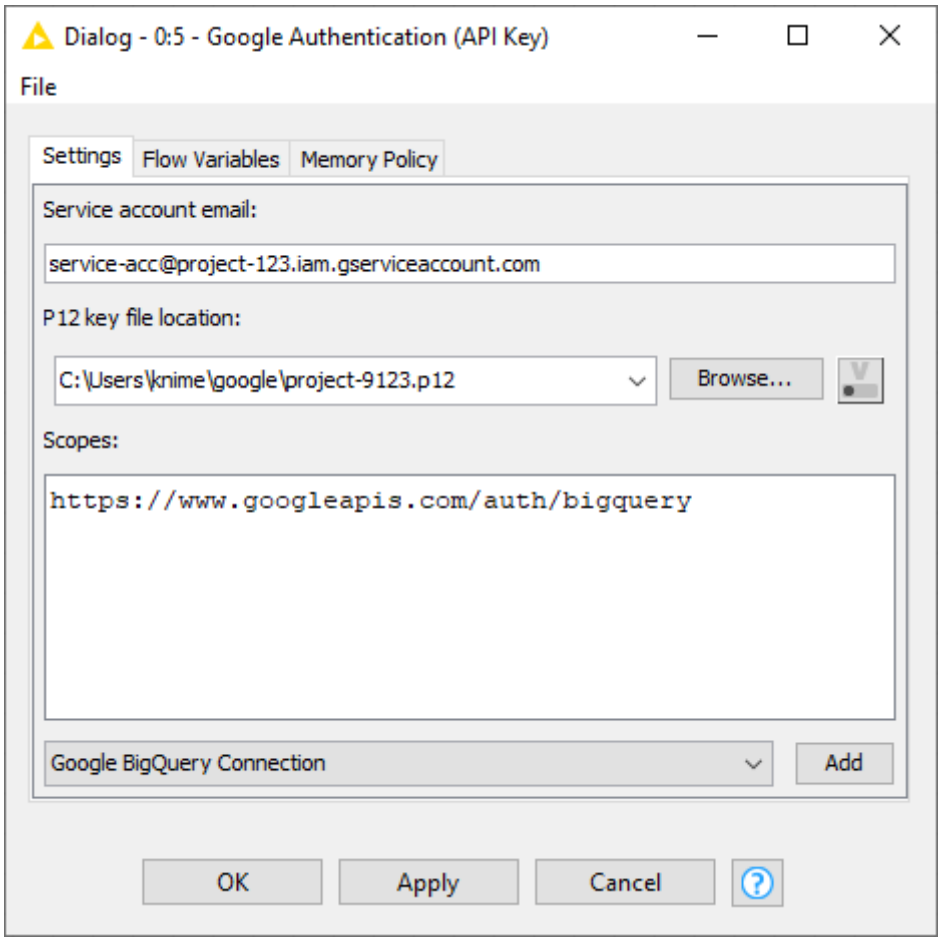

Figure 21. Configuration Window of Google Authentication (API Key) Node

After the connection is successfully established, it can be used as the input *Google Service Connection* for the *Google BigQuery Connector* node. The configuration window contains as follows:

- *Database Driver*: The name of the BigQuery driver we have given earlier when we registered the driver. In our example it is *BigQuery*.
- *Hostname*: The hostname (or IP address) of a Google BigQuery server.
- *Port*: The port on which the Google BigQuery server is listening. The default port is 443.
- *Database Name*: The name (project ID) of the database you want to connect to.

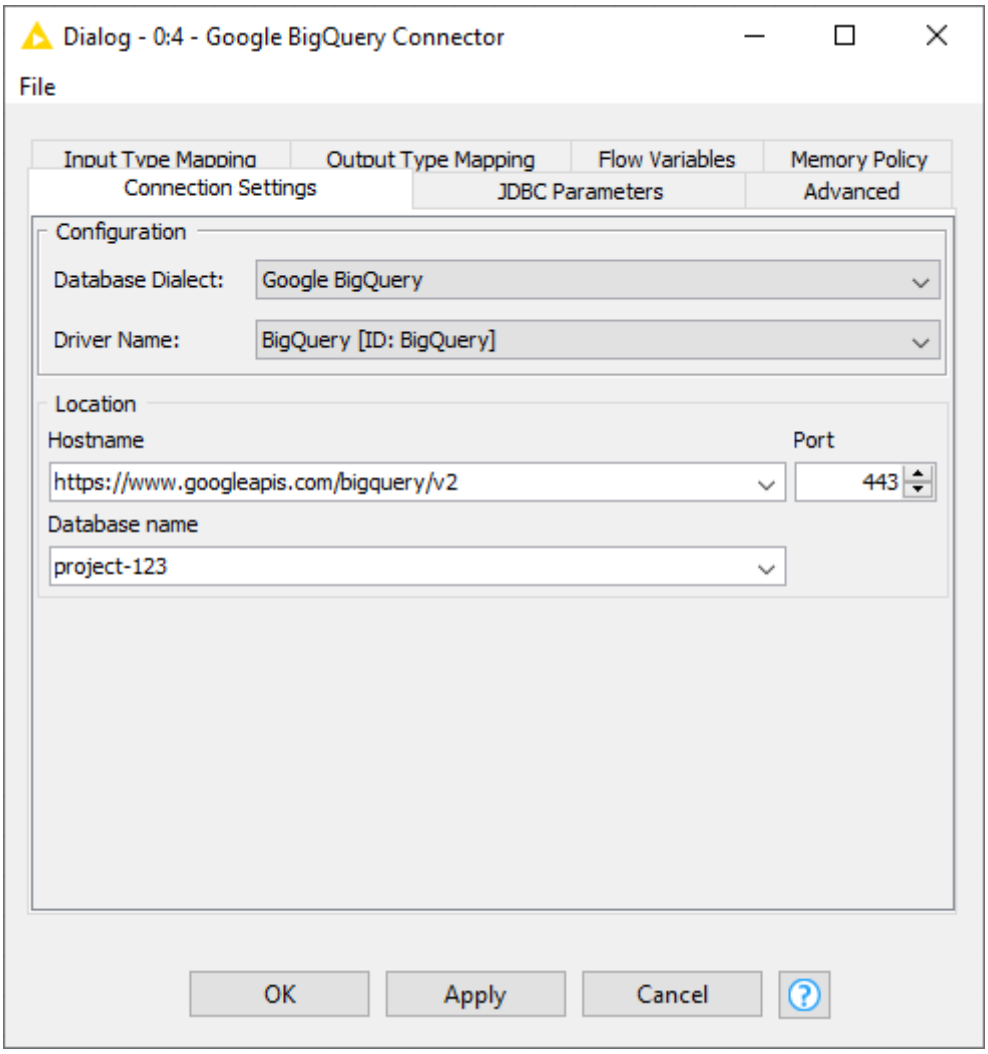

Figure 22. Configuration Window of Google BigQuery Connector Node

Connecting to Microsoft SQL Server

The dedicated *Microsoft SQL Server Connector* node is bundled by default with the jTDS for Microsoft SQL Server driver. If you want to use the official driver for Microsoft SQL Server instead, KNIME provides an additional plug-in to install the driver. Please refer to Third-party Database Driver Plug-in for more information about the plug-in and how to install it.

It is also possible to use your own Microsoft SQL Server driver. To register your own driver please refer to the Register your own JDBC drivers section. However, the Microsoft SQL Server driver might require several native libraries, such as DLLs. In that case you need to copy all the required native libraries into a single folder and then register this folder via the *Add directory* button in addition to the JDBC `.jar` file in the database driver window. This step is not required if you use the provided jTDS for Microsoft SQL Server driver or the official driver for Microsoft SQL Server installed through the plug-in.

All necessary files such as the sqljdbc_auth.dll or the ntlmauth.dll are part of the provided drivers.

After installing the JDBC driver, you can use the *Microsoft SQL Server Connector* node to start connecting to Microsoft SQL Server. Please refer to Connecting to predefined databases for more information on how to connect using dedicated Connector nodes.

Microsoft SQL Server supports the so-called Windows native authentication mode. If you want to use this mode, simply select *None/native authentication* in the *Authentication* setting in the configuration window. The following sections explain how to use this mode depending on which driver you use.

Windows native authentication only works on Windows. If you are running workflows on KNIME Hub or KNIME Server the KNIME Executor will most likely run on Linux which requires user login as well as additional JDBC parameters. For details see the following section.

Windows native authentication (NTLM) using the official driver for Microsoft SQL Server

To use this mode with the provided official driver for Microsoft SQL Server, KNIME Analytics Platform needs to run on a Windows machine and you need to be logged in a Windows domain that is accepted by the Microsoft SQL Server you are connecting to. In addition, you need to specify the following entry in the *JDBC Parameters* tab:

```
integratedSecurity=true
```

For more details see the Microsoft documentation.

If KNIME Analytics Platform runs on a non-Windows machine, you need to provide the user name and password of the Windows domain user you want to use for authentication. To do so please select either the *Credentials* option or the *Username & password* option.

In addition, you need to specify the following entries in the *JDBC Parameters* tab:

```
authenticationScheme=NTLM
domain=<<Windows domain name>>
```

For more details about the required parameters see the Microsoft JDBC driver documentation.

Windows native authentication (NTLM) using the jTDS driver for Microsoft SQL Server

If you are using the provided jTDS for Microsoft SQL Server driver and KNIME Analytics Platform runs on a Windows machine that is logged into a Windows domain that is accepted by the Microsoft SQL Server you are connecting to, then you don't need to specify any JDBC parameters.

If KNIME Analytics Platform runs on a non-Windows machine, you need to provide the user name and password of the Windows domain user you want to use for authentication. To do so please select either the *Credentials* option or the *Username & password* option. In addition, you need to specify the following entry in the *JDBC Parameters* tab:

```
domain=<<Windows domain name>>
```

For more details see the description of the domain property in the jTDS FAQ or see the README.sso.

Kerberos authentication using the official driver for Microsoft SQL Server

To use this mode with the provided official driver for Microsoft SQL Server, simply select *Kerberos* in the *Authentication* setting in the configuration window. In addition, you need to specify the following entries in the *JDBC Parameters* tab:

```
authenticationScheme=JavaKerberos
integratedSecurity=true
```

For more details see the Microsoft documentation.

Connecting to SQL Pools in Azure Synapse Analytics

You can use the dedicated *Microsoft SQL Server Connector node* to connect to a dedicated SQL pool or serverless SQL pool in Azure Synapse Analytics. Before connecting you need to install the official driver for Microsoft SQL Server that is provided as an additional plug-in. Please refer to Third-party Database Driver Plug-in for more information about the plug-in and how to install it. It is also possible to use your own Microsoft SQL Server driver. To register your own driver please refer to the Register your own JDBC drivers section.

To connect you need to obtain the serverless and/or dedicated SQL endpoint and a login user e.g. the SQL administration user first. This information is available via the Synapse workspace view in Azure Portal.

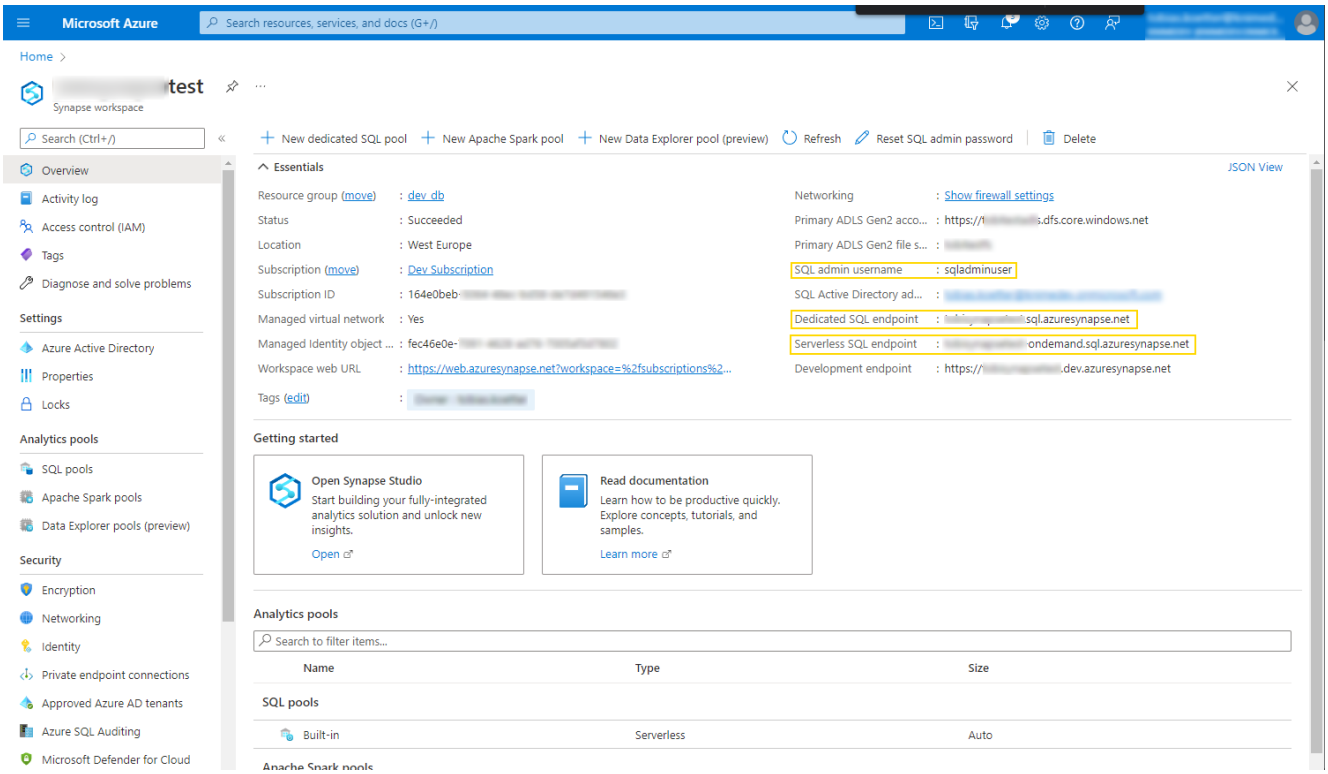

Figure 23. Synapse workspace view in Azure portal

Once you have the official driver installed and all necessary information available you can configure the Microsoft SQL Server Connector node. To do so open the node dialog and enter the serverless or dedicated SQL endpoint as *hostname* with *master* as default database. Enter the login information into the authentication section as shown below. Instead of a username and password you can also use Microsoft Entra ID login if this is configured in your Synapse workspace using the Microsoft Authentication node and the dynamic input port of the Microsoft SQL Server Connector.

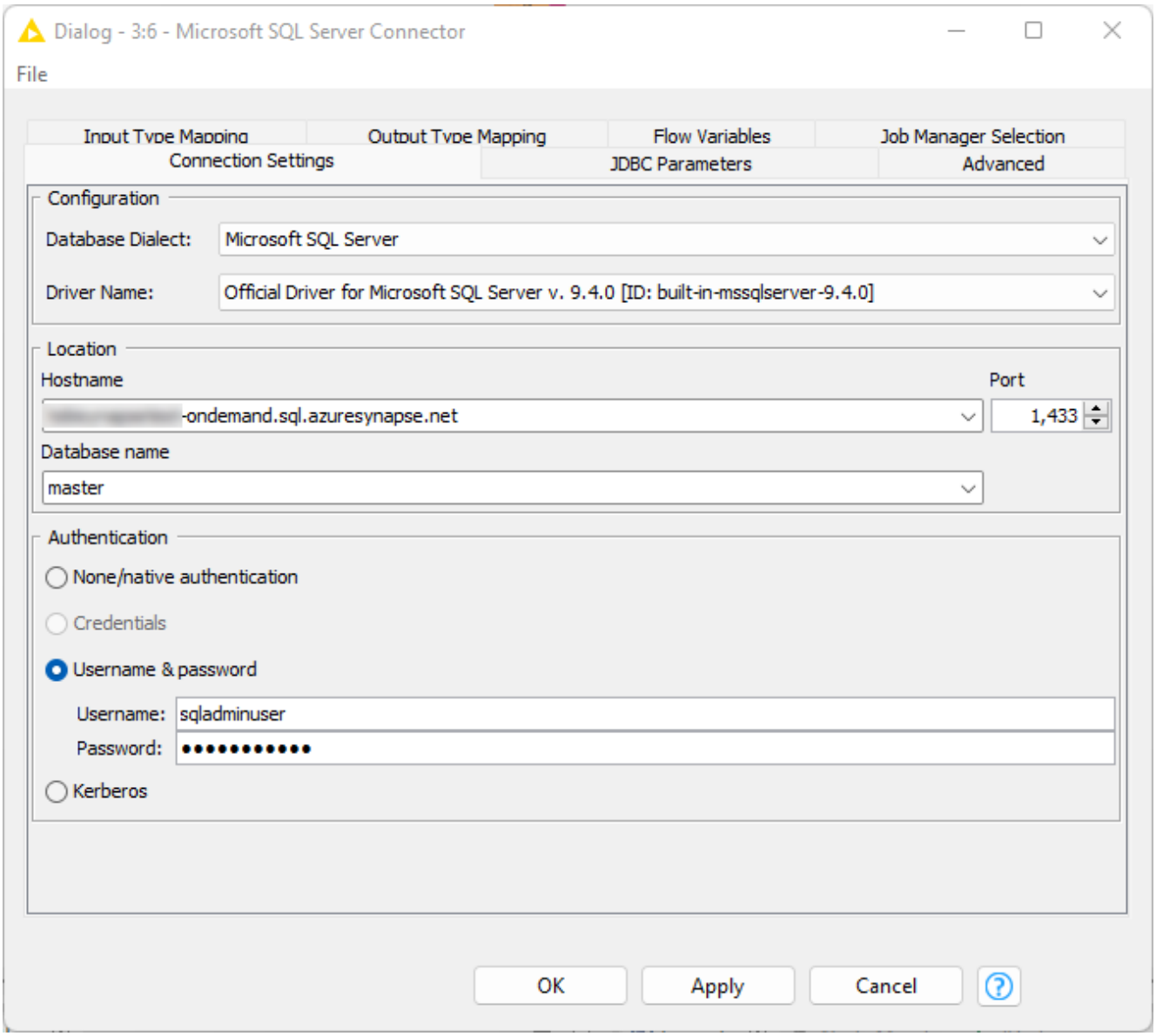

Figure 24. Microsoft SQL Server Connector dialog with connection settings

In addition you need to specify the following JDBC parameters via the *JDBC Parameters* tab:

- *TrustServerCertificate*: false
- *encrypt*: true
- *hostNameInCertificate*: *.sql.azuresynapse.net

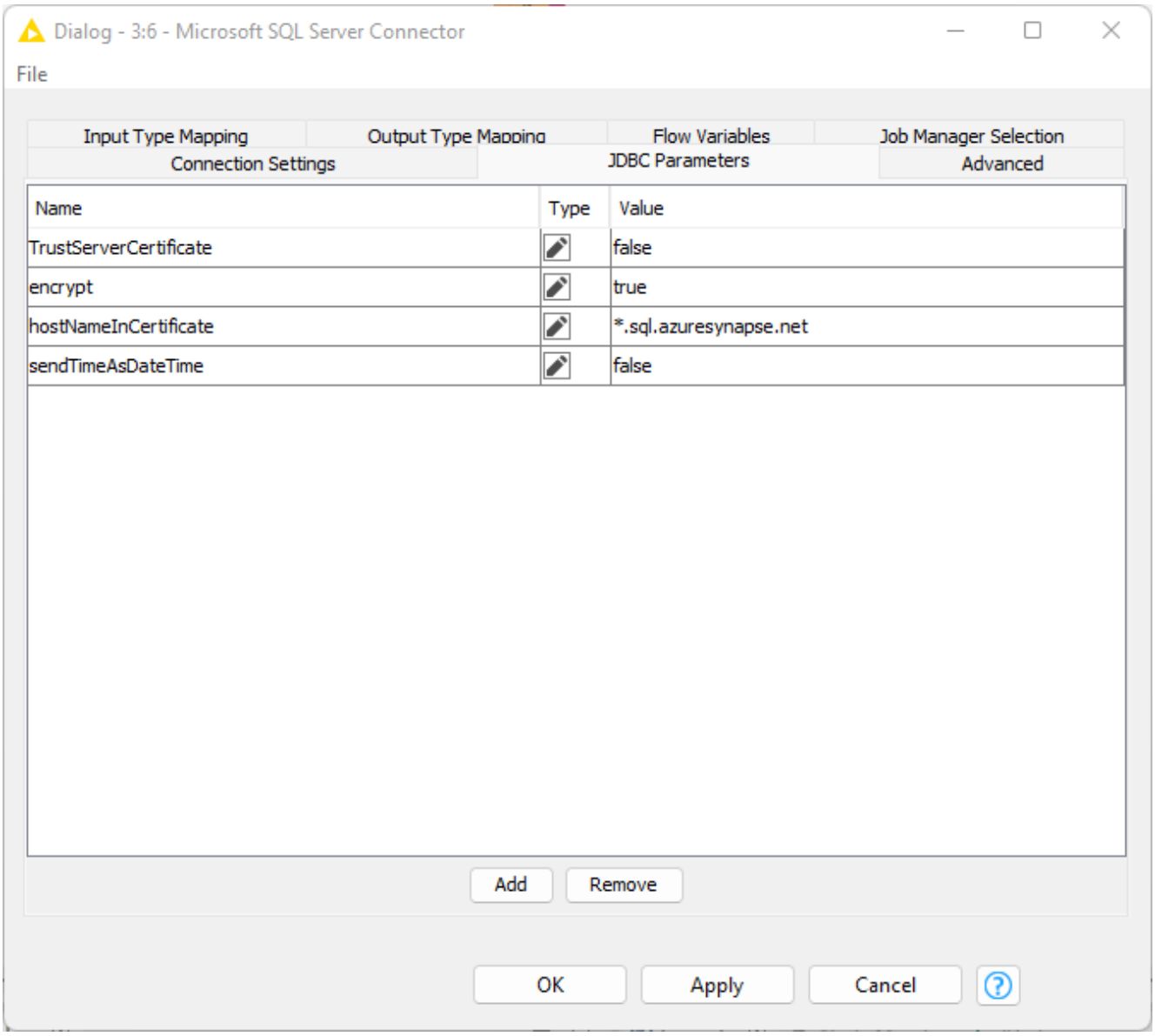

Figure 25. JDBC parameters required for Azure Synapse

For more details about the JDBC parameters see the Microsoft documentation.

Finally, for **serverless SQL pools** you need to disabled transaction since they are not support via the *Advanced Tab* by unselecting the *Enabled* value in the *Transaction* section.

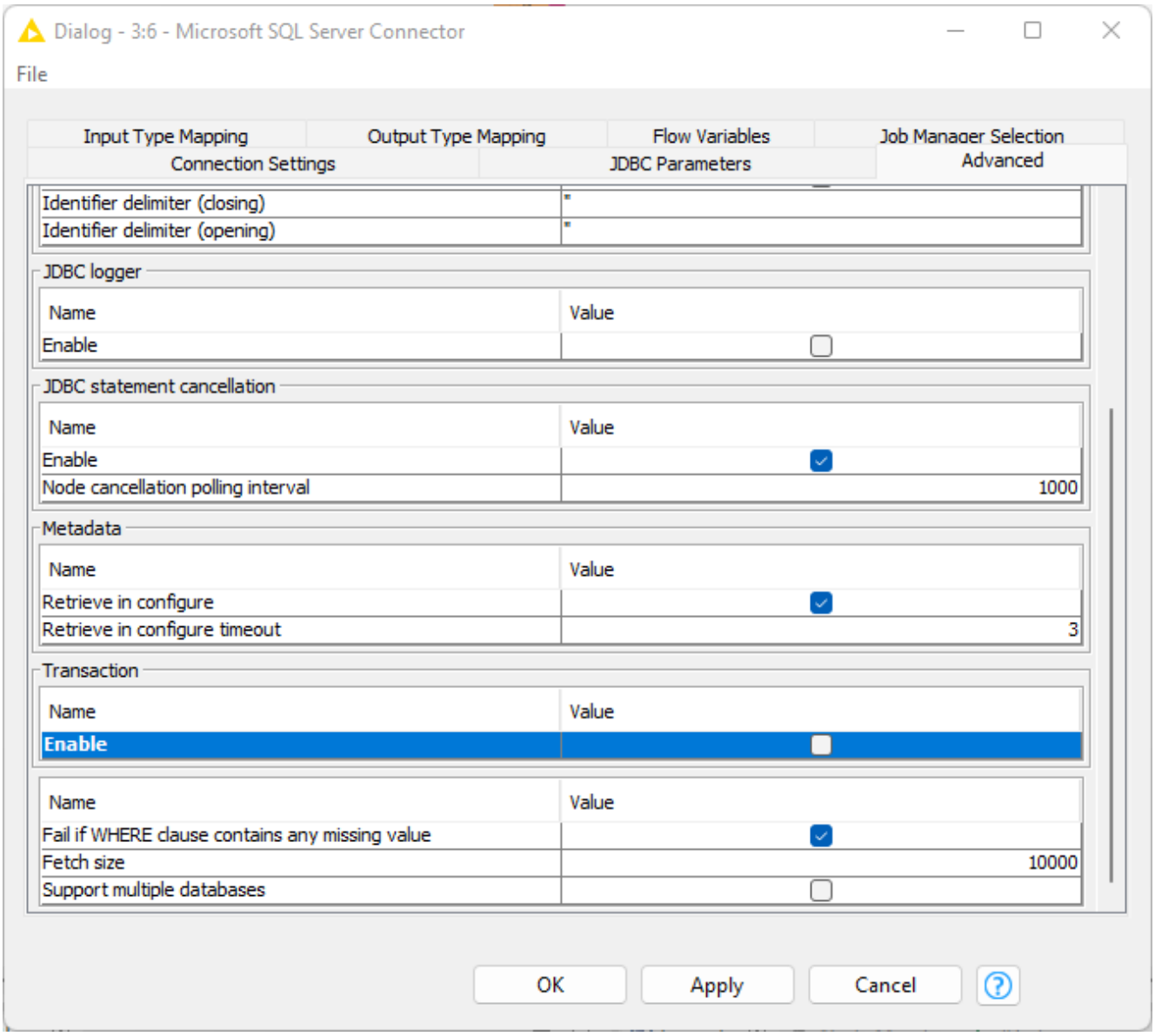

Figure 26. Disable transaction support via the Advanced tab

Connecting to Apache Hive™

To connect to Hive, you need to install the KNIME Big Data Connectors Extension.

The dedicated *Hive Connector* node is bundled by default with the open-source Apache Hive JDBC driver. Proprietary drivers are also supported, but need to be registered first, such as the Hive JDBC connector provided by Cloudera.

In this example we want to connect to Hive using the proprietary Cloudera Hive JDBC driver. The first step is to download the latest Hive JDBC driver for Cloudera Enterprise. Then go to *File* → *Preferences* → *KNIME* → *Databases*, and click *Add*.

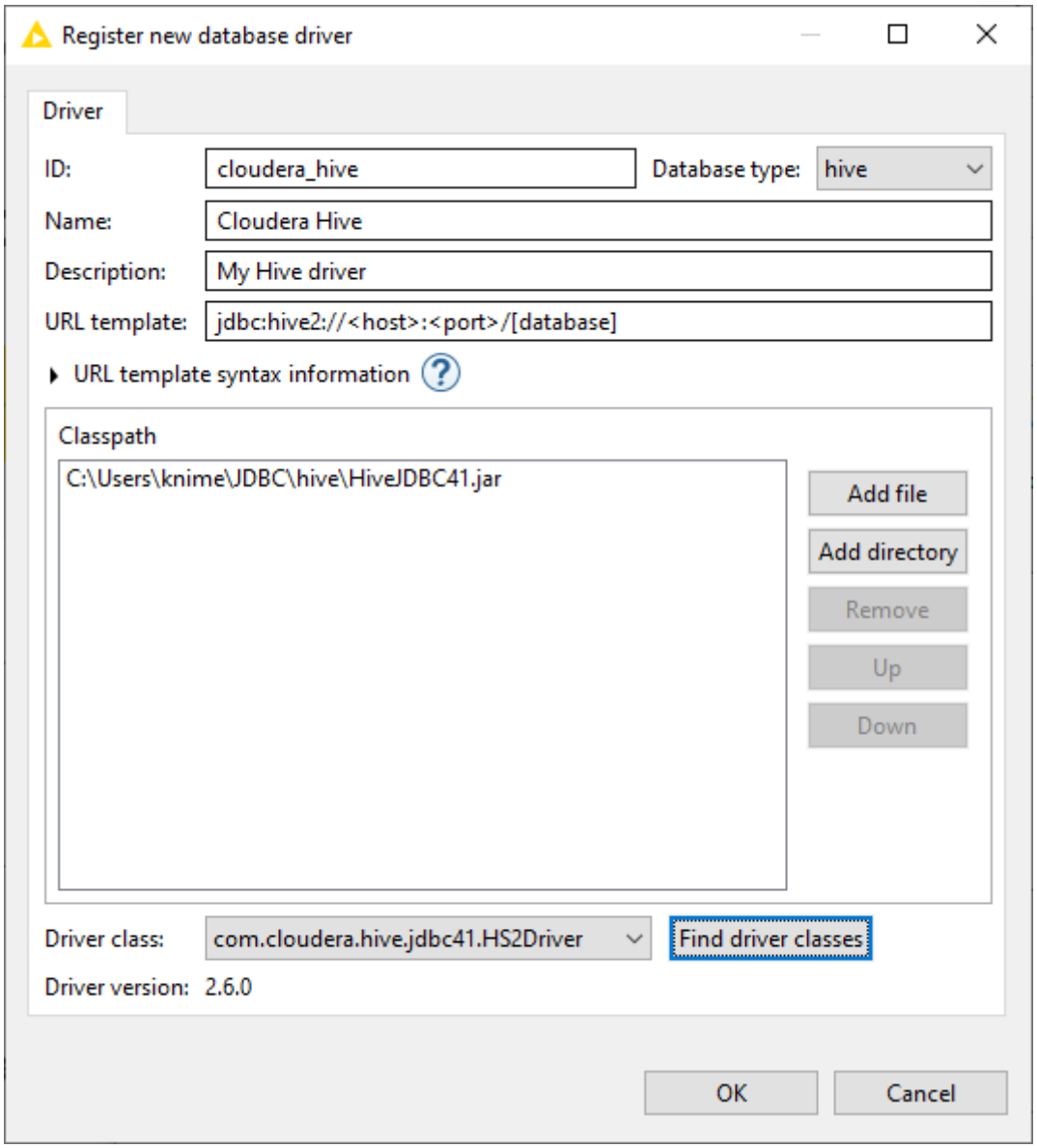

Figure 27. Register new Apache Hive driver

In the new database driver window, provide the following information:

- *ID*: cloudera_hive, but you can enter your own driver ID as long as it only contains alphanumeric characters and underscores.
- *Name*: Cloudera Hive, but you can enter your own driver name.
- *Database type*: Hive is available in the drop down list, so the database type is set to *hive*.
- *Description*: My Hive driver, for example.
- *URL template*: Please make sure that this field contains `jdbc:hive2://<host>:<port>/[database]`. Please refer to the JDBC URL Template section for more information on the supported tokens e.g. host, port and database.
- *Classpath*: Click *Add file* to add the `.jar` file that contains the Hive JDBC driver. The

path to the driver file will then appear in the Classpath area.

- *Driver class*: clicking *Find driver classes* will automatically detect all available JDBC driver classes and versions. Please make sure to select *com.cloudera.hive.jdbc41.HS2Driver*.

Finally, click *Ok* and the newly added driver will appear in the database driver preferences table. Click *Apply and Close* to apply the changes and you can start connecting to your Hive database.

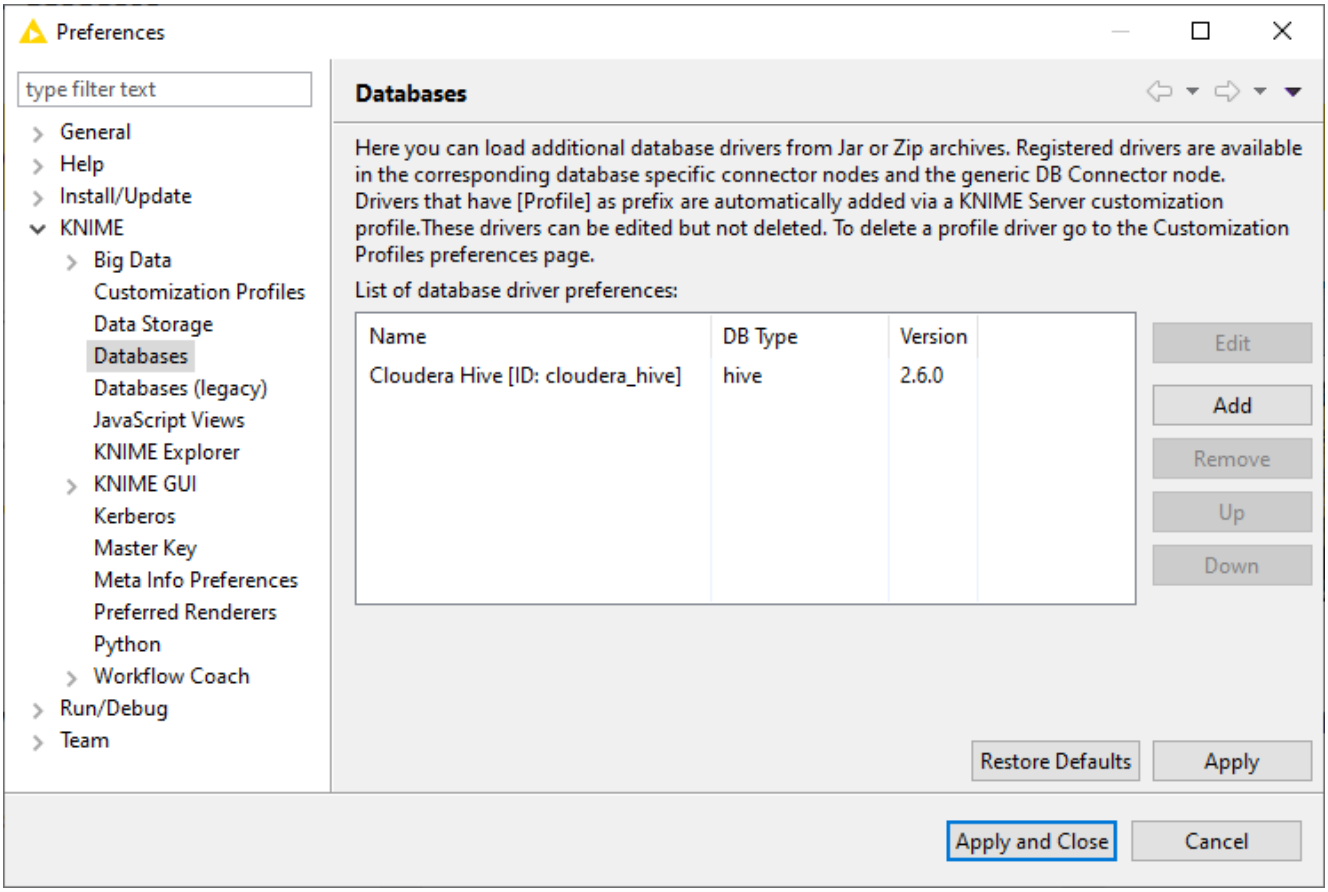

Figure 28. Database Preference page

Hive has a dedicated Connector node called *Hive Connector*, please refer to Connecting to predefined databases on how to connect using dedicated Connector nodes.

Connecting to Apache Impala™

To connect to Apache Impala, you need to install the KNIME Big Data Connectors Extension.

The dedicated *Impala Connector* node is bundled by default with the open-source Apache Hive JDBC driver, which is compatible with Impala. Proprietary drivers are also supported, but need to be registered first, such as the Impala JDBC connector provided by Cloudera.

In this example we want to connect to Impala using the proprietary Cloudera Impala JDBC

driver. The first step is to download the latest Impala JDBC driver for Cloudera Enterprise. Then go to *File* → *Preferences* → *KNIME* → *Databases*, and click *Add*.

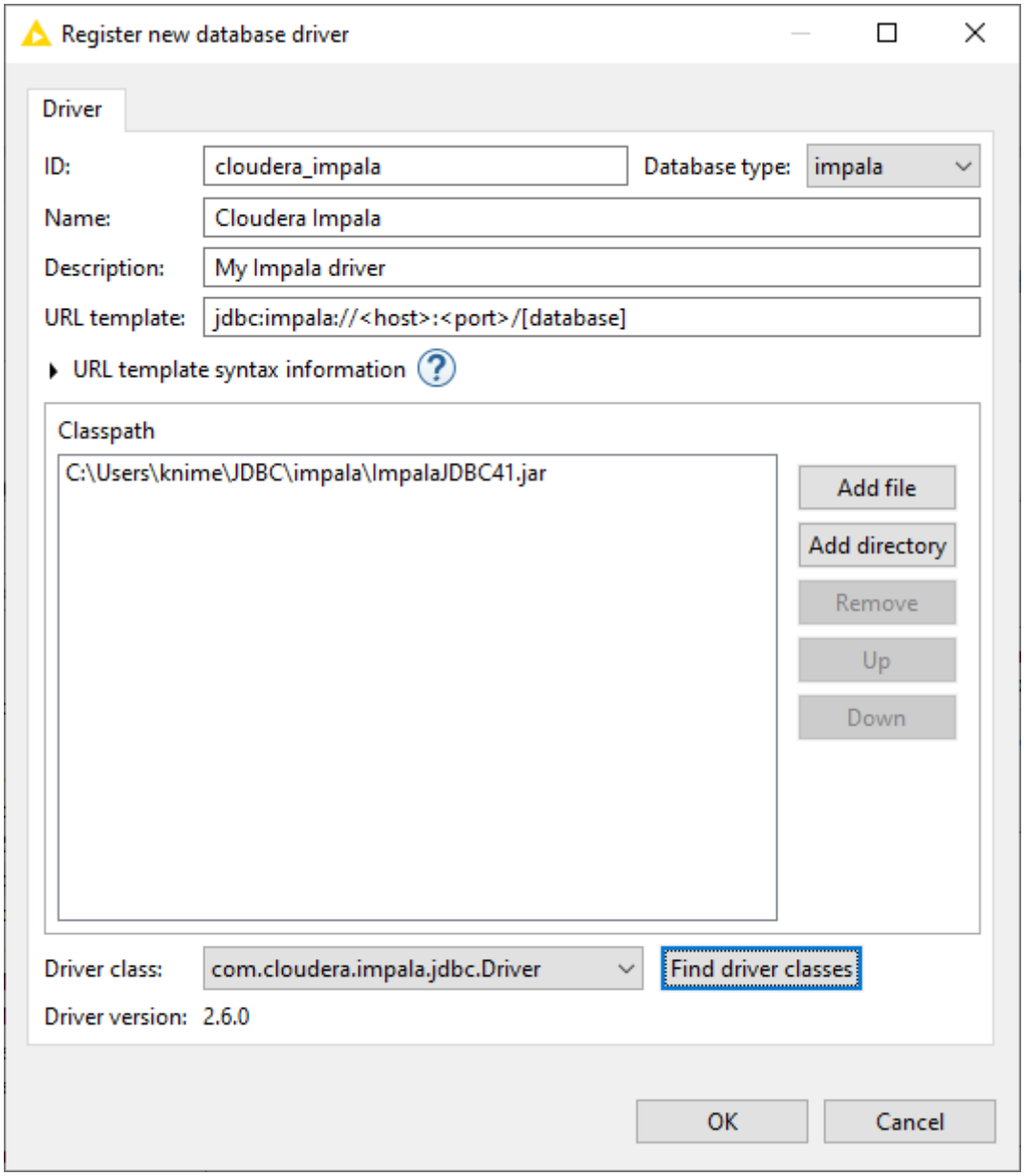

Figure 29. Register new Apache Impala driver

In the new database driver window, provide the following information:

- *ID*: cloudera_impala, but you can enter your own driver ID as long as it only contains alphanumeric characters and underscores.
- *Name*: Cloudera Impala, but you can enter your own driver name.
- *Database type*: Impala is available in the drop down list, so the database type is set to *impala*.
- *Description*: My Impala driver, for example.
- *URL template*: By selecting *impala* in the *Database type*, the URL template is

automatically preset to the default JDBC URL template for Impala, i.e `jdbc:impala://<host>:<port>/[database]`. For more possible templates, simply click on the URL Template syntax information directly below. Please refer to the JDBC URL Template section for more information on the supported tokens e.g. host, port and database.

- *Classpath*: Click *Add file* to add the `.jar` file that contains the Impala JDBC driver file. The path to the driver file will then appear in the Classpath area.
- *Driver class*: clicking *Find driver classes* will automatically detect all available JDBC driver classes and versions, which in this case is *com.cloudera.impala.jdbc.Driver* .

Finally, click *Ok* and the newly added driver will appear in the database driver preferences table. Click *Apply and Close* to apply the changes and you can start connecting to your Impala database.

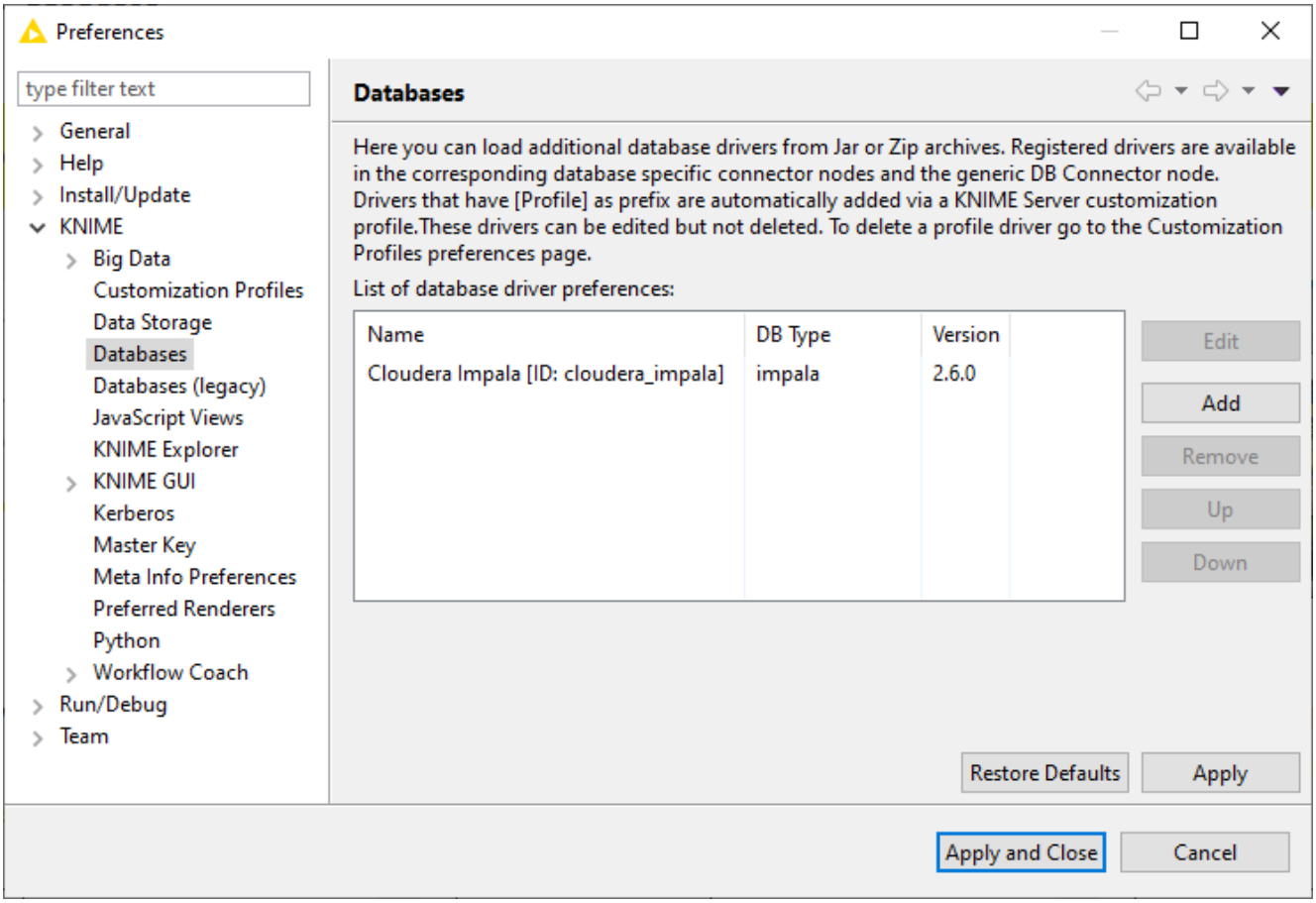

Figure 30. Database Preference page

Impala has a dedicated Connector node called *Impala Connector*, please refer to Connecting to predefined databases on how to connect using dedicated Connector nodes.

Reading from a database

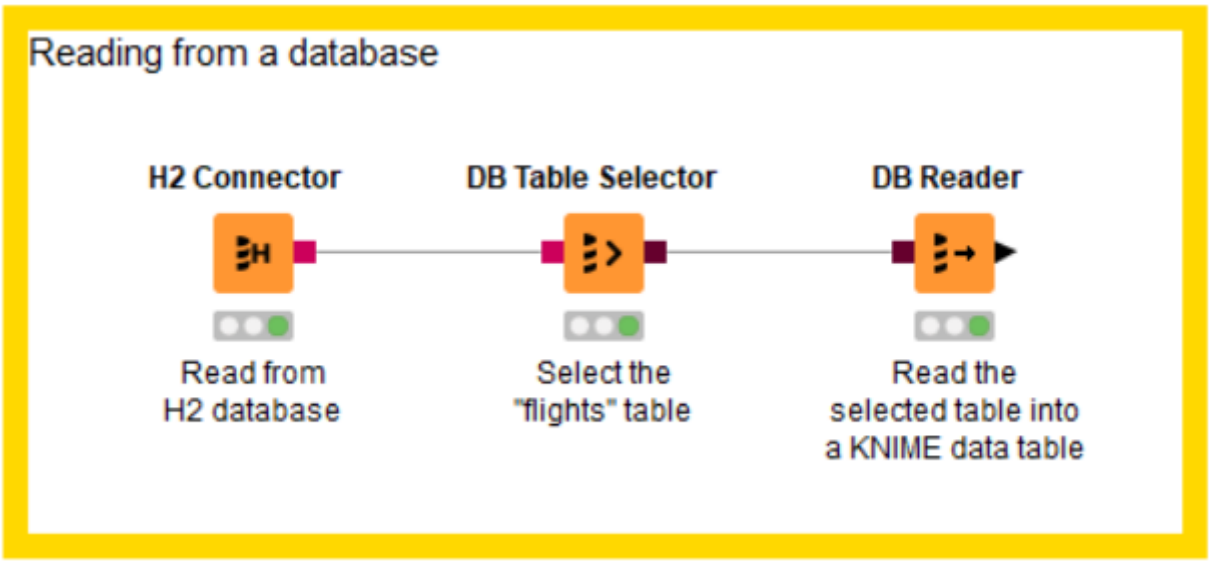

Figure 31. Reading from a database

The figure above is an example on how to read from a database. In this example we want to read the flights dataset stored in an H2 database into a KNIME data table.

First you need a connector node to establish a connection to the database, in the example above it is an H2 database. There are several dedicated connector nodes depending on which database we want to connect to. For further details on how to connect to a database refer to the Connecting to a database section .

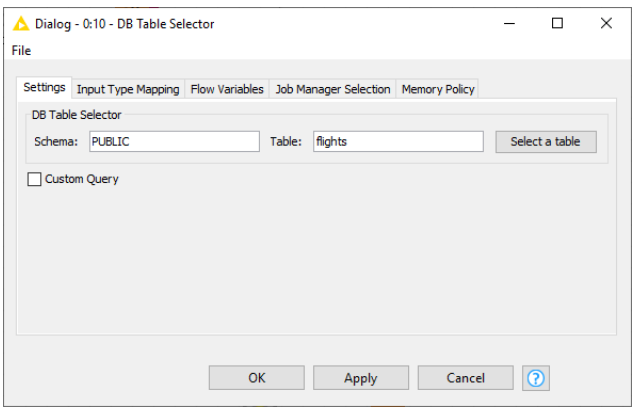

Figure 32. DB Table Selector configuration dialog

After the connection is established, the next step is to use the *DB Table Selector* node that allows selecting a table or a view interactively based on the input database connection.

The figure on the left side shows the configuration dialog of the *DB Table Selector* node. At the top part you can enter the schema and the table/view name that you want to select, in this example we want to select the "flights" table.

Pressing the *Select a table* button will open a Database Metadata Browser window that lists available tables/views in the database.

In addition, ticking the *Custom Query* checkbox will allow 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 table selected via the *Select a table* button.

The *Input Type Mapping* tab allows you to define mapping rules from database types to KNIME types. For more information on this, please refer to the section Type Mapping.

The output of this node is a DB Data connection that contains the database information and the SQL query automatically build by the framework that selects the entered table or the user entered custom query. To read the selected table or view into KNIME Analytics Platform, you can use the *DB Reader* node. Executing this node will execute the input SQL query in the database and the output will be the result stored in a KNIME data table which will be stored on the machine where KNIME Analytics Platform is running.

Database Metadata Browser

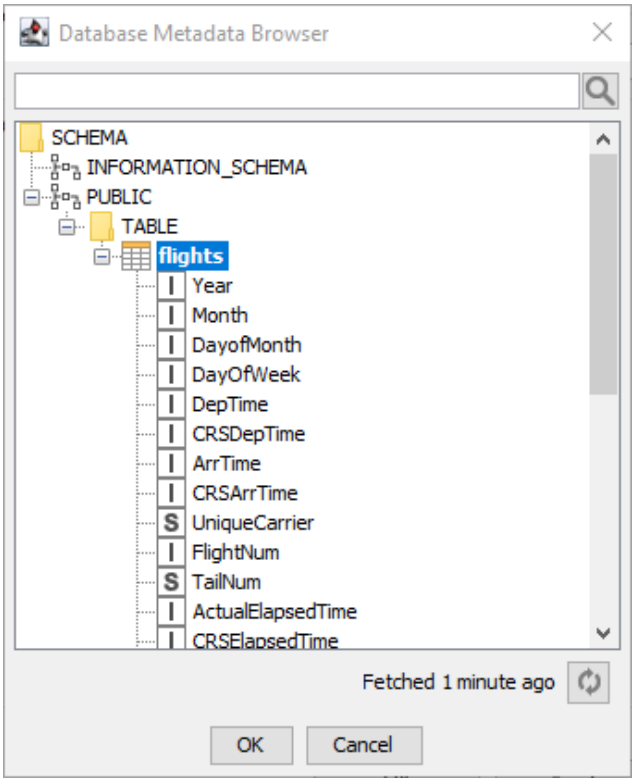

Figure 33. Database Metadata Browser

The Database Metadata Browser 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 a table or view select the name and click OK or double click the element.

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 with a time reference on how long ago the schema was last refreshed.

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 at the lower right corner.

Query Generation

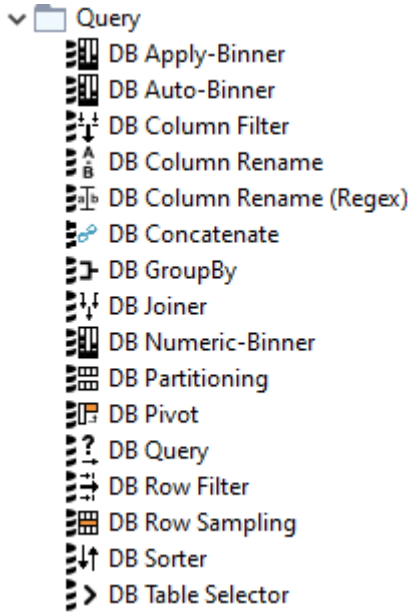

Figure 34. DB Query nodes

Once you have successfully connected to your database, there is a set of nodes that provide in-database data manipulation, such as aggregating, filtering, joining etc.

The database nodes come with a visual user interface and automatically build a SQL query in the background according to the user settings in the configuration window, so no coding is required to interact with the database.

The output of each node is a SQL query that corresponds to the operation(s) that are performed within the node. The generated SQL query can be extracted by using the DB Query Extractor node.

Visual Query Generation

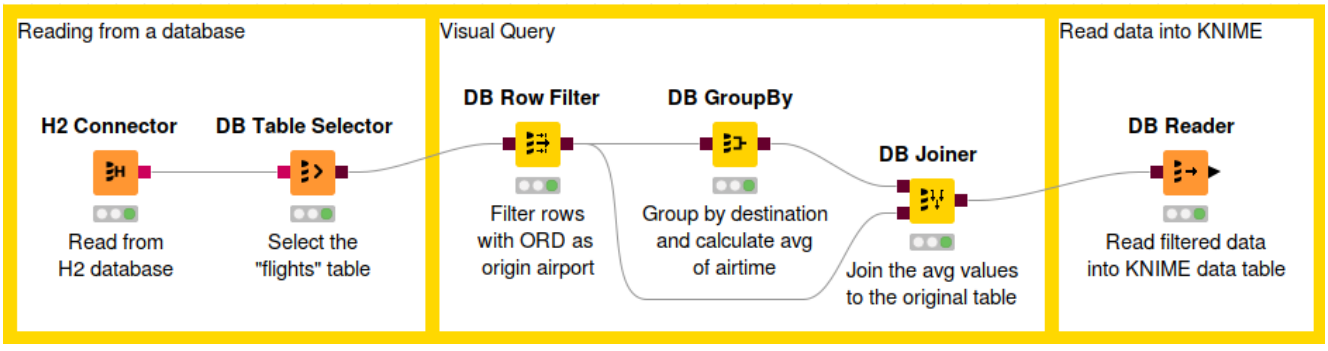

Figure 35. Example of a workflow that performs in-database data manipulation

The figure above shows an example of in-database data manipulation. In this example, we read the flights dataset from a H2 database. First we filter the rows so that we take only the flights that fulfil certain conditions. Then we calculate the average air time to each unique destination airport. Finally we join the average values together with the original values and then read the result into KNIME Analytics Platform.

The first step is to connect to a database and select the appropriate table we want to work with.

DB Row Filter

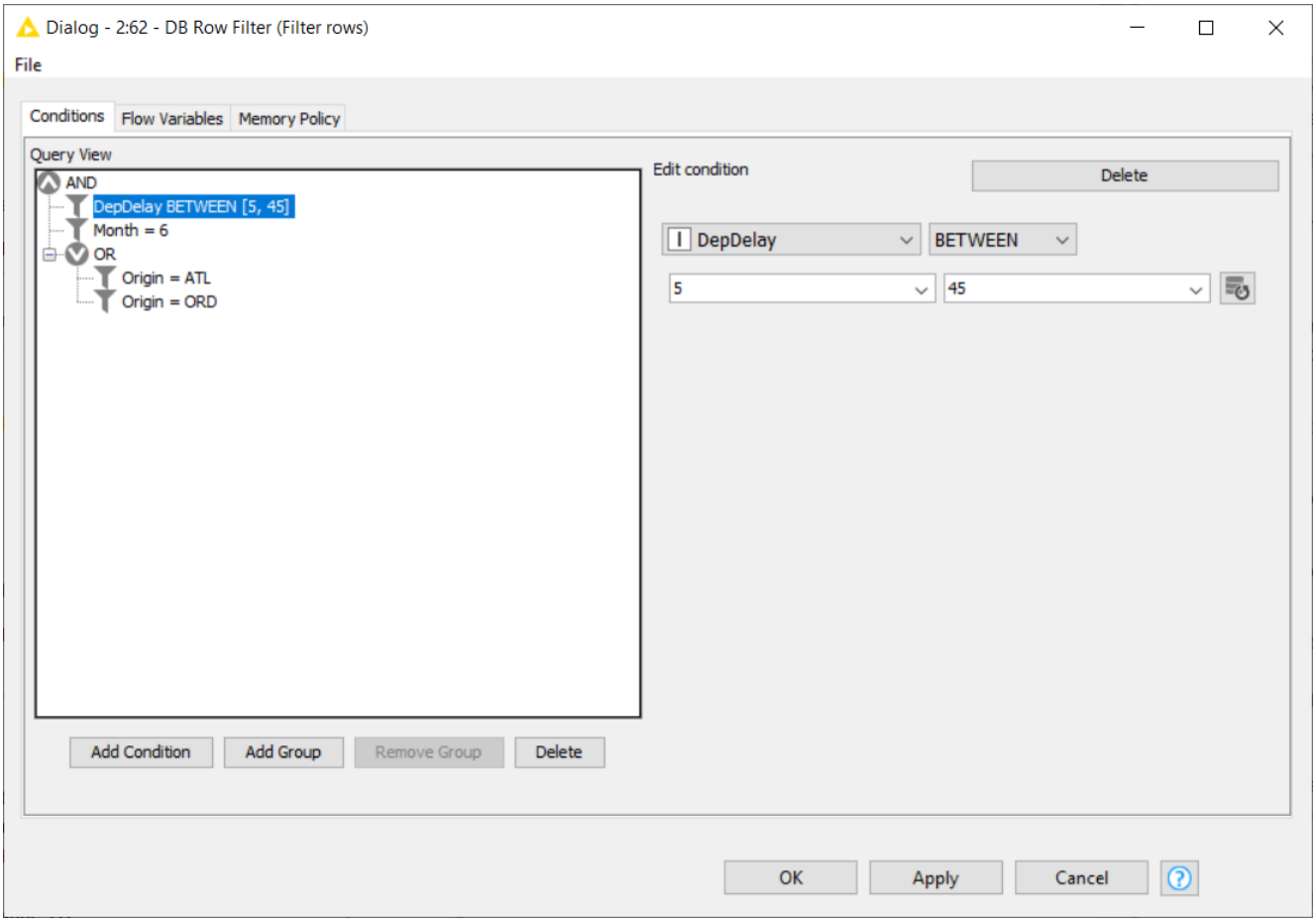

Figure 36. DB Row Filter configuration dialog

After selecting the table, you can start working with the data. First we use the *DB Row Filter* node to filter rows according to certain conditions. The figure above shows the configuration dialog of the *DB Row Filter*. On the left side there is a Preview area that lists all conditions of the filter to apply to the input data. Filters can be combined and grouped via logical operators such as AND or OR. Only rows that fulfil the specified filter conditions will be kept in the output data table. At the bottom there are options to:

- *Add Condition*: add more condition to the list
- *Group*: Create a new logical operator (AND or OR)
- *Ungroup*: Delete the currently selected logical operator
- *Delete*: Delete the selected condition from the list

To create a new condition click on the *Add_Condition* button. To edit a condition select in the condition list which will show the selected condition in the condition editor on the right. The editor consists of at least two dropdown lists. The most left one contains the columns from the input data table, and the one next to it contains the operators that are compatible with the selected column type, such as =, !=, <, >. Depending on the selected operation a third and

maybe fourth input field will be displayed to enter or select the filter values. The button next to the values fields fetches all possible values for the selected column which will then be available for selection in the value field.

Clicking on a logical operator in the Preview list would allow you to switch between AND or OR, and to delete this operator by clicking *Ungroup*.

As in our example, we want to return all rows that fulfil the following conditions:

- Originate from the Chicago O'Hare airport (ORD) OR Hartsfield-Jackson Atlanta Airport (ATL)
- AND occur during the month of June 2017
- AND have a mild arrival delay between 5 and 45 minutes

DB GroupBy

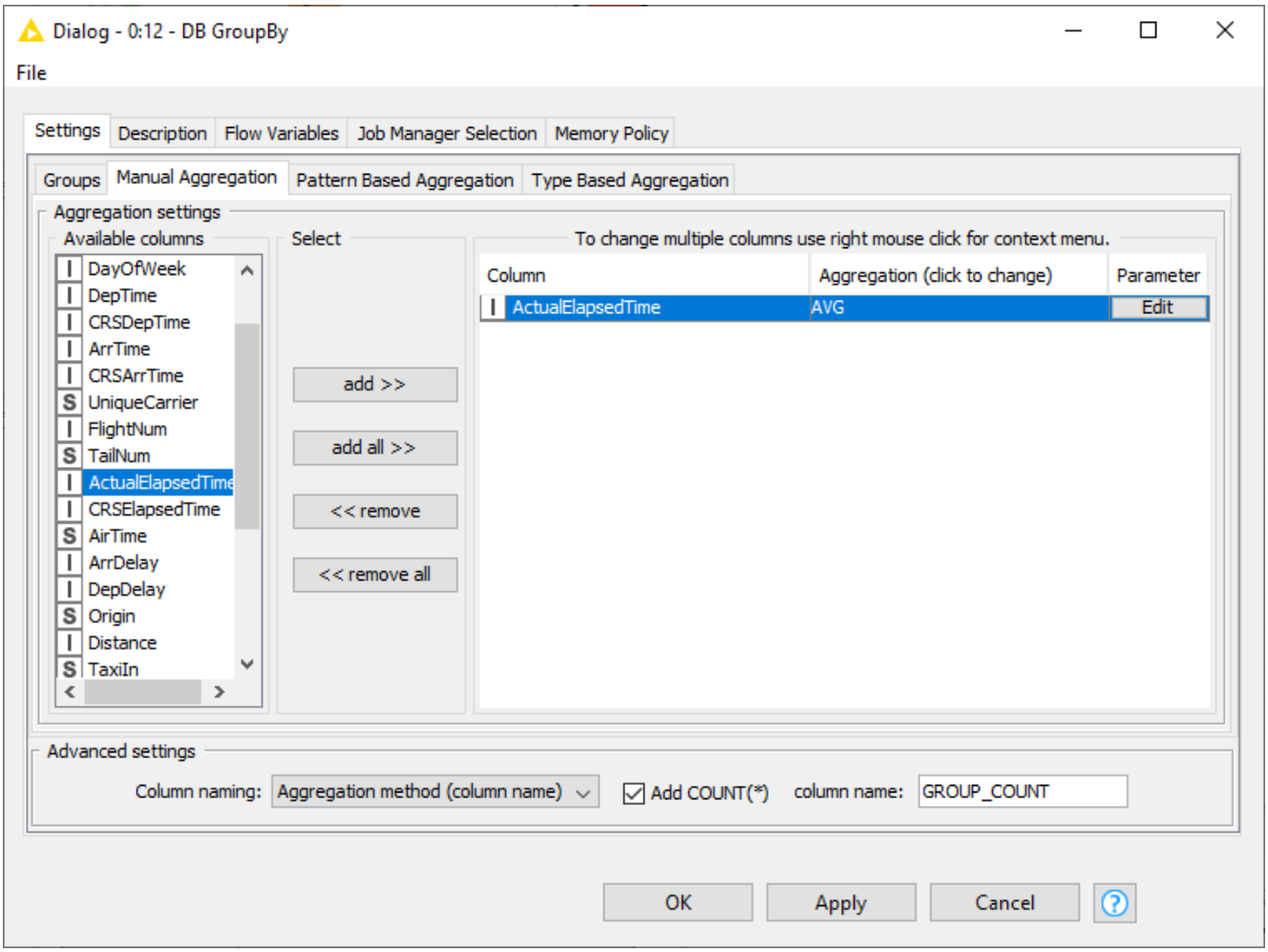

Figure 37. DB GroupBy: Manual Aggregation

The next step is to calculate the average air time to each unique destination airport using the *DB GroupBy* node. To retrieve the number of rows per group tick the *Add Count(*)* checkbox in

the *Advanced Settings*. The name of the group count column can be changed via the result column name field.

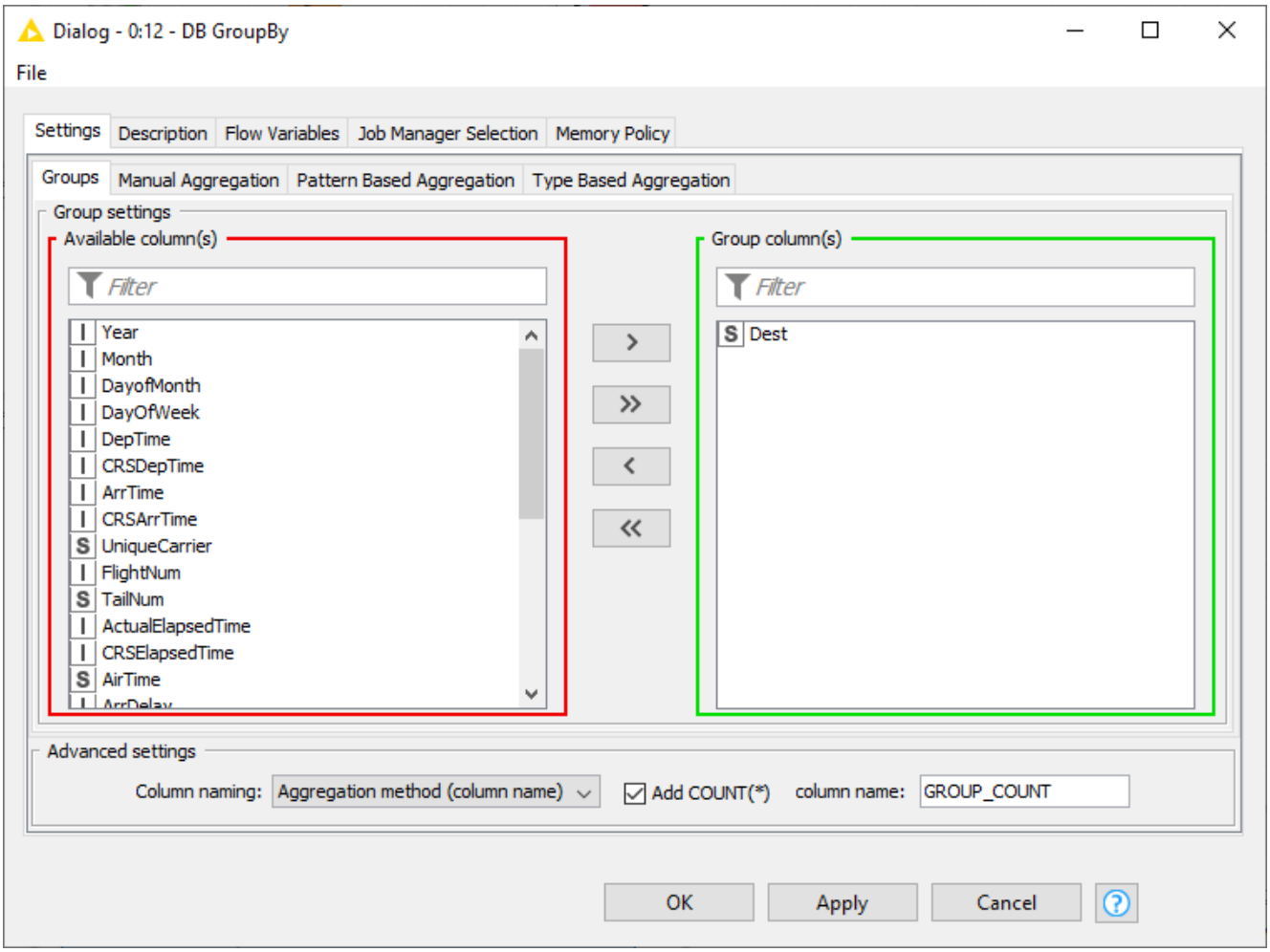

Figure 38. DB GroupBy: Group Settings

To calculate the average air time for each destination airport, we need to group by the *Dest* column in the *Groups* tab, and in *Manual Aggregation* tab we select the *ActualElapsedTime* column (air time) and *AVG* as the aggregation method.

DB Joiner

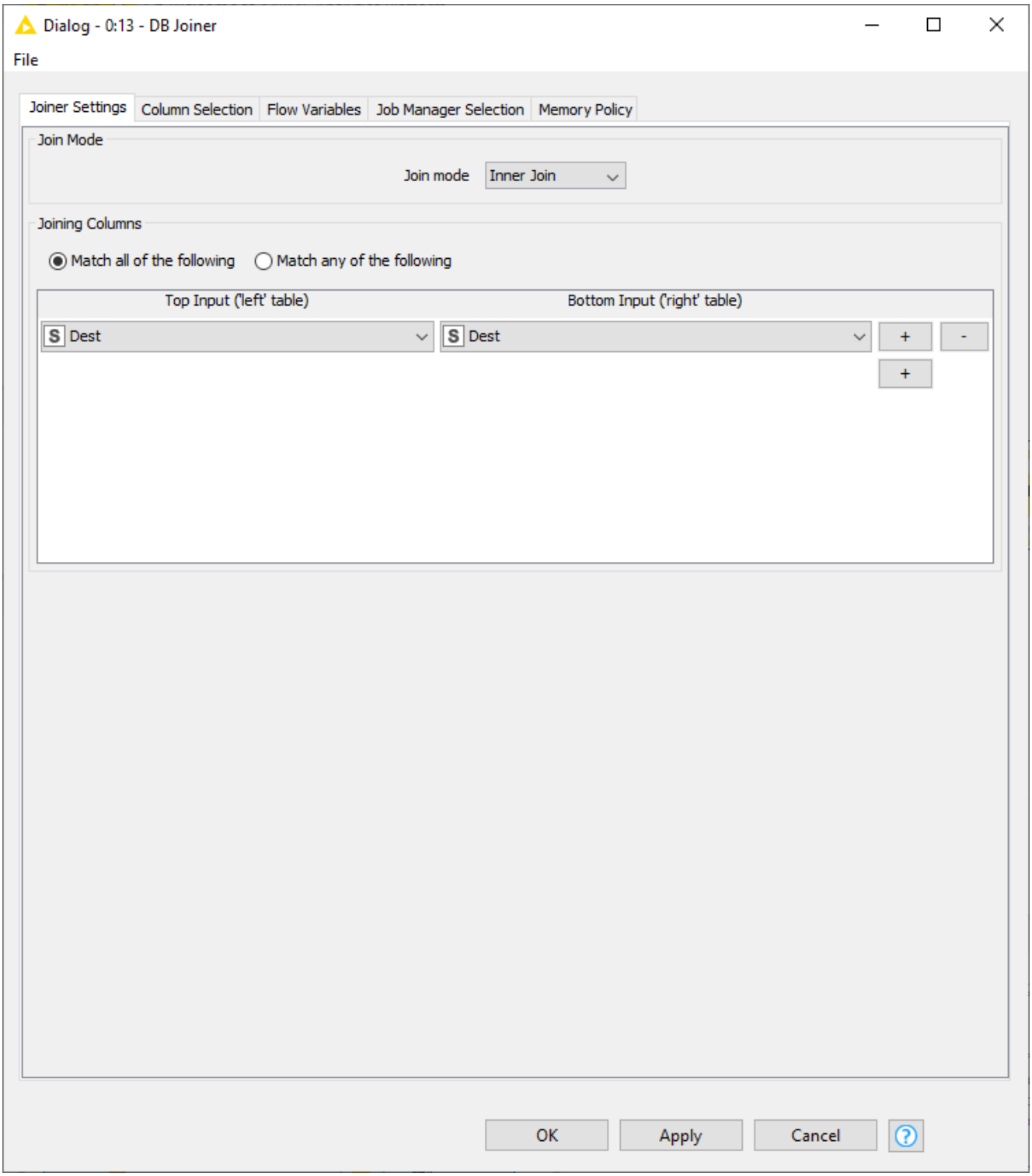

Figure 39. DB Joiner: Joiner Settings

To join the result back to the original data, we use the *DB Joiner* node, which joins two database tables based on joining column(s) of both tables. In the *Joiner* settings tab, there are options to choose the join mode, whether Inner Join, Full Outer Join, etc, and the joining column(s).

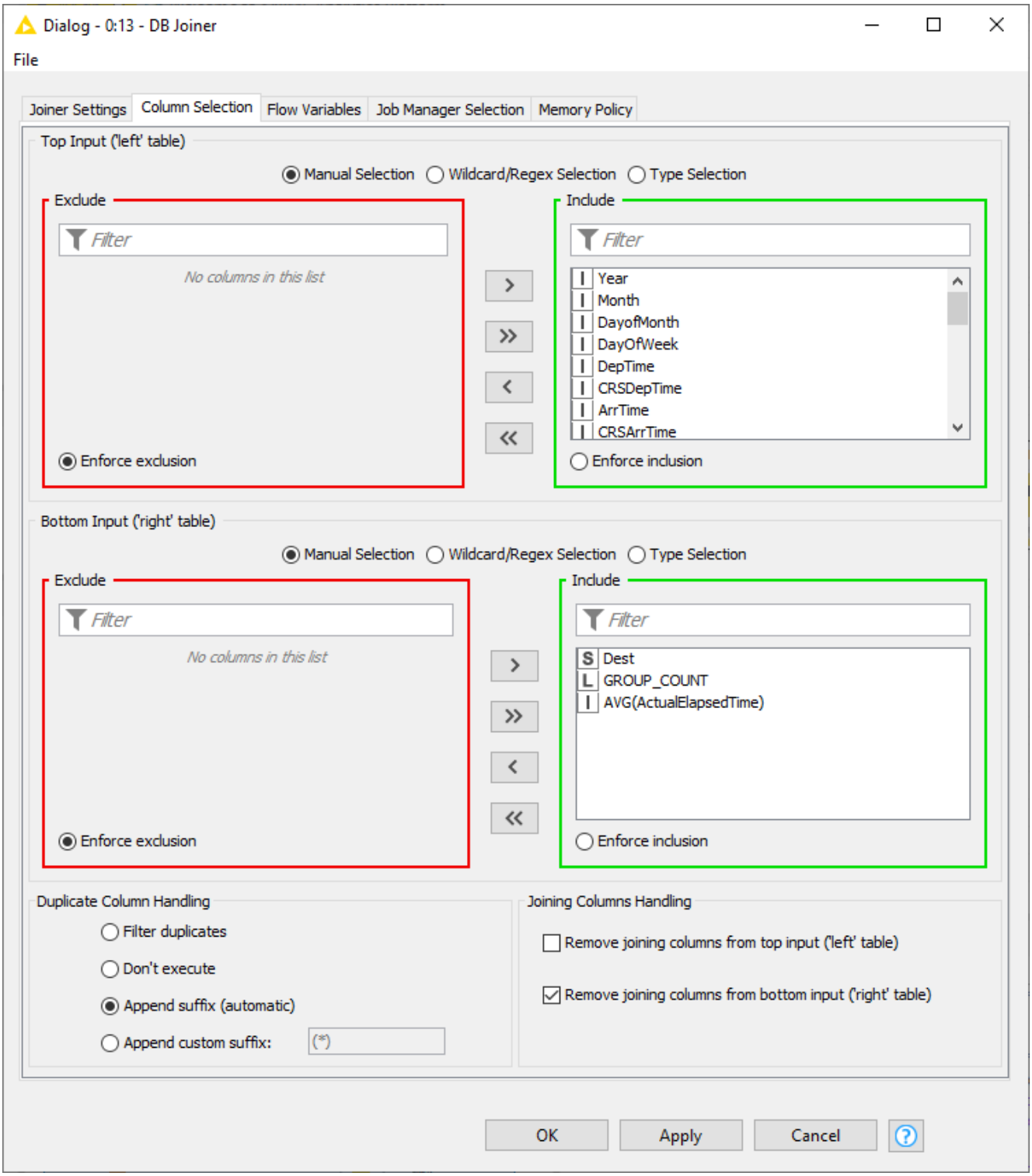

Figure 40. DB Joiner: Column Selection

In the *Column Selection* tab you can select which columns from each of the table you want to include in the output table. By default the joining columns from bottom input will not show up in the output table.

Advanced Query Building

Sometimes, using the predefined DB nodes for manipulating data in database is not enough. This section will explain some of the DB nodes that allow users to write their own SQL queries, such as *DB Query*, *DB Query Reader*, and *Parametrized DB Query Reader* node.

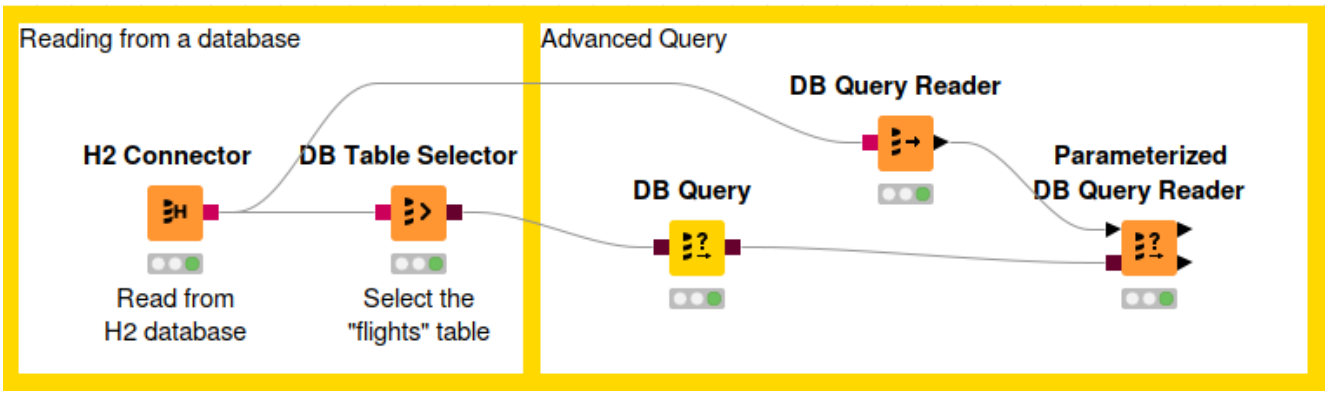

Figure 41. Example workflow with advanced query nodes

Each DB manipulation node, that gets a DB data object as input and returns a DB data object as output, wraps the incoming SQL query into a sub-query. However some databases don't support sub-queries, and if that is the case, please use the *DB Query Reader* node to read data from the database.

The figure below shows the configuration dialog of the *DB Query* node. The configuration dialog of other advanced query nodes that allow user to write SQL statements provide a similar user experience. There is a text area to write your own SQL statement, which provides syntax highlighting and code completion by hitting *Ctrl+Space*. On the lower side there is an *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 be shown in the Evaluate window. On the left side there is the Database Metadata Browser window that 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 in the connected database table. Double clicking any of the items will insert its name at the current cursor position in the SQL statement area.

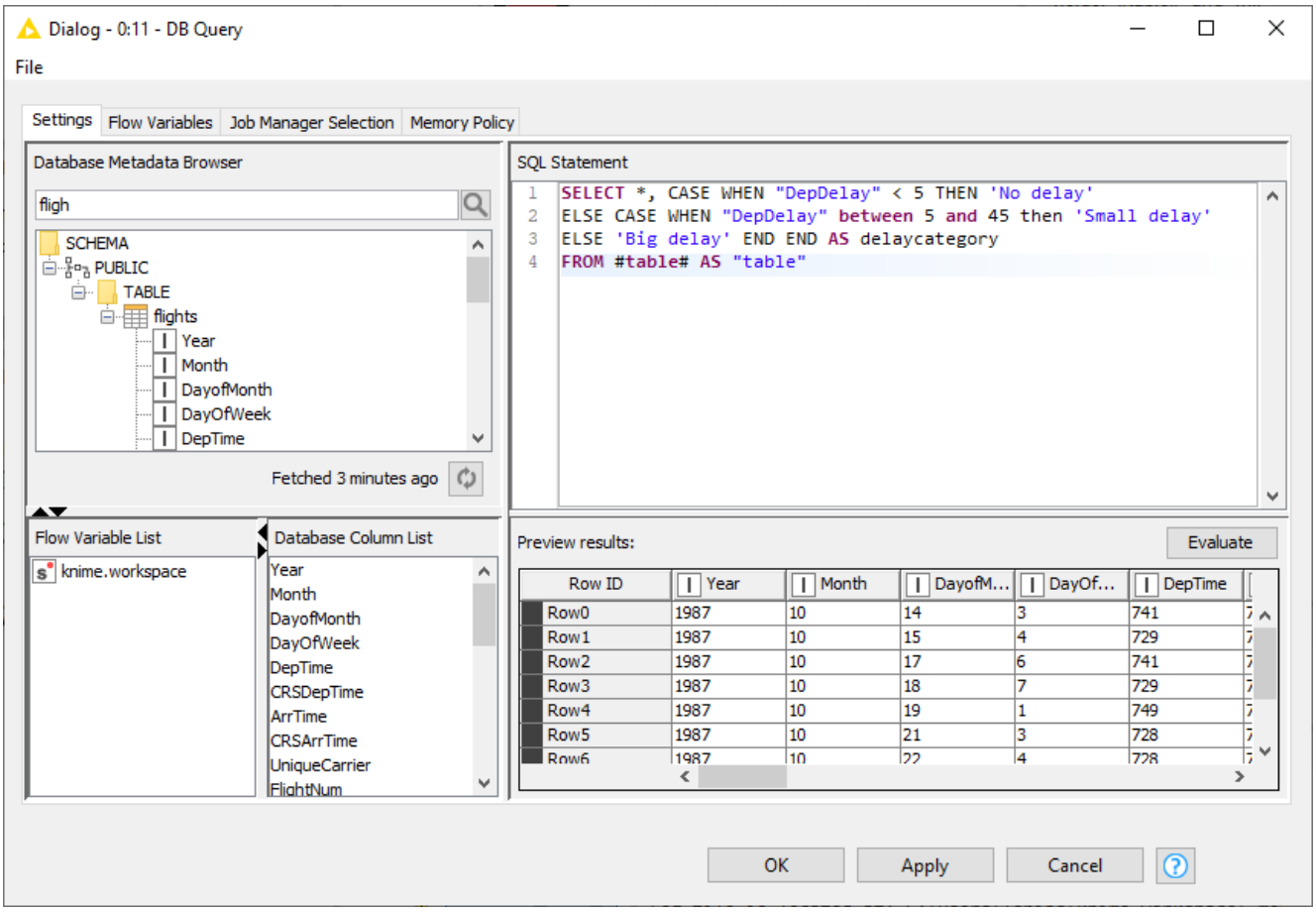

Figure 42. Configuration dialog of the DB Query node

DB Query

The *DB Query node* modifies the input SQL query from an incoming database data connection. The SQL query from the predecessor is represented by the place holder `#table#` and will be replaced during execution. The modified input query is then available at the outport.

DB Query Reader

Executes an entered SQL query and returns the result as KNIME data table. This node does not alter or wrap the query and thus supports all kinds of statements that return data.

This node supports other SQL statements beside SELECT, such as `DESCRIBE TABLE`.

Parameterized DB Query Reader

This node allows you to execute a SQL query with different parameters. It loops over the input KNIME table and takes the values from the input table to parameterise the input SQL query. Since the node has a KNIME data table input it provides a type mapping tab that allows you to change the mapping rules. For more information on the Type Mapping tab, please refer to the Type Mapping section.

DB Looping

This node runs SQL queries in the connected database restricted by the possible values given by the input table. It restricts each SQL query so that only rows that match the possible values from the input table are retrieved whereas the number of values per query can be defined. This node is usually used to execute IN queries e.g.

```
SELECT * FROM table
WHERE Col1 IN ($Col1_values$)
```

During execution, the column placeholder *$Col1_values$* will be replaced by a comma separated list of values from the input table. Since the node has a KNIME data table input it provides a type mapping tab that allows you to change the mapping rules. For more information on the Type Mapping tab, please refer to the Type Mapping section.

An example of the usage of this node is available on KNIME Hub.

Database Structure Manipulation

Database Structure Manipulation refers to any manipulation to the database tables. The following workflow demonstrates how to remove an existing table from a database using the *DB Table Remover* and create a new table with the *DB Table Creator* node.

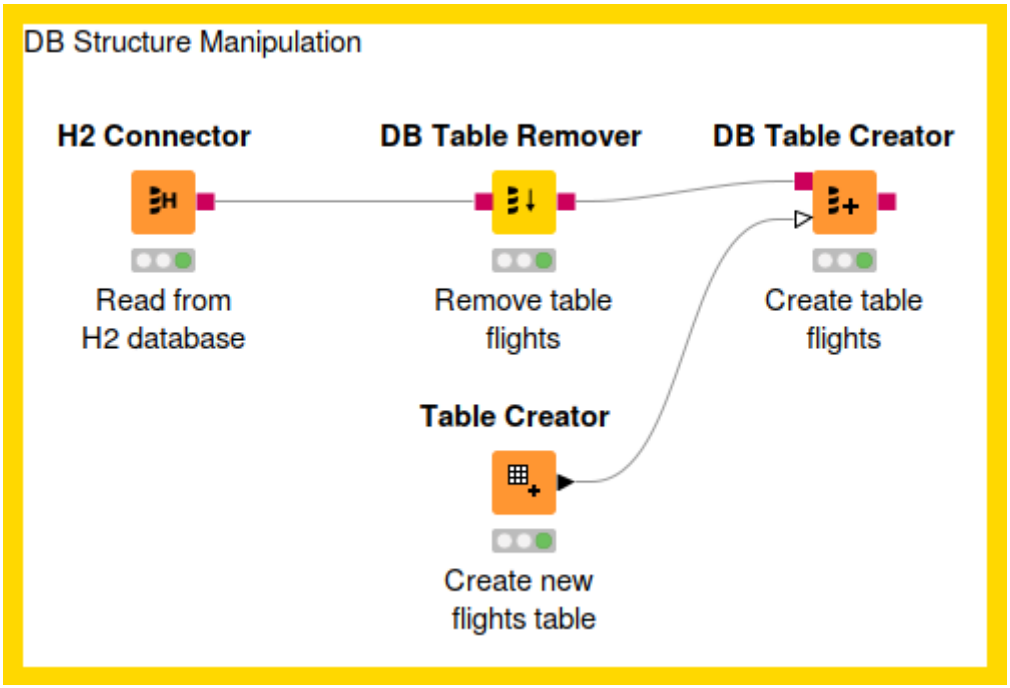

Figure 43. Example of a database structure manipulation workflow

DB Table Remover

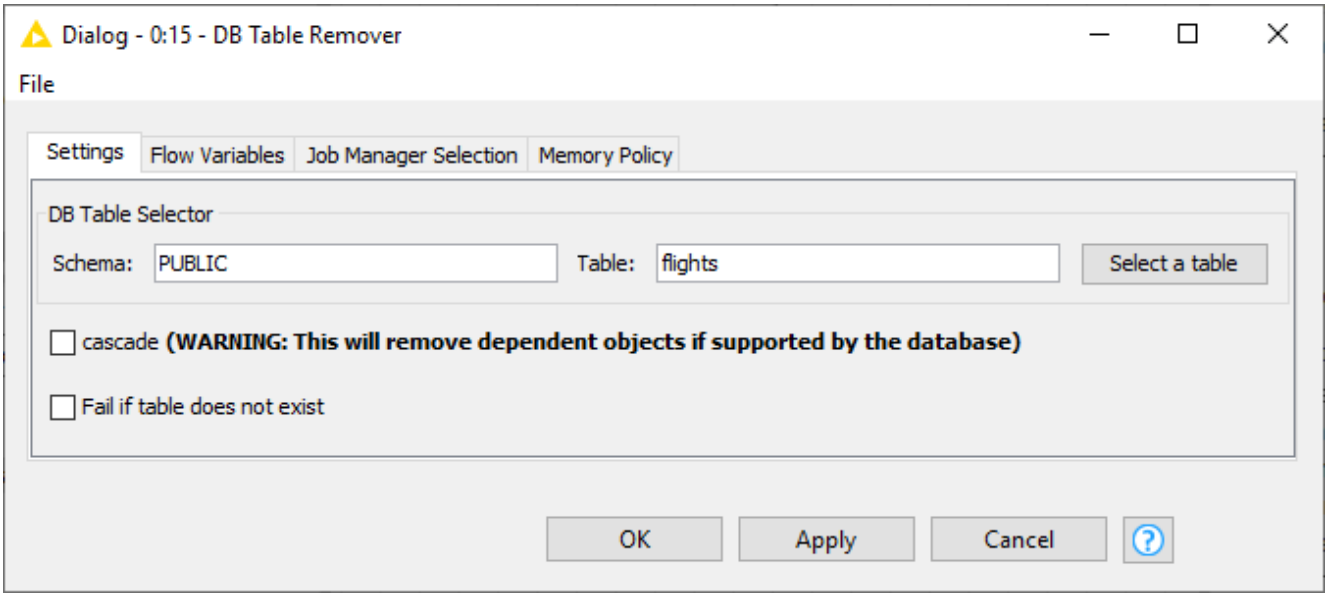

Figure 44. DB Table Remover configuration dialog

This node removes a table from the database defined by the incoming database connection.

Executing this node is equivalent to executing the SQL command DROP. In the configuration dialog, there is an option to select the database table to be removed. The configuration is the same as in the *DB Table Selector* node, where you can input the corresponding Schema and the table name, or select it in the Database Metadata Browser.

The following options are available in the configuration window:

Cascade: Selecting this option means that removing a table that is referenced by other tables/views will remove not only the table itself but also all dependent tables and views. If this option is not supported by your database then it will be ignored.

Fail if table does not exist: Selecting this option means the node will fail if the selected table does not exist in the database. By default, this option is not enabled, so the node will still execute successfully even if the selected table does not exist in the database.

DB Table Creator

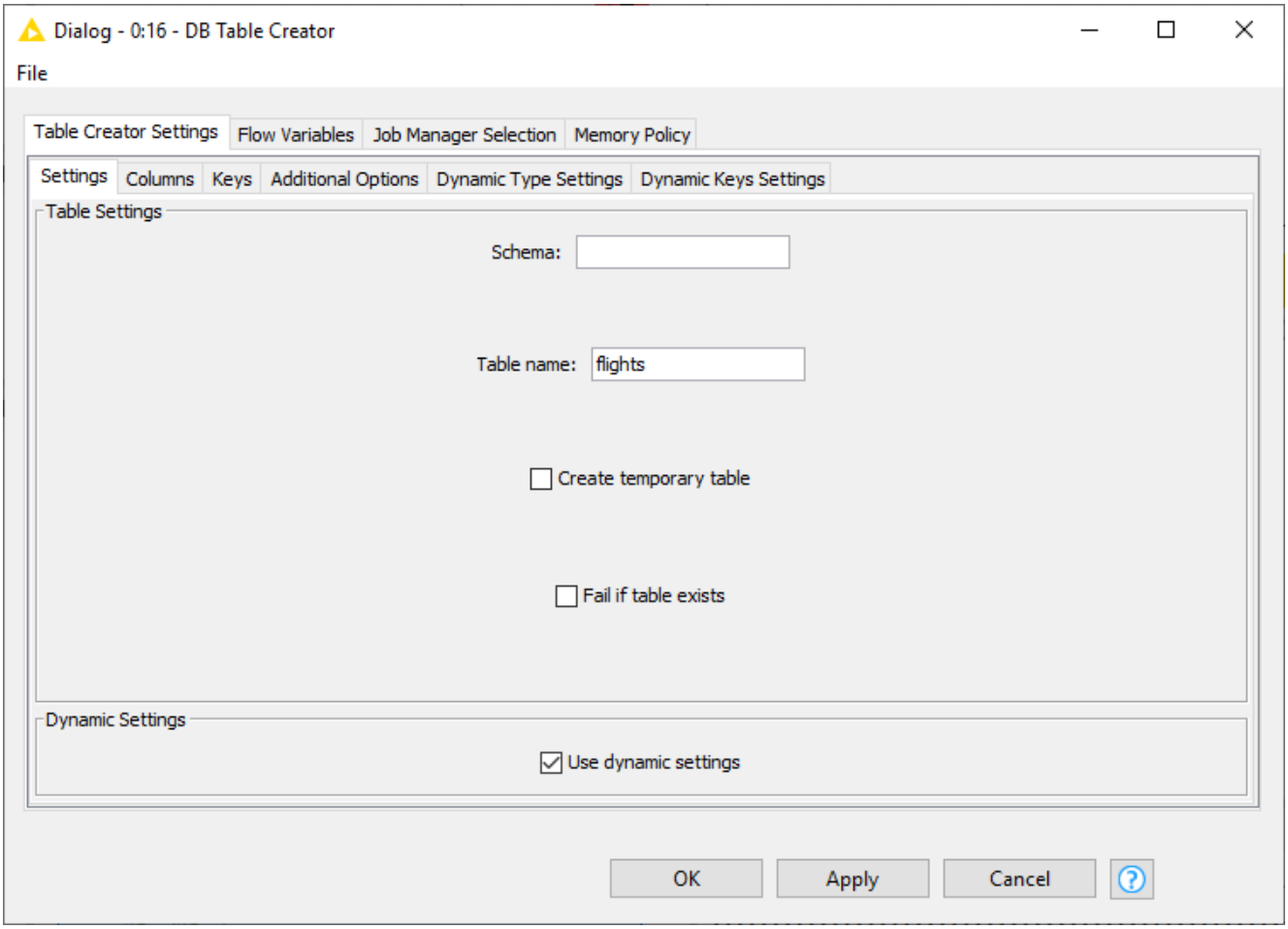

Figure 45. DB Table Creator: Settings

This node creates a new database table. The table can be created either manually, or dynamically based on the input data table spec. It supports advanced options such as

specifying if a column can contain null values or specifying primary key or unique keys as well as the SQL type.

When the *Use dynamic settings* option is enabled the database table structure is defined by the structure of the input KNIME data table. The *Columns* and *Keys* tabs are read only and only help to verify the structure of the table that is created. The created database table structure can be influenced by changing the type mapping e.g. by defining that KNIME double columns should be written to the database as string columns the *DB Table Creator* will choose the string equivalent database type for all double columns. This mapping and also the key generation can be further influenced via the *Dynamic Type Settings* and *Dynamic Key Settings* tabs.

In the Settings tab you can input the corresponding schema and table name. The following options are available:

Create temporary table: Selecting this will create a temporary table. The handling of temporary tables, such as how long it exists, the scope of it, etc depends on the database you use. Please refer to your database vendor for more details on this.

Fail if table exists: Selecting this will make the node fail with database-specific error message if the table already exists. By default, this option is disable, so the node will execute successfully and not create any table if it already existed.

Use dynamic settings: Selecting this will allow the node to dynamically define the structure of the database table e.g. column names and types based on the input KNIME table and the dynamic settings tabs. Only if this option is enabled will the *Dynamic Type Settings* and *Dynamic Column Settings* tab be available. The mappings defined in the *Name-Based SQL Type Mapping* have a higher priority than the mappings defined in the *KNIME-Based SQL Type Mapping*. If no mapping is defined in both tabs, the default mapping based on the Type Mapping definitions of the database connector node are used. Note that while in dynamic settings mode the *Columns* and *Keys* tab become read-only to allow you a preview of the effect of the dynamic settings.

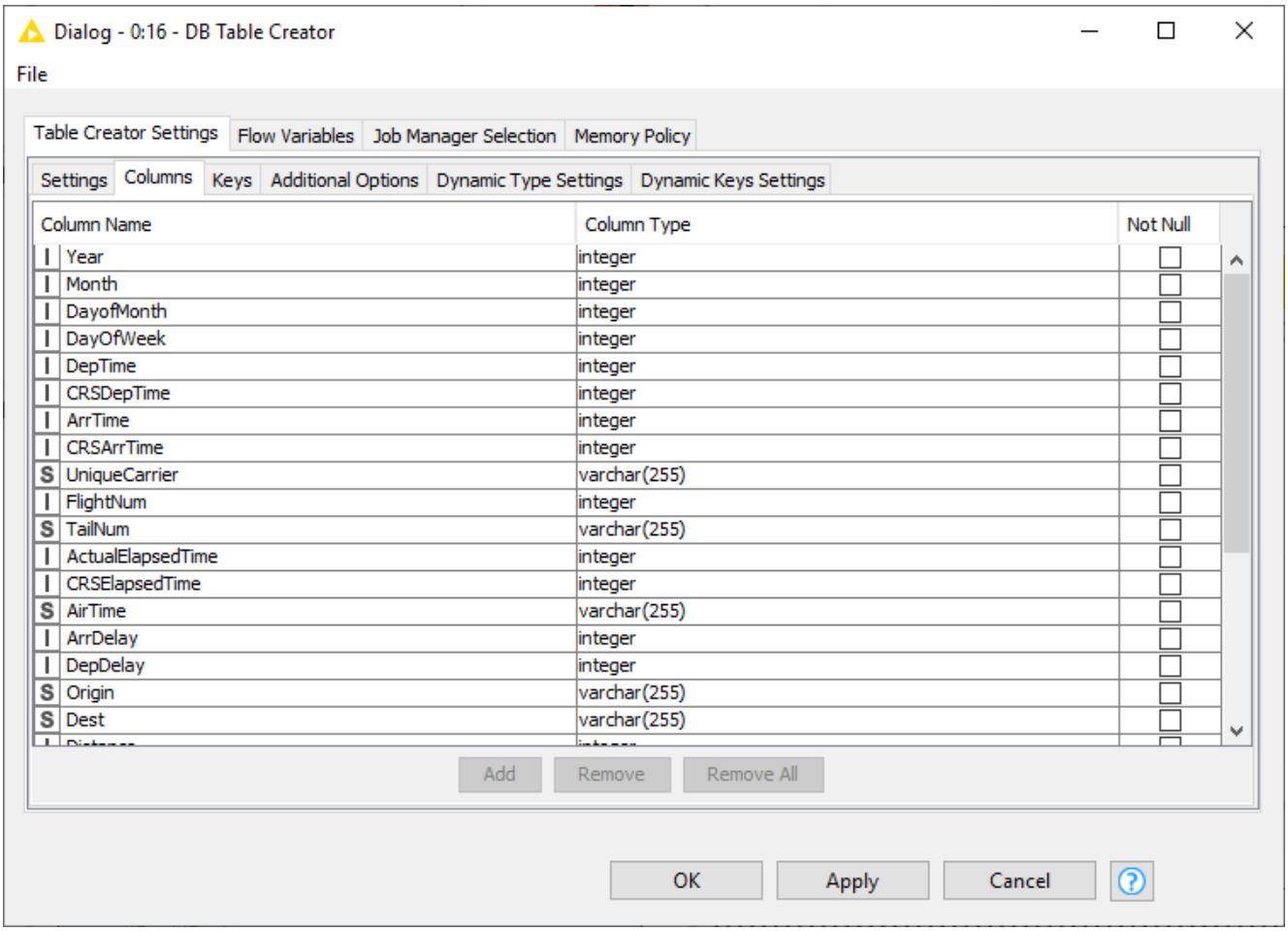

Figure 46. DB Table Creator: Columns

In the *Columns* tab you can modify the mapping between the column names from the input table and their corresponding SQL type manually. You can add or remove column and set the appropriate SQL type for a specific column. However, if the *Use dynamic settings* is selected, this tab become read-only and serves as a preview of the dynamic settings.

In the *Key* tab you can set certain columns as primary/unique keys manually. As in the *Columns* tab, if the *Use dynamic settings* is enabled, this tab become read-only and serves as a preview of the dynamic settings.

In the *Additional Options* tab you can write additional SQL statement which will be appended after the CREATE TABLE statement, e.g storage parameter. This statement will be appended to the end of the automatically generated CREATE TABLE statement and executed as a single statement.

In the *Dynamic Columns Settings* there are two types of SQL Type Mapping, the Name-Based and the KNIME-Based.

- In the *Name-Based SQL Type Mapping* you define the default SQL type mapping for a set of columns based on the column names. You can add a new row containing the name pattern of the columns that should be mapped. The name pattern can either be a

string with wildcard or a regular expression. The mappings defined in the *Name-Based SQL Type Mapping* have a higher priority than the mappings defined in the *KNIME-Based SQL Type Mapping*.

- In the *KNIME-Type-Based SQL Type Mapping* you can define the default SQL type mapping based on a KNIME data type. You can add a new row containing the KNIME data type that should be mapped.

In the *Dynamic Keys Settings* you can dynamically define the key definitions based on the column names. You can add a new row containing the name pattern of the columns that should be used to define a new key. The name pattern can either be a string with wildcard or a regular expression.

Supported wildcards are * (matches any number of characters) and ? (matches one character) e.g. KNI* would match all strings that start with KNI such as KNIME whereas KNI? would match only strings that start with KNI followed by a fourth character.

DB Manipulation

This section describes various DB nodes for in-database manipulation, such as *DB Delete (Table)*, *DB Writer*, *DB Insert*, *DB Update*, *DB Merge*, and *DB Loader* node, as well as the database transaction nodes.

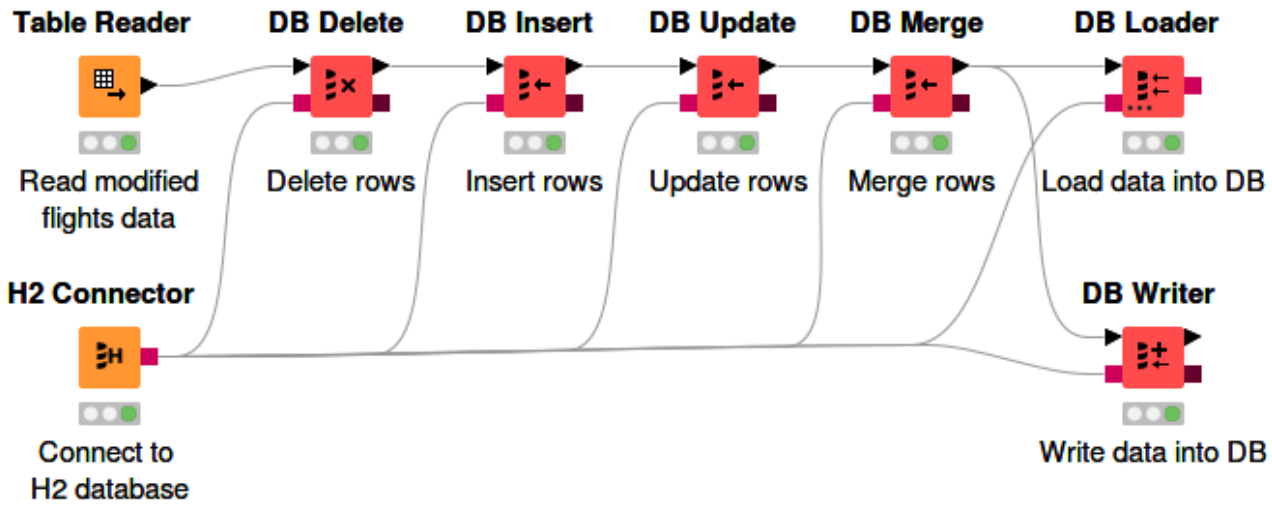

Figure 47. Example of DB Manipulation

DB Delete

The database extension provides two nodes to delete rows from a selected table in the database. The *DB Delete (Table)* node deletes all rows from the database table that match the values of an input KNIME data table whereas the *DB Delete (Filter)* node deletes all rows from the database table that match the specified filter conditions.

DB Delete (Table)

This node deletes rows from a selected table in the database. The input is a DB Connection port that describes the database, and also a KNIME data table containing the values which define which rows to delete from the database. It deletes data rows in the database based on the selected columns from the input table. Therefore all selected column names need to exactly match the column names inside the database. Only the rows in the database table that match the value combinations of the selected columns from the KNIME input data table will be deleted.

The figure below shows the configuration dialog of the *DB Delete (Table)* node. The configuration dialog of the other nodes for DB Manipulation are very similar. You can enter the table name and its corresponding schema or select the table name in the Database

Metadata Browser by clicking *Select a table*.

In addition the identification columns from the input table need to be selected. The names of the selected KNIME table columns have to match the names in the selected database table. All rows in the database table with matching values for the selected columns from the input KNIME data table will be deleted. In SQL this is equivalent to the `WHERE` columns. There are three options:

- *Fail on error*: if selected, the node will fail if any errors occur during execution otherwise it will execute successfully even if one of the input rows caused an exception in the database.
- *Append delete status columns*: if selected, it will add two extra columns in the output table. The first column contains the number of rows affected by the `DELETE` statement. A number greater or equal to zero indicates that the operation was performed successfully. A value of -2 indicates that the operation was performed successfully but the number of rows affected is unknown. The second column will contain a warning message if any exists.
- *Disable DB Data output port*: If selected, it will disable the DB Data output port and the execution of the metadata query at the end of the node execution which might cause problems with databases that do not support subqueries.

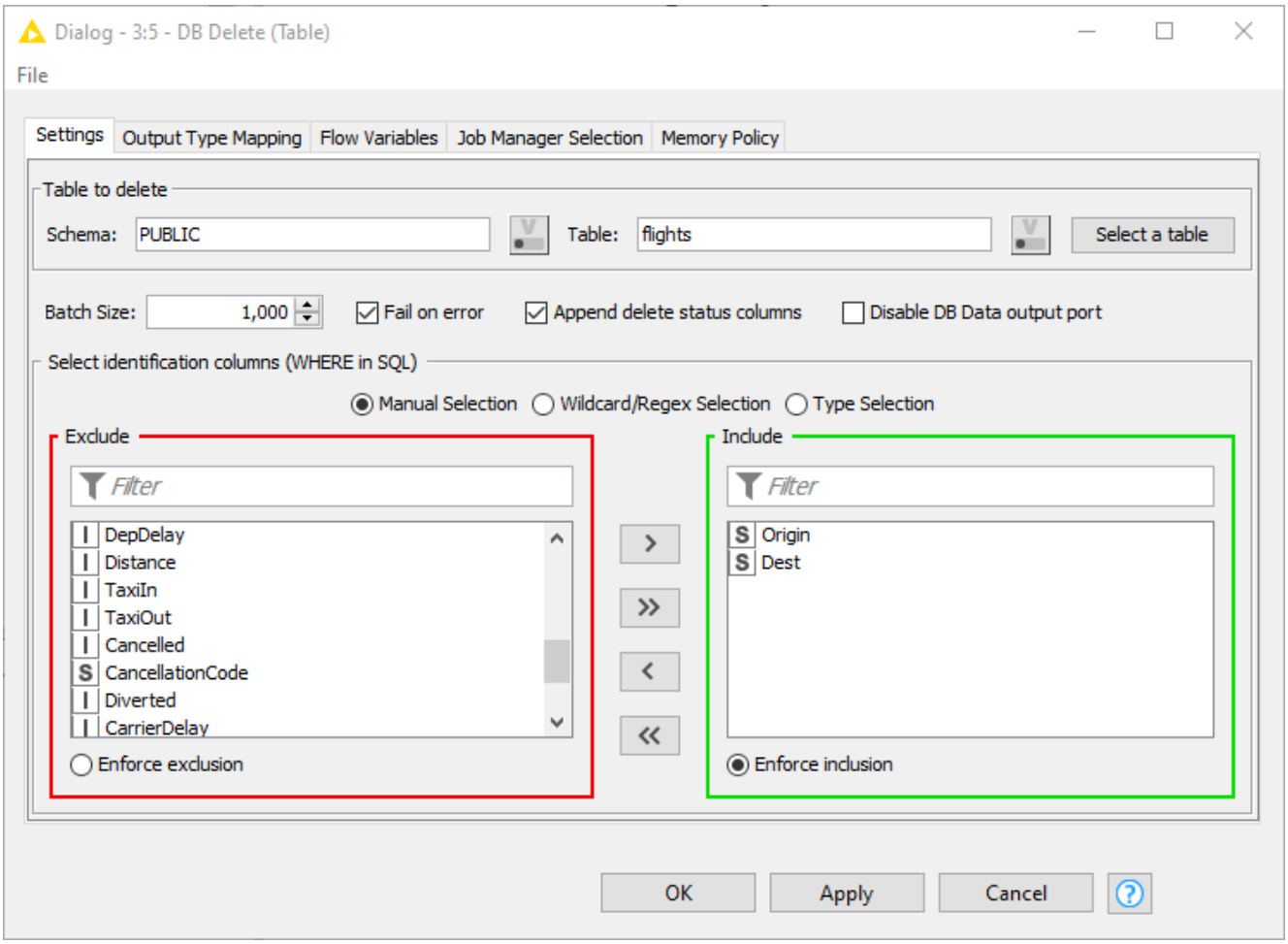

Figure 48. Configuration dialog of the DB Delete (Table) node

The *Output Type Mapping* tab allows you to define mapping rules from KNIME types to database types. For more information on this, please refer to the Type Mapping section.

DB Delete (Filter)

This node deletes rows from a selected table in the database that match the specified filter conditions. The input is a DB Connection port that can be exchanged to a DB Data port. In both cases the input port describes the database. The output port can be exchanged from a DB Connection to a DB Data port as well.

In the node dialog of the *DB Delete (Filter)* node you can enter the table name and its corresponding schema or select the table name in the Database Metadata Browser by clicking *Select a table*. In addition you can specify the filter conditions that are used to identify the rows in the database table that should be deleted. The conditions are used to generate the `WHERE` clause of the `DELETE` statement. All rows that match the filter conditions will be deleted.

For example, the settings specified in the figure shown below will delete all rows from the *flights* table that took place prior the *Year* 2017 and that where neither *Diverted* nor *Cancelled*.

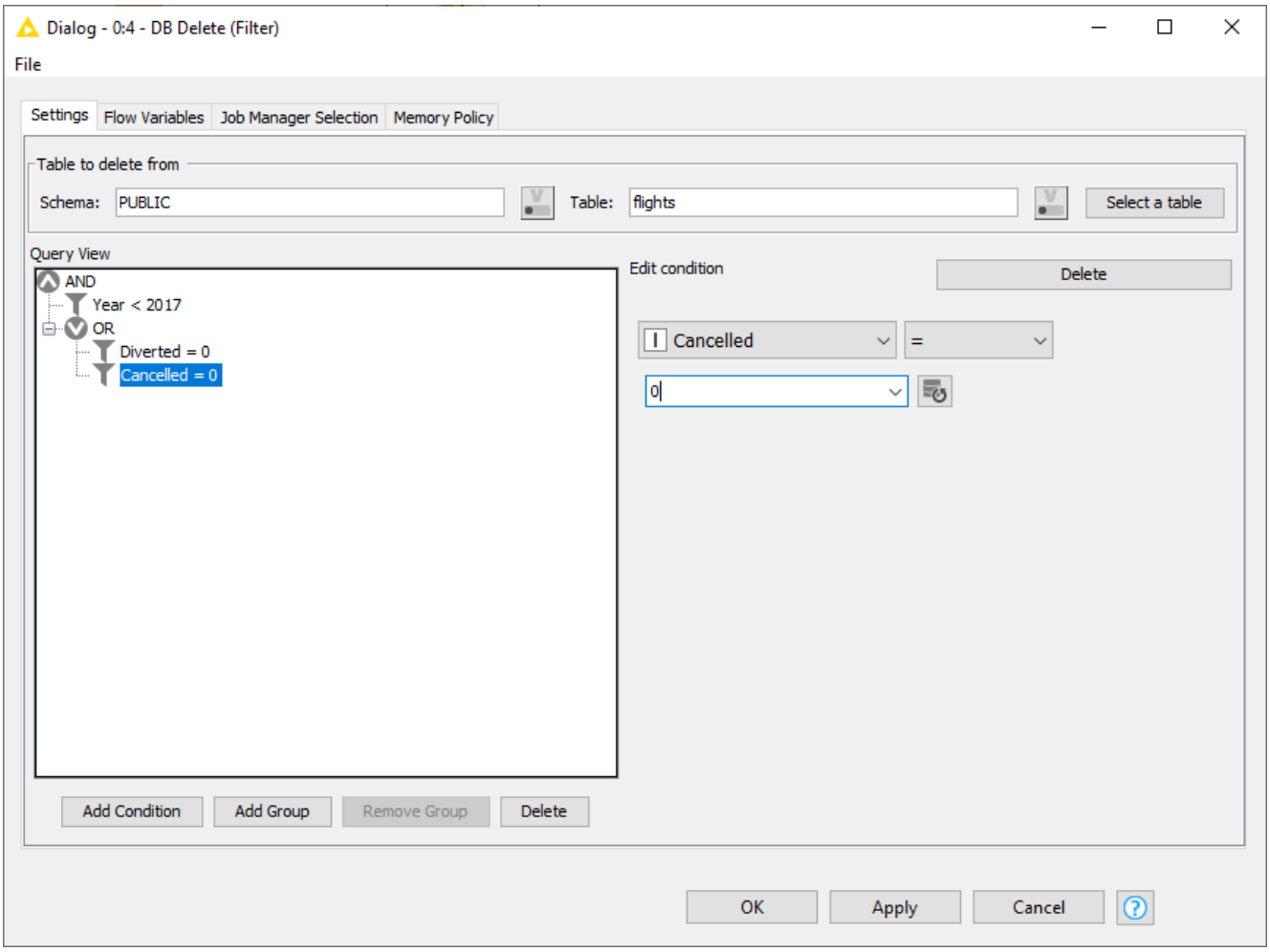

Figure 49. Configuration dialog of the DB Delete (Filter) node

An example of the usage of this node is available on KNIME Hub.

DB Writer

This node inserts the selected values from the input KNIME data table into the specified database tables. It performs the same function as the DB Insert node, but in addition it also creates the database table automatically if it does not exist prior inserting the values. The newly created table will have a column for each selected input KNIME column. The database column names will be the same as the names of the input KNIME columns. The database column types are derived from the given KNIME types and the Type Mapping configuration. All database columns will allow missing values (e.g. `NULL`).

Please use the DB Table Creator node if you want to control the properties of the created database table.

There is also an option to overwrite an existing table by enabling the option *Remove existing table* in the configuration window. Enabling the option will remove any table with the given name from the database and then create a new one. If this option is not selected, the new

data rows will be appended to an existing table. Once the database table exists the node will write all KNIME input rows into the database table in the same way as the DB Insert node.

DB Insert

This node inserts the selected values from the input KNIME data table into the specified database tables. All selected column names need to exactly match the column names within the database table.

DB Update

This node updates rows in the specified database table with values from the selected columns of the input KNIME data table. The identification columns are used in the `WHERE` part of the SQL statement and identify the rows in the database table that will be updated. The columns to update are used in the `SET` part of the SQL statement and contain the values that will be written to the matching rows in the selected database table.

DB Merge

The *DB Merge* node is a combination of the *DB Update* and *DB Insert* node. If the database supports the functionality it executes a `MERGE` statement that inserts all new rows or updates all existing rows in the selected database table. If the database does not support the merge function the node first tries to update all rows in the database table and then inserts all rows where no match was found during the update. The names of the selected KNIME table columns need to match the names of the database table where the rows should be updated.

DB Loader

Starting from 4.3, the DB Loader node employs the new file handling framework, which allows seamless migration between various file systems. For more details, please check out the KNIME File Handling Guide.

This node performs database-specific bulk loading functionality that only some databases (e.g. Hive, Impala, MySQL, PostgreSQL and H2) support to load large amounts of data into an existing database table.

Most databases do not perform data checks when loading the data into the table which might lead to a corrupt data table. The node does perform some preliminary checks such as checking that the column order and column names of the input KNIME data table are compatible with the selected database table. However it does not check the column type compatibility or the values itself. Please make sure that the column types and values of the KNIME table are compatible with the database table.

Depending on the database an intermediate file format (e.g. CSV, Parquet, ORC) is often used for efficiency which might be required to upload the file to a server. If a file needs to be uploaded, any of the protocols supported by the file handling nodes and the database can be used, e.g. SSH/SCP or FTP. After the loading of the data into a table, the uploaded file gets deleted if it is no longer needed by the database. If there is no need to upload or store the file for any reason, a file connection prevents execution.

Some databases such as MySQL and PostgreSQL support file-based and memory-based uploading which require different rights in the database. For example, if you do not have the rights to execute the file-based loading of the data try the memory-based method instead.

If the database supports various loading methods (file-based or memory-based), you can select the method in the *Options* tab, as shown in the example below. Otherwise the *Loader mode* option will not appear in the configuration dialog.

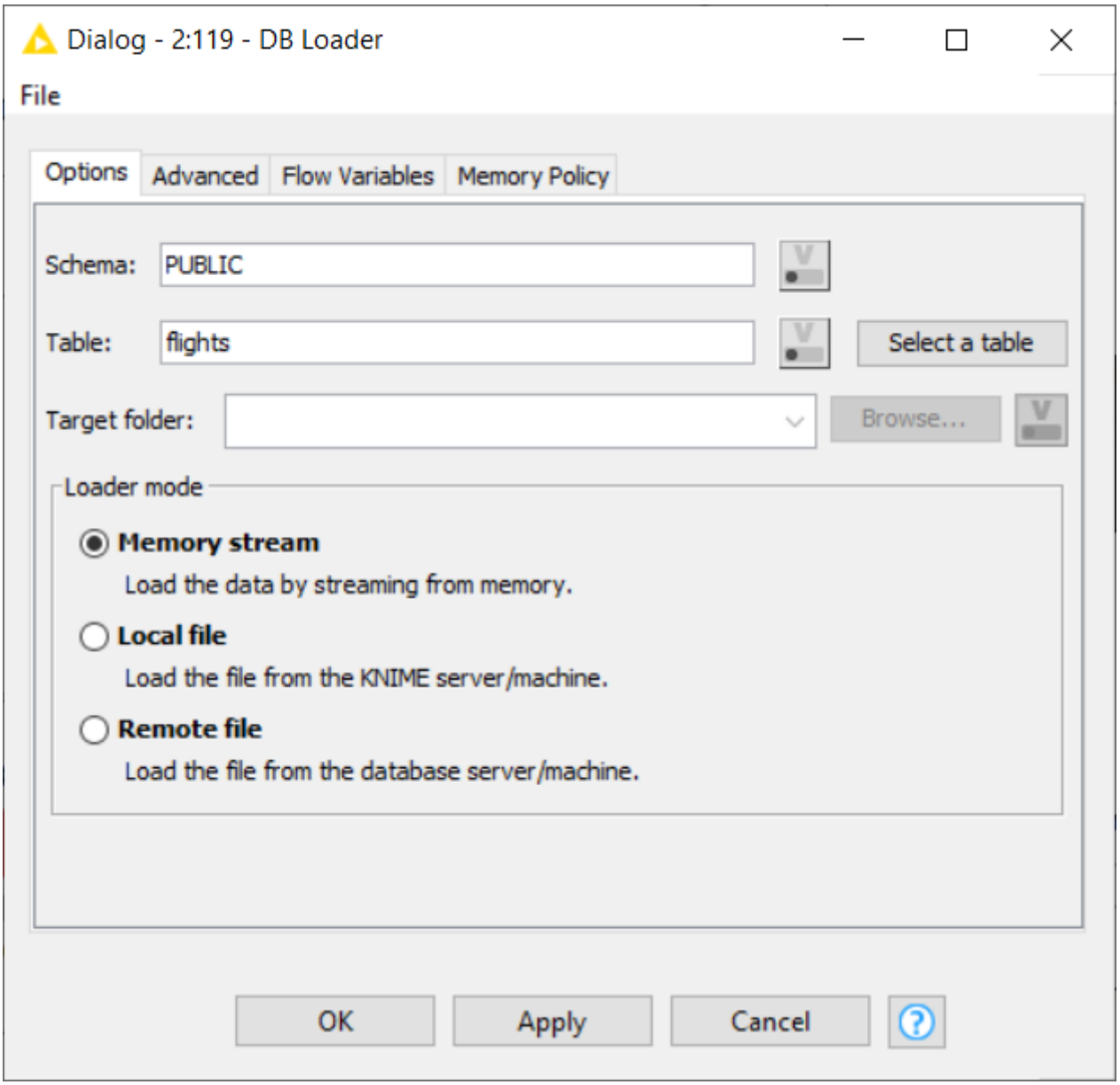

Figure 50. DB Loader: Option

Depending on the connected database the dialog settings may change. For example, MySQL and PostgreSQL use a CSV file for the data transfer. In order to change how the CSV file is created go to the *Advanced* tab.

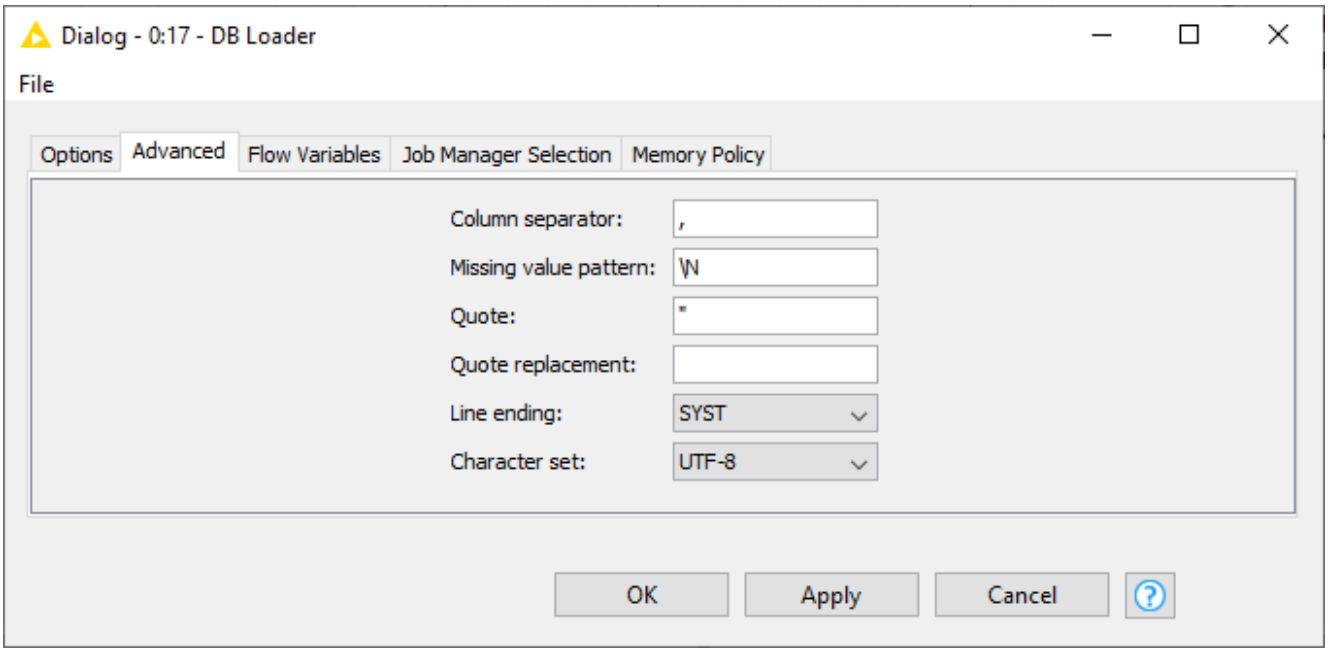

Figure 51. DB Loader: Advanced

DB Transaction Nodes

The database extension also provides nodes to simulate database transaction. A transaction allows you to group several database data manipulation operations into a single unit of work. This unit either executes entirely or not at all.

DB Transaction Start

The *DB Transaction Start* node starts a transaction using the input database connection. As long as the transaction is in process, the input database connection cannot be used outside of the transaction. Depending on the isolation level, other connections might not see any changes in the database while the transaction is in process. The transaction uses the default isolation level of the connected database.

DB Transaction End

The *DB Transaction End* node ends the transaction of the input database connection. The node ends the transaction with a commit that makes all changes visible to other users if executed successfully. Otherwise the node ends the transaction with a rollback returning the database to the state at the beginning of the transaction.

This node has 2 input ports. The first one is the transactional DB connection port which should be connected to from the end of the transaction chain. The second port should contain the transactional DB connection from the output of the *DB Transaction Start* node. If

the transaction is successful and a commit is executed, the DB connection from the first input port will be forwarded to the output port, otherwise in case of a rollback, the DB connection from the second input port will be forwarded.

The figure below shows an example of the transaction nodes. In this example, the transaction consists of two DB Writer nodes that write data to the same table consecutively. If an error occurs during any step of the writing, the changes will not be executed and the database will be returned to the previous state at the beginning of the transaction. If no error occurs, the changes to the database will be commited.

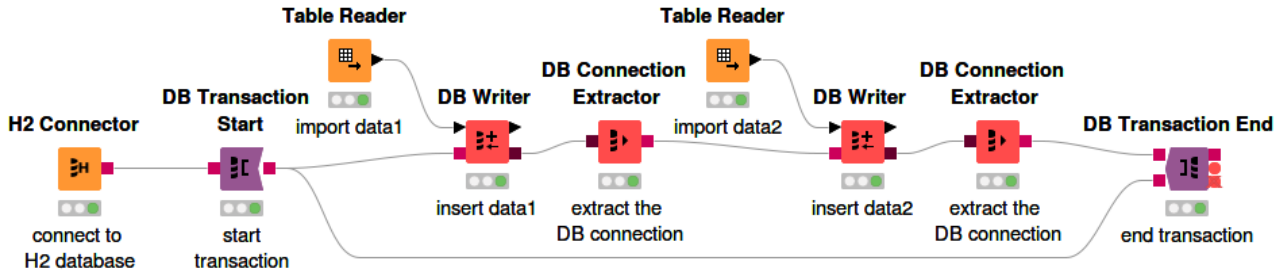

Figure 52. DB Transaction Nodes

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 e.g. Oracle only has one numeric type with different precisions to represent integers but also 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. For example 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. But 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 direction has two sets of rules:

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

The type mapping can be defined and altered at various places in the analysis workflow. The basic configuration can be done in the different connector nodes. They come with a sensible database specific default mapping. The type mapping rules are part of the *DB Connection* and *DB Data* connections and inherited from preceding nodes. In addition to the connector nodes provide all database nodes with a KNIME data table a *Output Type Mapping* tab to map the data types of the nodes input KNIME columns to the types of the corresponding database columns.

The mapping of database types to KNIME types can be altered for any *DB Data* connection via the *DB Type Mapper* node.

DB Type Mapper

The *DB Type Mapper* node changes the database to KNIME type mapping configuration for subsequent nodes by selecting a KNIME type to the given database Type. The configuration dialog allows you to add new or change existing type mapping rules. All new or altered rules are marked as bold.

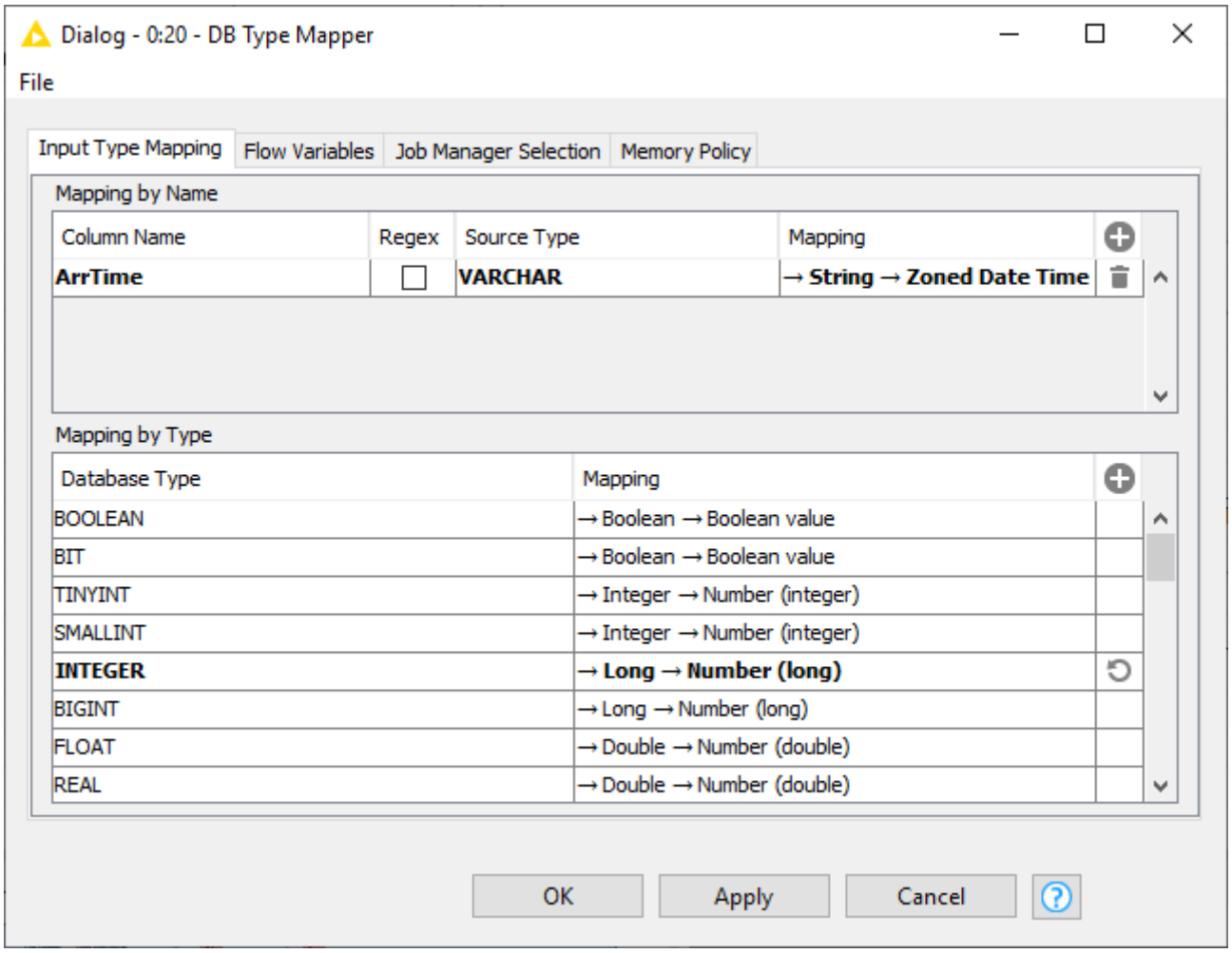

Figure 53. DB Type Mapper configuration dialog

Rules from preceding nodes can not be deleted but only altered.

Migration

This section explains how to migrate your workflow that contains deprecated database nodes to the new database framework. The Workflow Migration Tool can be used to guide you through the process and convert the deprecated database nodes to the corresponding new database nodes. For the mapping between the deprecated and the new nodes, please look at the list in the Node Name Mapping section.

All previously registered JDBC drivers need to be re-registered. For more information on how to register a driver in the new database framework, please refer to the Register your own JDBC drivers section.

Workflow Migration Tool

The workflow migration tool is still in preview. We will continue to add new and revise existing functionality.

The workflow migration tool assists you to migrate existing workflows that contain deprecated database nodes to the new database nodes. The tool does not change any existing workflow but performs the migration on a copy of the original workflow.

As an example, we can see in the figure below a workflow that contains deprecated database nodes. The goal is to use the Workflow Migration Tool to help us migrating the deprecated nodes to the new database nodes.

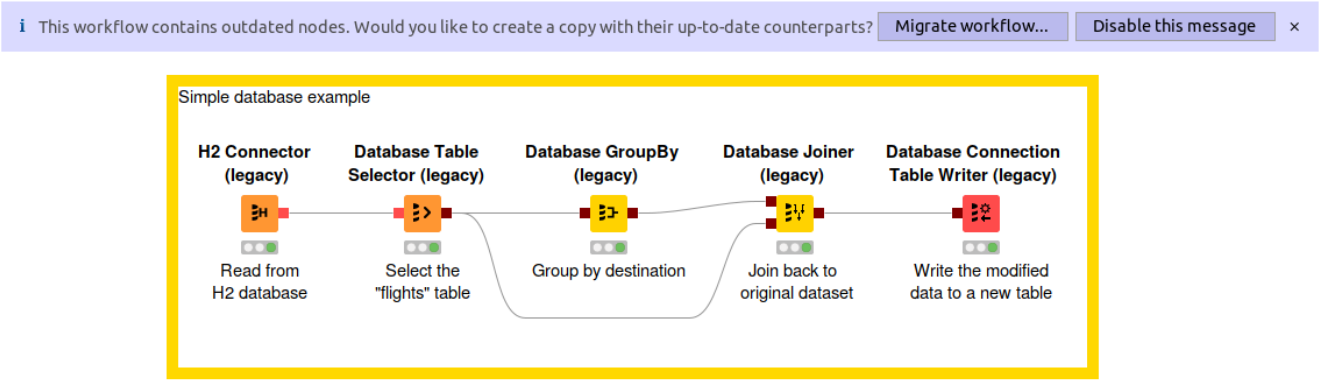

Figure 54. Workflow containing deprecated Database nodes

In order to start the Workflow Migration tool we simply need to open the workflow containing the deprecated database nodes that you want to migrate. A message will appear at the top of the workflow with the option to migrate the workflow (see figure above).

Clicking on *Migrate workflow…* will open the migration wizard window as shown below. In this window, you can change the workflow to migrate (the one containing the deprecated

database nodes), and enter the name for the new workflow, which is a copy of the old workflow but with the deprecated database nodes replaced with the new ones (if available). The default name for the new workflow is the name of the old workflow with *(migrated)* attached as suffix.

The original workflow will not be modified throughout the migration process.

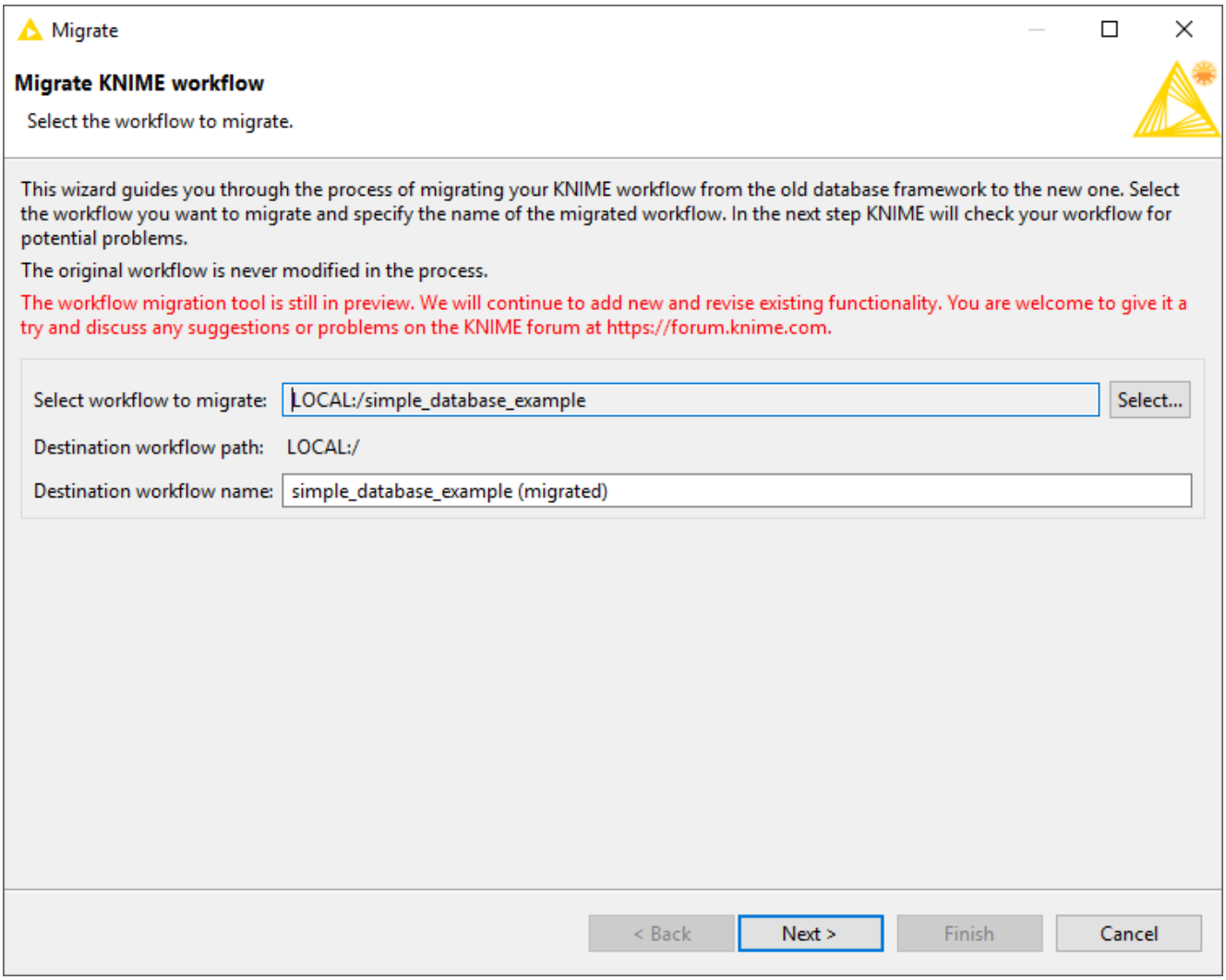

Figure 55. Migration Tool: Select the workflow

Click *next* to get to the next page, as shown below. At this stage the workflow will be analysed, and all deprecated database nodes for which a migration rule exists will be listed here, along with their equivalent new nodes. The tool also performs a preliminary check and shows any potential problems. If you agree with the mapping suggestion, click *Next* to perform the migration process.

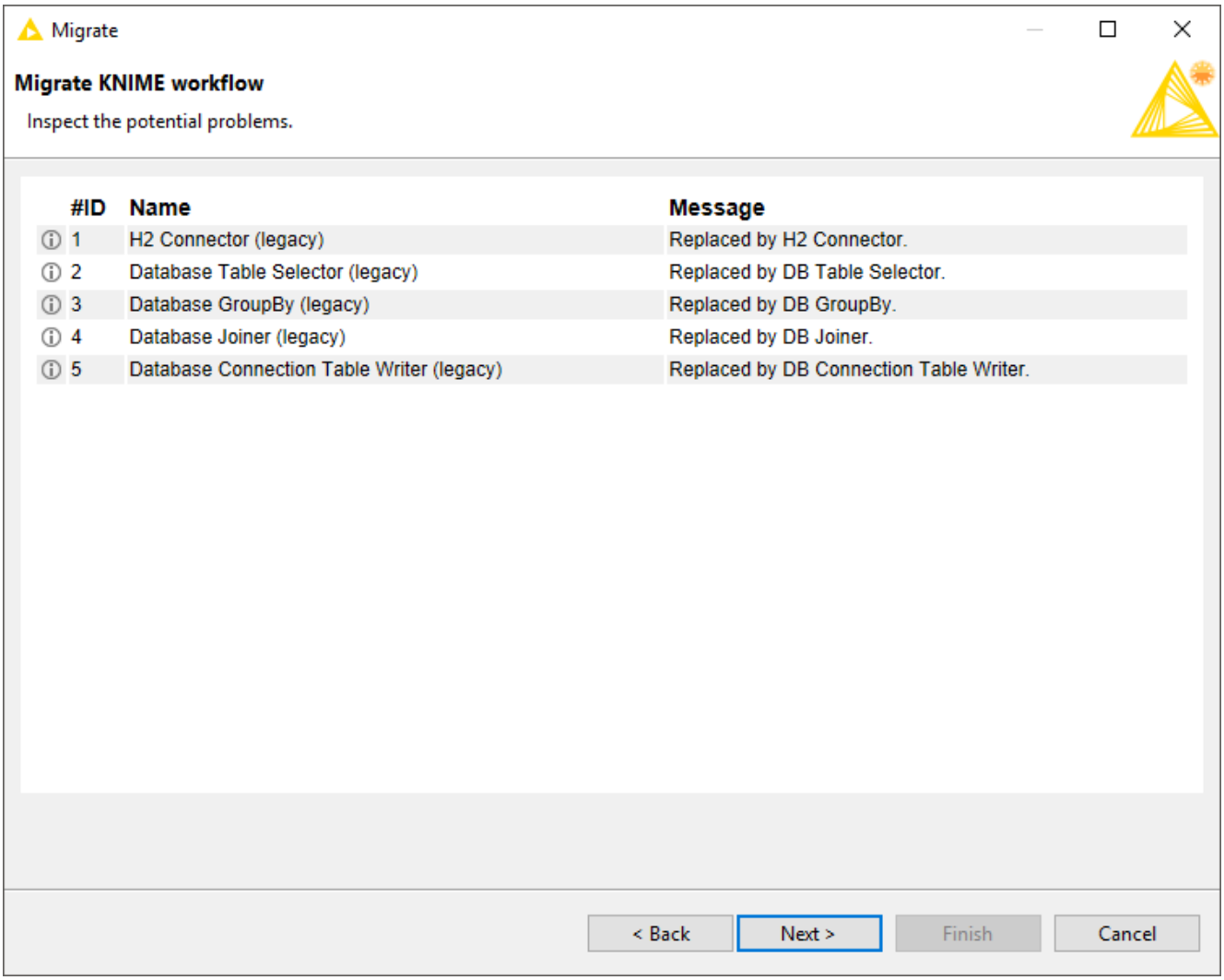

Figure 56. Migration Tool: Show the potential mapping

After the migration process is finished, you can see the migration report like the one shown below. If any warnings or problems happened during the migration process, corresponding messages will be shown in the report. You also have the option to save and open the migration report in HTML format.

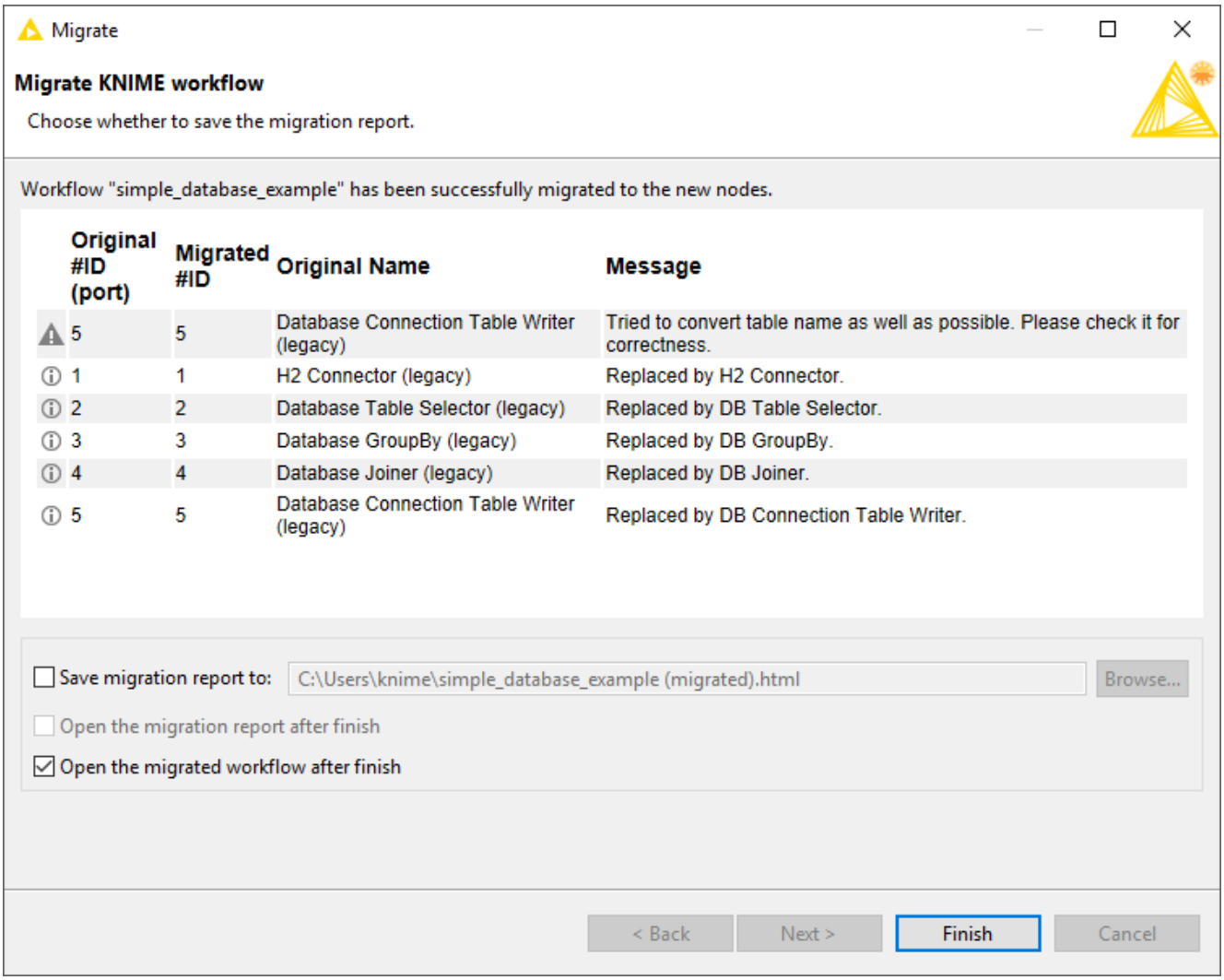

Figure 57. Migration Tool: Migration report

The figure below shows the migrated workflow where all deprecated database nodes are replaced by the new database nodes while keeping all the settings intact.

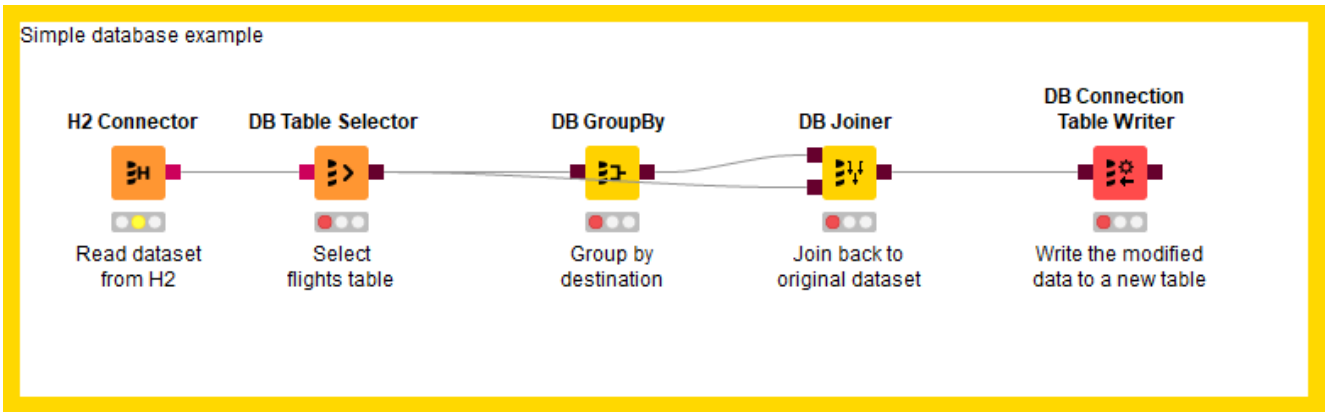

Figure 58. New workflow containing migrated DB nodes

Disabling the workflow migration message

If you don't want to migrate and want to disable the migration message, click on *Disable this message* in the message. The *Preferences* → *Databases* page will open where you can uncheck the option *Offer the migration of workflows that contain deprecated database nodes* as shown in the figure below. Click *Apply and Close* to save the setting and the message will not appear anymore if you open a workflow containing deprecated database nodes. To reverse this setting, simply open *Preferences* → *Databases* page again and enable the check box.

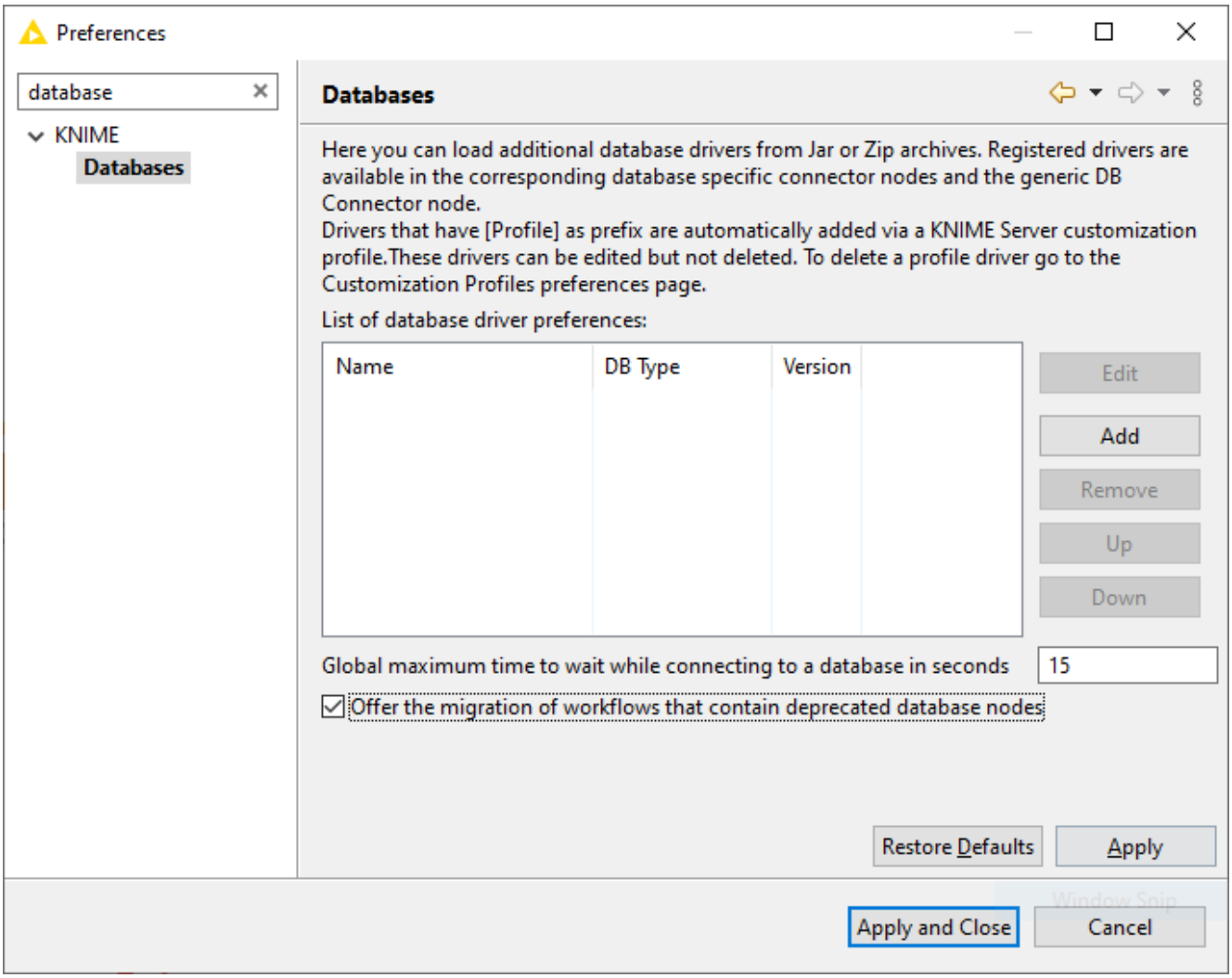

Figure 59. Disabling migration message

Node Name Mapping

The table below shows the mapping between the deprecated database nodes and the new database nodes.

Deprecated Database nodes	New Database nodes
Amazon Athena Connector	Amazon Athena Connector
Amazon Redshift Connector	Amazon Redshift Connector
Database Apply-Binner	DB Apply-Binner
Database Auto-Binner	DB Auto-Binner
Database Column Filter	DB Column Filter
Database Column Rename	DB Column Rename
Database Connection Table Reader	DB Reader
Database Connection Table Writer	DB Connection Table Writer
Database Connector	DB Connector
Database Delete	DB Delete (Table) or DB Delete (Filter)
Database Drop Table	DB Table Remover
Database GroupBy	DB GroupBy
Database Joiner	DB Joiner
Database Looping	DB Looping
Database Numeric-Binner	DB Numeric-Binner
Database Pivot	DB Pivot
Database Query	DB Query
Database Reader	DB Query Reader
Database Row Filter	DB Row Filter
Database Sampling	DB Row Sampling
Database Sorter	DB Sorter
Database SQL Executor	DB SQL Executor

Deprecated Database nodes	New Database nodes
Database Table Connector	Can be replaced with DB Connector and DB Table Selector
Database Table Creator	DB Table Creator
Database Table Selector	DB Table Selector
Database Update	DB Update
Database Writer	DB Writer
H2 Connector	H2 Connector
Hive Connector	Hive Connector
Hive Loader	DB Loader
Impala Connector	Impala Connector
Impala Loader	DB Loader
Microsoft SQL Server Connector	Microsoft SQL Server Connector
MySQL Connector	MySQL Connector
Parameterized Database Query	Parameterized DB Query Reader
PostgreSQL Connector	PostgreSQL Connector
SQL Extract	DB Query Extractor
SQL Inject	DB Query Injector
SQLite Connector	SQLite Connector
Vertica Connector	Vertica Connector
-	Microsoft Access Connector
-	DB Insert
-	DB Merge
-	DB Column Rename (Regex)

Deprecated Database nodes	New Database nodes
-	DB Partitioning
-	DB Type Mapper

Register your own JDBC drivers for the deprecated database framework

The JDBC driver registration page for the deprecated database framework is no longer visible in the KNIME preferences. However, drivers can still be registered for the deprecated database framework.

This section is only valid if you use the Database Connector (deprecated) node. To register driver for the new database framework e.g. the DB Connector node refer to the Register your own JDBC drivers section.

To register new drivers, you need to create a new text file with the name *driver.epf* and the following content:

```
\!/=
/instance/org.knime.workbench.core/database_drivers=<PATH_TO_THE_JDBC_DRIVER>
file_export_version=3.0
```

Here `<PATH_TO_THE_JDBC_DRIVER>` refers to the path that points to the drivers jar file. You can register several drivers by concatenating the paths with *;* as separator. The following shows an example that registers two different JDBC driver (DB2 and Neo4J) with windows file path notation.

```
\!/=
/instance/org.knime.workbench.core/database_drivers=C\:\\KNIME\\JDBC\\db2\\db2jcc4.jar;C
\:\\KNIME\\JDBC\\Neo4J\\neo4j-jdbc-driver-3.4.0.jar
file_export_version=3.0
```

If you use windows path notation in the `<PATH_TO_THE_JDBC_DRIVER>` that includes backslashes each backslash has to be escaped by a second backslash.

Once created you need to import the `driver.epf` file into the KNIME Analytics Platform via *File→Import Preferences*. After importing the driver preferences file, you need to restart KNIME Analytics Platform to have the legacy database framework pickup the new driver files.

Business Hub / Server Setup

This section contains everything related to executing workflows that contain database nodes on KNIME Hub and KNIME Server.

JDBC drivers on KNIME Hub and KNIME Server

KNIME Hub allows you to upload customization profiles to set up JDBC drivers on KNIME Hub executors and KNIME Analytics Platform clients.

KNIME Server also allows you to define customization profiles to automatically set up JDBC drivers on its own executors as well as KNIME Analytics Platform clients.

Instead of customization profiles, it is also possible to register JDBC drivers directly on the executor with a preferences file. In this case however, the preferences of the server executor and all KNIME Analytics Platform clients need to be kept in sync manually, so that the same drivers are available on both ends.

Server-side steps

1. Create a *profile folder* inside `<server-repository>/config/client-profiles`. The name of the folder corresponds to the name of the profile. The folder will hold preferences and other files to be distributed.
2. Copy the `.jar` file that contains the JDBC driver into the profile folder.
3. Inside the profile folder, create a preferences file (file name ends with `.epf`) with the following contents:

   ```
   /instance/org.knime.database/drivers/<DRIVER_ID>/database_type=<DATABASE>
   /instance/org.knime.database/drivers/<DRIVER_ID>/driver_class=<DRIVER_CLASS_NAME>
   /instance/org.knime.database/drivers/<DRIVER_ID>/paths/0=${profile:location}/<DRIV
   ER_JAR>
   /instance/org.knime.database/drivers/<DRIVER_ID>/url_template=<URL_TEMPLATE>
   /instance/org.knime.database/drivers/<DRIVER_ID>/version=<DRIVER_VERSION>
   ```

   Where:

   - `<DRIVER_ID>`: A unique ID for the JDBC driver, consisting only of alphanumeric characters and underscores.
   - `<DATABASE>`: The database type. Please consult the preference page shown in Register your own JDBC drivers for the list of currently available types.

- `<DRIVER_CLASS_NAME>`: The JDBC driver class, for example `oracle.jdbc.OracleDriver` for Oracle.
- `<DRIVER_JAR>`: The name of the `.jar` file (including the file extension) that contains the JDBC driver class. Note that the variable `${profile:location}` stands for the location of the profile folder on each client that downloads the customization profile. It will be automatically replaced with the the correct location by each client.
- `<URL_TEMPLATE>`: The JDBC URL template to use for the driver. Please refer to the JDBC URL Template section for more information. Note that colons (`:`) and backslashes (`\`) have to be escaped with a backslash. Example:

```
jdbc\:oracle\:thin\:@<host>\:<port>/<database>
```

- `<DRIVER_VERSION>`: The version of the JDBC driver e.g. `12.2.0`. The value can be chosen at will.

4. KNIME Server executors need to be made aware of a customization profile, by adding this information to the `knime.ini` file, so that they can request it from KNIME Server. Please consult the respective section of the KNIME Server Administration Guide on how to set this up.

Please note that there are database-specific examples at the end of this section.

Client-side steps

KNIME Analytics Platform clients need to be made aware of a customization profile so that they can request it from KNIME Server. Please consult the respective section of the KNIME Server Administration Guide for a complete reference on how to set this up.

In KNIME Analytics Platform, you can go to *File* → *Preferences* → *KNIME* → *Customization Profiles*. This opens the *Customization Profiles* page where you can choose which KNIME Server and profile to use. The changes will take effect after restarting KNIME Analytics Platform.

To see whether the driver has been added, go to *File* → *Preferences* → *KNIME* → *Databases*. In this page, drivers that are added via a customization profile are marked as *origin: profile* after the driver ID (see figure below). These drivers can be edited but not deleted. To delete a profile driver, please go to the *Customization Profiles* page and unselect the respective profile.

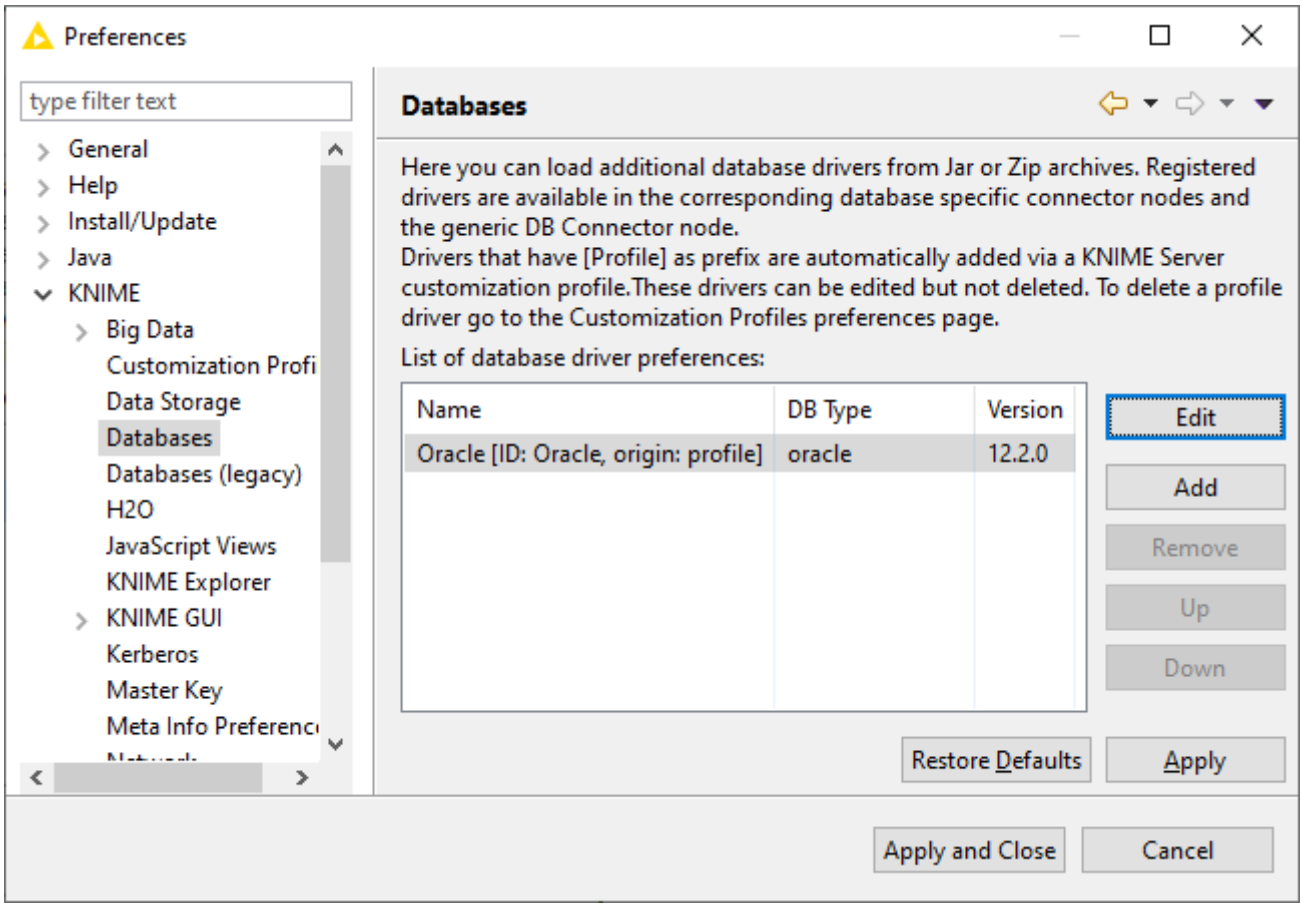

Figure 60. DB Preferences page

Default JDBC Parameters

Default JDBC Parameters provide a way for hub admins to inject JDBC parameters on JDBC connections made from workflows running on KNIME Hub. These parameters take precedence over values specified in the connector node. To specify an additional JDBC parameter, add the following lines to the `.epf` file of your customization profile:

```
/instance/org.knime.database/drivers/<DRIVER_ID>/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connection.jdbc.properties/<JDBC_PARAMETER>/type=<TYPE>
/instance/org.knime.database/drivers/<DRIVER_ID>/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connection.jdbc.properties/<JDBC_PARAMETER>/value=<VALUE>
```

Where:

- `<DRIVER_ID>`: The unique ID for the JDBC driver.
- `<JDBC_PARAMETER>`: The name of the JDBC parameter to set.
- `<TYPE>` and `<VALUE>`: A type and value that specifies what to set the JDBC parameter to.

<TYPE> and <VALUE> can be chosen as follows:

- Setting <TYPE> to CONTEXT_PROPERTY allows to specify workflow context related properties. <VALUE> can be set to one of:
  - context.workflow.name: The name of the KNIME workflow.
  - context.workflow.path: The mountpoint-relative workflow path.
  - context.workflow.absolute-path: The absolute workflow path.
  - context.workflow.username: The name of the KNIME Hub or KNIME Server user that executes the workflow.
  - context.workflow.temp.location: The path of the workflow temporary location.
  - context.workflow.author.name: The name of the workflow author.
  - context.workflow.last.editor.name: The name of the user who last edited the workflow.
  - context.workflow.creation.date: The creation date of the workflow.
  - context.workflow.last.time.modified: The last modified time of the workflow.
  - context.job.id: The job id when run on KNIME Hub or KNIME Server.
- Setting <TYPE> to CREDENTIALS_LOGIN allows to specify the *login name* from a credentials flow variable. Set <VALUE> to the name of the flow variable. Please note that the value is the **name** of the credentials flow variable to get the login from but not the login itself.
- Setting <TYPE> to CREDENTIALS_PASSWORD allows to specify the *password* from a credentials flow variable. Set <VALUE> to the name of the flow variable. Please note that the value is the **name** of the credentials flow variable to get the password from but not the password itself.
- Setting <TYPE> to DELEGATED_GSS_CREDENTIAL allows to pass in a delegated GSS credential. The credential is only available for Kerberos authentication and during KNIME Hub execution. The KNIME Hub executor creates the delegated GSS credential in the name of the current KNIME Hub user that executes the workflow. Do not set any <VALUE> for this parameter type. For an example on how to use this parameter see the Microsoft SQL Hub example.
- Setting <TYPE> to FLOW_VARIABLE allows to specify a flow variable. Set <VALUE> to the name of the flow variable.
- Setting <TYPE> to GSS_PRINCIPAL_NAME allows to pass in the principal name of the

(delegated) Kerberos user e.g., `User@REALM.COM`. The Kerberos principal name is only available for Kerberos authentication. Do not set any `<VALUE>` for this parameter type. For an example on how to use this parameter see the PostgreSQL example.

- Setting `<TYPE>` to `GSS_PRINCIPAL_NAME_WITHOUT_REALM` allows to pass in the principal name without the REALM of the (delegated) Kerberos user. For example, if the Kerberos principal name is `User@REALM.COM` the value will be `User`. The Kerberos principal name is only available for Kerberos authentication. Do not set any `<VALUE>` for this parameter type. For an example on how to use this parameter see the PostgreSQL example.
- Setting `<TYPE>` to `LITERAL` allows to specify a literal value that does not undergo any further substitution. Set `<VALUE>` to the literal value.
- Setting `<TYPE>` to `LOCAL_URL` allows to specify a URL in `<VALUE>`, such as a "knime" URL.

Please note that there are database-specific examples at the end of this section.

Reserved JDBC Parameters

Certain JDBC parameters can cause security issues when a workflow is executed on KNIME Hub, e.g. `DelegationUID` for Impala/Hive connections using a Simba based driver. Such parameters can be marked as *reserved* to prevent workflows from using them on KNIME Hub. To set a parameter as reserved, add the following lines to the `.epf` file of your customization profile:

```
/instance/org.knime.database/drivers/<DRIVER_ID>/attributes/reserved/org.knime.database.
util.DerivableProperties/knime.db.connection.jdbc.properties/<JDBC_PARAMETER>=true
```

Or the shorter version:

```
/instance/org.knime.database/drivers/<DRIVER_ID>/attributes/reserved/*/knime.db.connecti
on.jdbc.properties/<JDBC_PARAMETER>=true
```

Where:

- `<DRIVER_ID>`: The unique ID for the JDBC driver.
- `<JDBC_PARAMETER>`: The name of the JDBC parameter.

Please note that there are database-specific examples at the end of this section.

Connection Initialization Statement

The connection initialization statement provides a way for hub admins to inject a SQL statement that is executed when a JDBC connection is created from a workflow running on KNIME Hub. This statement takes precedence over all other statements executed within the workflow. To specify an initialization statement, add the following line to the `.epf` file of your customization profile:

```
/instance/org.knime.database/drivers/<DRIVER_ID>/attributes/additional/java.lang.String/
knime.db.connection.init_statement/value=<VALUE>
```

Where:

- `<DRIVER_ID>`: The unique ID for the JDBC driver.
- `<VALUE>`: A value that specifies the statement to execute. It is possible to use `CONTEXT_PROPERTY` variables inside the `<VALUE>`. These variables are replaced by the workflow context related properties right before the statement is executed. These variables have the following format: `${variable-name}`. The following variable names are available:
  - `context.workflow.name`: The name of the KNIME workflow.
  - `context.workflow.path`: The mountpoint-relative workflow path.
  - `context.workflow.absolute-path`: The absolute workflow path.
  - `context.workflow.username`: The name of the KNIME Hub or KNIME Server user that executes the workflow.
  - `context.workflow.temp.location`: The path of the workflow temporary location.
  - `context.workflow.author.name`: The name of the workflow author.
  - `context.workflow.last.editor.name`: The name of the user who last edited the workflow.
  - `context.workflow.creation.date`: The creation date of the workflow.
  - `context.workflow.last.time.modified`: The last modified time of the workflow.
  - `context.job.id`: The job id when run on KNIME Hub or KNIME Server.

Kerberos Constrained Delegation

This section describes how to configure KNIME Hub to perform Kerberos constrained delegation (or user impersonation) when connecting to a Kerberos-secured database such as Apache Hive, Apache Impala, Microsoft SQL Server or PostgreSQL. Constrained delegation allows the KNIME Hub to execute database operations on behalf of the user that executes a workflow on a KNIME Hub Executor rather than the KNIME Executor Kerberos ticket user.

To get started, you need to configure the KNIME Hub to authenticate itself against Kerberos. To do so you need to setup the KNIME Hub Executors as described in the Kerberos Admin Guide.

Once all KNIME Hub Executors are setup to obtain a Kerberos ticket you can enable Kerberos constrained delegation for individual JDBC drivers using one of the following methods.

Please note that there are database-specific examples at the end of this section.

Default JDBC Parameters

Default JDBC Parameters provide a way for hub admins to inject JDBC parameters on JDBC connections made from workflows running on KNIME Hub. Depending on the database a different parameter and value is used to perform constrained delegation. Some drivers only require the name of the user that should be impersonated. This can be done using the name of the KNIME Hub user that executes the workflow (`context.workflow.username`) as the value of the driver specific property. See Hive or Impala for an example. Other drivers require the delegated GSS credential as parameter which can automatically be passed to the driver via the value type `DELEGATED_GSS_CREDENTIAL`. The GSS credential is obtained using the MS-SFU Kerberos 5 Extension. See Microsoft SQL Server for an example.

Service Ticket Delegation

If the driver does not provide a dedicated parameter to do constrained delegation the KNIME Executor can request a Kerberos service ticket on behalf of the user that executes the workflow using the MS-SFU Kerberos 5 Extension. The obtained service ticket is then used by the JDBC driver when establishing the connection to the database.

To request the service ticket the KNIME Executor requires the service name and the fully qualified hostname. To specify the service name add the following line to the `.epf` file of your KNIME Executors customization profile:

```
/instance/org.knime.database/drivers/<DRIVER_ID>/attributes/additional/java.lang.String/
knime.db.connection.kerberos_delegation.service/value=<VALUE>
```

Whereas `<VALUE>` is the name of the service to request the Kerberos ticket for. See PostgreSQL for an example.

The fully qualified hostname is automatically extracted from the hostname setting for all dedicated connector nodes. For the generic DB Connector node the name is extracted from the JDBC connection string using a default regular expression (`.*(?:@|/)([^:;,/\\]*).*`) that should work out of the box for most JDBC strings. However if necessary it can be changed by adding the following line to the `.epf` file of your KNIME Executors customization profile:

```
/instance/org.knime.database/drivers/<DRIVER_ID>/attributes/additional/java.lang.String/
knime.db.connection.kerberos_delegation.host_regex/value=<VALUE>
```

The `<VALUE>` should contain a regular expression that extracts the fully qualified hostname from the JDBC URL with the first group matching the hostname.

Connection Initialization Statement

The connection initialization statement provides a way for Hub admins to inject a SQL statement that is executed when a JDBC connection is created from a workflow running on KNIME Hub. This function can be used for constrained delegation in some databases such as Exasol by executing a specific SQL statement with the `context.workflow.username` as variable e.g.

```
IMPERSONATE ${context.workflow.username};
```

Example: Apache Hive™

Connections to Apache Hive require further setup steps depending on the used JDBC driver. In this example we will show how to:

1. Register the proprietary Hive JDBC driver provided by Cloudera on KNIME Hub.
2. Configure user impersonation on KNIME Hub (for both embedded and proprietary Hive JDBC driver).

Proprietary Simba-based JDBC driver registration

If the use of embedded open-source *Apache Hive JDBC Driver* is preferred, skip to the next section.

1. Download the proprietary Hive JDBC driver from the Cloudera website.
2. Create the profile folder inside `<server-repository>/config/client-profiles` and name it `ClouderaHive` (for example).
3. Copy `HiveJDBC41.jar` from the downloaded JDBC driver into the newly created profile folder.
4. In the profile folder, create a new preferences file called `hive.epf` (for example) with the following contents:

```
/instance/org.knime.database/drivers/cloudera_hive/database_type=hive
/instance/org.knime.database/drivers/cloudera_hive/driver_class=com.cloudera.hive.
jdbc41.HS2Driver
/instance/org.knime.database/drivers/cloudera_hive/paths/0=${profile:location}/Hiv
eJDBC41.jar
/instance/org.knime.database/drivers/cloudera_hive/url_template=jdbc\:hive2\://<ho
st>\:<port>/[database]
/instance/org.knime.database/drivers/cloudera_hive/version=2.6.0
```

5. If, as recommended, KNIME Hub has to impersonate workflow users, go to the next section.

User impersonation on Hive

This example sets up the Hive JDBC driver (embedded or proprietary) so that KNIME Hub will impersonate workflow users on JDBC connections.

Activating user impersonation on Hive depends on the used JDBC driver:

- For the embedded Apache Hive JDBC driver, add the following lines to the KNIME Hub preferences file.

```
/instance/org.knime.database/drivers/hive/attributes/additional/org.knime.database.util.
DerivableProperties/knime.db.connection.jdbc.properties/hive.server2.proxy.user/type=CON
TEXT_PROPERTY
/instance/org.knime.database/drivers/hive/attributes/additional/org.knime.database.util.
DerivableProperties/knime.db.connection.jdbc.properties/hive.server2.proxy.user/value=co
ntext.workflow.username
/instance/org.knime.database/drivers/hive/attributes/reserved/*/knime.db.connection.jdbc
.properties/hive.server2.proxy.user=true
```

- For the proprietary Simba-based JDBC driver, add the following lines to the preferences file (use the preferences file created in the previous step (step 4).

```
/instance/org.knime.database/drivers/cloudera_hive/attributes/additional/org.knime.datab
ase.util.DerivableProperties/knime.db.connection.jdbc.properties/DelegationUID/type=CONT
EXT_PROPERTY
/instance/org.knime.database/drivers/cloudera_hive/attributes/additional/org.knime.datab
ase.util.DerivableProperties/knime.db.connection.jdbc.properties/DelegationUID/value=con
text.workflow.username
/instance/org.knime.database/drivers/cloudera_hive/attributes/reserved/*/knime.db.connec
tion.jdbc.properties/DelegationUID=true
```

Example: Apache Impala™

In this example we will register the proprietary Impala JDBC driver provided by Cloudera on KNIME Hub. This example sets up the driver so that KNIME Hub will impersonate workflow users on JDBC connections.

If you use the embedded open-source *Apache Hive JDBC Driver (for Impala)*, you don't need to do this step. However, please note that in this case user impersonation on KNIME Hub is not possible due to limitations of the driver.

1. Download the proprietary Impala JDBC from the Cloudera website.
2. Create the profile folder inside `<server-repository>/config/client-profiles` and name it `ClouderaImpala` (for example).
3. Copy `ImpalaJDBC41.jar` from the downloaded JDBC driver into the newly created profile folder.
4. In the profile folder, create a new preferences file called `impala.epf` (for example) with the following contents:

```
/instance/org.knime.database/drivers/cloudera_impala/database_type=impala
/instance/org.knime.database/drivers/cloudera_impala/driver_class=com.cloudera.impala.jdbc.Driver
/instance/org.knime.database/drivers/cloudera_impala/paths/0=${profile:location}/ImpalaJDBC41.jar
/instance/org.knime.database/drivers/cloudera_impala/url_template=jdbc\:impala\://<host>\:<port>/[database]
/instance/org.knime.database/drivers/cloudera_impala/version=2.6.0
/instance/org.knime.database/drivers/cloudera_impala/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connection.jdbc.properties/DelegationUID/type=CONTEXT_PROPERTY
/instance/org.knime.database/drivers/cloudera_impala/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connection.jdbc.properties/DelegationUID/value=context.workflow.username
/instance/org.knime.database/drivers/cloudera_impala/attributes/reserved/*/knime.db.connection.jdbc.properties/DelegationUID=true
```

Note that the last three lines use the `DelegationUID` JDBC parameter to force user impersonation (**recommended**). If you do not want KNIME Hub to impersonate workflow users you can remove the last three lines.

Example: Microsoft SQL Server

Connections to Microsoft SQL Server require further setup steps. In this example we will show how to:

1. Register the SQL Server JDBC driver provided by Microsoft on KNIME Hub.
2. Configure user impersonation on KNIME Hub which is recommended for Kerberos authentication.

Microsoft driver installation

The SQL Server JDBC driver from Microsoft requires some special licensing that you need to agree to. That is why KNIME provides an additional plug-in to install the driver. In order to install the plug-in follow the steps as described in the Third-party Database Driver Plug-in section.

Constrained delegation on Microsoft SQL Server

If you use Kerberos based authentication for Microsoft SQL Server, we recommend to also setup user impersonation. This example sets up the Microsoft SQL Server JDBC driver so that KNIME Hub will impersonate workflow users on JDBC connections. For further details

about using Kerberos integrated authentication with Microsoft SQL Server see the Microsoft SQL Server documentation.

Activate the user impersonation for the embedded Microsoft SQL Server driver, by adding the following line to the KNIME Hub preferences file.

```
/instance/org.knime.database/drivers/built-in-mssqlserver-
9.4.0/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connect
ion.jdbc.properties/gsscredential/type=DELEGATED_GSS_CREDENTIAL
```

In the Microsoft SQL Server Connector ensure that you have selected Kerberos in the *Authentication* setting in the configuration window and added the following two parameters to the JDBC Parameters tab:

```
authenticationScheme=JavaKerberos
integratedSecurity=true
```

These parameters can be set automatically as Default JDBC Parameters via the KNIME Hub preferences file by adding the following lines:

```
/instance/org.knime.database/drivers/built-in-mssqlserver-
9.4.0/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connect
ion.jdbc.properties/authenticationScheme/type=literal
/instance/org.knime.database/drivers/built-in-mssqlserver-
9.4.0/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connect
ion.jdbc.properties/authenticationScheme/value=JavaKerberos

/instance/org.knime.database/drivers/built-in-mssqlserver-
9.4.0/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connect
ion.jdbc.properties/integratedSecurity/type=literal
/instance/org.knime.database/drivers/built-in-mssqlserver-
9.4.0/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connect
ion.jdbc.properties/integratedSecurity/value=true
```

If you want to use your own driver you must exchange the built-in driver id (*built-in-mssqlserver-9.4.0*) with the ID of your driver. For further information on how to setup your own driver see the JDBC drivers on KNIME Hub and KNIME Server section.

For further details about the parameters that are used for constrained delegation see the Microsoft SQL Server documentation.

Example: Oracle Database

Connections to Oracle Database require further setup steps. In this example we will show how to:

1. Register the Oracle Database driver provided by Oracle on KNIME Hub.
2. Configure user impersonation on KNIME Hub which is recommended for Kerberos authentication.

Oracle Database driver installation

The Oracle Database JDBC driver requires some special licensing that you need to agree to. That is why KNIME provides an additional plug-in to install the driver. In order to install the plug-in follow the steps as described in the Third-party Database Driver Plug-in section.

Constrained delegation on Oracle Database

If you use Kerberos based authentication for Oracle Database, we recommend to also setup constrained delegation.

Activate the Kerberos constrained delegation for the installed Oracle driver by adding the following lines to the KNIME Hub preferences file (e.g. `oracle.epf`).

```
/instance/org.knime.database/drivers/built-in-oracle-
19.14.0/attributes/additional/java.lang.String/knime.db.connection.kerberos_delegation.s
ervice/value=oracle
```

If you want to use your own driver you must exchange the built-in driver id (*built-in-oracle-19.14.0*) with the ID of your driver. For further information on how to setup your own driver see the JDBC drivers on KNIME Hub and KNIME Server section.

The service name itself is not used so any non-empty string will enable constrained delegation.

Example: PostgreSQL

If you use Kerberos based authentication for PostgreSQL, we recommend to also setup constrained delegation.

Activate the Kerberos constrained delegation for the embedded PostgreSQL driver by adding the following line to the KNIME Hub preferences file.

```
/instance/org.knime.database/drivers/built-in-postgres-
42.3.5/attributes/additional/java.lang.String/knime.db.connection.kerberos_delegation.se
rvice/value=postgres
```

The default service name for PostgreSQL is *postgres*, but it might be different depending on your database setup. In that case, you need to change *postgres* in the above-mentioned line to the service name.

By default the PostgreSQL driver uses the operation system user as user during login when using Kerberos authentication which will cause problems when the workflow is executed on the KNIME Hub. Depending on your database setup you either want to set it to the Kerberos principal name with or without REALM or the `context.workflow.username` depending on your database setup.

To set the user name to the principal name including the REALM e.g. *User@REALM.COM* use the following line:

```
/instance/org.knime.database/drivers/built-in-postgres-
42.3.5/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connec
tion.jdbc.properties/user/type=GSS_PRINCIPAL_NAME
```

To use the principal name without the REALM use this line instead:

```
/instance/org.knime.database/drivers/built-in-postgres-
42.3.5/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connec
tion.jdbc.properties/user/type=GSS_PRINCIPAL_NAME_WITHOUT_REALM
```

If you want to use your own driver you must exchange the built-in driver id (*built-in-postgres-42.3.5*) with the ID of your driver. For further information on how to setup your own driver see the JDBC drivers on KNIME Hub and KNIME Server section.

KNIME AG
Talacker 50
8001 Zurich, Switzerland
www.knime.com
info@knime.com