Open for Innovation
KNIME

KNIME Database Extension Guide

KNIME AG, Zurich, Switzerland
Version 4.0 (last updated on 2019-09-17)

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.

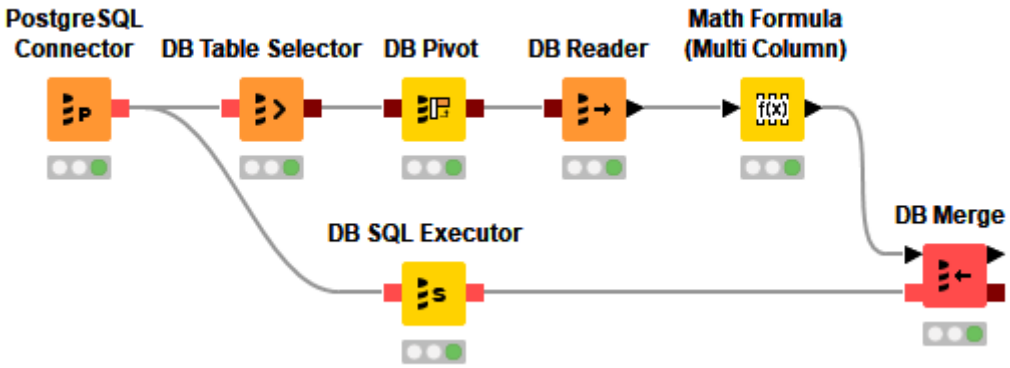


Figure 1. Example workflow using DB nodes

Port Types

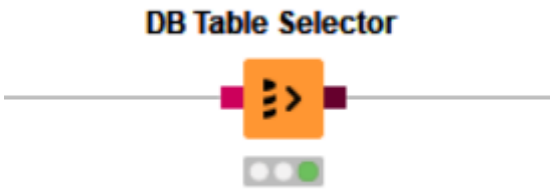


Figure 2. Two types of Database port

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 contains not only the DB Session, but also the DB Data object, which is

described by a SQL query.

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.

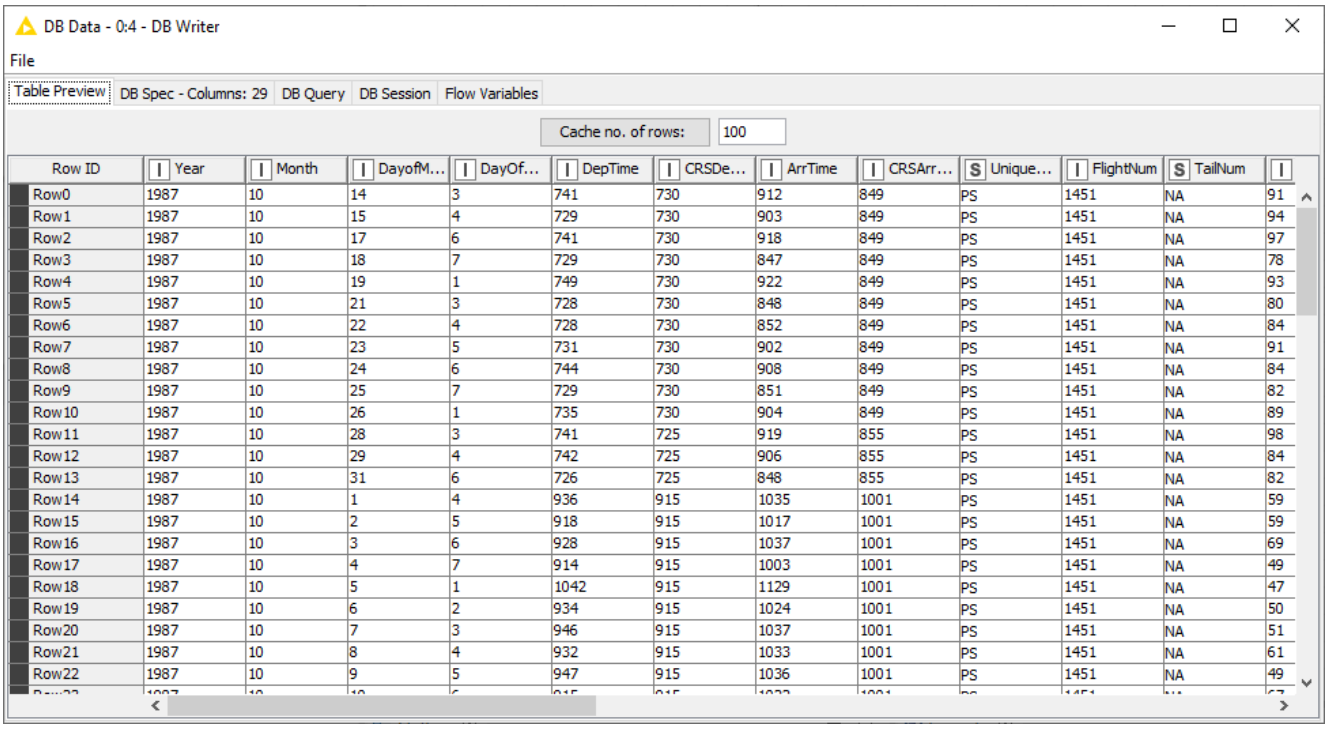
DB Connection outport view

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 subset of the intermediate result via the DB Data outport view. In addition to information about the *DB Session*, the DB Data outport view contains the preview and specification of the output data.

The *Table Preview* tab in the outport view shows an empty table at the beginning. Clicking on *Cache no. of rows:* will execute the intermediate SQL query and cache a subset of the output which then will be shown in the outport view. By default only the first 100 rows are cached, but you can set your own value at the top.



Row ID	Year	Month	DayofM...	DayOf...	DepTime	CRSDe...	ArrTime	CRSArr...	Unique...	FlightNum	TailNum	
Row0	1987	10	14	3	741	730	912	849	PS	1451	NA	91
Row1	1987	10	15	4	729	730	903	849	PS	1451	NA	94
Row2	1987	10	17	6	741	730	918	849	PS	1451	NA	97
Row3	1987	10	18	7	729	730	847	849	PS	1451	NA	78
Row4	1987	10	19	1	749	730	922	849	PS	1451	NA	93
Row5	1987	10	21	3	728	730	848	849	PS	1451	NA	80
Row6	1987	10	22	4	728	730	852	849	PS	1451	NA	84
Row7	1987	10	23	5	731	730	902	849	PS	1451	NA	91
Row8	1987	10	24	6	744	730	908	849	PS	1451	NA	84
Row9	1987	10	25	7	729	730	851	849	PS	1451	NA	82
Row10	1987	10	26	1	735	730	904	849	PS	1451	NA	89
Row11	1987	10	28	3	741	725	919	855	PS	1451	NA	98
Row12	1987	10	29	4	742	725	906	855	PS	1451	NA	84
Row13	1987	10	31	6	726	725	848	855	PS	1451	NA	82
Row14	1987	10	1	4	936	915	1035	1001	PS	1451	NA	59
Row15	1987	10	2	5	918	915	1017	1001	PS	1451	NA	59
Row16	1987	10	3	6	928	915	1037	1001	PS	1451	NA	69
Row17	1987	10	4	7	914	915	1003	1001	PS	1451	NA	49
Row18	1987	10	5	1	1042	915	1129	1001	PS	1451	NA	47
Row19	1987	10	6	2	934	915	1024	1001	PS	1451	NA	50
Row20	1987	10	7	3	946	915	1037	1001	PS	1451	NA	51
Row21	1987	10	8	4	932	915	1033	1001	PS	1451	NA	61
Row22	1987	10	9	5	947	915	1036	1001	PS	1451	NA	49

Figure 3. DB Outport View with retrieved rows

Depending on the complexity of the SQL query, caching the first 100 rows might take a long time.

The table specification can be inspected in the *DB Spec* tab. It contains 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. In order to get the table specification, a query that only fetches the metadata but not the data itself is executed during configuration of the node or during execution. Execution of the metadata query during configure can be disabled via the Advanced Tab of the Connector node.

The *DB Query* tab contains the intermediate SQL query that defines the data at this outport. The query consists of the queries that were created in the preceding database nodes, and will only be executed when you want to retrieve the result in a KNIME data table, for example using the *DB Reader* node.

Session Handling

The DB Session lifecycle 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.

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 location of the database, and the authentication method if available.

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:

- H2
- Microsoft Access
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- SQLite

Some dedicated Connector nodes, such as Oracle 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 type, dialect, location, or authentication. Then click *Ok* and execute the node to establish a connection.

The 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

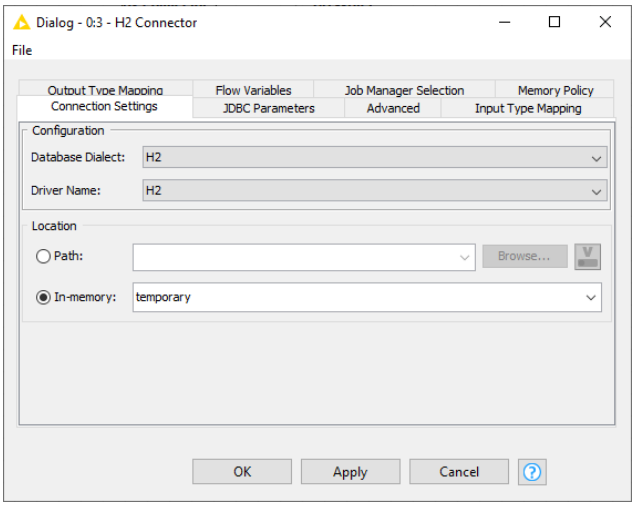


Figure 4. H2 Connector configuration dialog

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

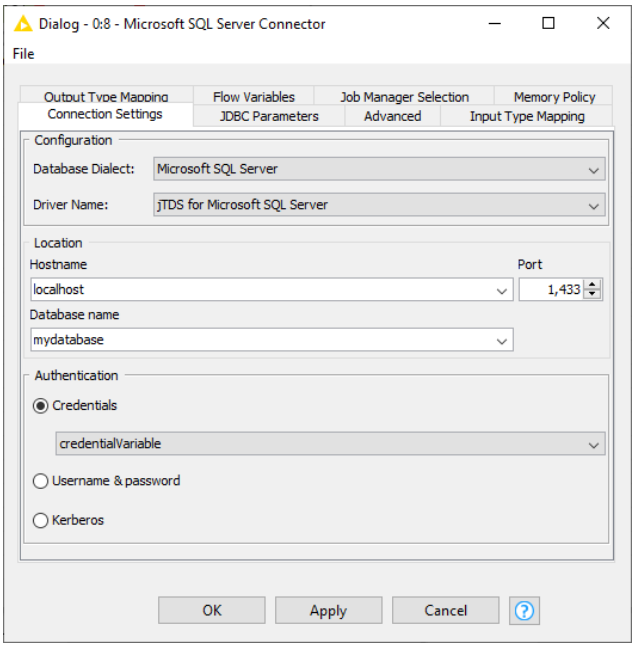


Figure 5. MS SQL Server Connector configuration dialog

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 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 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 (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 driver name list.

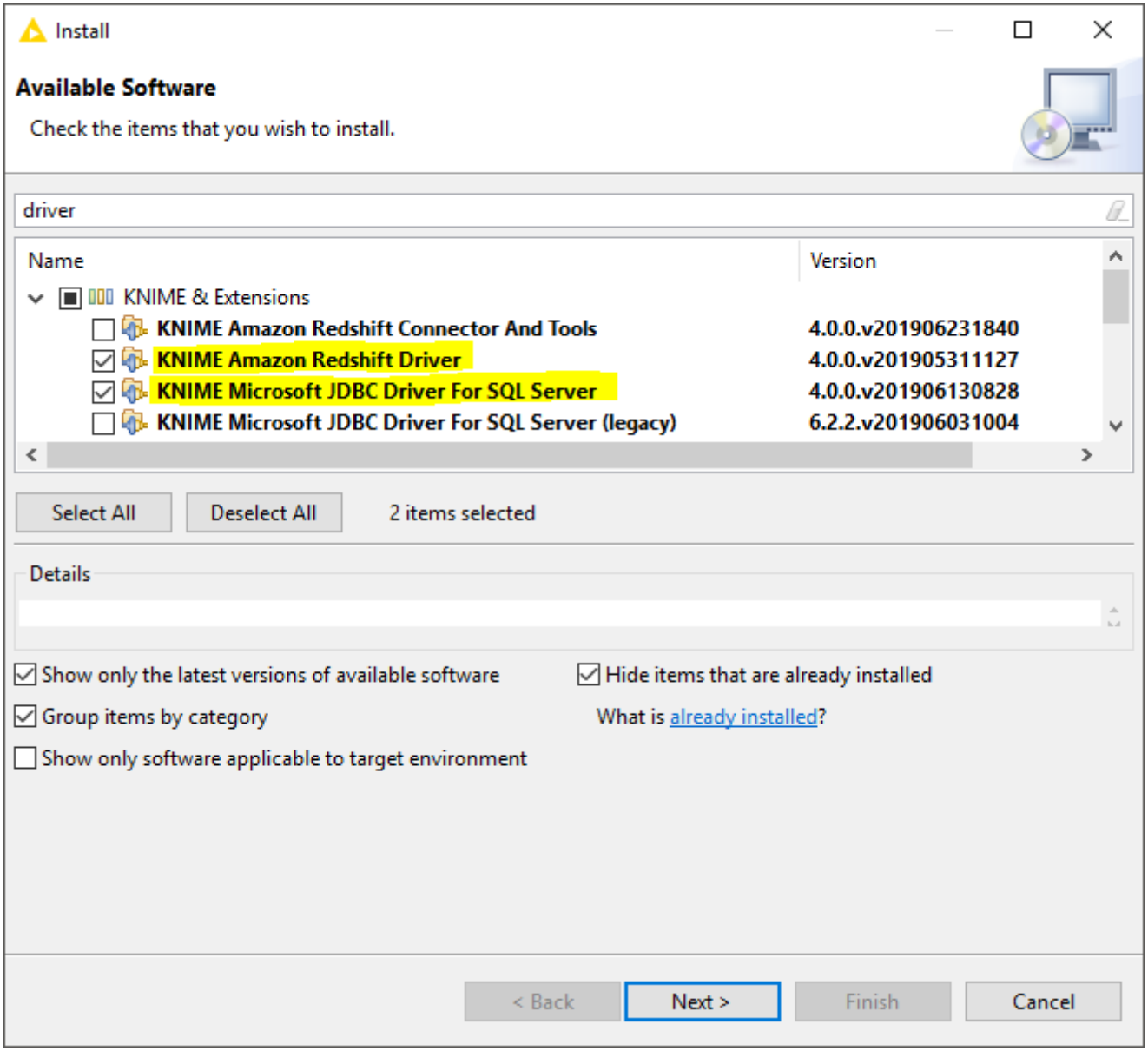


Figure 6. Install Window

Connecting to other databases

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

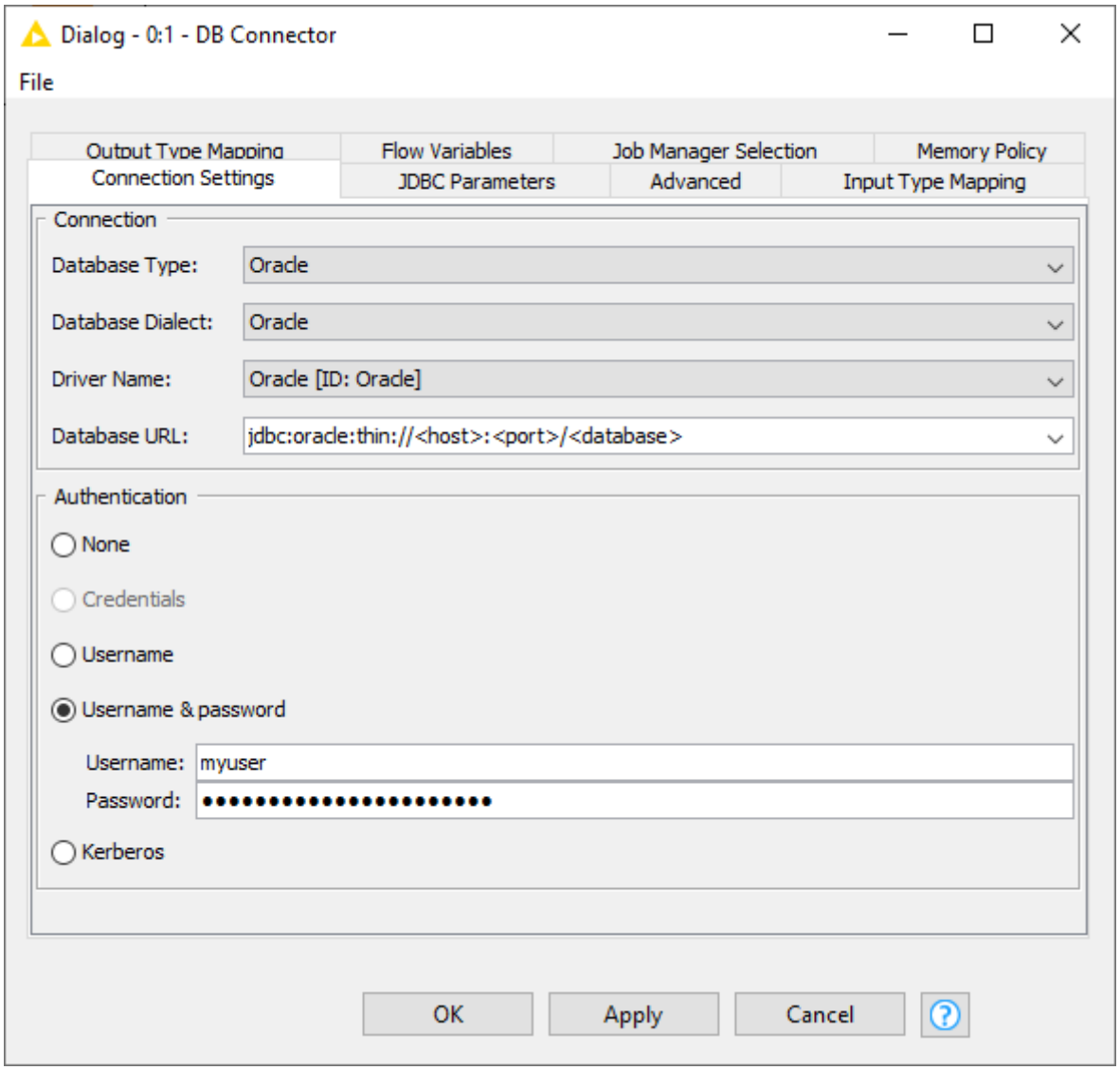


Figure 7. Database Connector configuration dialog

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.

Driver Name: Select an appropriate driver for your specific database. 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.

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 JDBC 4.2 compliant.

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

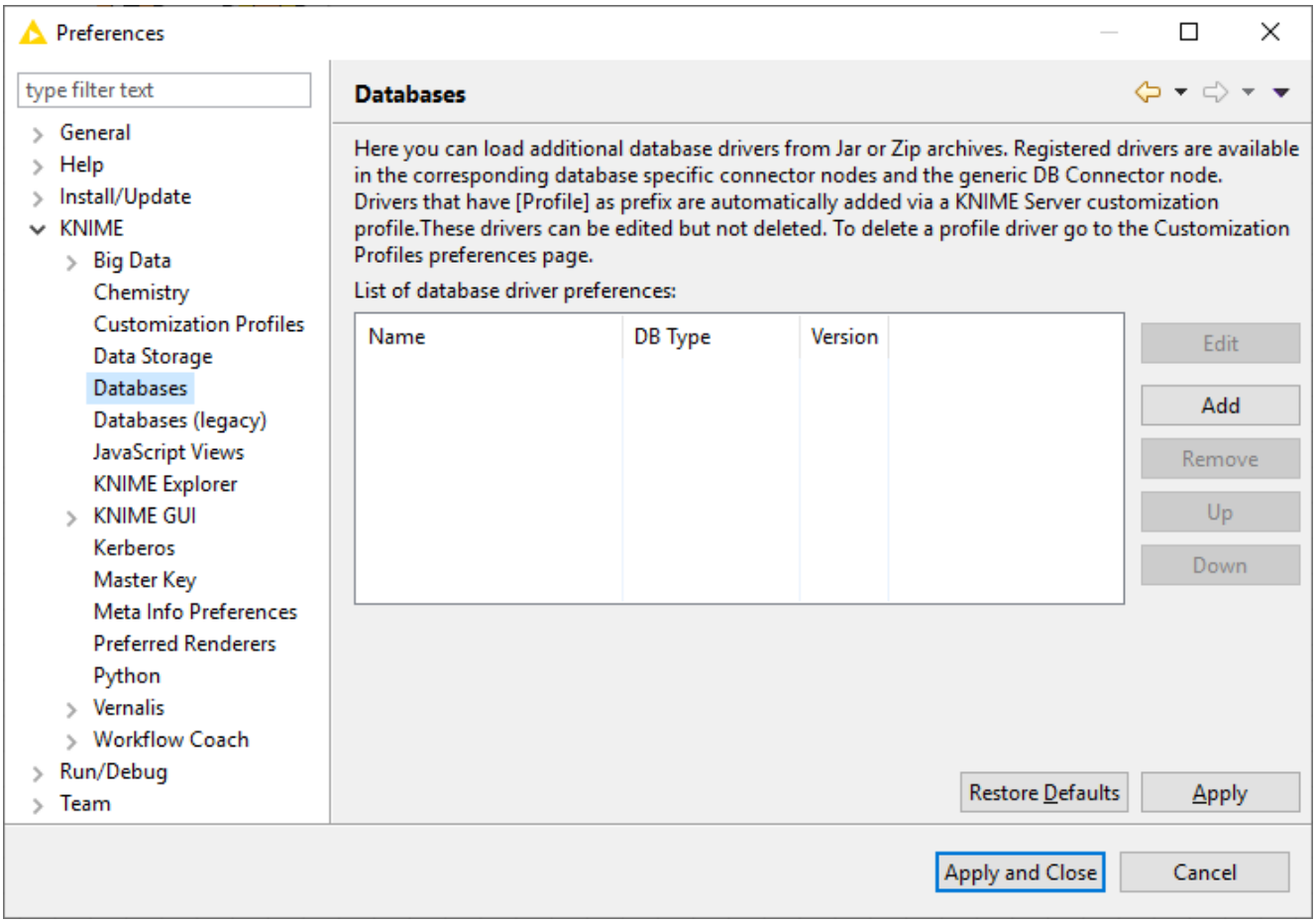


Figure 8. DB Preference page

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 JBC driver connection URL format which is used in the dedicated connector nodes. Please refer to the JDBC URL Template section for more information.
- *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.

- *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.

Let's do an example here and try to add the Oracle JDBC driver.

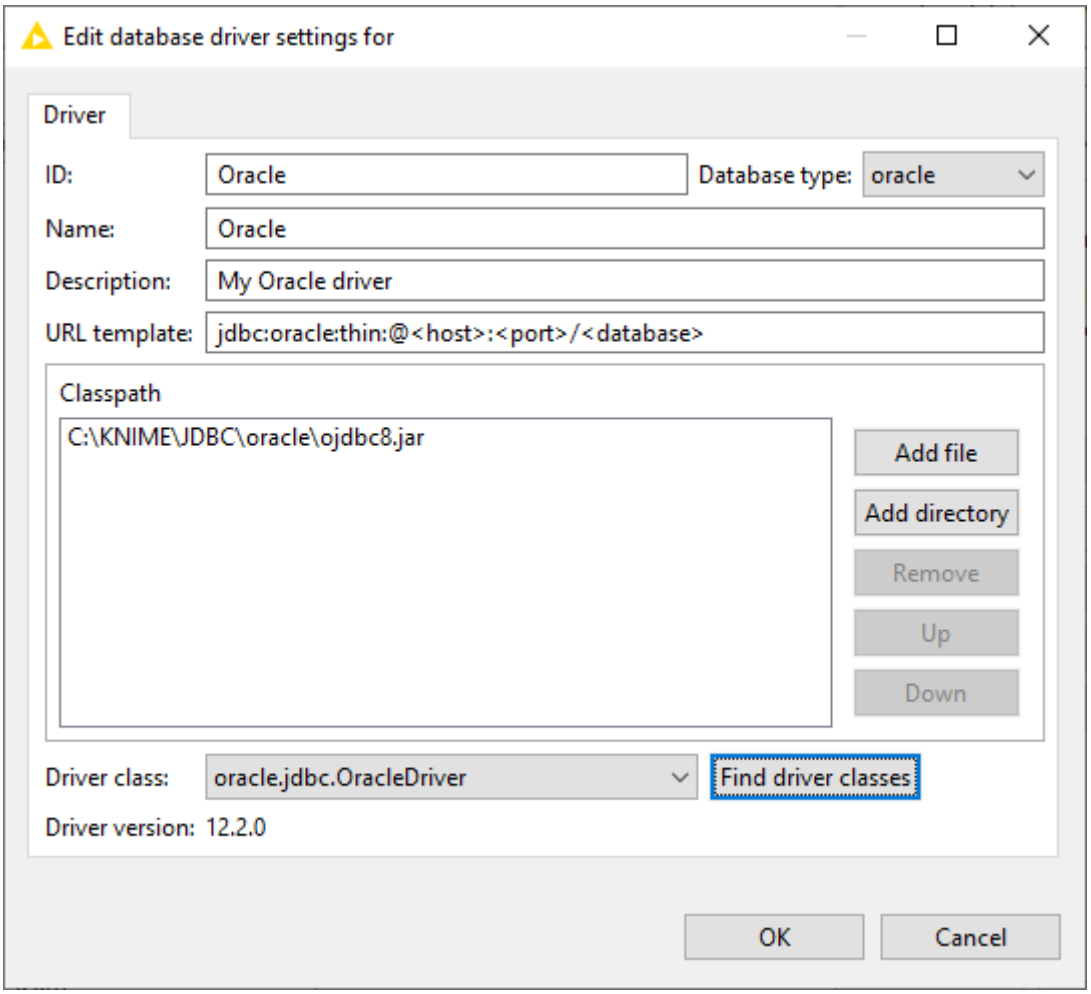


Figure 9. Edit database driver settings

- *ID*: Oracle, but you can enter your own driver ID as long as it only contains alphanumeric characters and underscores.
- *Name*: Oracle, but you can enter your own driver name.
- *Database type*: Oracle is available in the drop down list, so the database type is set to Oracle.
- *Description*: My Oracle driver.
- *URL template*: The JDBC URL template for an Oracle Thin driver is as the following `jdbc:oracle:thin:@<host>:<port>/<database>` according to the documentation. 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 ojdbc8.jar Oracle 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 *oracle.jdbc.OracleDriver* in version 12.2.0.

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.

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 and you can start connecting to your database.

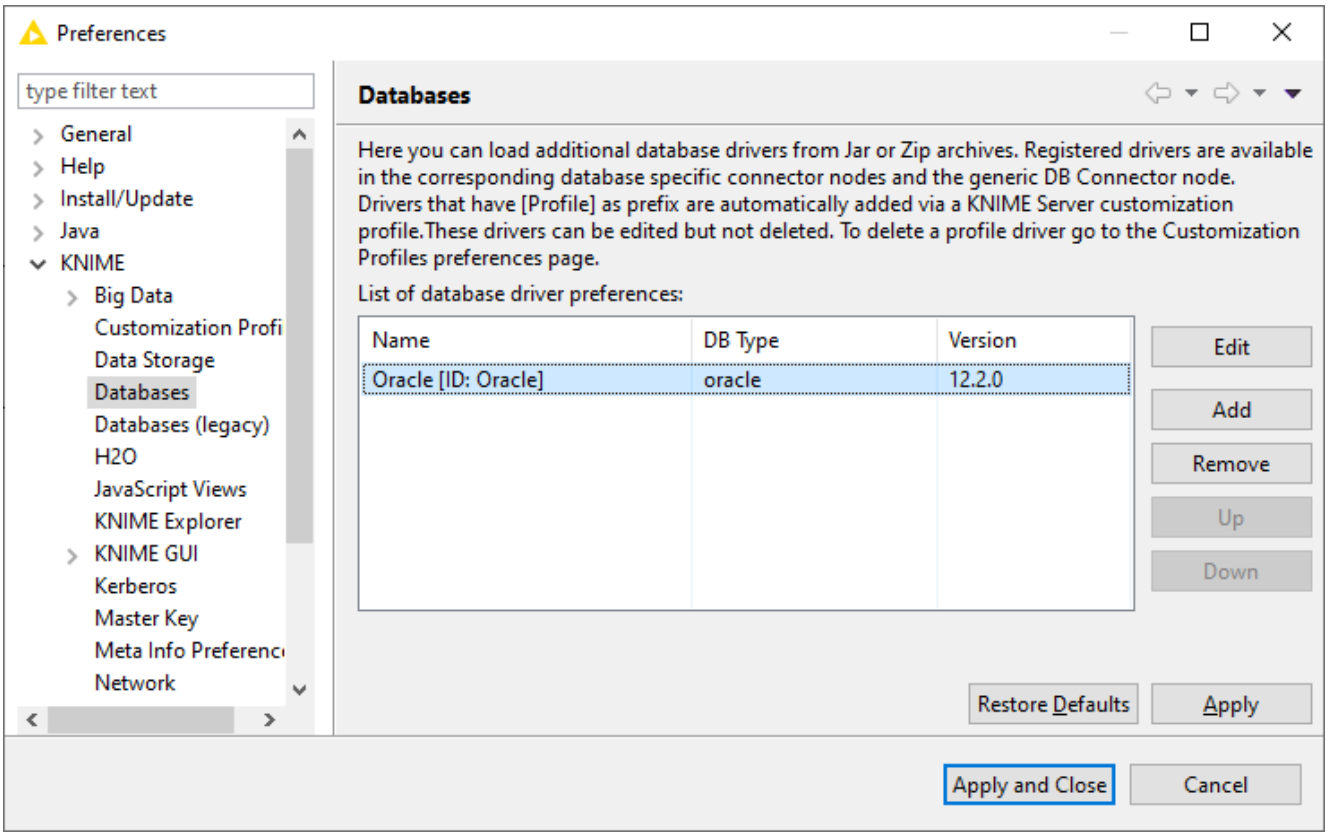


Figure 10. Database Preference page

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. The values of the variables, 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
- Oracle
- SAP HANA
- SnowFlake

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.

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 or KNIME URL. 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.

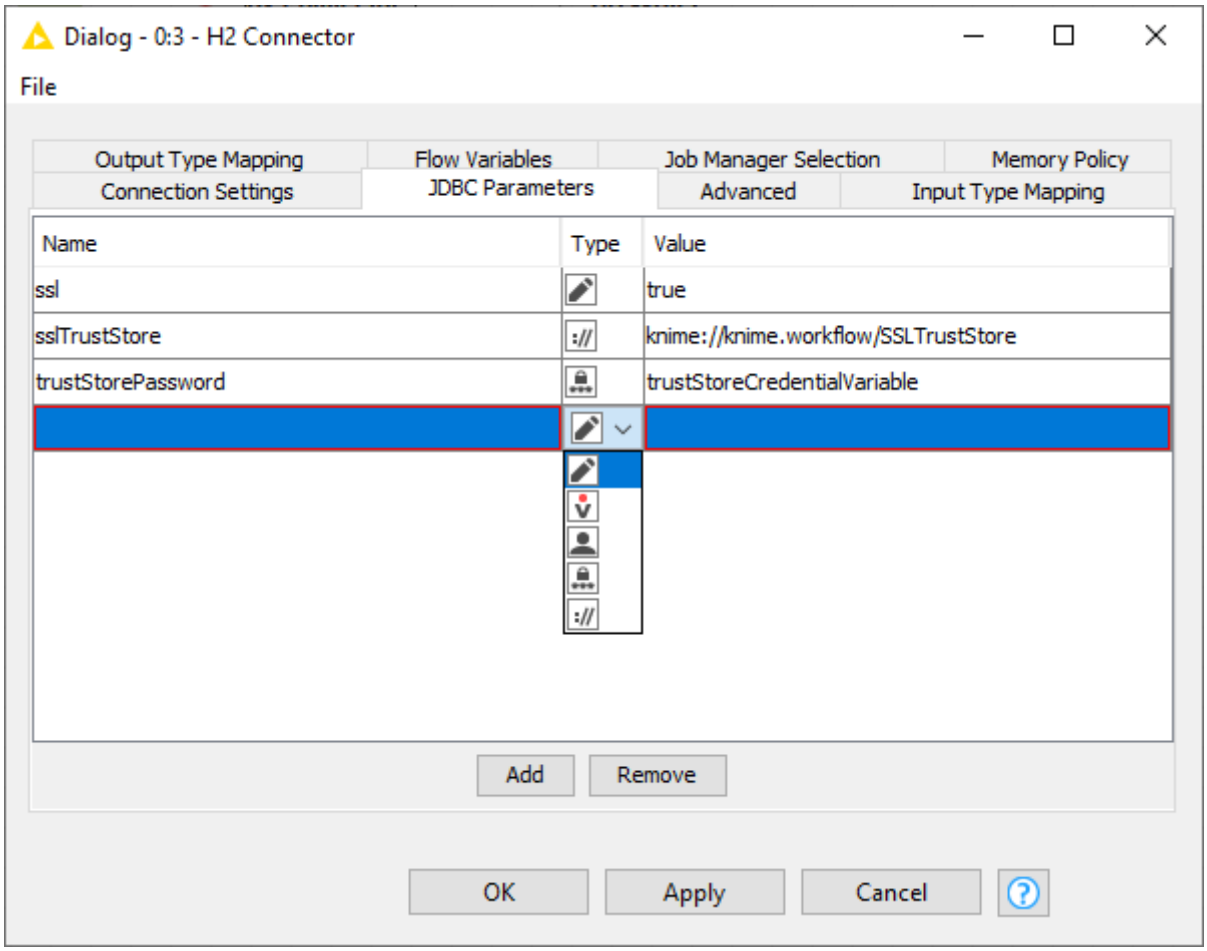


Figure 11. JDBC Parameters Tab

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.

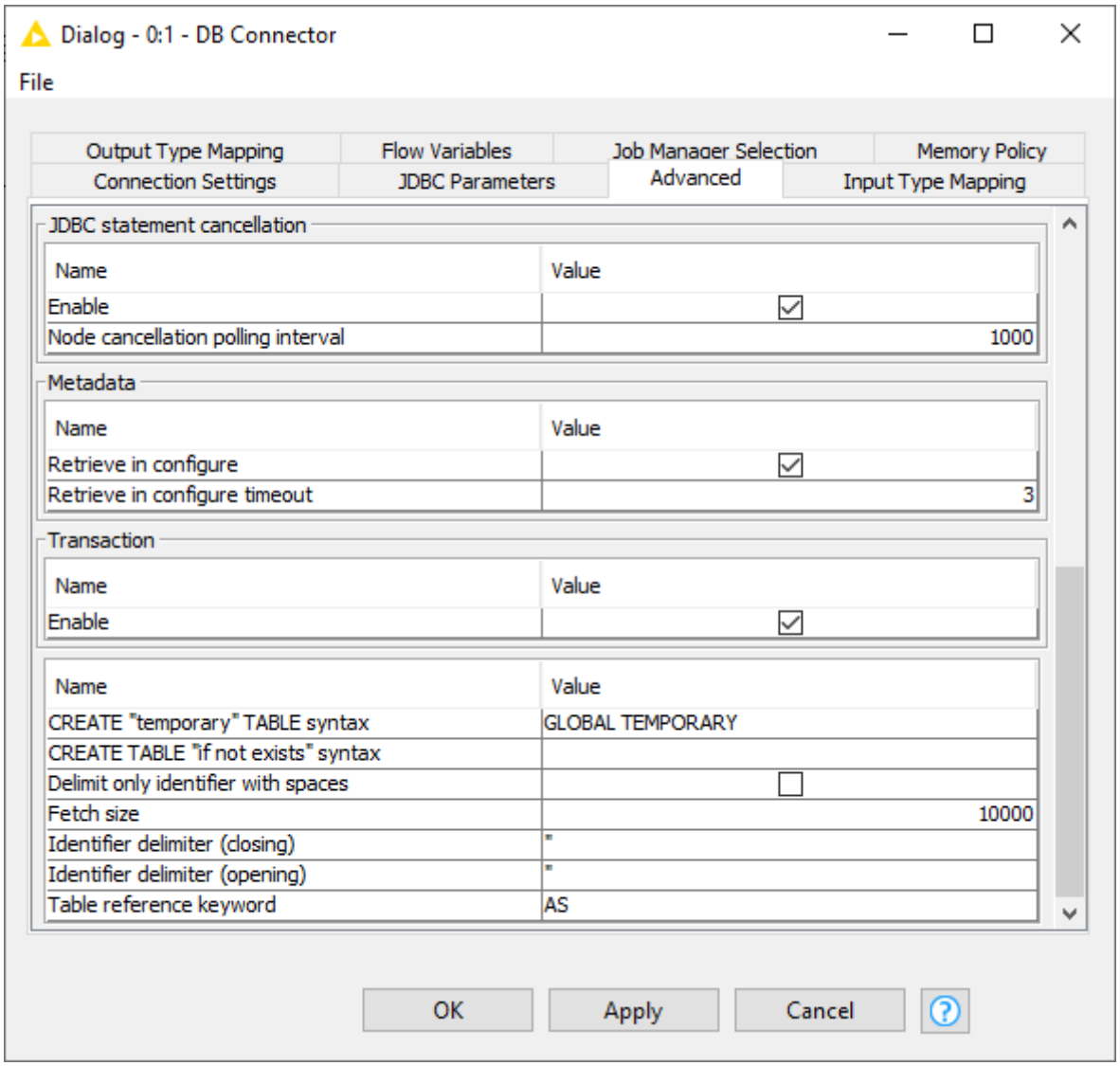


Figure 12. Advanced Tab

The full available options are described as follow:

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

- *JDBC statement cancellation*: 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].
- *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*: Enables or disables database transactions e.g. commit/rollback. If the database you want to connect to e.g. Google Big Query or Dremio does not support transaction please disable this option.
- *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.
- *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,).
- *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.
- *Table reference keyword*: The keyword before correlation names in table references.
- *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 URL last parameter suffix*: The character to be appended after the last parameter in the JDBC URL.

- *Minus operation*: Whether MINUS set operations are supported or not.

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

Reading from a database

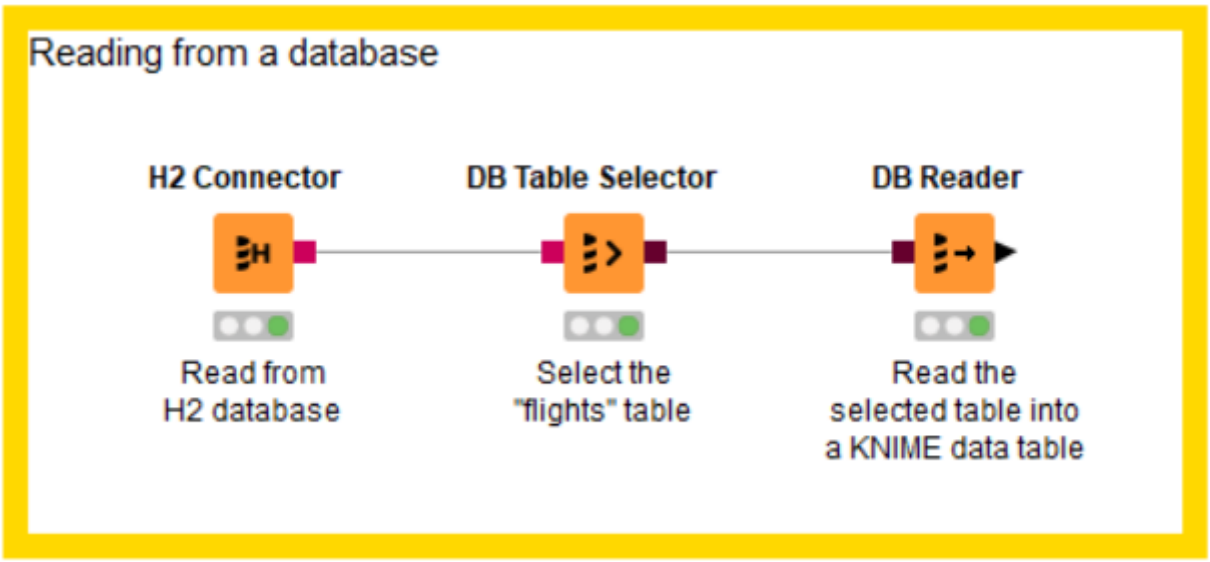


Figure 13. Reading from a database

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 .

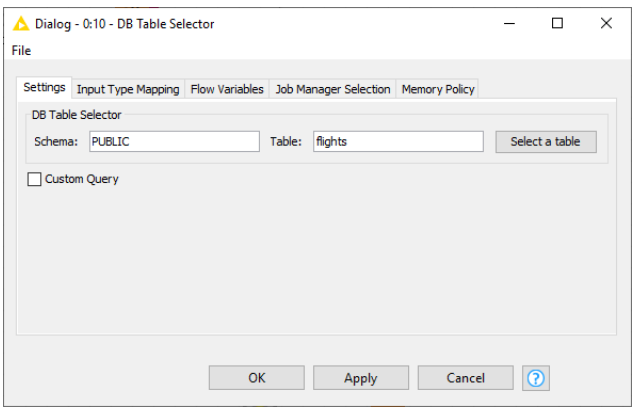


Figure 14. DB Table Selector configuration dialog

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 the KNIME Analytics Platform is running.

Database Metadata Browser

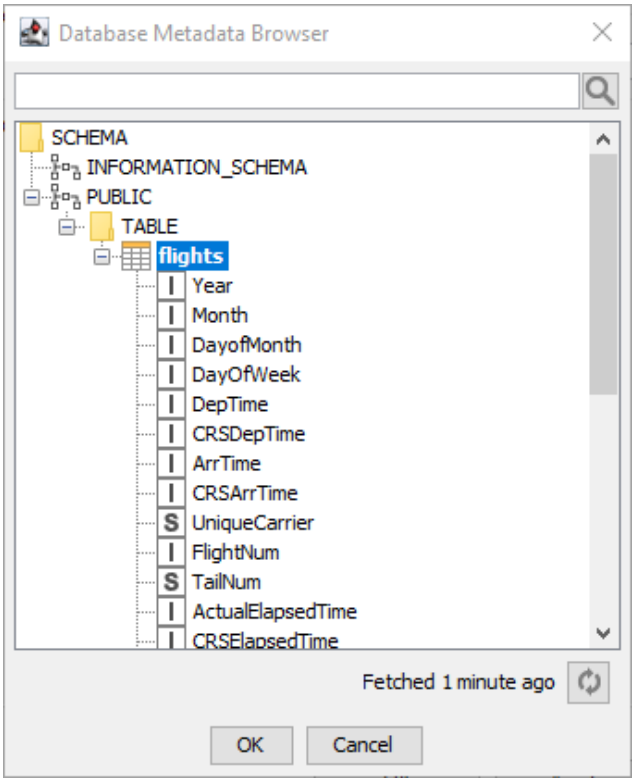


Figure 15. Database Metadata Browser

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

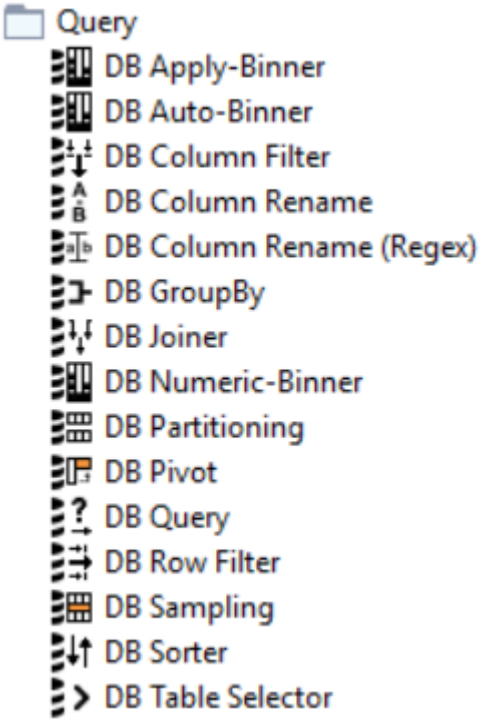


Figure 16. DB Query nodes

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 viewed via the DB Data outport view .

Visual Query Generation

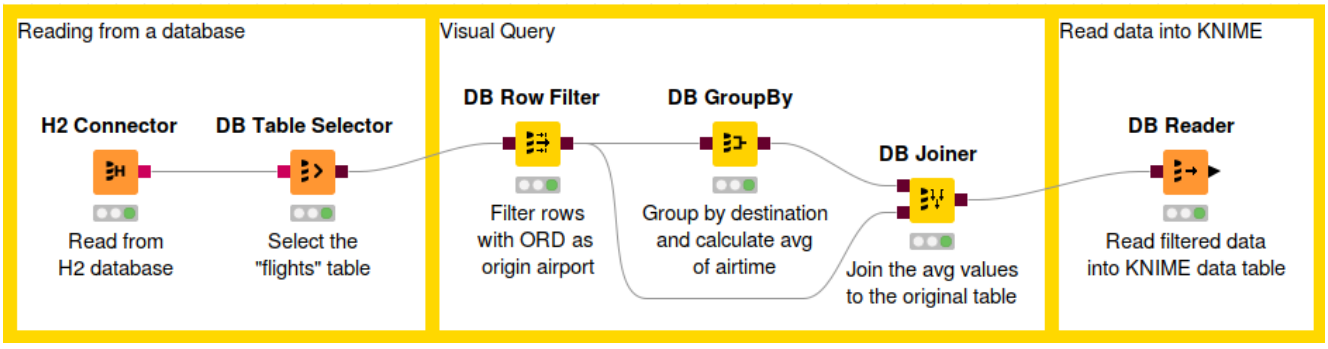


Figure 17. Example of a workflow that performs in-database data manipulation

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

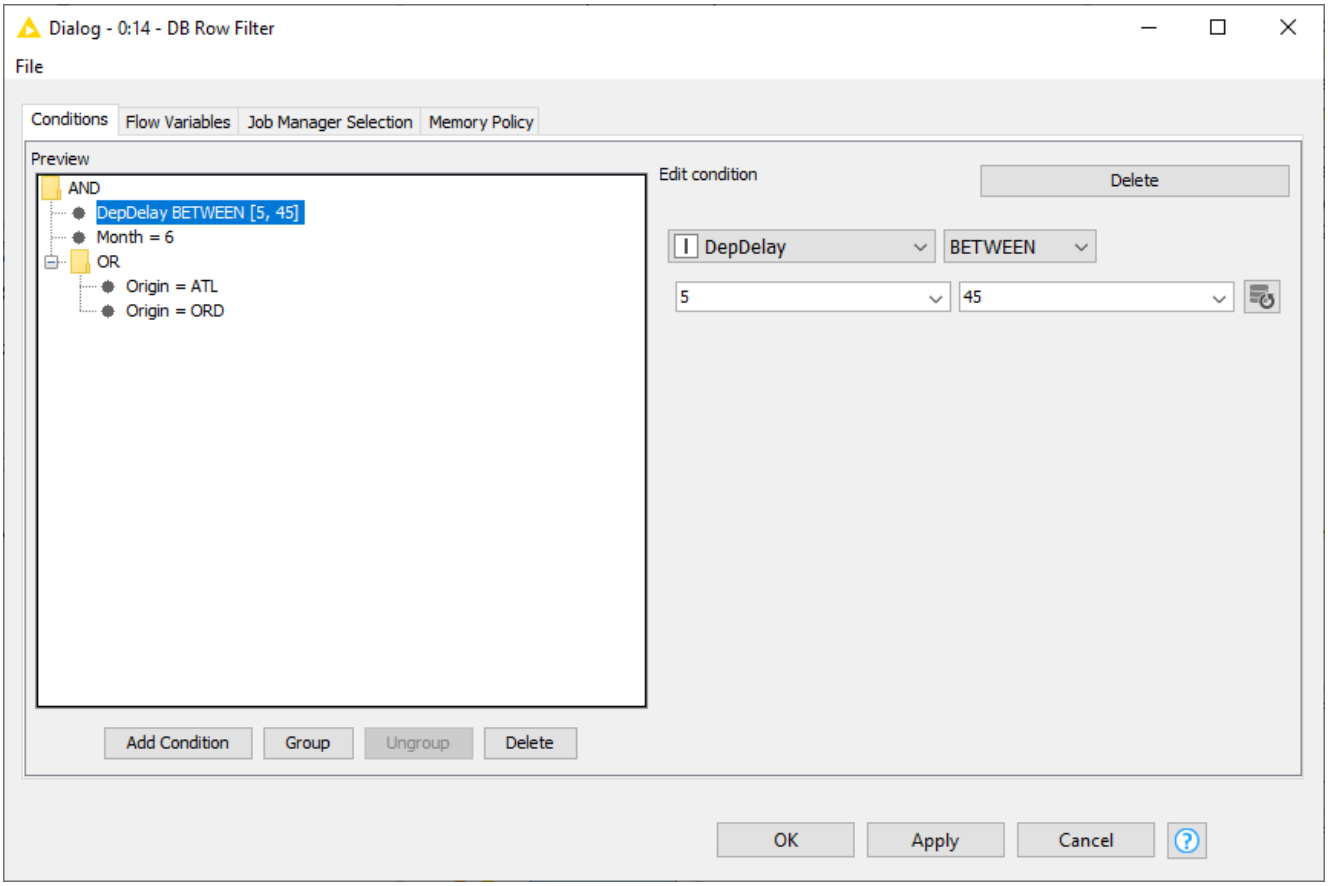


Figure 18. DB Row Filter configuration dialog

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

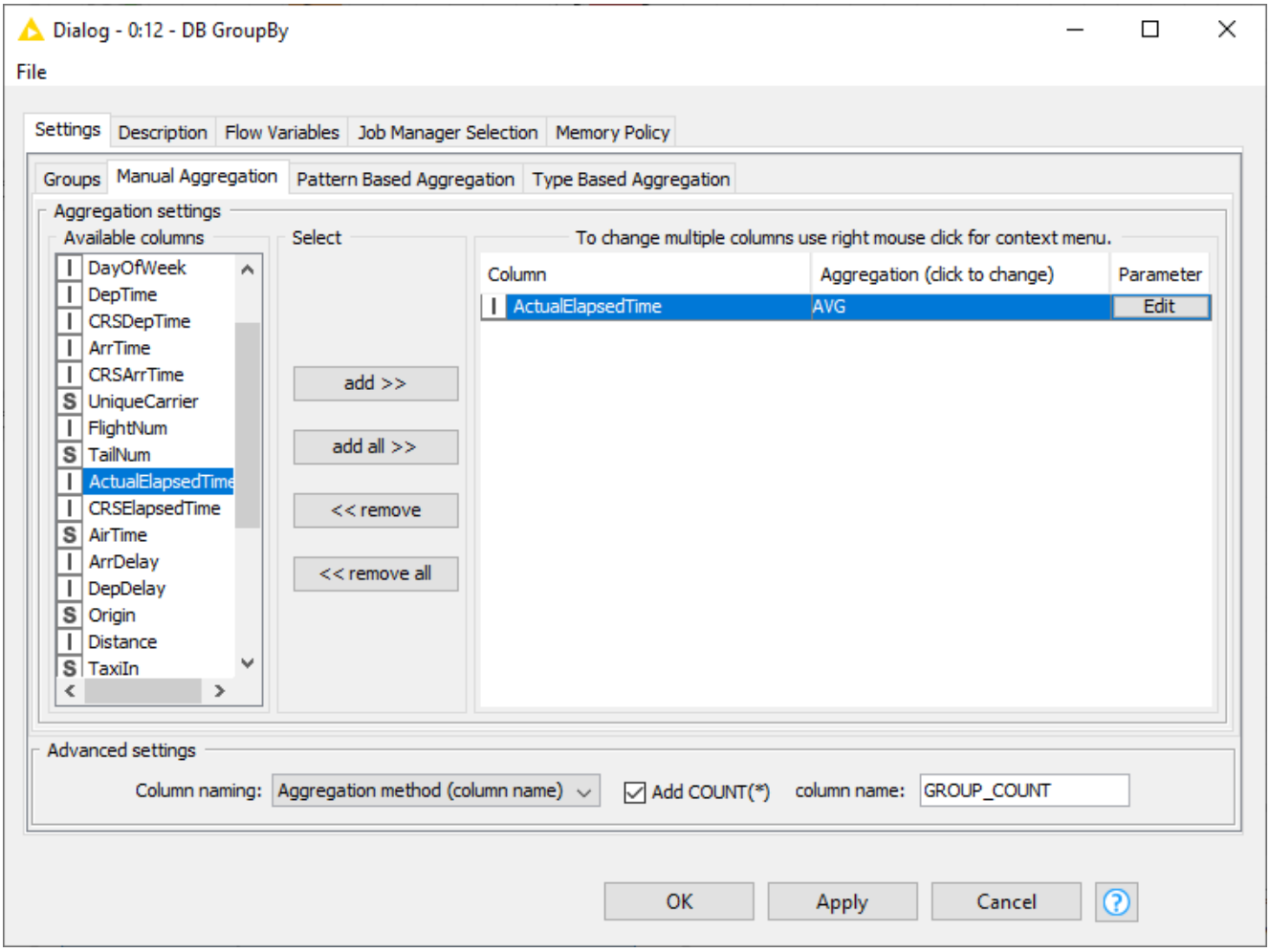


Figure 19. DB GroupBy: Manual Aggregation

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.

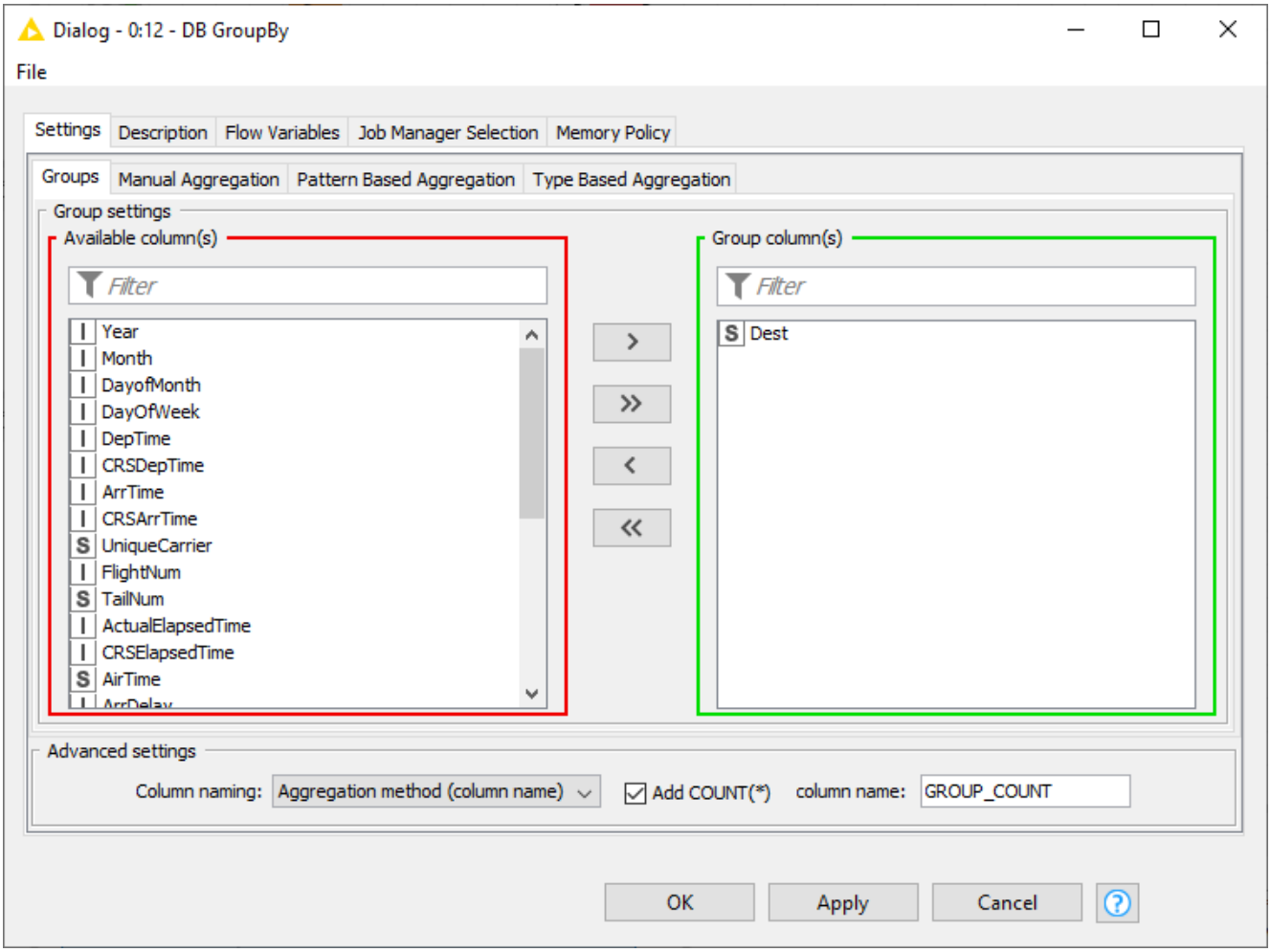


Figure 20. DB GroupBy: Group Settings

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

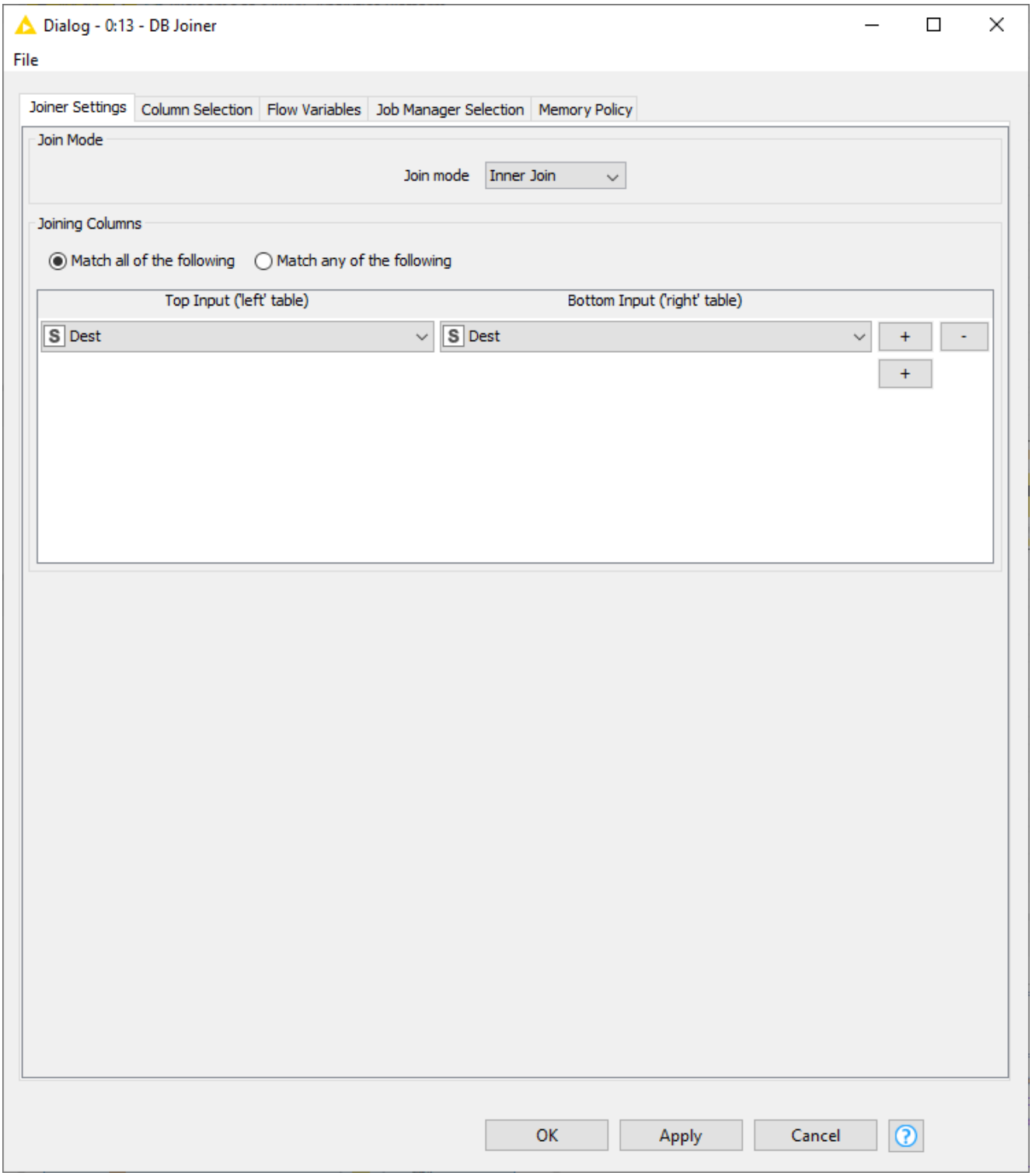


Figure 21. DB Joiner: Joiner Settings

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).

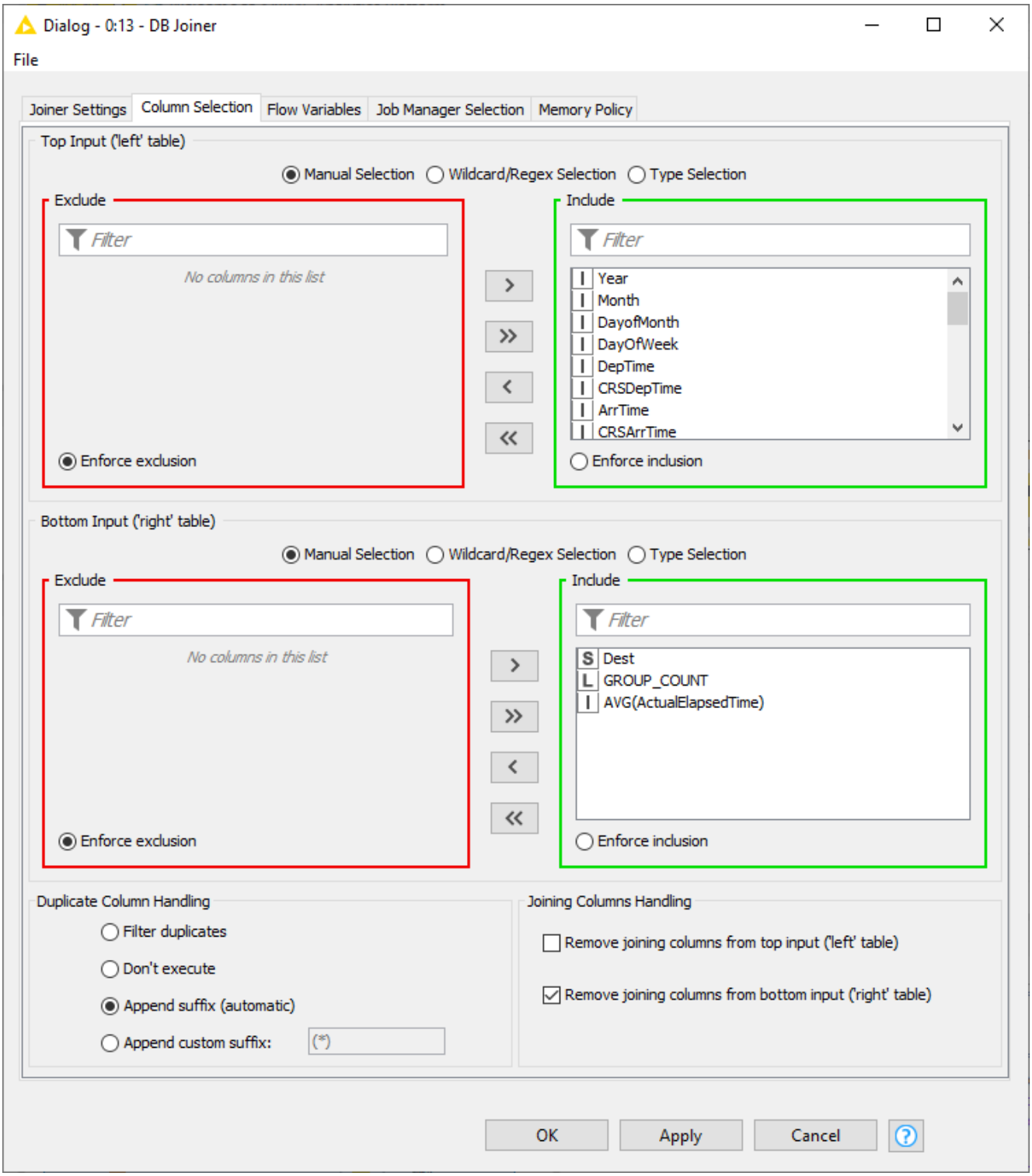


Figure 22. DB Joiner: Column Selection

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.

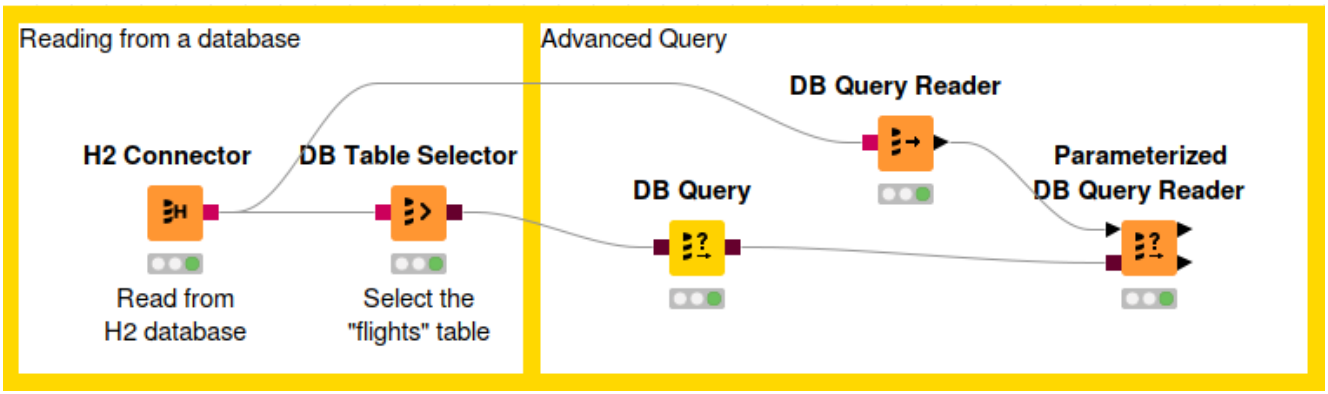


Figure 23. Example workflow with advanced query nodes

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.

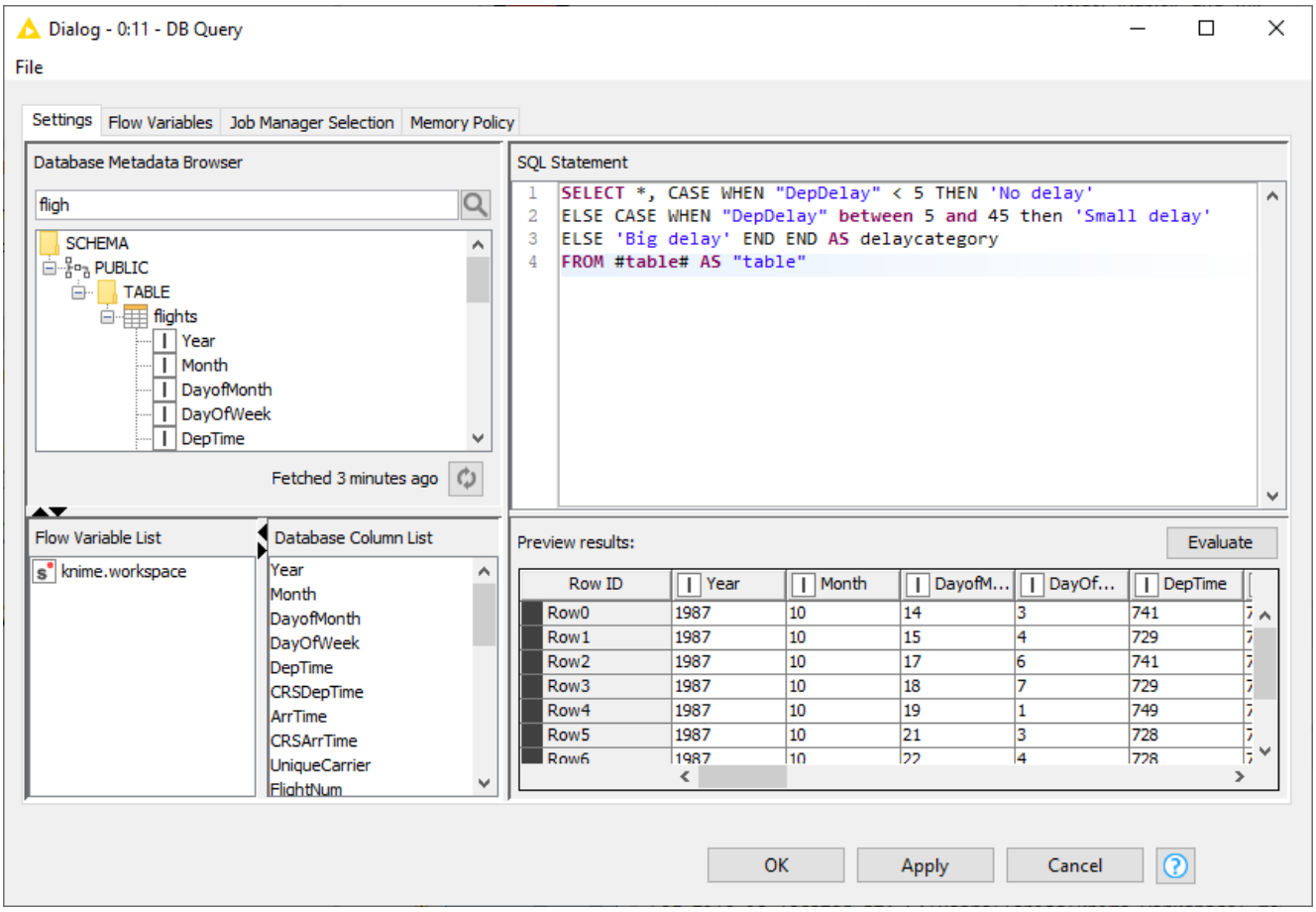


Figure 24. Configuration dialog of the DB Query node

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.

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.

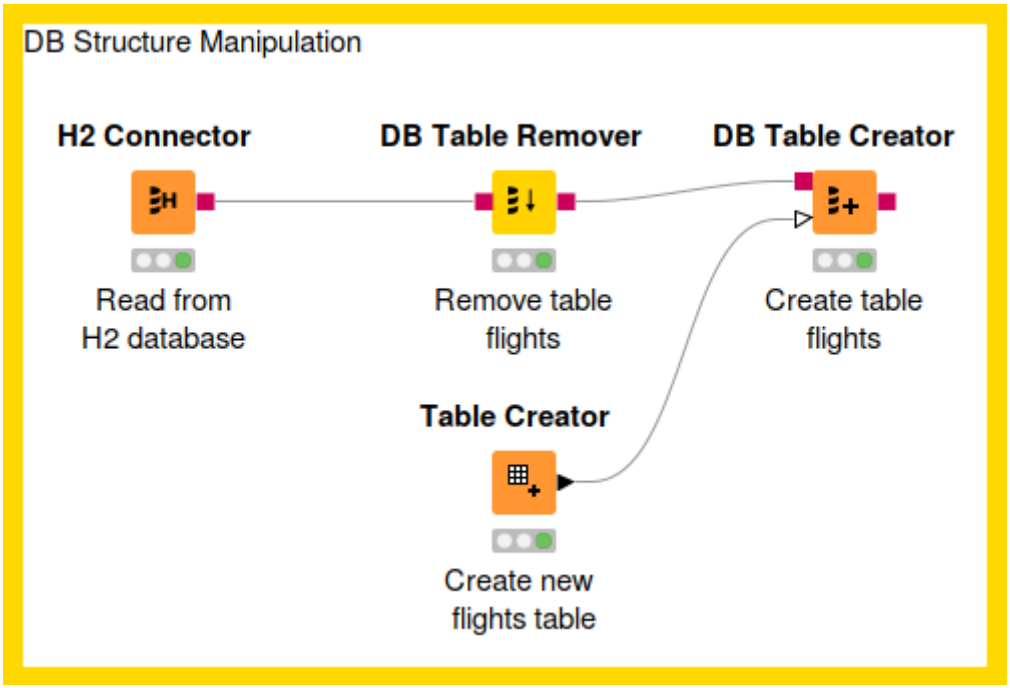


Figure 25. Example of a database structure manipulation workflow

DB Table Remover

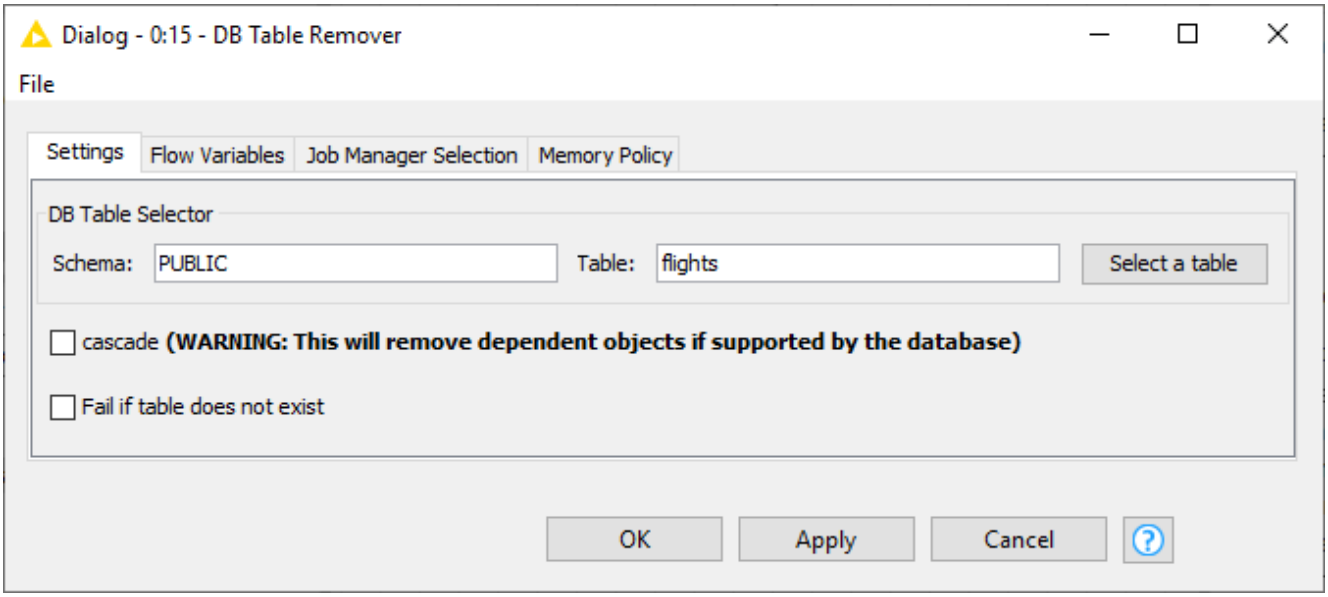


Figure 26. DB Table Remover configuration dialog

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

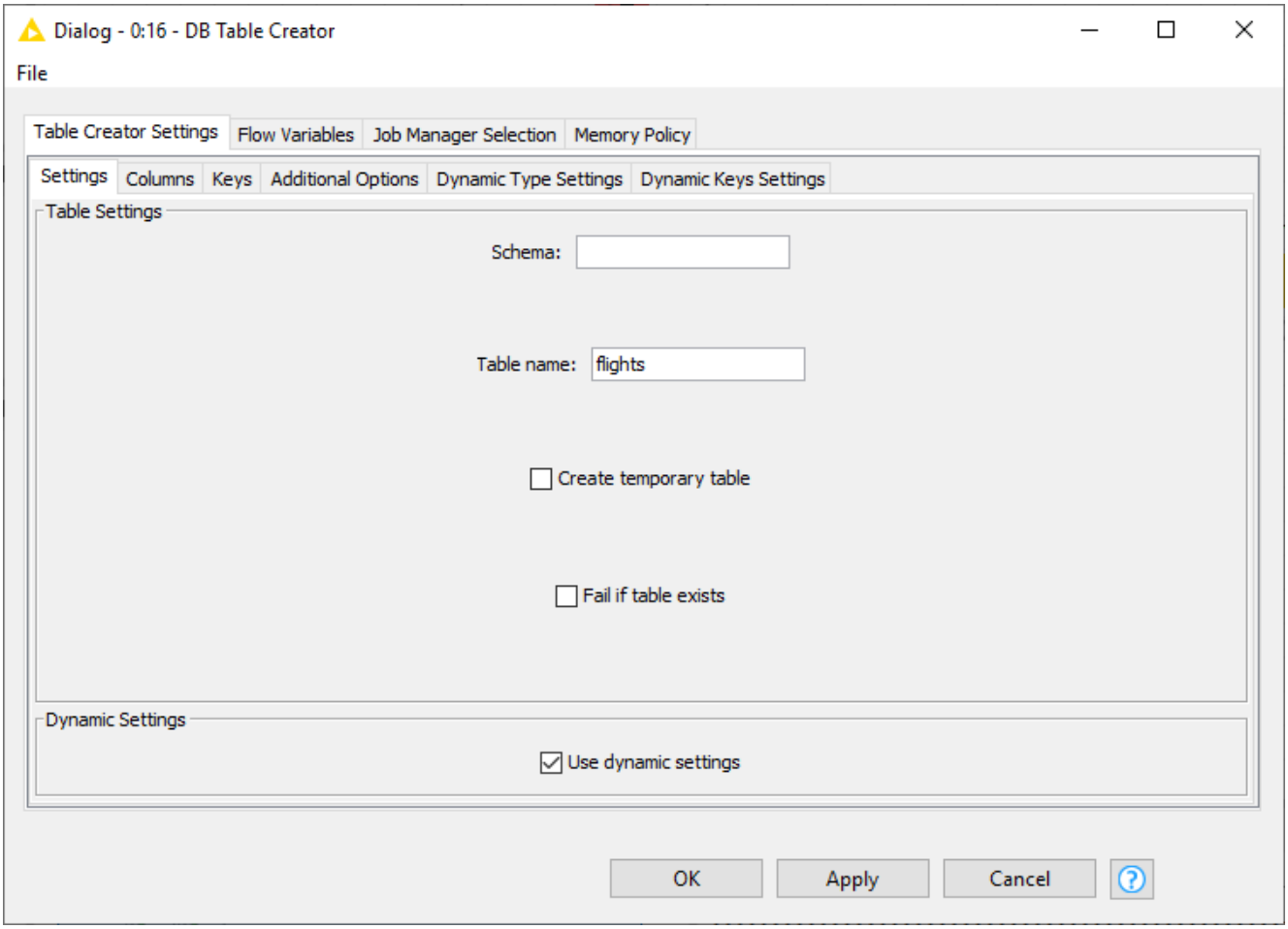


Figure 27. DB Table Creator: Settings

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.

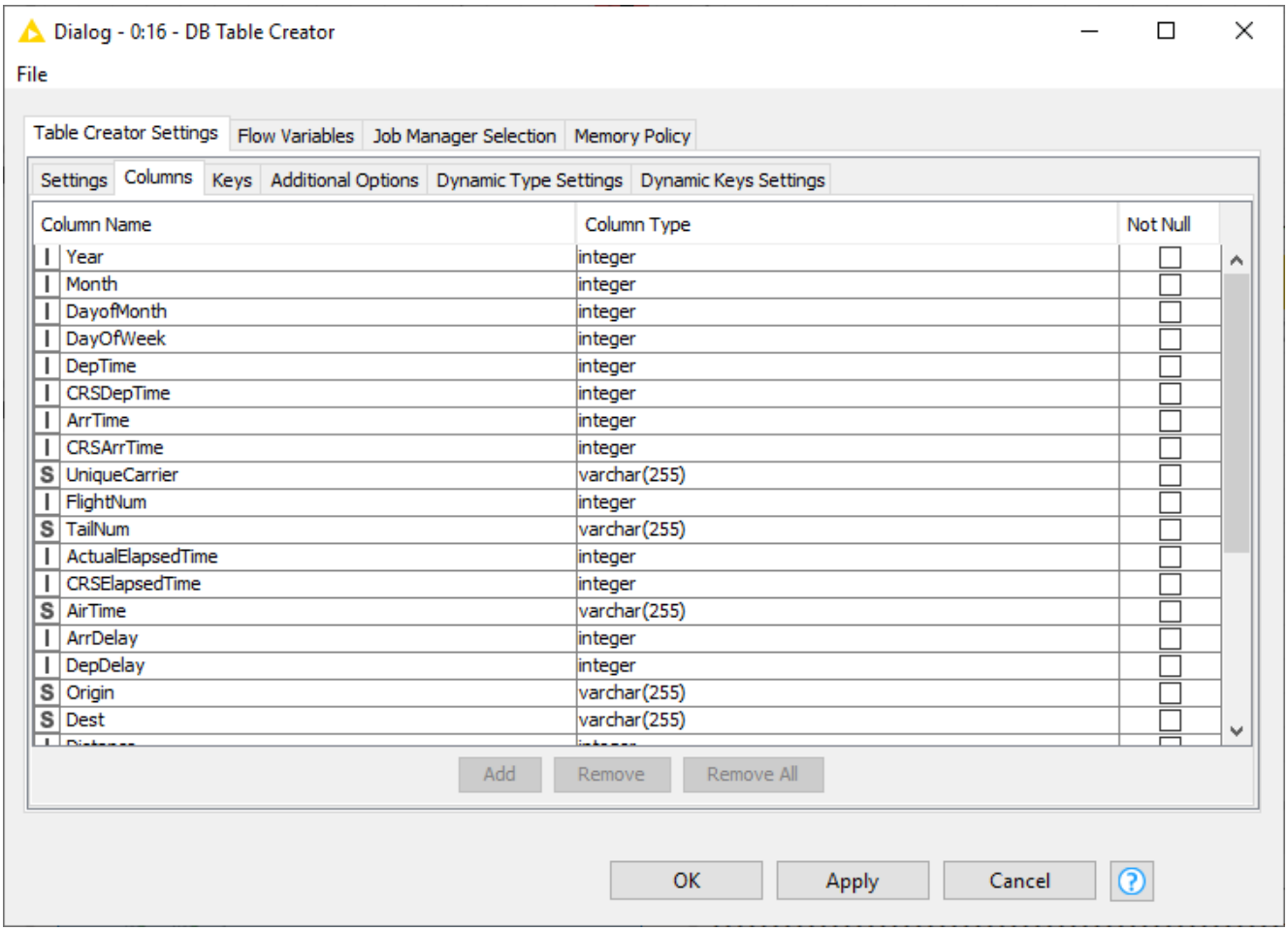


Figure 28. DB Table Creator: Columns

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*, *DB Writer*, *DB Insert*, *DB Update*, *DB Merge*, and *DB Loader* node.

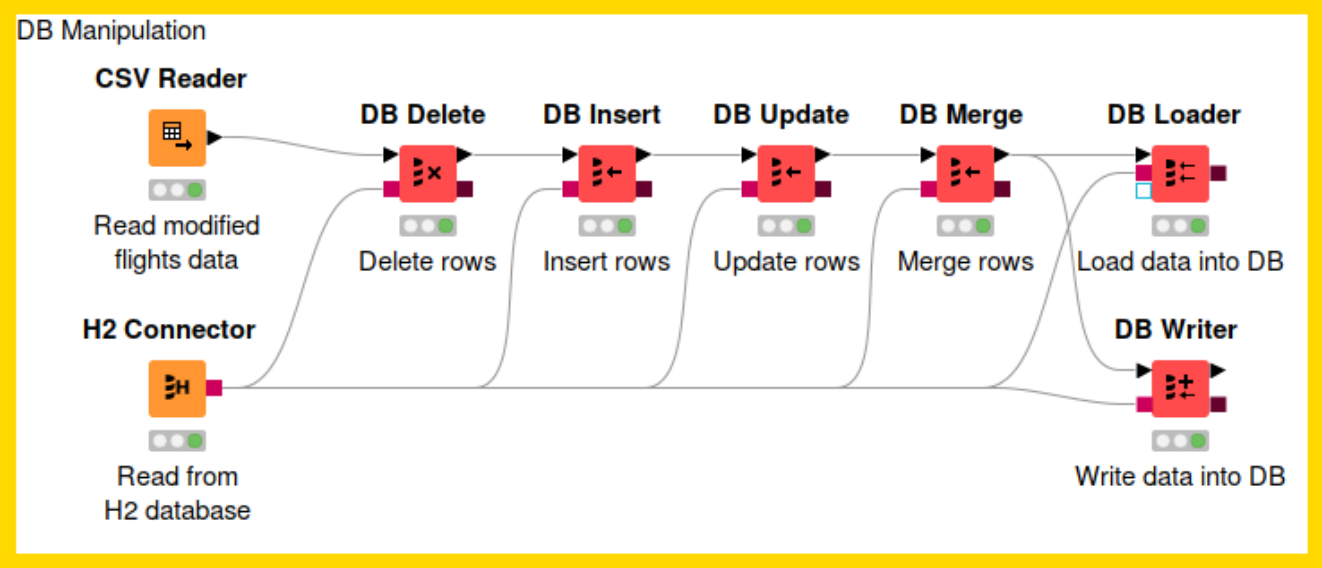


Figure 29. Example of DB Manipulation

DB Delete

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* 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 two options:

- *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.
- *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.

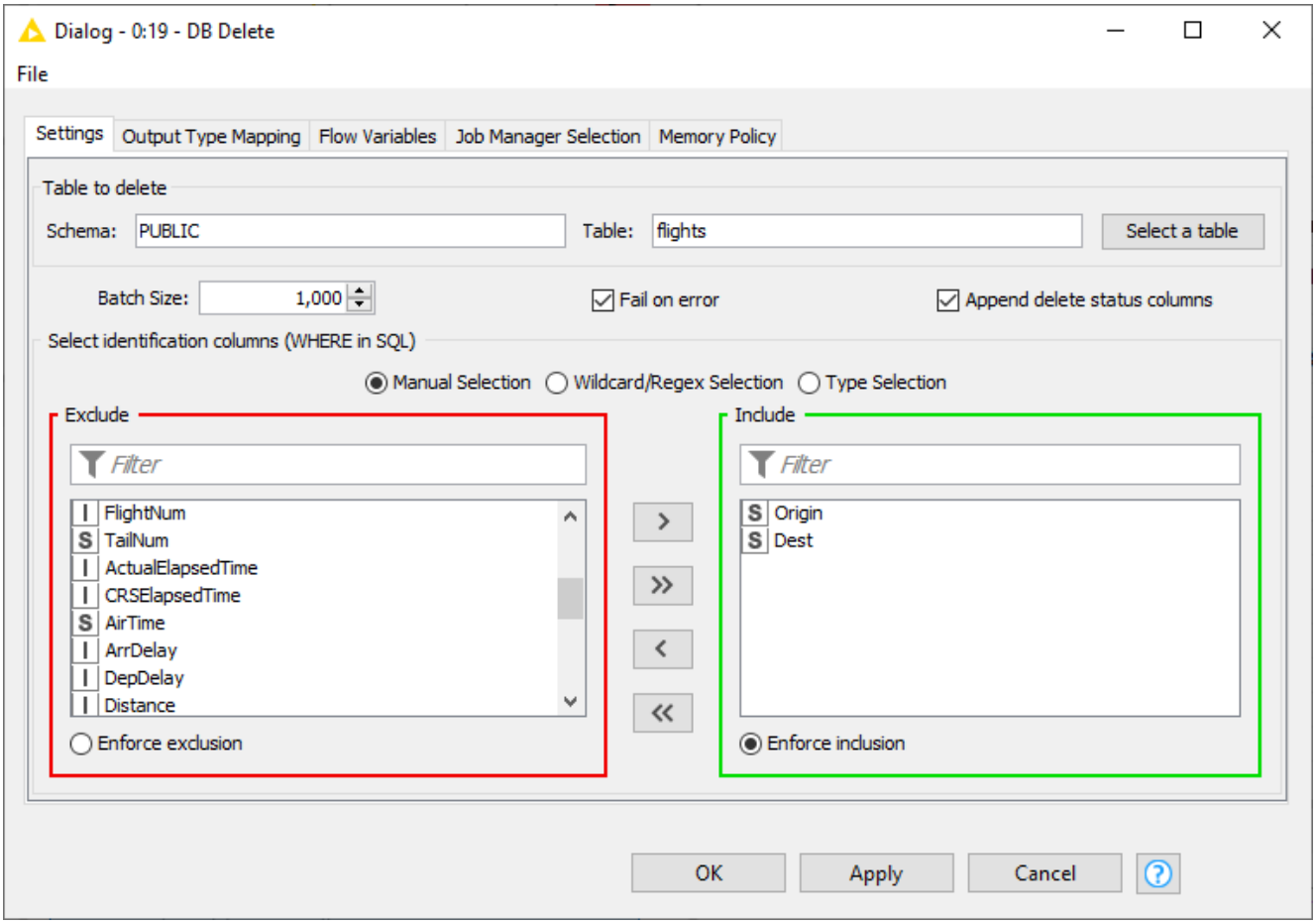


Figure 30. Configuration dialog of the DB Delete node

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 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.

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

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 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.

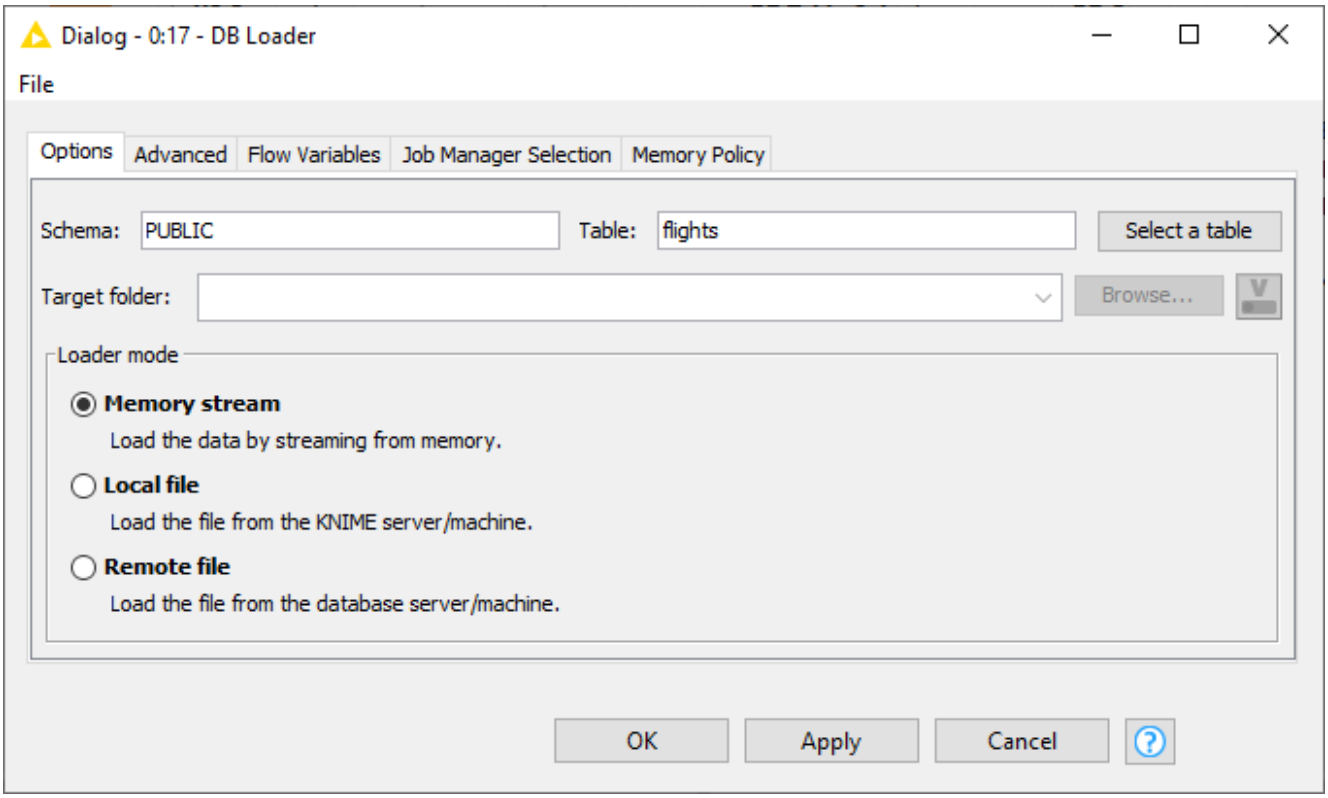


Figure 31. DB Loader: Option

Depending on the connected database the advanced 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.

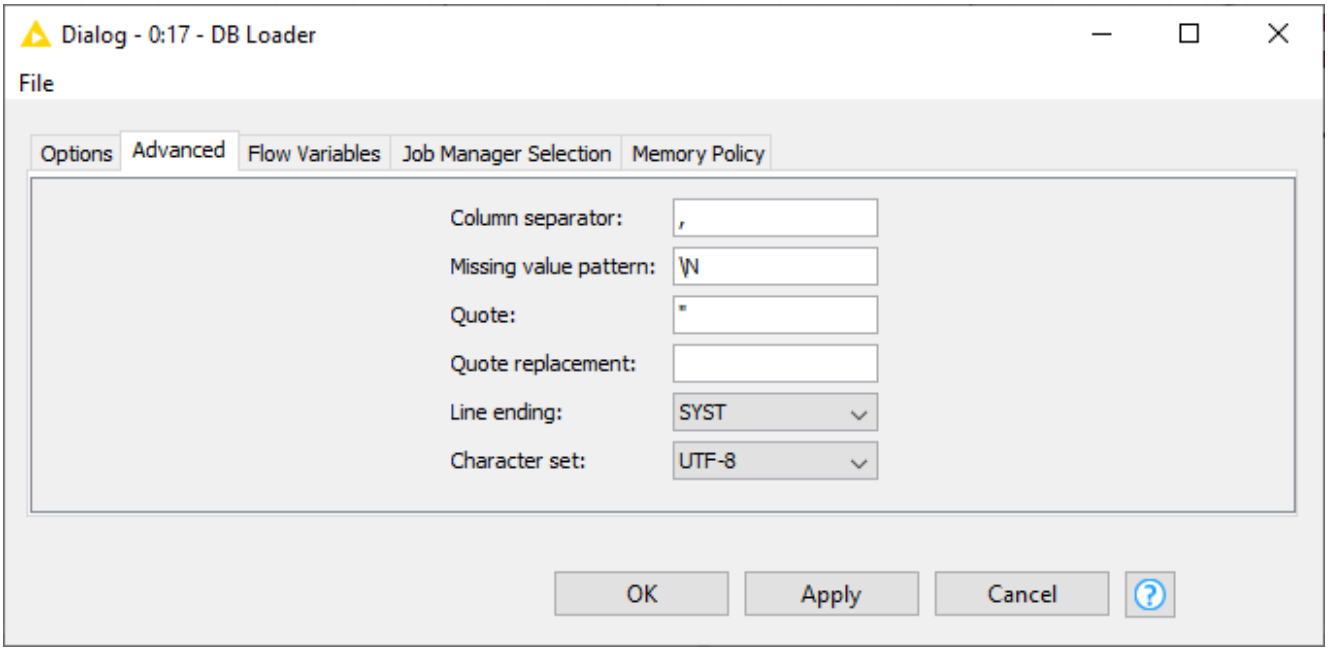


Figure 32. DB Loader: Advanced

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.

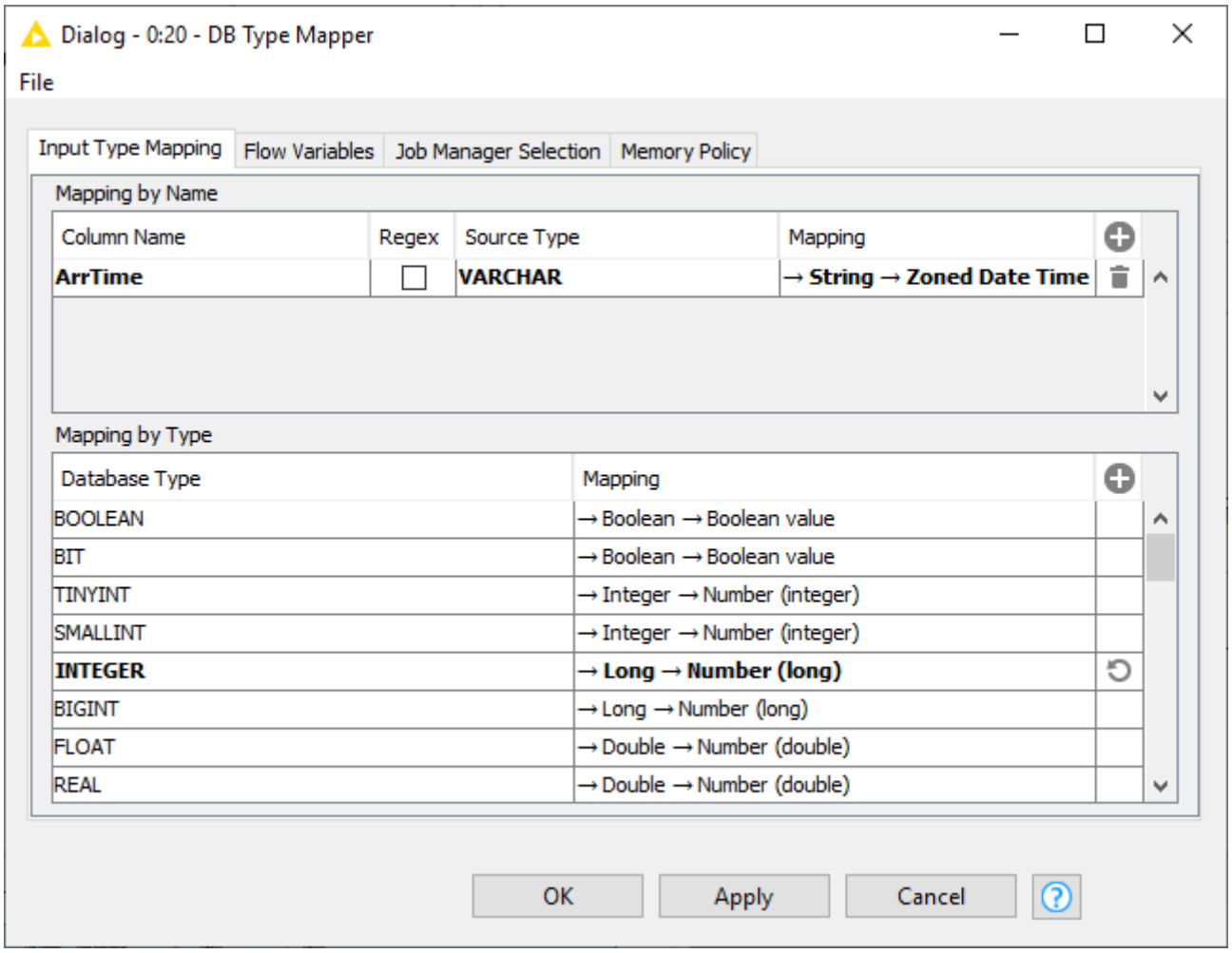


Figure 33. DB Type Mapper configuration dialog

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

Migration

This section explains how to migrate your workflow that contains old database nodes (legacy) to the new database framework. The Workflow Migration Tool can be used to guide you through the process and convert the database legacy nodes to the corresponding new database nodes. For the mapping between the legacy 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 legacy 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 database legacy nodes. The goal is to use the Workflow Migration Tool to help us migrating the legacy nodes to the new database nodes.

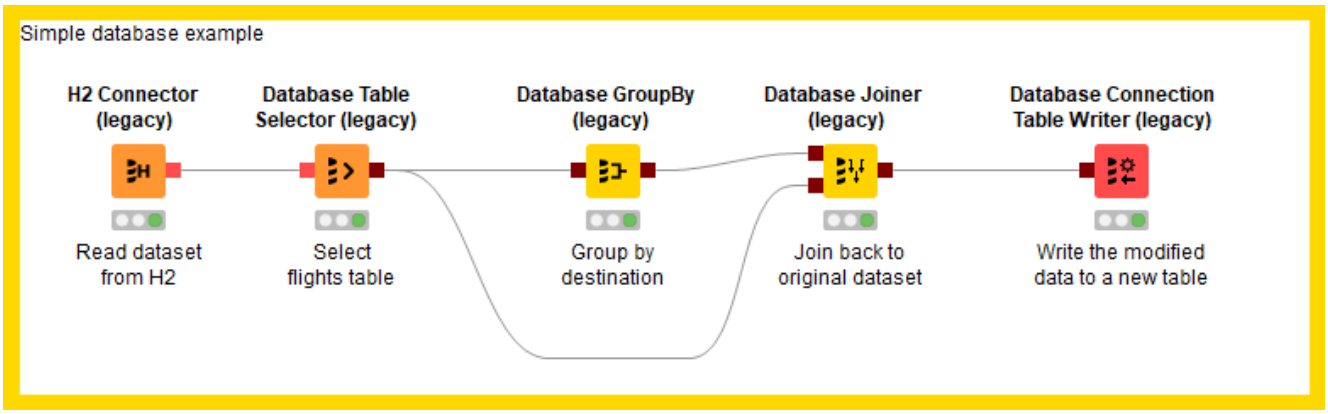


Figure 34. Workflow containing Database Legacy nodes

In order to start the Workflow Migration tool go to the KNIME Explorer, find the workflow containing the database legacy nodes that you want to migrate, and right click on the workflow. You can see the option *Migrate legacy DB workflow…* at the bottom (see figure below).

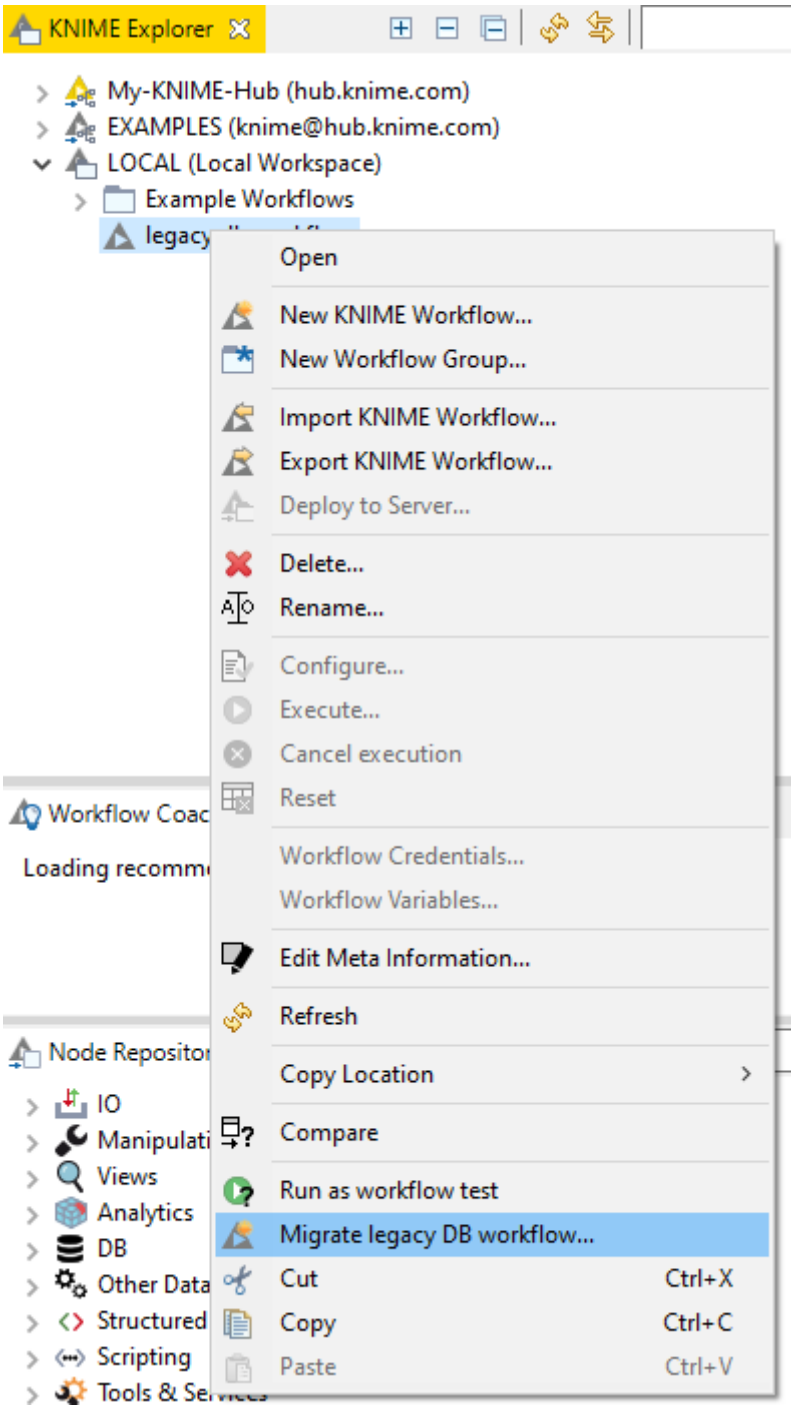


Figure 35. Migration Tool option by right-clicking the workflow

Clicking on *Migrate legacy DB workflow…* will open the migration wizard window as shown below. In this window, you can change the workflow to migrate (the one containing the database legacy nodes), and enter the name for the new workflow, which is a copy of the old workflow but with the database legacy 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.

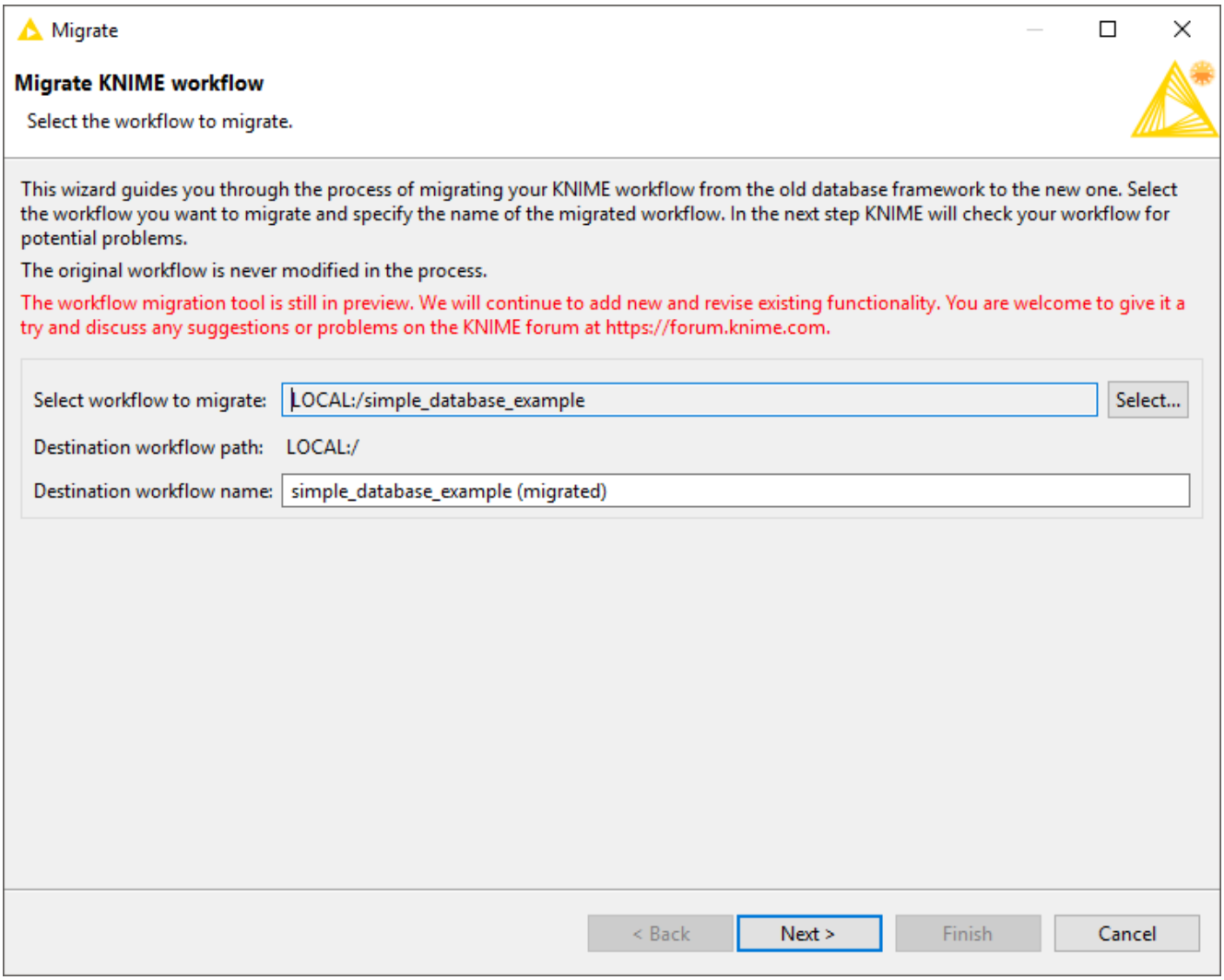


Figure 36. Migration Tool: Select the workflow

Click *next* to get to the next page, as shown below. At this stage the workflow will be analysed, and all database legacy 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.

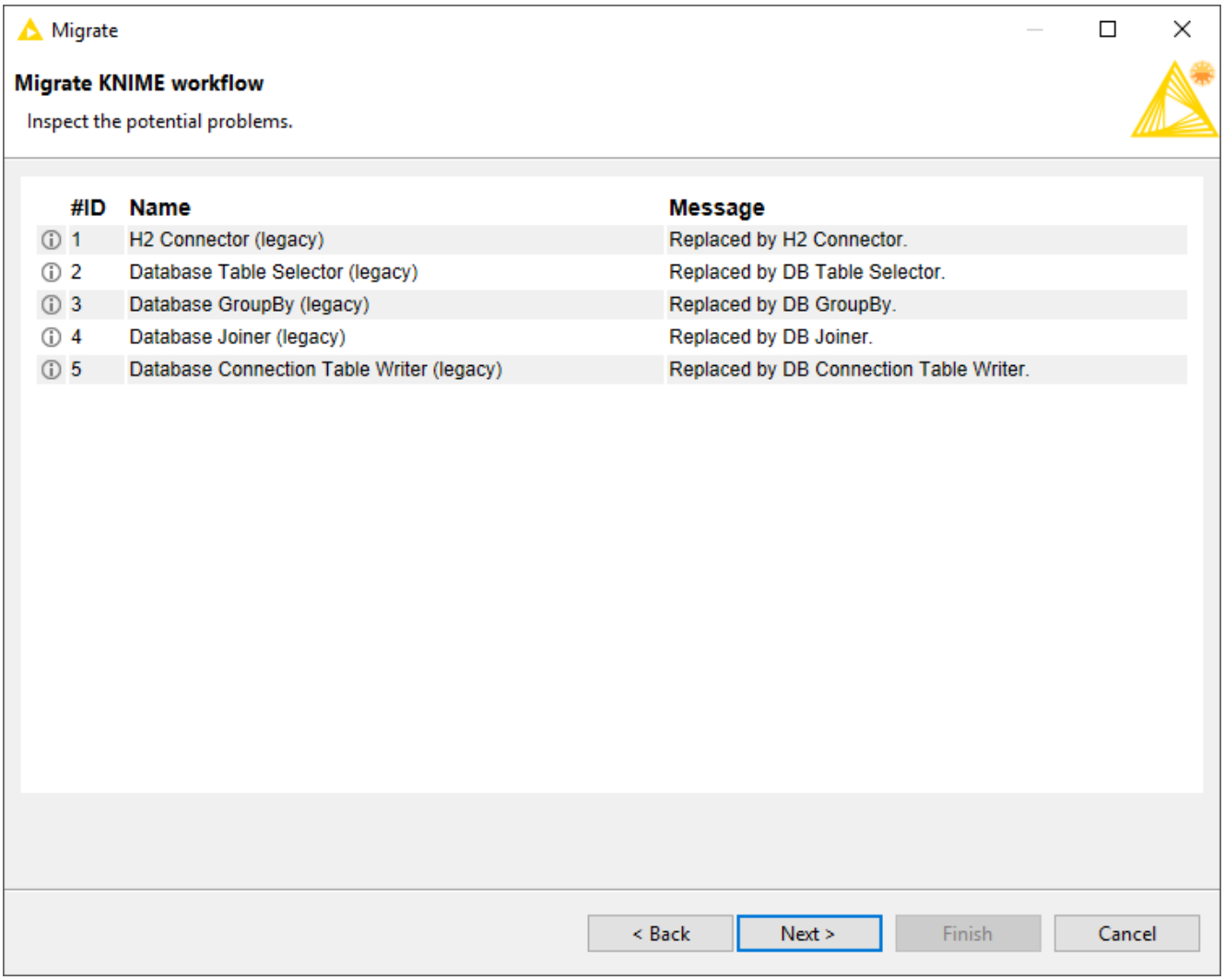


Figure 37. Migration Tool: Show the potential mapping

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.

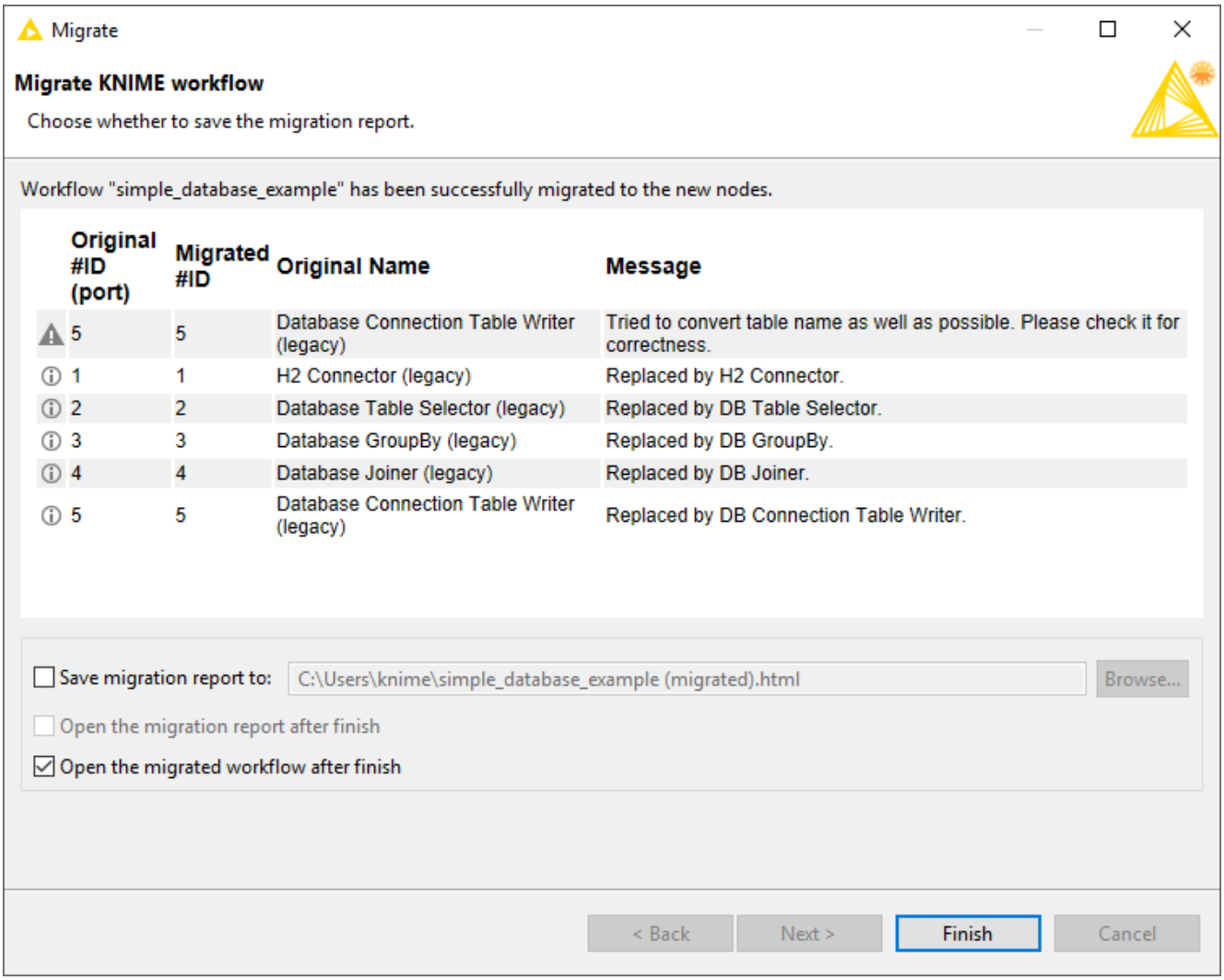


Figure 38. Migration Tool: Migration report

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

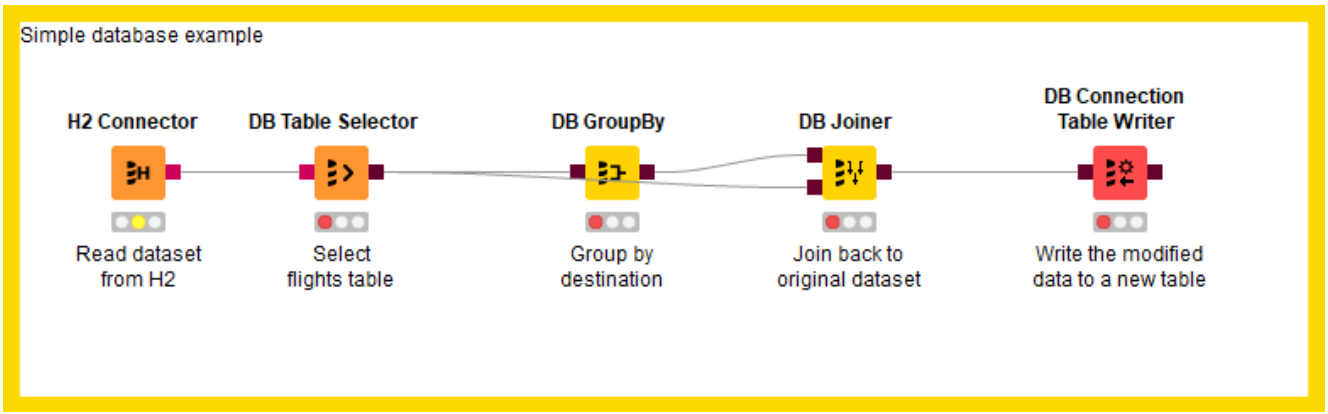


Figure 39. New workflow containing migrated DB nodes

Node Name Mapping

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

Database Legacy nodes	New Database nodes
Amazon Athena Connector	(planned)
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
Database Drop Table	DB Table Remover
Database GroupBy	DB GroupBy
Database Joiner	DB Joiner
Database Looping	Can be replaced with Parameterized DB Query Reader
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 Legacy nodes	New Database nodes
Database Sampling	DB Sampling
Database Sorter	DB Sorter
Database SQL Executor	DB SQL Executor
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	(planned)
-	Microsoft Access Connector

Database Legacy nodes	New Database nodes
-	DB Insert
-	DB Merge
-	DB Column Rename (Regex)
-	DB Partitioning
-	DB Type Mapping

Server Setup

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

Register your own JDBC drivers on the KNIME Server

In order to be able to execute workflows containing database nodes that use custom or proprietary JDBC driver files on KNIME Server, you need to register the JDBC driver not only on the KNIME Analytics Platform but also on the KNIME Server.

The following is the recommended route for systems that have graphical access to the KNIME Analytics Platform (executor):

1. Register the JDBC driver on KNIME Analytics Platform. Please refer to Register your own JDBC drivers section for more details.
2. Export the preference file, rename it to `preferences.epf`, and copy it into the server repository `<server-repository>/config`.
3. The `preferences.epf` file must contain the path to the JDBC jar file, or the folder containing the JDBC driver. Please refer to Preferences file section of the KNIME Server Administration Guide for a more in-depth instruction.

Some systems do not have graphical access to the KNIME Analytics Platform (executor) GUI. In that case the `preferences.epf` can be manually created, or created on an external machine and copied into the folder on the server. The relevant lines that must be contained in the `preferences.epf` file are:

```
file_export_version=3.0
\!/=
/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=<PATH_TO_DRIVER_FILE_1>
/instance/org.knime.database/drivers/<DRIVER_ID>/paths/1=<PATH_TO_DRIVER_FILE_2>
/instance/org.knime.database/drivers/<DRIVER_ID>/url_template=jdbc\:protocol\://<HOST>\:
<PORT>/?databaseName=<DATABASE>
/instance/org.knime.database/drivers/<DRIVER_ID>/version=<DRIVER_VERSION>
```

The colons (":") and backslash ("\") in the url template have to be escaped with a backslash ("\") so `jdbc\:protocol\://<HOST>\:<PORT>\\databaseName=<DATABASE>` will be resolved to `jdbc:protocol://<HOST>:<PORT>\databaseName=<DATABASE>`.

With:

- <DRIVER_ID>: The unique ID of the JDBC driver that needs to match the ID in all KNIME Analytics Platform that are connected to the server.
- <DRIVER_CLASS_NAME>: The JDBC driver class, e.g oracle.jdbc.OracleDriver for Oracle.
- <PATH_TO_DRIVER>: The path to the JDBC driver files. It may also reference a folder in some cases (e.g. if the driver requires companion libraries). The digit at the end of the key is the index of the file. It starts at 0 and needs to be incremented for each additional file as show in the example above.
- <HOST>, <PORT>, <DATABASE>: Tokens in the JDBC URL template. Please refer to the JDBC URL Template section for more information.
- <DRIVER_VERSION>: The version of the JDBC driver e.g. 12.2.0.

We've bundled a file called preferences.epf.template into the <server-repository>/config folder. In order for those preferences to be used, you must edit the file as appropriate, and rename it so that it is named preferences.epf.

Now let's do an example with the Oracle JDBC driver where we use Oracle as driver ID.

1. Register the Oracle JDBC driver on the KNIME Analytics Platform. Please refer to the Register your own JDBC drivers section where we have covered this part.
2. Export the preferences in KNIME Analytics Platform via *File → Export Preferences*. Set the name as preferences.epf and store it in the server directory <server-repository>/config.
3. Make sure that the following lines are contained in the preferences.epf (with the path adapted to your own path).

```
file_export_version=3.0
\!/=
/instance/org.knime.database/drivers/Oracle/database_type=oracle
/instance/org.knime.database/drivers/Oracle/driver_class=oracle.jdbc.OracleDriver
/instance/org.knime.database/drivers/Oracle/paths/0=/home/knime/Downloads/ojdbc8.jar
/instance/org.knime.database/drivers/Oracle/url_template=jdbc\:oracle\:thin\://<HOST>\:<
PORT>/<DATABASE>
/instance/org.knime.database/drivers/Oracle/version=12.2.0
```

Server-managed Customization Profiles

KNIME Server allows you to distribute customization profiles, which can be used to

automatically distribute JDBC drivers to all connected KNIME Analytics Platform clients. In order to enable server-managed customization on the server side you have to create one subdirectory inside `<server-repository>/config/client-profiles`. Inside this subdirectory, we need to create a profile file ending with `.epf` with all the necessary JDBC driver settings:

```
/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}/<DRIVER_FIL
E>
/instance/org.knime.database/drivers/<DRIVER_ID>/url_template=jdbc\:protocol\://<HOST>\:
<PORT>/<DATABASE>
/instance/org.knime.database/drivers/<DRIVER_ID>/version=<DRIVER_VERSION>
```

The description of these lines are already explained in previous section. `<DRIVER_FILE>` should be replaced with the name of the driver file (including the extension).

Most of these lines can be copied directly from the `preferences.epf` as explained in the previous section. The only difference is the path to the JDBC driver. The driver files needs to be put inside the same folder (this should be the subdirectory we created earlier) as the `.epf` file, where together they will be distributed to the KNIME Analytics Platform clients. Since the path to this profile folder (where the `.epf` file and the driver files reside) might be different in every client, we use the variable `${profile:location}` to refer to the location of the profile folder on each client. This variable will then be replaced with the location of the folder on each client.

From the client-side, they simply have to request this profile from the KNIME Server. There are three possibilities to do that, which are described in more details in the Client-side setup section. One possibility to do that is the *Preferences* page in the KNIME Analytics Platform. Go to *File* → *Preferences* → *Customization Profiles* to open the *Customization Profiles* page. In this page you can choose which KNIME Server to use, and select the profile that we have created. The changes will take effect after restarting.

Let's continue the example with the Oracle JDBC driver from the previous section. Now we want to automatically distribute the Oracle JDBC driver to all KNIME Analytics Platform clients that are connected to the KNIME Server to ensure that all clients use the same version and driver id. To do that, we have to:

1. Create a subdirectory inside `<server-repository>/config/client-profiles` and name it `Oracle`.
2. Copy the Oracle JDBC driver into this newly created folder.
3. Create a new empty `.epf` file, give it a name (let's say `oracle.epf`), and copy all database-related lines from the `preferences.epf` (see previous section to find out how

to get this file). The new `oracle.epf` file should look like this:

```
/instance/org.knime.database/drivers/Oracle/database_type=oracle
/instance/org.knime.database/drivers/Oracle/driver_class=oracle.jdbc.OracleDriver
/instance/org.knime.database/drivers/Oracle/paths/0=${profile:location}/ojdbc8.jar
/instance/org.knime.database/drivers/Oracle/url_template=jdbc\:oracle\:thin\://<HOST>\:<
PORT>/<DATABASE>
/instance/org.knime.database/drivers/Oracle/version=12.2.0
```

From the server-side everything is now set up. A client can request this profile from the KNIME Server. One possibility to do that is the *Preferences* page in the KNIME Analytics Platform. Go to *File* → *Preferences* → *Customization Profiles* to open the *Customization Profiles* page. In this page you can choose which KNIME Server to use, and select the profile that we have created. The changes will take effect after restarting.

To see whether the driver has been added, go to *File* → *Preferences* → *Databases*. In this page, drivers that are added via a KNIME Server customization profile are marked *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.

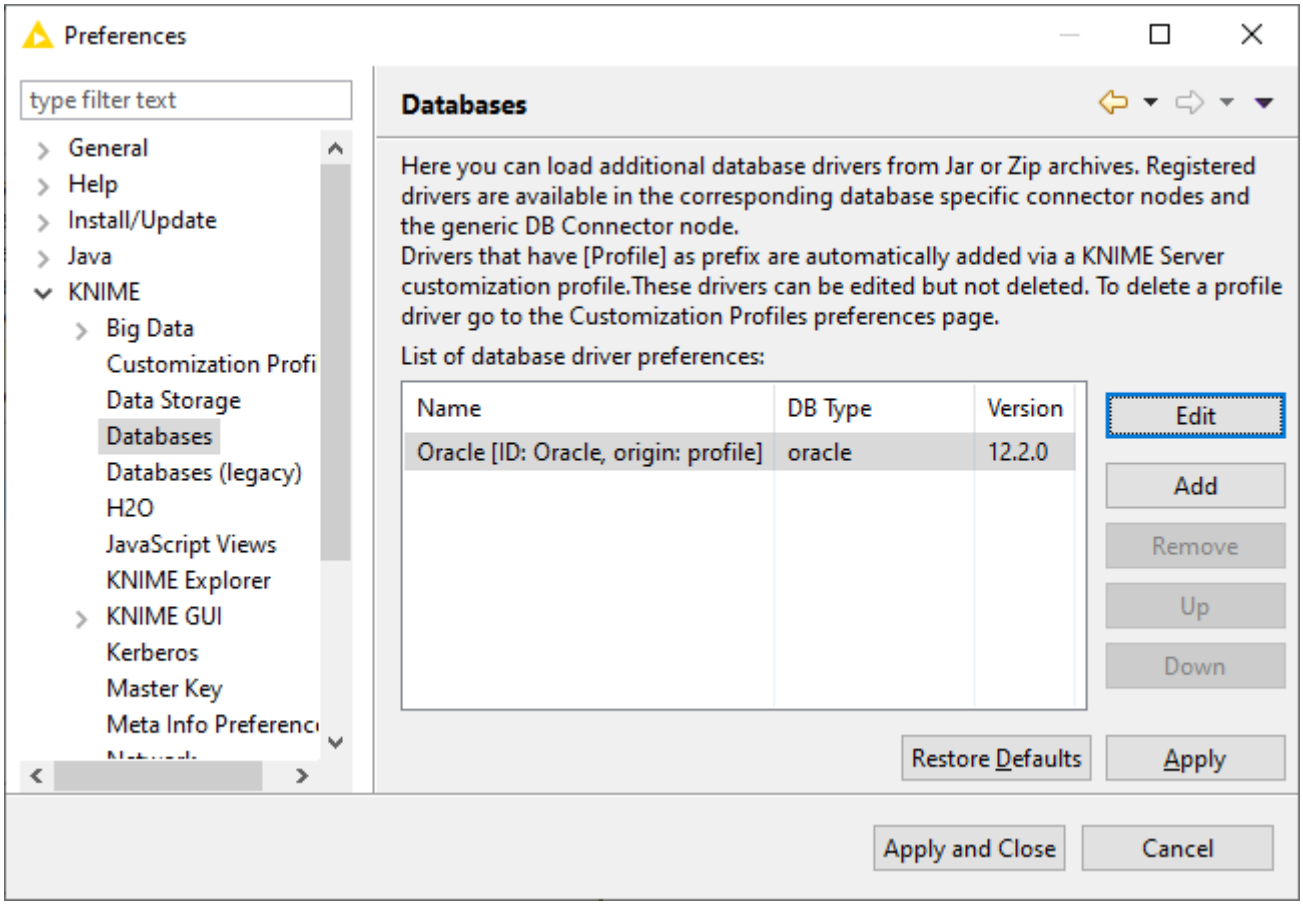


Figure 40. DB Preferences page

Default JDBC Parameters

Users are also allowed to specify JDBC parameters that will be added during connection creation on the KNIME Server, by adding them to the `preferences.epf` file located in the KNIME Server repository (please refer to the section Register your own JDBC drivers on the KNIME Server to find out more about this file). These parameters take precedence over values specified in the workflow to execute. To specify an additional JDBC parameter, add the following lines to the KNIME Server `preferences.epf` file:

```
/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 of the JDBC driver the default parameter should be used whenever the driver is used to establish a connection to the database.
- `<JDBC_PARAMETER>`: The name of the JDBC parameter.
- `<VALUE>`: The value of the JDBC parameter.
- `<TYPE>`: The type of the JDBC parameter. The following types are supported:
  - `CONTEXT_PROPERTY`: Represents the value of a workflow context related property. The supported context variables are:
    - `context.workflow.name`: The name of the KNIME workflow.
    - `context.workflow.path`: The mount-point-relative workflow path.
    - `context.workflow.absolute-path`: The absolute workflow path.
    - `context.workflow.user`: The name of the workflow user that executes the workflow.
    - `context.workflow.temp.location`: The path to the workflow temporary location.
    - `context.workflow.author.name`: The workflow author's name.
    - `context.workflow.last.editor.name`: The workflow last editor's name.
    - `context.workflow.creation.date`: The creation date of the workflow.
    - `context.workflow.last.time.modified`: The last modified time of the

workflow.

- `CREDENTIALS_LOGIN`: The login name from a credentials object whereas the value contains the name of the credential object to use.
- `CREDENTIALS_PASSWORD`: The password from a credentials object whereas the value contains the name of the credential object to use.
- `FLOW_VARIABLE`: Represents the value of a workflow variable.
- `LITERAL`: The value that represents itself.
- `LOCAL_URL`: Represents a local, possibly "knime", URL.

KNIME needs to be restarted after importing the preferences keys because they are only loaded during start-up.

As an example, the following lines are used for user impersonation to access a Kerberos-secured big data cluster via Apache Impala using a JDBC driver with the ID `impala`:

```
/instance/org.knime.database/drivers/impala/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connection.jdbc.properties/DelegationUID/type=CONTEXT_PROPERTY

/instance/org.knime.database/drivers/impala/attributes/additional/org.knime.database.util.DerivableProperties/knime.db.connection.jdbc.properties/DelegationUID/value=context.workflow.user
```

Reserved JDBC Parameters

Certain JDBC parameters could cause security issues when a workflow is executed on the KNIME Server, e.g. *delegationUID* for Impala/Hive connections using a Simba based driver. Such parameters can be reserved to prevent users from modifying the values in the client before executing the workflow on the KNIME Server. To set a parameter as reserved, add the following lines to the KNIME Server `preferences.epf`:

```
/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.connection.jdbc.properties/<JDBC_PARAMETER>=true
```

Where:

- `<DRIVER_ID>`: The unique ID of the JDBC driver the parameter should be reserved whenever the driver is used to establish a connection to the database.
- `<JDBC_PARAMETER>`: The name of the JDBC parameter.

KNIME needs to be restarted after importing the preferences keys because they are only loaded during start-up.

As an example, to set the *delegationUID* parameter from the previous example (see Default JDBC Parameters) to reserved for the JDBC driver with the ID `impala`, add the following line to the `preferences.epf`:

```
/instance/org.knime.database/drivers/impala/attributes/reserved/org.knime.database.util.
DerivableProperties/knime.db.connection.jdbc.properties/DelegationUID=true
```

KNIME AG
Technoparkstrasse 1
8005 Zurich, Switzerland
www.knime.com
info@knime.com