KNIME WORKFLOW
Exploring Targets Associated with Liver Diseases from ChEMBL Database

1 Objectives

The main objective of this workflow is to explore KNIME as a low-code programming platform, by creating a pipeline that connects to an existing database, extracts the data which is necesssary to investigate the scientific questions for the project in hand, curates the data for further analysis, and inserts it into a new database.

For this purpose, the PostgreSQL version of the ChEMBL database is used as starting point.

As a reminder, the scientific questions to be investigated include:

1. What are the target types and how many compounds have been tested against each target type associated with liver diseases in humans?
2. What is the distribution of activities for compounds targeting the top 5 most frequent targets associated with liver diseases, and how does this distribution vary by target type?

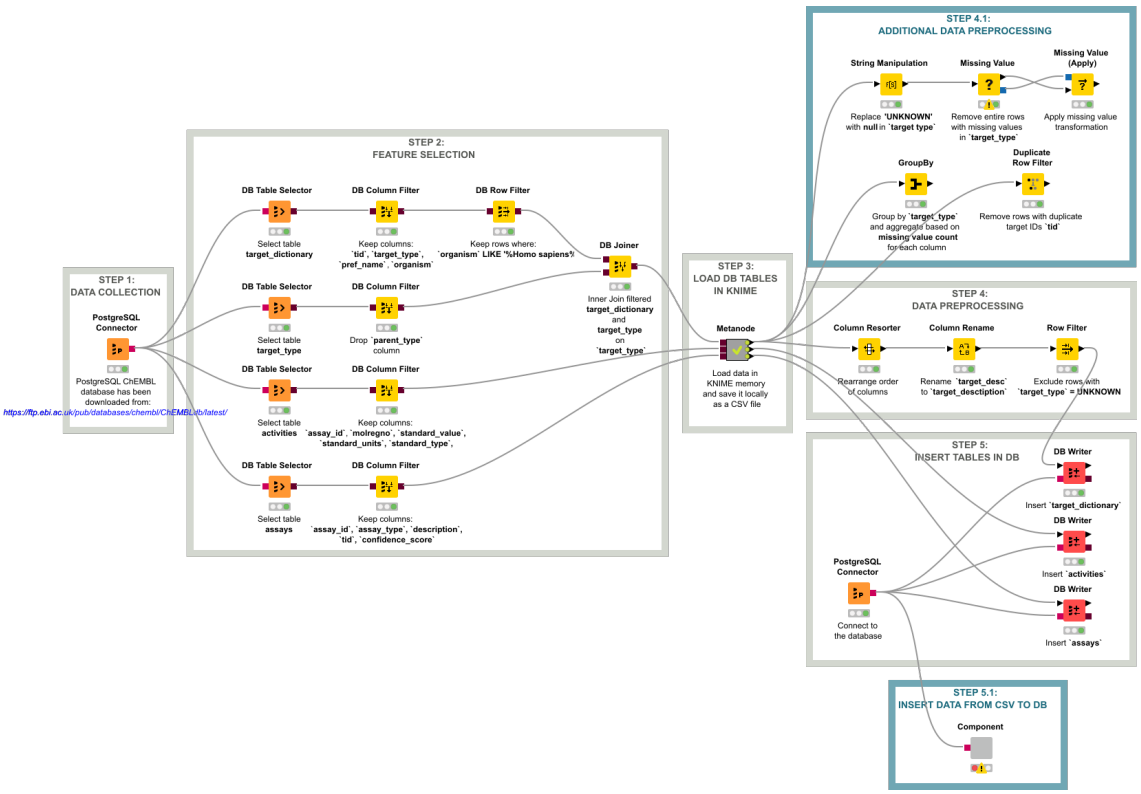
2 Methodology

The overall KNIME workflow consists of five steps:

- **Step 1:** data collection
- **Step 2:** feature selection
- **Step 3:** loading data in KNIME memory
- **Step 4:** data preprocessing
- **Step 5:** inserting tables in the database

summarized in the figure below.

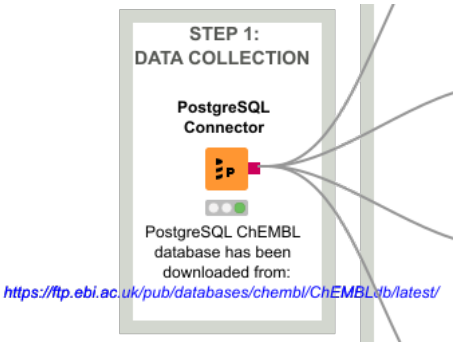
The annotations in grey present the main workflow, whereas the annotations in blue present alternative or additional operations.



2.1 Step 1: Data collection

Initially, the PostgreSQL version of the ChEMBL database has been downloaded from https://ftp.ebi.ac.uk/pub/databases/chembl/ChEMBLdb/latest/.

A *PostgreSQL Connector* node is used to import the whole database in KNIME, by selecting "PostgreSQL" as the database type and providing the necessary connection details:



An alternative of integrating the ChEMBL database in KNIME is using its REST services and the corresponding API key to extract the data from the latest version.

2.2 Step 2: Feature selection

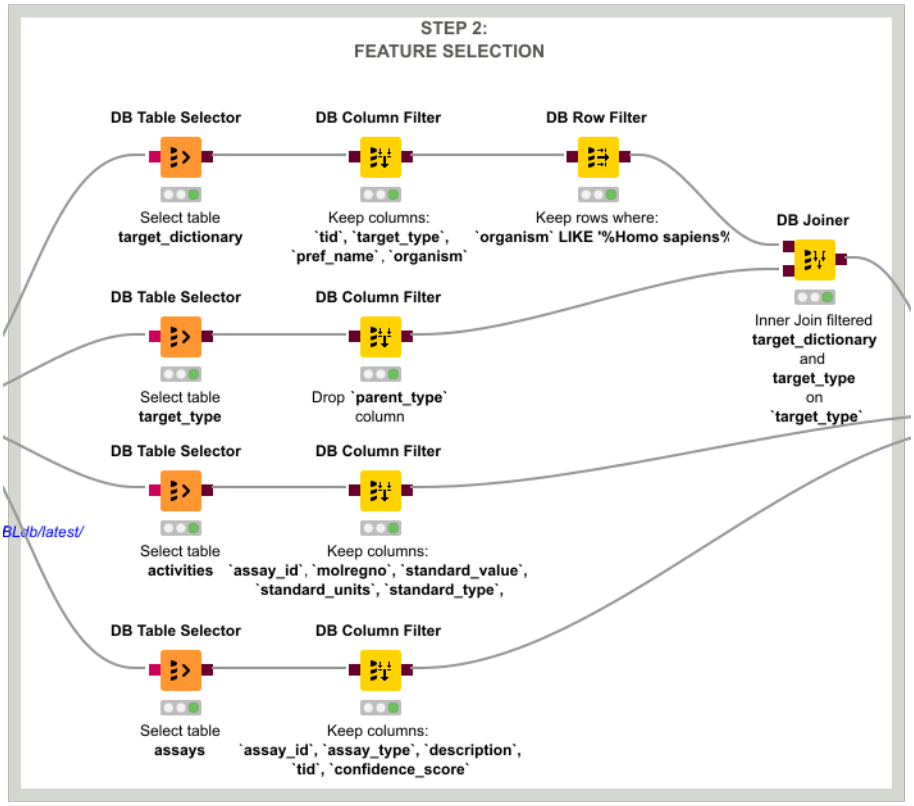
To answer the above-mentioned scientific question, several tables need to be selected and filtered from the ChEMBL database. These include:

- **target_dictionary** - contains information about the biological system in which the bioactivity data is measured. For the present project, the most relevant columns from this table are: 'tid' (target ID), 'target_type' (type of target, e.g., protein, enzyme, receptor), 'pref_name' (the preferred name of the target), and 'organism' (organism in which the target is expressed).

- **activities** - contains information about the activity of compounds in various assays. For the present project, the most relevant columns from this table are: 'molregno' (the molecule ID), 'standard_value' (numerical value of the activity measurement), 'standard_type' (type of activity measurement), 'standard_units' (units of the activity measurement), and 'assays_id' (the assay ID, through which activities can be related to the assays).

- **assays** - contains information about the experimental assays that have been performed on compounds. For the present project, the most relevant columns from this table are: 'assay_id' (the assay ID), 'assay_type' (type of assay, e.g., binding (B), functional(F)) , 'description' (description of the assay), and 'tid' (the target ID, through which assays can be linked to targets).

A *DB Table Selector* is used to select the **target_dictionary, activities,** and **assays** tables, followed by a *DB Column Filter* node which filters each in a such a way that only the most relevant columns for the present project are kept. This step is necessary in order to reduce the amount of data (and memory) that needs to be processed (stored) by KNIME so as to improve the overall performance of the workflow:



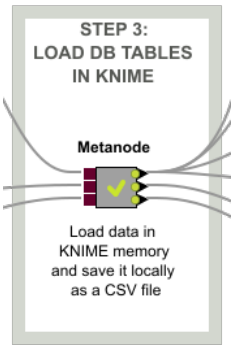
To limit the study for targets expressed in humans only, a *DB Row Filter* with the SQL query **organism LIKE '%Homo sapiens%'** is used to extract only rows containing the 'Homo sapiens' word.

Moreover, a fourth table - the **target_type** has been selected from the ChEMBL database. This table contains more detailed information about the specific target, and consists of three columns: 'target_type', 'target_desc' (description of the target), and 'parent_type', of which only the former two are kept. While the table **target_type** does not directly contribute in answering the posed scientific questions, it has been included in the workflow to help in demonstrating the function of the *DB Joiner* node in KNIME. Using the inner join mode of *DB Joiner*, the tables **target_dictionary** and **target_type** have been combined into a single one (hereafter denoted as the modified **target_dictionary** table) based on the common 'target_type' column .

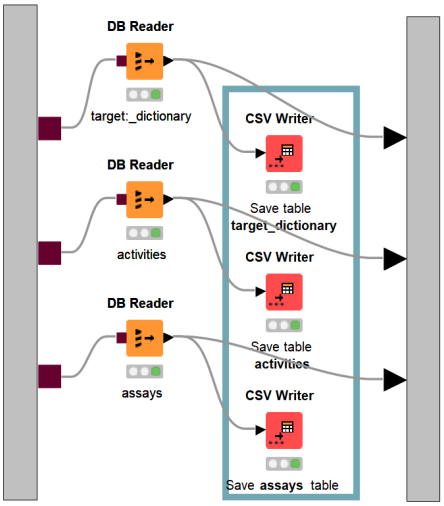
To demonstrate the functionality of the *DB Joiner* node, the **target_dictionary** table could have been joined with the **assays** one the 'tid' column, and the **assays** table could have been joined with the **activties** on the 'assay_id'. However, to avoid data redundancy and the generation of too large data sets, I decided to keep the **activities** and **assays** tables separate and perform any necessary joins of these tables during the analysis phase in PostgreSQL.

2.3 Step 3: Loading data in KNIME memory

In the next step, the tables have been cached in the memory of KNIME using the *DB Reader* node. For this purpose, *DB Reader* nodes have been encapsulated in a single *Metanode*, simplifying the overall view of the workload.



The workflow which has been encapsulated in a metanode is given here:



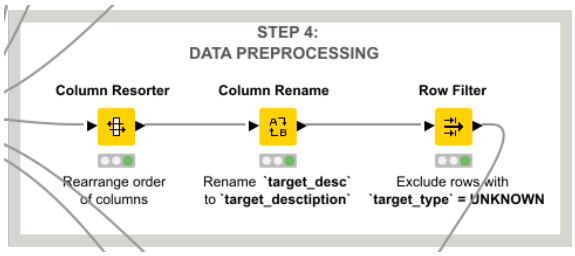
Three 'DB Data' input ports have been added to the Metanode to connect the three different tables (the modified **target_dictionary, activities,** and **assays** with the corresponding *DB Reader*. The outputs of the *DB Reader* nodes are then connected to three different 'Table' output ports of the Metanode, allowing the data from the three tables to be accessed separately.

Assuming that no further data preprocessing takes place in the steps that follow, another operation that could be performed at this stage is to save the tables locally, e.g., in a CSV (Comma Separated Value) format using the *CSV Writer* node.

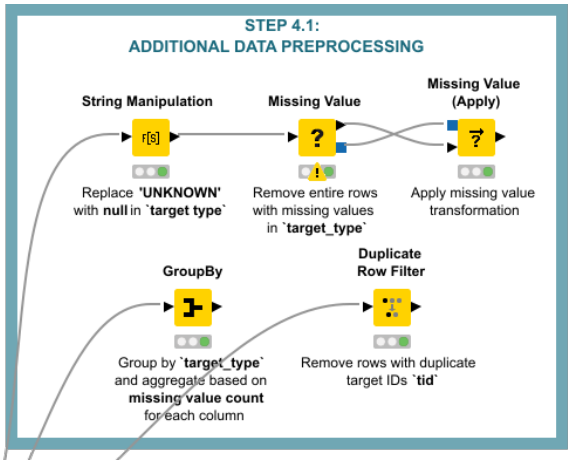
2.4 Step 4: Data preprocessing

Once the tables have been cached in KNIME's memory, the columns of the modified **target_dictionary** table have been rearranged in the following order: 'tid', 'organism', 'target_type', 'target_desc', 'pref_name', using the *Column Resorter* node. Additionally, the column 'target_desc' has been

renamed to 'target_description' using the KNIME *Column Rename* node. Finally, a *Row Filter* is used to exclude unknown targets from the modified **target_dictionary** table. The operations of step 4 are shown here:



To demonstrate the functionality of other commonly used nodes in data preprocessing with KNIME, an alternative operation of excluding unknown target types is presented in the blue annotation (namely in Step 4.1):



Initially, a *String Manipulation* node has used to replace the 'UNKNOWN' string to Null value, in the 'target_type' column, using the expression:

toNull(replace($target_type$, "UNKNOWN", ""))

and replacing the 'target_type' column with the modified one. Then, the *Missing Value* node has been used to drop rows which contain Null values in the 'target_type' column, by choosing this column in the Column Settings and the Remove Row* option. The transformation is applied using the *Missing Value (Apply)* node.

Nodes such as: *GroupBy* and *Duplicate Row Filter* are also applied to the modified **target_dictionary** table to check for missing values and duplicate rows (there are none as the database is well curated). In the *GroupBy* node the results have been grouped by the 'target_type', and for the remaining columns the aggregation was set to 'Missing value count', with the column naming corresponding to the aggregation method. For the latter node, only the 'tid' column was scanned for duplicates.

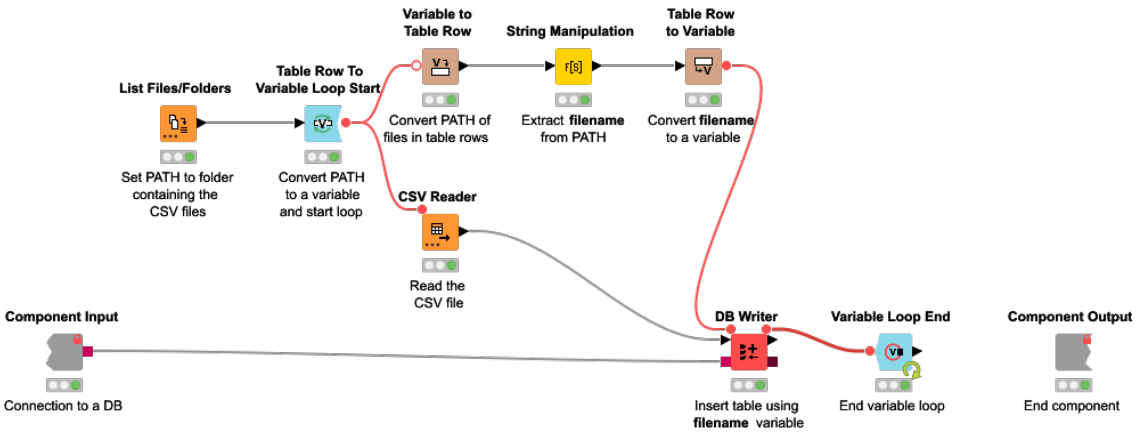
A quick comparison between step 2 and step 4, reveals two possible ways of data processing: one using database-related KNIME nodes (step 2) and the other using KNIME nodes after caching the data in KNIME memory (step 4). Both ways have their merits and drawbacks. Database-related KNIME nodes can be faster for large data sets as the processing is done on the database server side and only the relevant data is loaded into KNIME. However, they have limited functionality compared to KNIME nodes and may not provide all the operations required for preprocessing the data. On the other hand KNIME nodes that work with data cached in KNIME memory have a wide range of functionality and may provide greater flexibility and control over the data, although

loading data into KNIME memory can be time-consuming, memory-intensive and not feasible for very large data sets. It is also important to note, that performing data preprocessing with KNIME nodes may lead to data inconsistencies if the database is updated while the data is being processed, and in such cases one should rely on database-related KNIME nodes.

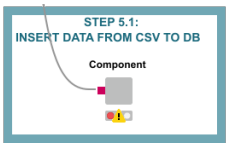
2.5 Step 5: Inserting tables in the database

Three *DB Writer* nodes (one for each table: modified **target_dictionary, activities** and **assays**) have been connected to a second *PostgreSQL Connector* and used to insert the tables into a new (already created) database in PostgreSQL. Alternatively, the *DB Update* node could have been used in combination with the *PostgreSQL Connector* from step 1 to update the existing tables in the ChEMBL database.

Furthermore, I have created a separate workflow (step 5.1 in the blue annotation), which writes the data from a CSV file to a PostgreSQL database:



This workflow has been encapsulated into a single *Component* as a reusable building block which could be shared across and incorporated in multiple workflows:



The *Component* has a 'DB Session' input port connected to a *PostgreSQL Connector* (can also be any other DB connector, as long as it is properly configured) and no output port. The path for the three CSV files containing the data from each table (modified **target_dictionary, activities** and **assays**) separately, is specified in the *List Files/Folders* node, in which the files to be loaded are filtered based on the CSV file extension. This node is then connected to the *Table Row to Variable Loop Start* which initiates the iteration over the files. The variable output port of the loop start follows two routes. One, where the path variable is converted to a table row (*Variable to Table Row*) and the filename is extracted using a *String Manipulation* node and the *regexReplace()* function with the following expression **regexReplace($Path$, ".*\\\\([^ \\.]+).*", "$1")**. This pattern uses a negative character class [^\\.] to match any character except a period (.), and the + quantifier to match one or more of those characters. The $1 replacement captures the matched characters and replaces the entire string with just the filename. The extracted filename is then converted to a variable (*Table Row to Variable*) to be used as a table name in the *DB Writer*. Since this route is placed within the variable loop, it ensures that the correct filename (table name) is assigned at each iteration. On the second route, the variable output port from the loop start is connected to a *CSV Reader* which reads the data from each file and directs it to the *DB Writer*.

In the *CSV Reader*, the 'Support changing file schemas' option has been enabled since the input file structure changes between different invocations. The end of the loop is controlled using the *Variable Loop End* node.