DbVisualizer 25.3

# User Guide

# Table of contents


Getting Started 3
Getting the Most Out of the GUI 37
Managing Database Objects 71
Working with Tables 87
Working with Views 197
Working with Procedures, Functions and Other Code Objects 205
Working with Schemas 222
Working with SQL 233
Working with Result Sets 416
Working with Charts 425
Working with Workspaces 434
Working with Git 437
Exporting a Grid 450
Opening a Grid in an External Application 457
Comparing Data 459
Monitoring Data Changes 465
Accessing Frequently Used Objects 473
Delimited Identifiers and Qualifiers 483
Handling Transactions 484
Database Connection Options 487
Finding Database Objects and Data 550
Transfer DbVisualizer settings 562
Exporting and Importing Settings 563
Command Line Interface (CLI) 568
Telemetry 584
Database Profiles 585
Troubleshooting 718
Reference Material 731

# Getting Started

## Getting Started

This section will help you get going with DbVisualizer - from downloading and installing the software, to creating a database connection and working with the data.


- Downloading
- Installing
- Starting DbVisualizer
- Evaluating the Pro Edition
- Installing a Pro Edition License
- Installing the Demo Database
- Creating a Connection - basics
- Creating a Table - basics
- Viewing a Table - basics
- Editing a Table - basics
- Executing SQL - basics
- Updating DbVisualizer
- Printing

# Downloading

DbVisualizer installers are available on the DbVis Software web site at https://www.dbvis.com/download/.

**We recommend choosing the Installer with Java VM** - this is the safest option and ensures DbVisualizer works as expected, regardless of your current Java setup.

Other options:

- **Installer without Java VM** – For advanced users who already have a compatible Java version installed.
- **Archive formats** – Only needed for special installation scenarios.

**In most cases, use the Installer with Java VM for the best experience.**

# Installing

There are two ways to install DbVisualizer, using an Installer or extracting from an archive file.

## Installing with a Setup Installer

To install DbVisualizer, just execute the Installer you have downloaded and follow the instructions in the screens. The setup installer optionally creates a desktop shortcut used to launch DbVisualizer which is not the case for the other installers below.

## Installation from an archive file

### ZIP archives (Windows)

Unpack the distribution file with the built-in zip archive extraction utility in Windows.

The ZIP archive installer will not add any entries to the **Start** menu, add desktop launchers or register the software in the Windows registry. Start DbVisualizer by running `DbVisualizer\dbvis`.

To uninstall, simply delete the DbVisualizer directory.

### DMG archives (macOS)

Open the installer file:

```
open dbvis_macos_<version>.dmg
```

In the mount window drag the DbVisualizer.app to the Applications folder.

To uninstall, remove **/Applications/DbVisualizer.app**.

### TGZ archives (macOS)

Open the installer file:

```
open dbvis_macos_<version>.tgz
```

A **DbVisualizer.app** directory is created in the same directory. Start DbVisualizer:

```
open DbVisualizer.app
```

To uninstall DbVisualizer installed via a TGZ archive, simply delete the **DbVisualizer.app** directory.

### TAR.GZ archives (Linux)

All files are contained in an enclosing folder named `DbVisualizer`.

Unpack the distribution file in a terminal window with:

```
tar xvfz dbvis_linux_<version>.tar.gz
```

Start DbVisualizer by executing the command:

```
sh DbVisualizer/dbvis.sh
```

To uninstall DbVisualizer installed via a TAR archive, simply delete the complete DbVisualizer directory.

### RPM archives (Linux)

Install the RPM package with:

```
sudo rpm -i dbvis_linux-<version>.rpm
```

Start DbVisualizer by either finding the application and double-clicking on its icon or by executing the dbvis command in a shell.

To uninstall DbVisualizer installed via an RPM archive, run:

```
sudo rpm -e dbvis
```

DEB archives (Linux)

Install the package with the following command in a terminal window (or use your favorite package manager):

```
sudo dpkg -i dbvis_linux_<version>.deb
```

Start DbVisualizer by either finding the application and double-clicking on its icon or by executing thedbvis command in a shell.

To uninstall DbVisualizer installed via an DEB archive, run sudo dpkg --remove dbvis.

## Silent Install

In order to start a silent installation, the installer has to be invoked with the **-q** argument. The installer will perform the installation as if the user had accepted all default settings.

There is no user interaction on the terminal. The installer will install the application to the default installation directory, unless you pass the **-dir** parameter to the installer. The parameter after -dir must be the desired installation directory. Example:

```
dbvis_windows-x64_<version>_jre.exe -q -dir "d:\myapps\DbVisualizer"
```

The output of the installer is not printed to the command line for silent installation. If you pass the **-console** parameter after the **-q** parameter, a console will be allocated that displays the output to the user. This is useful for debugging purposes.

If the installation was successful, the exit code of the installer will be 0, if no suitable JRE could be found it will be 83, for other types of failure it will be 1.

For more options check the command line options for the installer.

See the following for examples on respective operating system.

**Windows**

```
dbvis_windows-x64_<version>_jre.exe -q -console
echo %errorlevel%
```

**macOS**

```
DbVisualizer Installer.app/Contents/MacOS/JavaApplicationStub -q -console
echo $?
```

**Linux**

```
dbvis_linux_<version>.sh -q -console
echo $?
```

The **-console** argument may be used for debugging purposes. The **echo** command verifies the exit code from the installer which may be useful if automating the installation.

## Upgrading

When upgrading DbVisualizer from one major version to a new major version, user settings for the old version of DbVisualizer are migrated for use with the new version.

> When upgrading from one major version to another major version, migration of the user settings is not performed when running the command-line interface, **dbviscmd**. You must launch the DbVisualizer GUI to migrate your current settings for use with the new version. dbviscmd can not be run until this is done.

# Starting DbVisualizer

How to start DbVisualizer depends on the operating system you are using.

- **Windows** In the **Start** menu, select the **DbVisualizer** menu item.
- **Linux/Unix** Open a shell and change directory to the DbVisualizer installation directory. Execute the dbvis program. If DbVisualizer was installed using the setup installer an optional desktop icon can be used to launch the app.
- **macOS** Double click on the **DbVisualizer** application or the **DbVisualizer.app** application bundle.

You can also start DbVisualizer with the bundled script files, please see the GUI Command Line Arguments page for details. For tasks that do not require a GUI, such as tasks scheduled via the operating system's scheduling tool, you can also use the pure command-line interface.

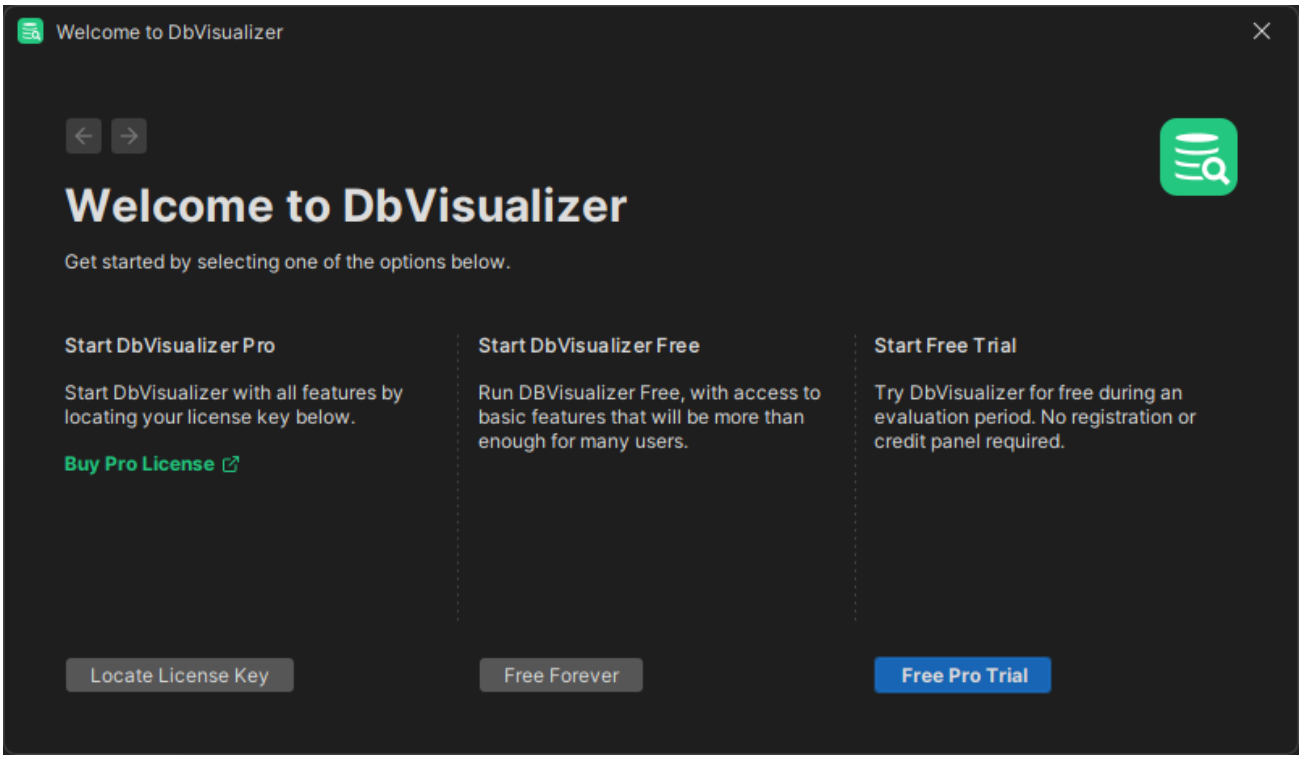
## EULA (End User License Agreement)

When you start DbVisualizer for the first time, or after upgrading to a more recent version, you are prompted to accept the End User License Agreement.

## Free or Pro Version

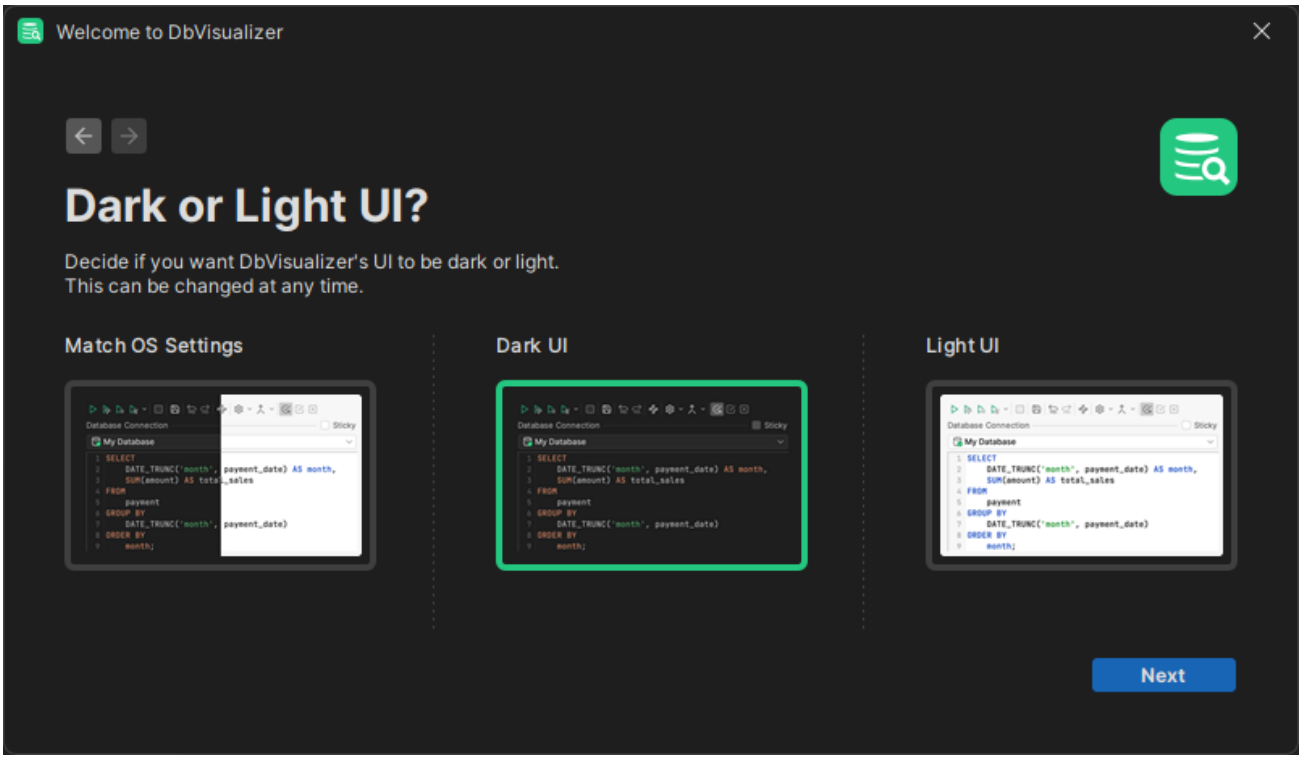
If you start DbVisualizer without a valid license, you are presented with a dialog that prompts you to choose how to proceed; you can proceed and run the Free version, request an Evaluation license, or register a license for the Pro version.

You can also proceed without entering a license and do that later using the **Help** menu (see Evaluating the Pro Edition and Installing a Pro Edition License).



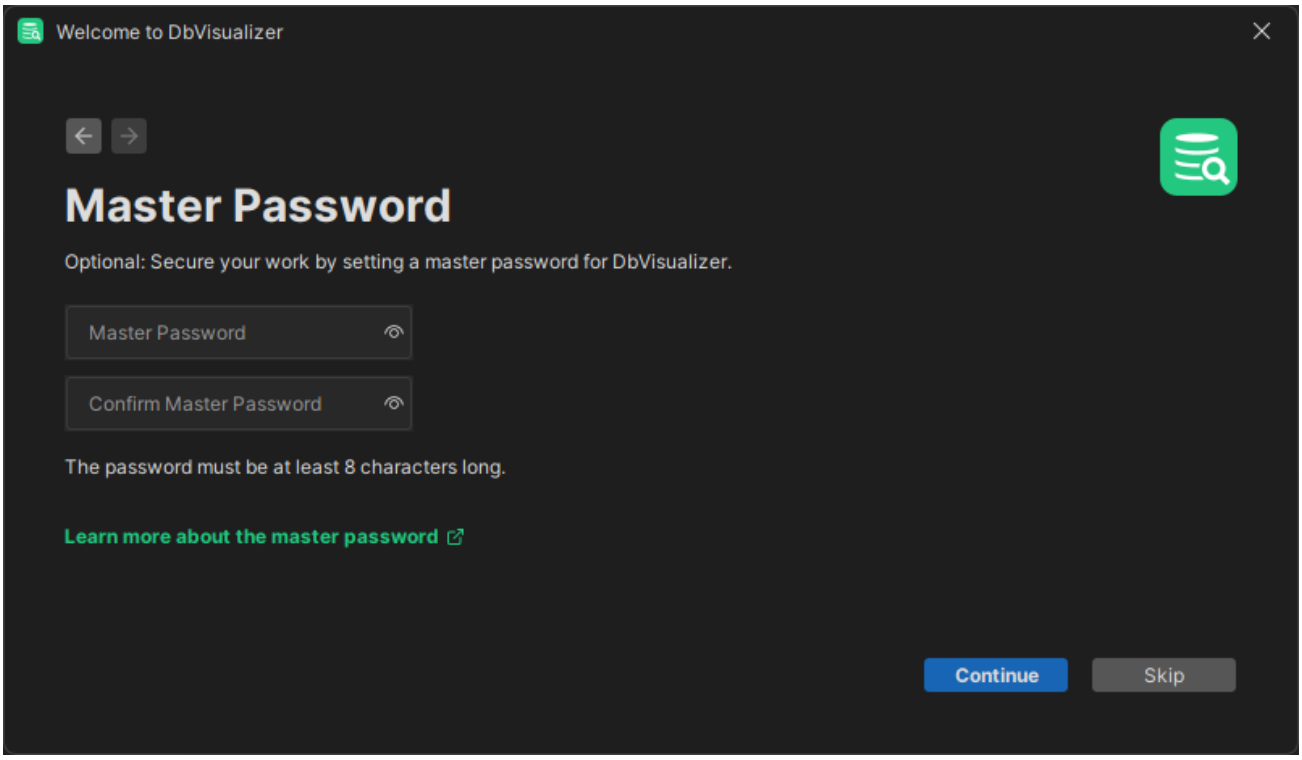

## Theme

Next, you may choose to change the theme. This setting may also be changed later (see Changing the GUI Appearance).



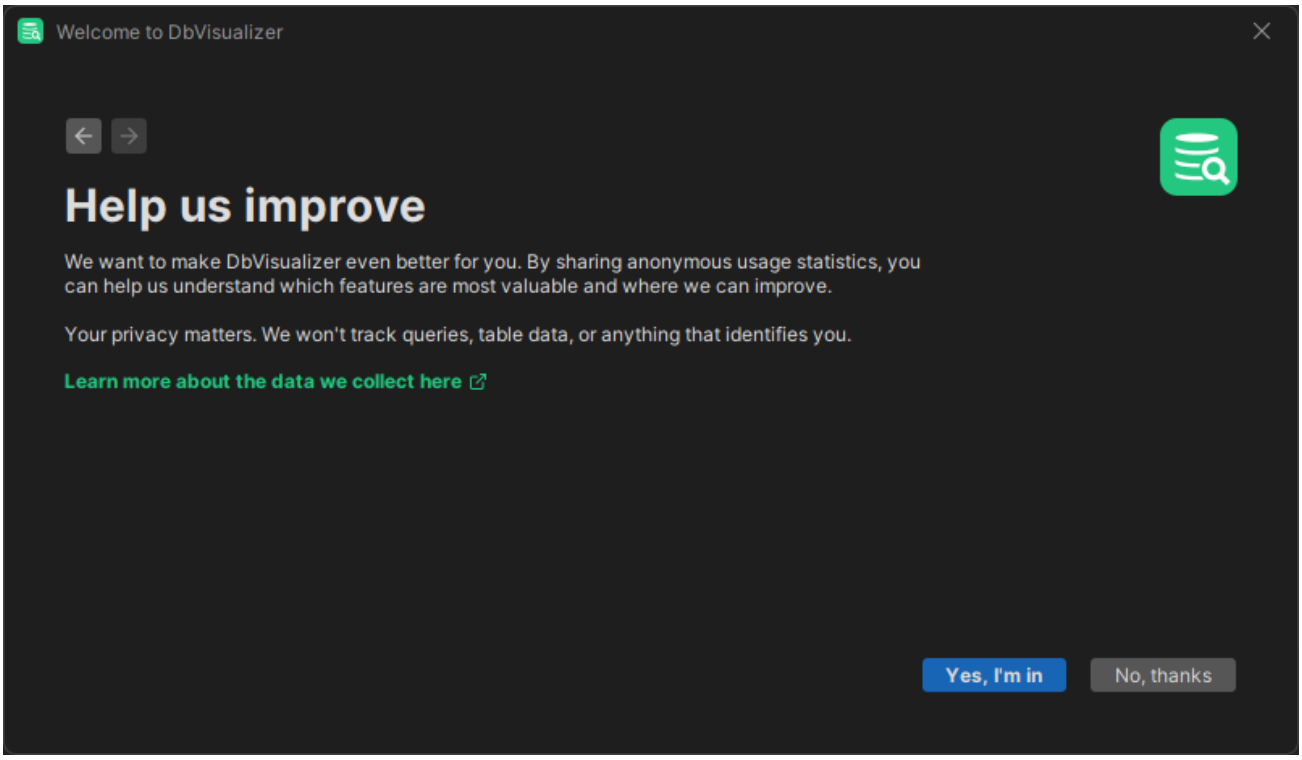

## Master Password

A master password option comes next. It is not required to set a master password, but it can be useful to prevent unauthorized access to your work. This is another setting that can be changed later (see Setting a Master Password)




## Anonymous Statistics

Lastly, we ask you to help us improve, by collecting anonymous user data to help us analyze which features are valuable. This is yet again something that can be reconfigured later (see Telemetry).




## Background Panel

When you first open DbVisualizer, or if you close all tabs at any time later, the backgrund panel becomes visible; this holds a few useful shortcuts and links.

# Evaluating the Pro Edition

The DbVisualizer Pro edition offers far more features than the Free edition. If you are using the Free edition, it is easy to activate a Pro edition evaluation to see if suits your needs.

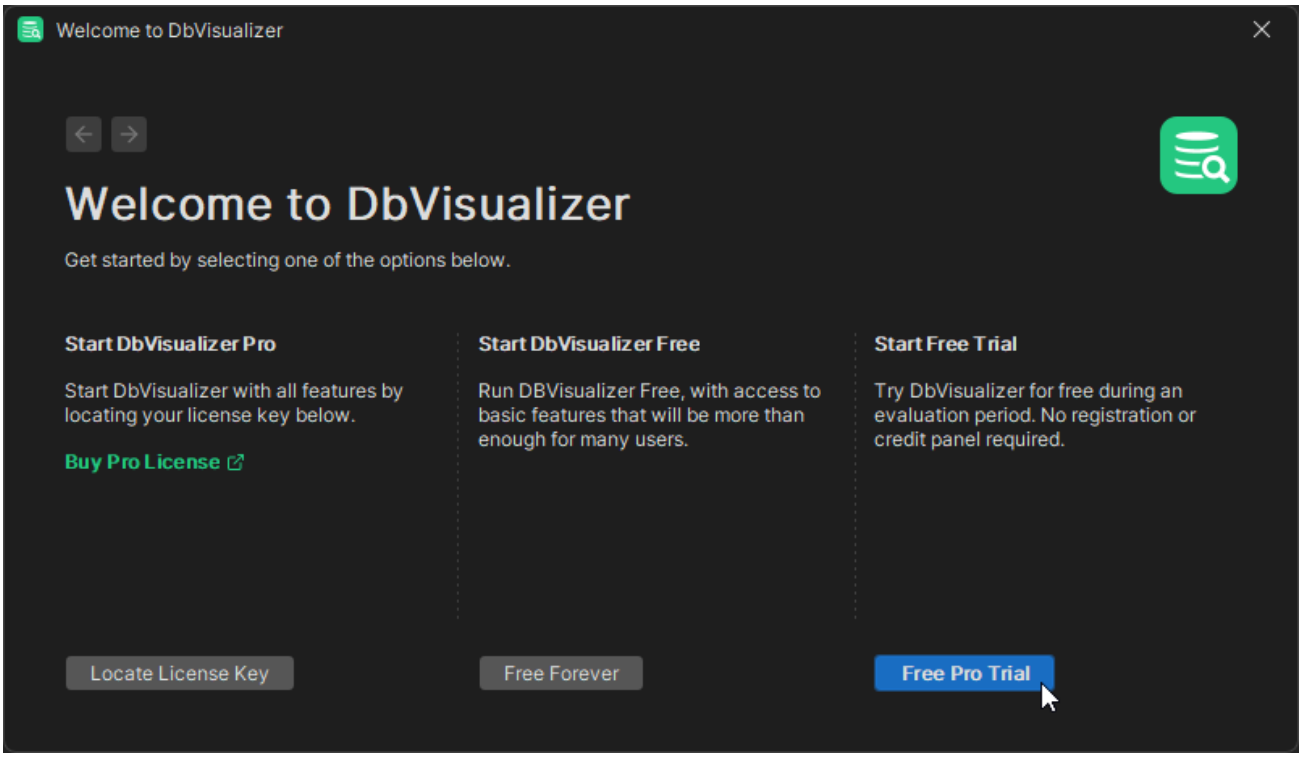
1. Request an evaluation license using any of the methods below
2. Enter your email address and click **Evaluate**
3. Click **Restart** when prompted after the activation of the evaluation.

> If you start DbVisualizer with one of the bundled scripts rather than with the launcher, you need to manually restart DbVisualizer after the activation.

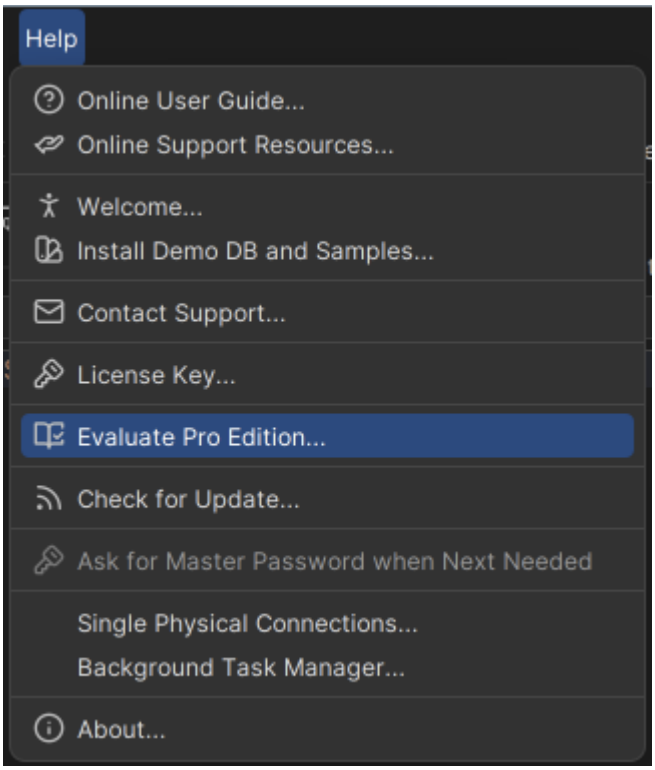
## Request an evaluation license

There are several ways to request an evaluation license:

### A. When DbVisualizer starts for the first time

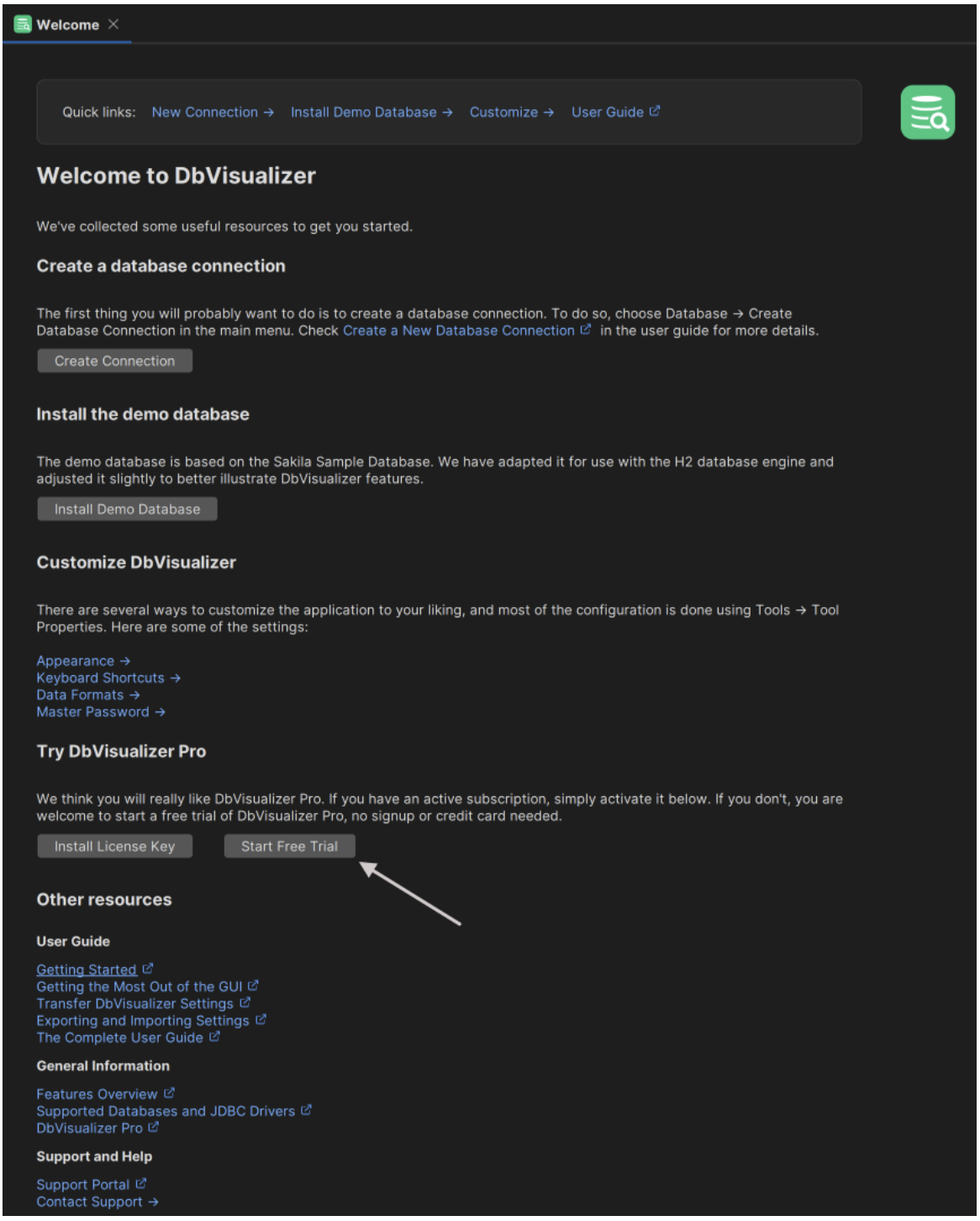


## B. From the Help Menu




## C. From the Welcome Screen

Click **Start Free Trial** to request an evaluation license.


Welcome
Quick links: New Connection → Install Demo Database → Customize → User Guide
Welcome to DbVisualizer
We've collected some useful resources to get you started.
Create a database connection
The first thing you will probably want to do is to create a database connection. To do so, choose Database → Create Database Connection in the main menu. Check Create a New Database Connection in the user guide for more details.
Create Connection
Install the demo database
The demo database is based on the Sakila Sample Database. We have adapted it for use with the H2 database engine and adjusted it slightly to better illustrate DbVisualizer features.
Install Demo Database
Customize DbVisualizer
There are several ways to customize the application to your liking, and most of the configuration is done using Tools → Tool Properties. Here are some of the settings:
Appearance →
Keyboard Shortcuts →
Data Formats →
Master Password →
Try DbVisualizer Pro
We think you will really like DbVisualizer Pro. If you have an active subscription, simply activate it below. If you don't, you are welcome to start a free trial of DbVisualizer Pro, no signup or credit card needed.
Install License Key
Start Free Trial
Other resources
User Guide
Getting Started
Getting the Most Out of the GUI
Transfer DbVisualizer Settings
Exporting and Importing Settings
The Complete User Guide
General Information
Features Overview
Supported Databases and JDBC Drivers
DbVisualizer Pro
Support and Help
Support Portal
Contact Support →

# Installing a Pro Edition License

To enable the Pro edition features, you need to install the License Key String or License Key File that you received after purchasing the license.

## Installing a License Key String

1. Select and copy the License Key String included in the email,
2. Start DbVisualizer and select the **Help → License Key** main menu choice,
3. Select License Key String as the **License Type**,
4. Paste the key string into the text area,
5. Click **Install License**,
6. Restart DbVisualizer when prompted to do so.

The DbVisualizer main window should now say **DbVisualizer Pro** in the window title. You're ready to go.

## Installing a License Key File

1. Save the dbvis.license file attached to the email to disk,
2. Start DbVisualizer and select the **Help → License Key** main menu choice,
3. Select License Key File as the **License Type**,
4. In the **License Key File** field, enter the path to the newly saved dbvis.license file or click the button to the right of the field to open a file browser to locate the file,
5. Click **Install License**,
6. Restart DbVisualizer when prompted to do so.

> An option to saving the dbvis.license file to disk is to drag it from you mail application (or elsewhere) into the **License Key File** field in the **Help → License Key** window.

The DbVisualizer main window should now say DbVisualizer Pro in the window title. You're ready to go.

## Uninstalling the license key

If you ever need to uninstall the license key, you can do so by removing the license file. Its path is listed in DbVisualizer Pro, license file location.

# DbVisualizer Pro, license file location

The license key file for DbVisualizer Pro is located in the following paths depending on used operating system:

| Operating System | Filename |
| --- | --- |
| Windows | C:\Users\<user>\.dbvis\dbvis.license |
| UNIX/Linux | /home/<user>/.dbvis/dbvis.license |
| macOS | /Users/<user>/.dbvis/dbvis.license |

# Installing

The quickest way to get started and explore the DbVisualizer features is probably to install the demo database.

The demo database is based on the Sakila Sample Database, adapted for use with the H2 database engine, and adjusted to better illustrate DbVisualizer features.

**Note:** The available features are restricted by the capabilities of the H2 engine; in order to get the full picture, you should connect DbVisualizer to the database(s) that you intend to use.

## Installing/Uninstalling

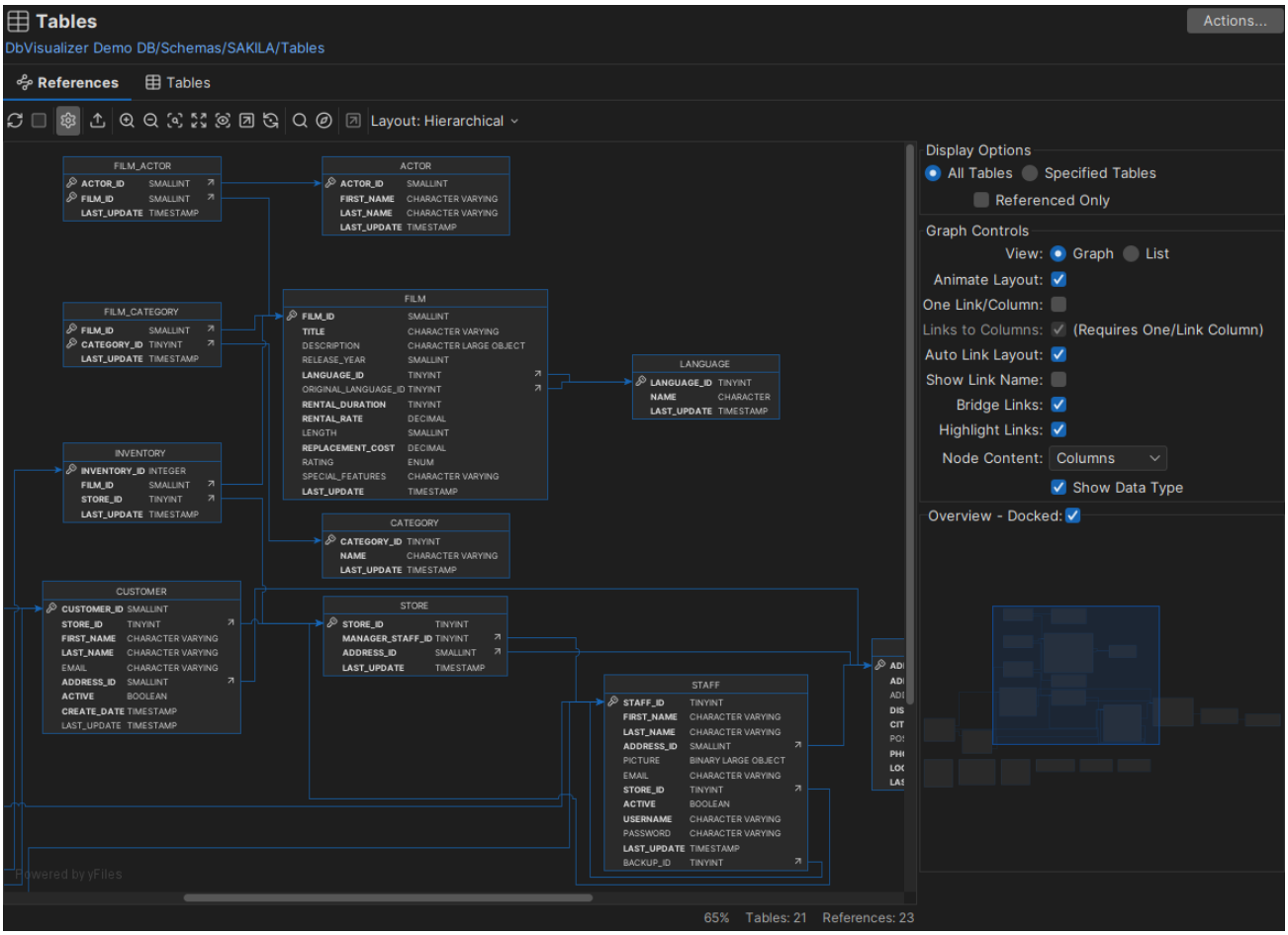
You can install, uninstall or reinstall the demo database from the background panel (visible when all tabs are closed) or from the **Help** menu. The installation dialog shows you the details, but in essence, you will get a local file for the database (initialized with the sample data), a connection for the database, and a set of sample scripts. You can use this database as you please to experiment; if you mess things up, you can always reinstall it to get a clean start.

## The Database

The demo database represents a DVD rental store with fictitious actors, movies, customers, etc. Besides tables and views, the sample includes code objects (triggers and functions). You can view the schema and DDL, both graphically and textually, and add/edit/delete objects.

The demo database is installed with its own driver, the **sakila-dbvis.jar** file with the triggers and functions, and an open connection to the database. The driver is actually the standard H2 driver, but we keep it separate to ensure that the version is compatible with the demo database, and to avoid conflicts with any other H2 connections.

If you open the **References** tab on **Tables** node, you get a graphical view of the schema:




## Samples

In addition to the database, you also get a number of scripts in the **Files** tab under the **Bookmarks/DbVisualizer Demo Scripts** folder; most of them are based on the demo database and were used to create the screenshots and examples in the User Guide.

The **Sakila** folder holds a few variants of the Sakila database, ported to a few different database engines; you should be able to install them on your own database.

**Disclaimer:** Please note that the Sakila scripts were downloaded from various open source repositories and provided for your convenience; only the **H2** scripts are developed and supported by the DbVisualizer team.

## Sources

Finally, in the **.dbvis/DEMO** folder, you find the actual database file (**sakila-dbvis.mv.db**) and the Java™ source code for the triggers and functions used in the demo database (the code that is compiled into **sakila-dbvis.jar**).

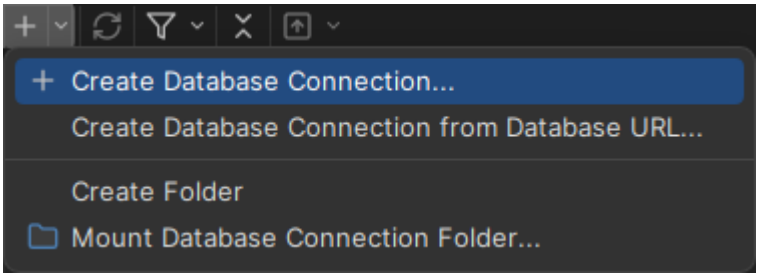
The driver used for the demo is the driver named **DbVisualizer Demo Driver** which is a h2 driver.

# Create a New Database Connection

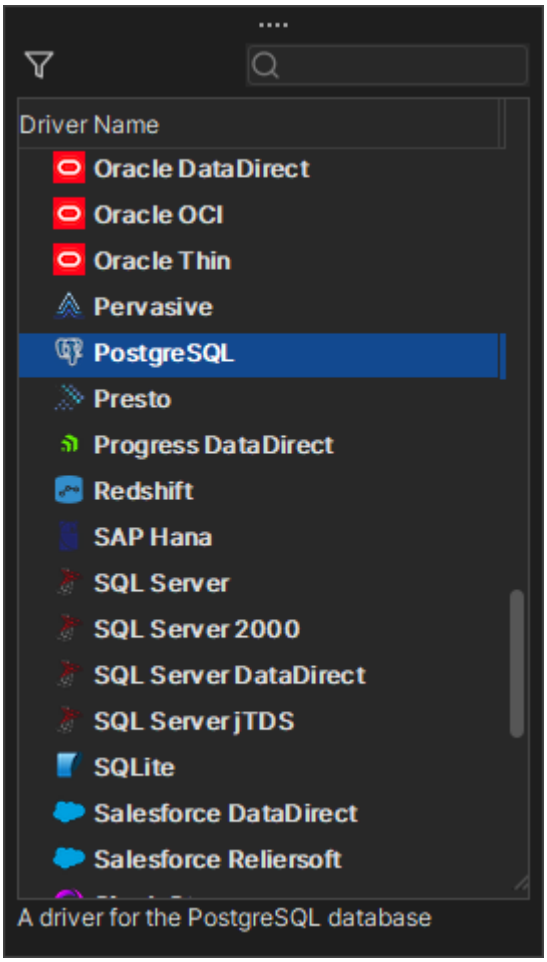
To connect to a database, you must first create and setup a Database Connection.

## Create a database connection

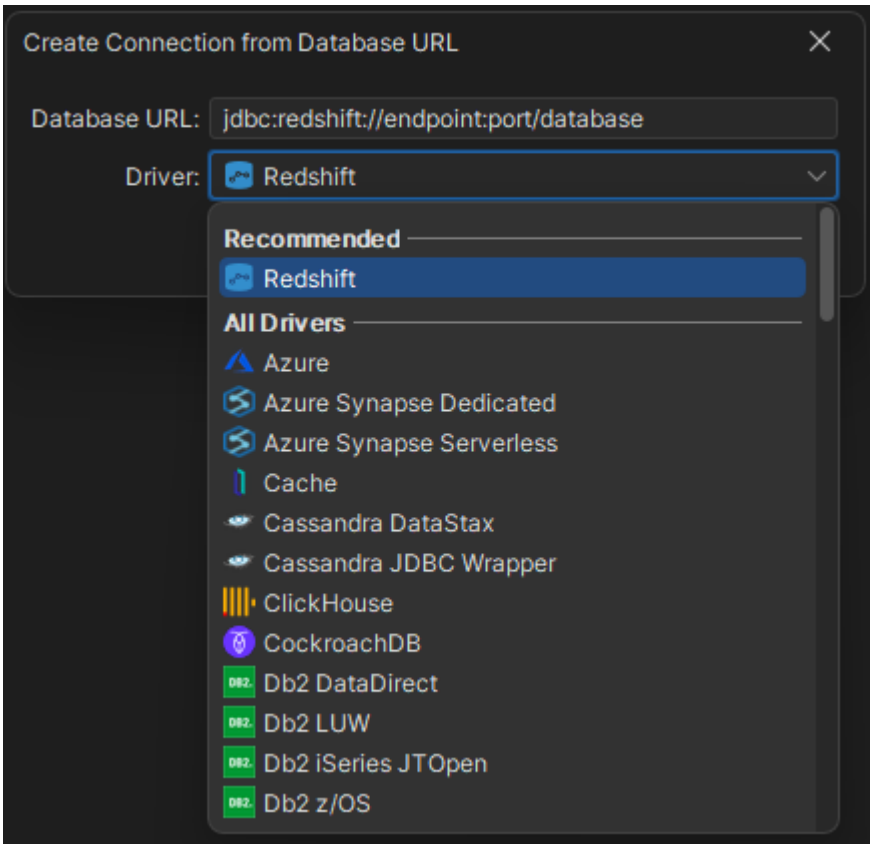
Database connections are created using the **Database → Create Database Connection** main menu or using the plus icon in the **Databases** tab toolbar.



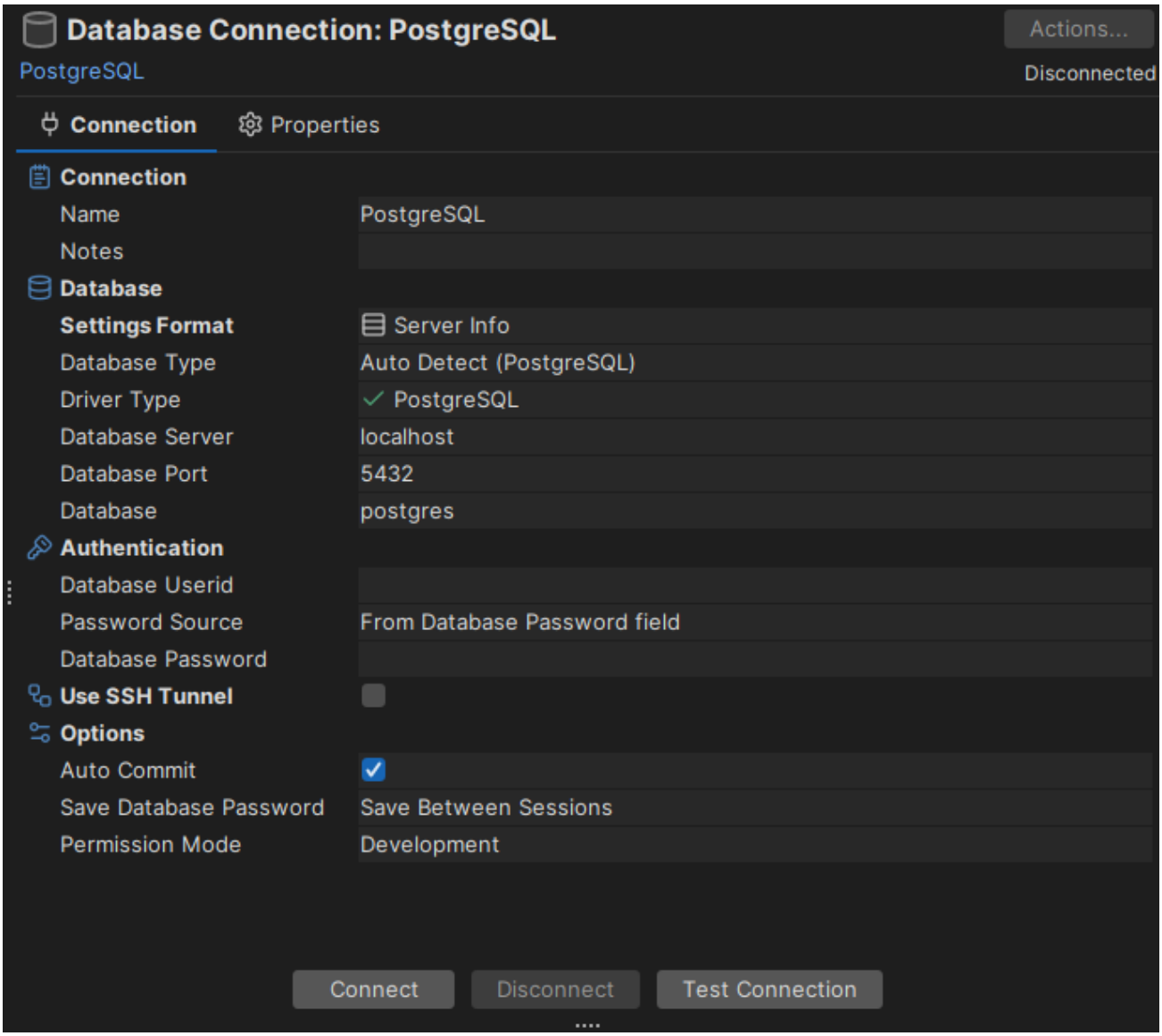

The first choice, **Create Database Connection**, opens a window in which the driver type should be selected by double-click:




The second choice, **Create Database Connection from Database URL** shows a dialog in which a JDBC URL is edited or pasted from clipboard. Based on the URL, the driver type is selected automatically:




Next step is to setup the connection details:




1. Enter a name for the connection in the **Name** field, and optionally enter a description of the connection in the **Notes** field,
2. Leave the **Database Type** as **Auto Detect,**
3. If the selected driver in **Driver Type** is marked with a green checkmark then it is ready to use. If it is not marked with a green checkmark, you may have to configure the driver in the **Driver Manager** (see Installing a JDBC Driver how to install a JDBC driver),
4. Enter information about the database server in the remaining fields,
5. Verify that a connection can be established by clicking the **Test Connection** button,
6. Click **Connect** to connect to the database server.

> See Fixing Connection Issues for some tips if you have problems connecting to the database.

Alternatively, you can set the **Settings Format** to **Database URL** (this is the only choice for some JDBC drivers). This replaces the fields for information about the database server with a single **Database URL** field, where you can enter the JDBC URL.

The information about the database server that needs to be entered depends on the which JDBC driver you use. For most drivers, you need to specify:

| Field | Description |
|---|---|
| **Database Server** | The IP address or DNS name for the server where the database runs. |
| **Database Port** | The TCP/IP port used by the database. |
| **Database Userid** | The database user account name. Enter `(null)` to not send an account name. |
| **Database Password** | The database user account password. Enter `(null)` to not send a password. |

For some database such as Oracle, you may use a TNS name instead of specifying the server and port. Other drivers may add more fields that are driver specific.

You may also optionally specify SSH tunneling information and Options, such as:

| Option | Description |
|---|---|
| **Auto Commit** | Check if you want to enable auto commit in the SQL Commander by default for the connection. |
| **Save Database Password** | Check if you want the password to be saved (encrypted) during the session, between sessions, or cleared when you disconnect. |
| **Permission Mode** | One of **Development**, **Test** or or **Production** to select which set of Permissions to use. |

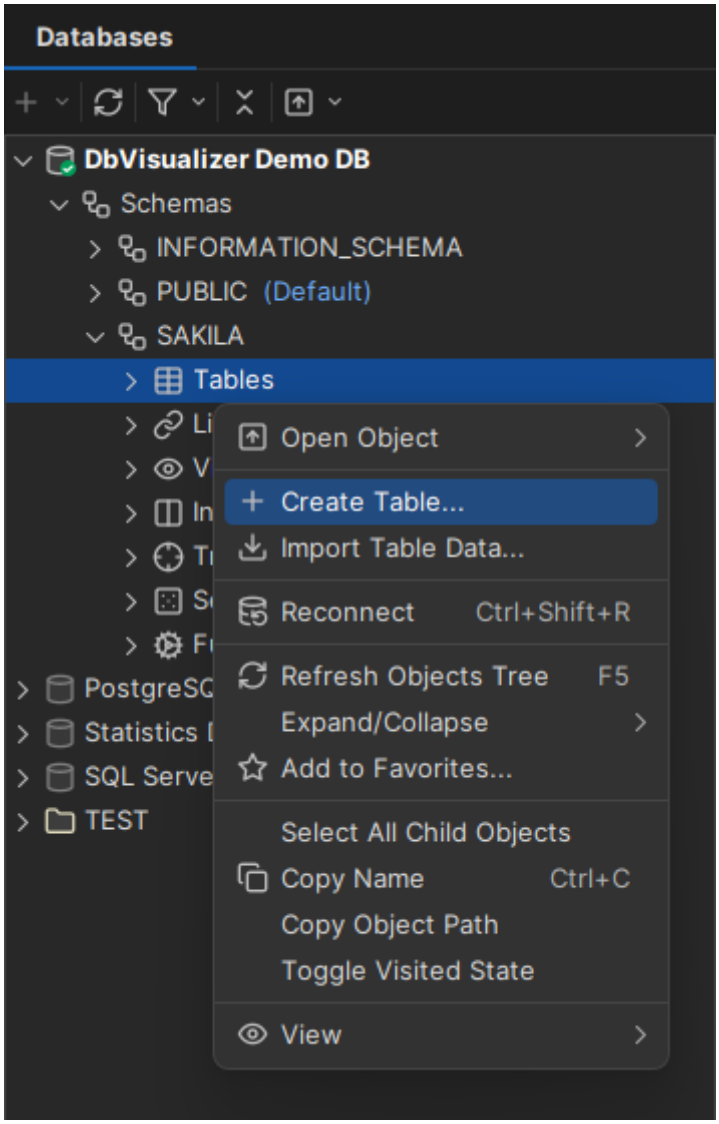
See the Configuring Connection Properties page for related topics.

# Creating a Table - basics

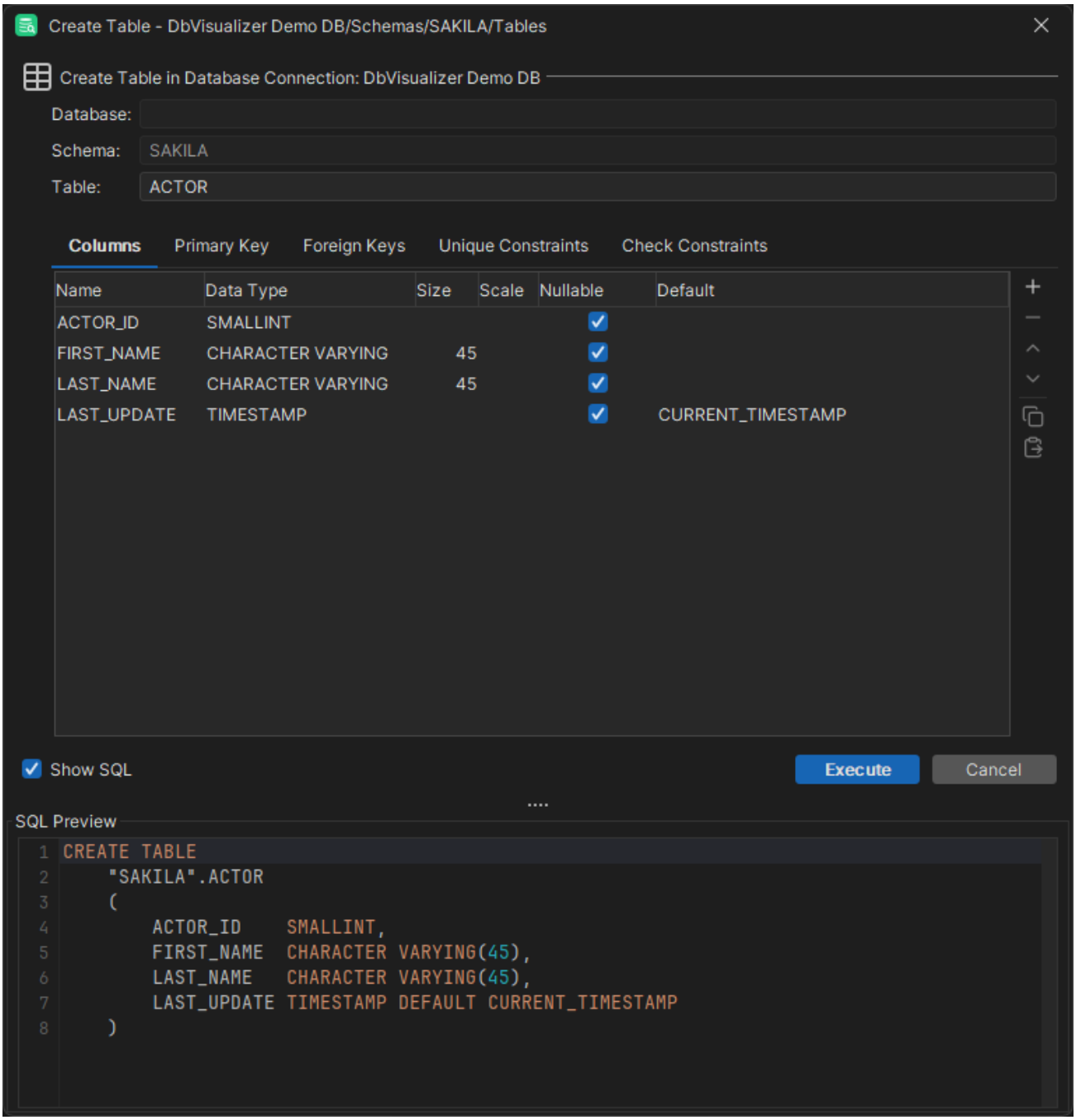
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

To create a new table:

1. Expand nodes in the **Databases** tab tree under the connection node until you reach the **Tables** node, select the **Tables** node and launch the **Create Table** dialog from the right-click menu:




2. Add columns and constraints in the different tabs, then click the **Execute** button to create the table.

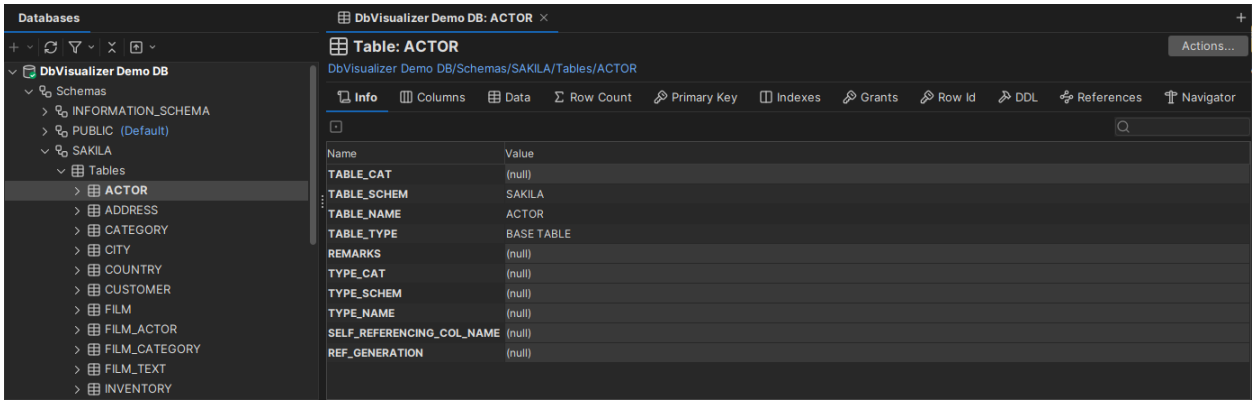



You can learn more about the Create Table dialog in the Creating a Table page.

# Viewing a Table - basics

To view details about a database table:

1. Expand nodes in the **Databases** tab tree under the connection node until you find the table,
2. Double-click on the table node to open its **Object View** tab.




The Object View has a number of sub tabs. Exactly which sub tabs are available depends on the database type, but these are common for all types:

| Subtab | Description |
|---|---|
| **Info** | Brief information about the table. |
| **Columns** | Information about all table columns, e.g. data types and sizes. |
| **Data** | Then table data. Here you can view and edit the data. |
| **Row Count** | The number of rows in the table. |
| **Primary Key** | Information about the table's primary key columns, if any. |
| **Indexes** | Information about the table's indexes, if any. |
| **Grants** | Information about granted privileges for the table. |
| **DDL** | Shows the CREATE statement for the table. |
| **References** | Shows declared primary/foreign key relationships to other tables. Please read more in Viewing Table Relationships. |
| **Navigator** | Navigate through the declared relationships. Please read more in Navigating Table Relationships. |

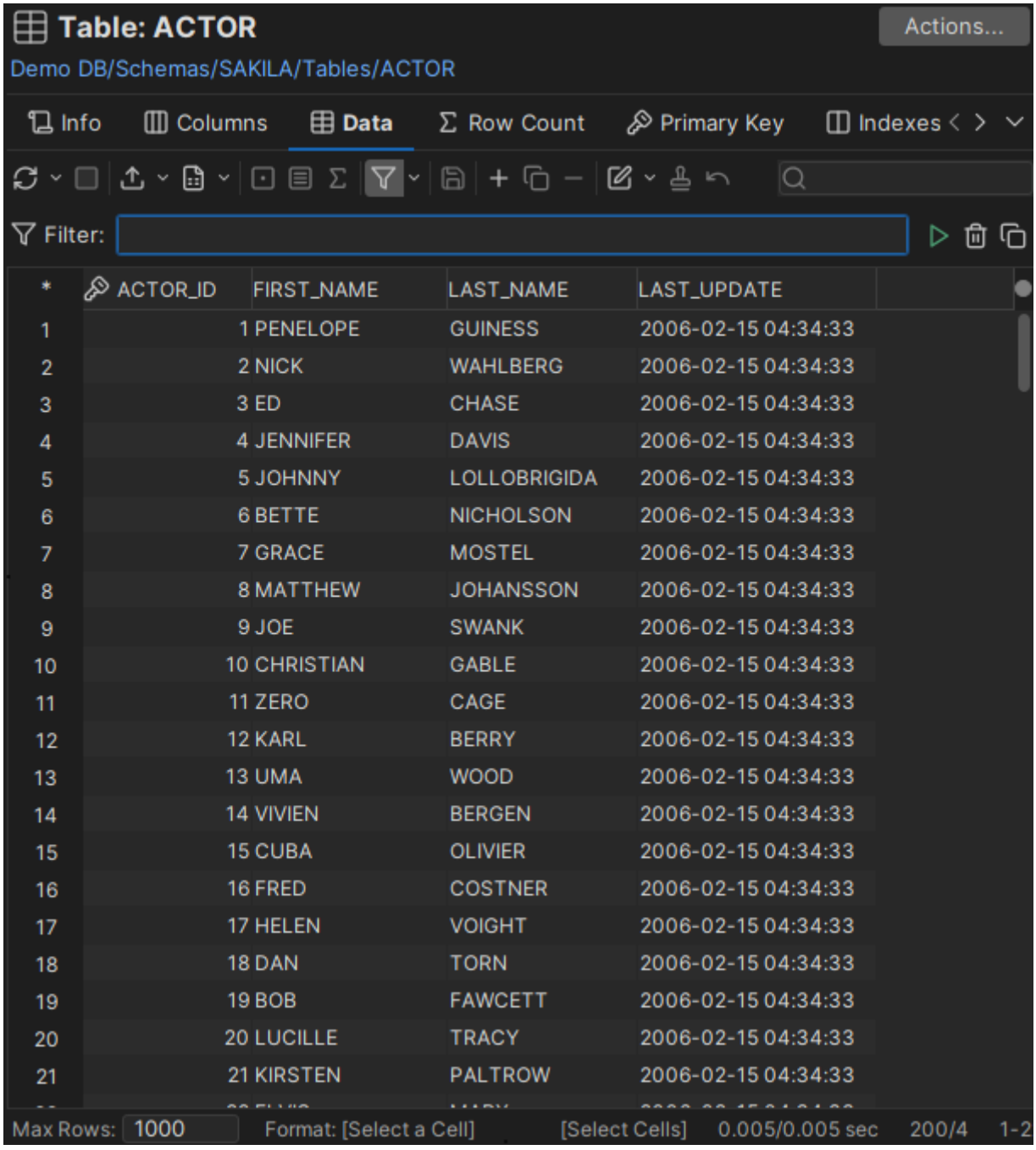
# Editing a Table - basics

**Only in DbVisualizer Pro**

This feature is only available in the DbVisualizer Pro edition.

To edit table data:

1. Expand nodes in the **Databases** tab tree under the connection node until you find the table, Double-click on the table node to open its **Object View** tab, Open the **Data** sub-tab




2. Edit column values directly in the grid, add and remove rows by clicking on the buttons in the toolbar,
3. Save the edited data by clicking on the **Save** button in the toolbar.

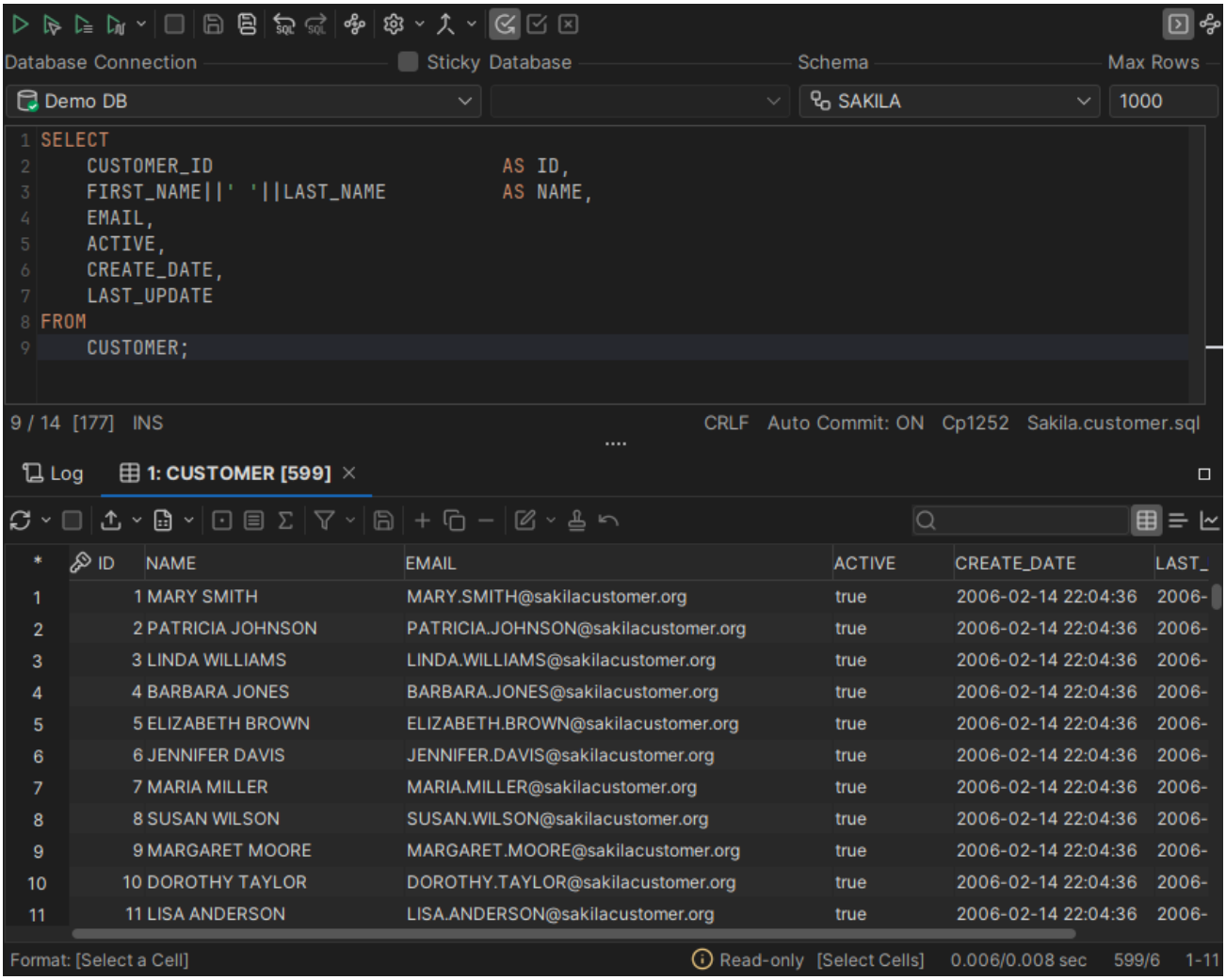
> Note that if the table does not have any declared Primary Key, you will be prompted to select the column(s) that uniquely identify a row.

You can learn more about the editing features in Editing Table Data.

# Executing SQL - basics

To execute SQL statements:

1. Open an SQL Commander window from **SQL Commander → New SQL Commander** or by clicking the **Plus** button in the main toolbar.




1. Select the database connection, catalog and schema to use.
2. Enter the SQL statements in the editor area.
3. Execute the statements by clicking the **Execute** button in the toolbar or choosing **SQL Commander → Execute**.
4. The execution log and possible result sets are shown as tabs in the results area below the editor.

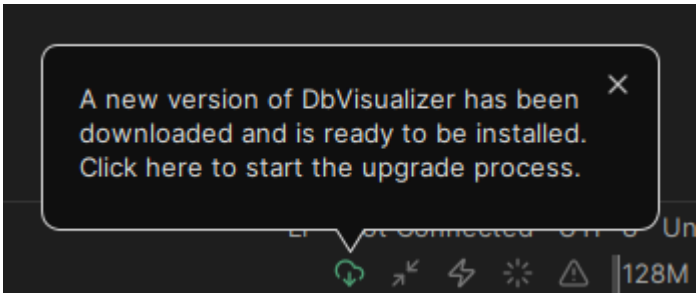
You can learn more about editing, saving and executing SQL statements in the Working with SQL section.

# Updating DbVisualizer

To make sure you have the latest functions, database support and security updates, we recommend staying up to date with the official releases and updating your installation of DbVisualizer regularly.

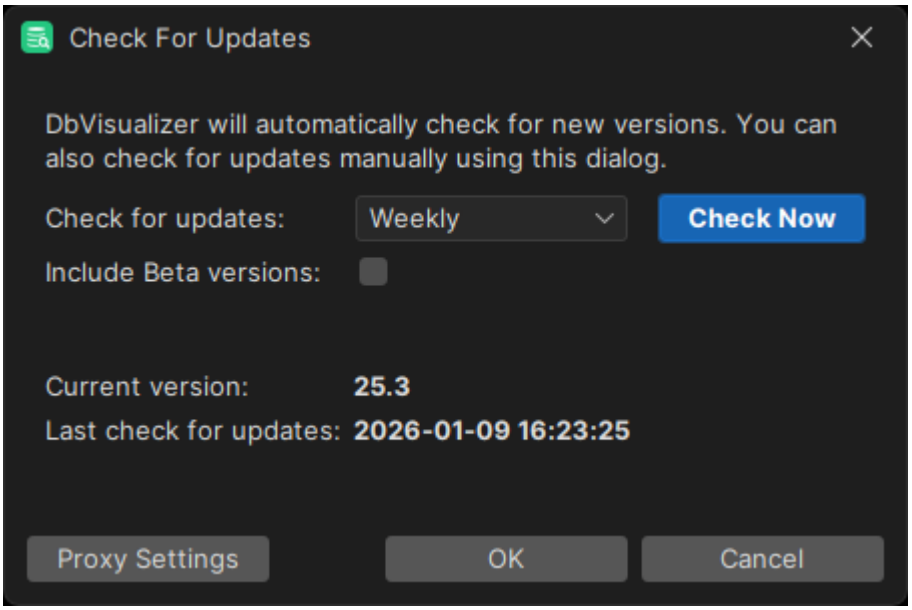
## Automatic Updates

By default, DbVisualizer checks for available updates on a weekly basis. When an update is available, it will be downloaded automatically and a notification will be displayed in the status bar. Click on the notification to see the details and initiate the upgrade.




## Checking for Updates Manually

If you want to check for updates at any given time, open **Help → Check for Updates** and click the **Check Now** button to see if there is a new version available right now.




If an update is available, a dialog is displayed from where you can install the new version, read release notes or, in case your license has expired, open the purchase page in your web browser.

## Beta Releases

If you are interested in getting information about available **Beta** versions to get early access to upcoming applications features and help us fine-tune new functionality, check the corresponding option in the **Check for Updates** dialog.

## Configuring Check for Updates

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

### Setting a Custom Update Interval

To choose a custom update interval, open **Help → Check for Updates** and select one of **Every Start-Up**, **Daily**, **Weekly**, **Monthly,** or **Never**.

## Disabling Updates

In some organizations, decisions about software updates and installation are handled centrally and individual users are not allowed to upgrade or install software. Checking for updates can be disabled by adding the following row in DBVIS-HOME/resources/dbvis-custom.prefs:

```
dbvis.url.clientapi.checkforupdate=
```

# Printing

DbVisualizer supports printing of graphs and charts. To launch the print dialog, choose **Print** from the right-click menu. This menu also contains the **Print Preview** option, which is useful if you want to see what the printout will look like before you actually print.

## Printer setup

If you want to set the page orientation (e.g., portrait or landscape) and paper size, you must launch the Printer Setup dialog, using the **File → Printer Setup** main menu option, before you print. Printing varies widely between platforms, so even though the Print dialog (as opposed to the Printer Setup dialog) on some platforms also lets you choose a page orientation and other options, they may be ignored if specified in that dialog. The only supported way to specify the page orientation and other options is via the Printer Setup dialog.

## Printing a chart

For a chart, DbVisualizer launches the platform's native Print dialog, so it looks different on different platforms. The two options available on all platforms are a choice of printer and the page range. On some platforms, the dialog may offer additional options, but they may be ignored by DbVisualizer. Use the Printer Setup dialog to set other options besides which printer to use and the page range, as described above.

## Printing a graph

Printing a graph adds a custom dialog before the native Print dialog is displayed. You can specify the number of rows (pages) and columns (pages) that the complete image will be split into. You can also select whether the view as it appears on the screen or the complete graph should be printed. If you run a dark theme, you are presented with an option to **Use Light Theme** (good for printing on white paper).

When you click OK, the native Print dialog is displayed, where you can select the printer.

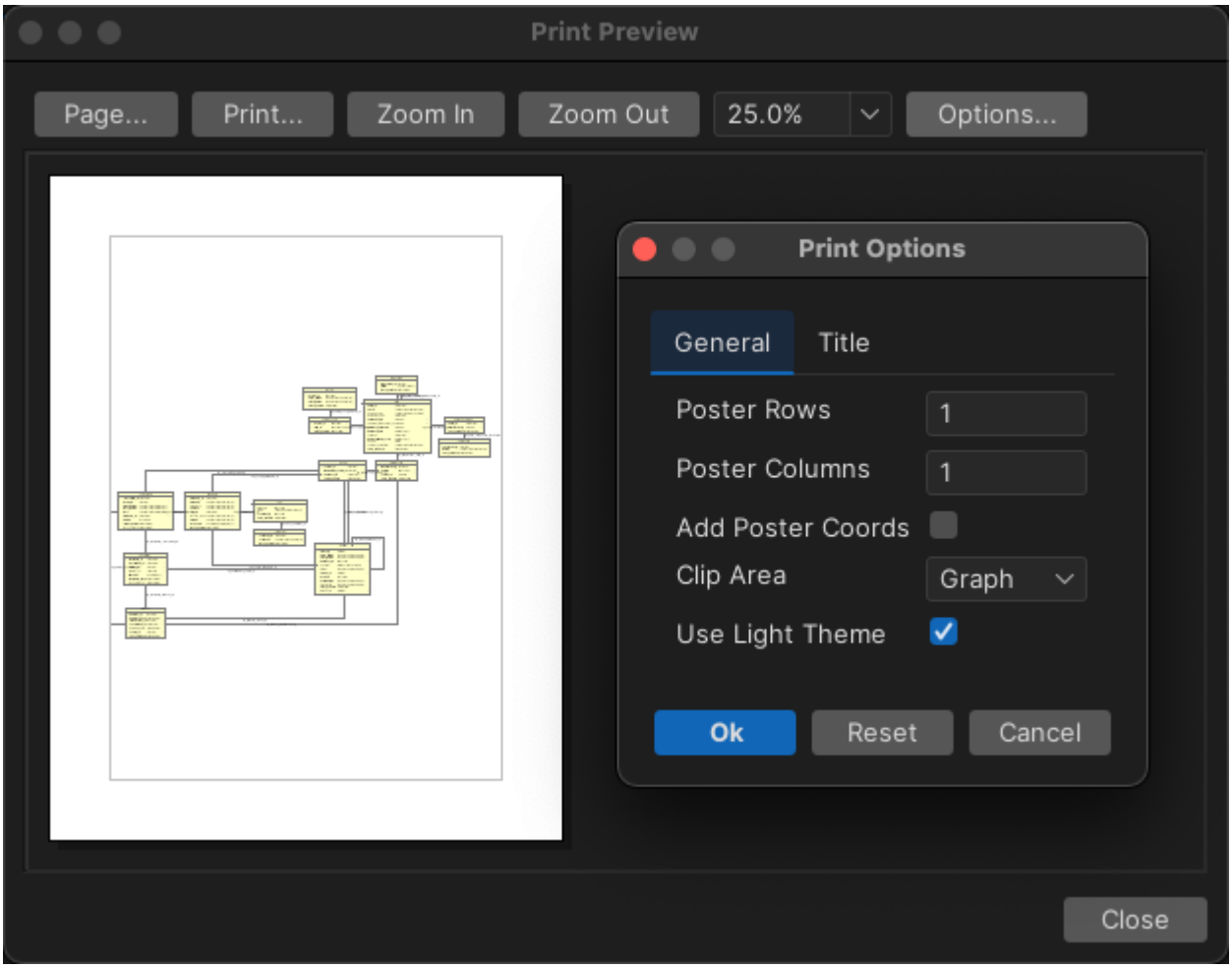
Printing a chart scales the chart to the size of the paper.

## Printing a grid

Printing a grid is not supported directly in DbVisualizer. A convenient way to do this is to use **Open in Web Browser** or **Open Selection in Web Browser**, and then use the web browser to print the result.

## Print preview

Right-click a graph and choose **Print → Print Preview** feature to preview what the printout will look like before you actually print it.


Print Preview
Page...
Print...
Zoom In
Zoom Out
25.0%
Options...
Print Options
General
Title
Poster Rows
1
Poster Columns
1
Add Poster Coords
Clip Area
Graph
Use Light Theme
Ok
Reset
Cancel
Close

# Getting the Most Out of the GUI

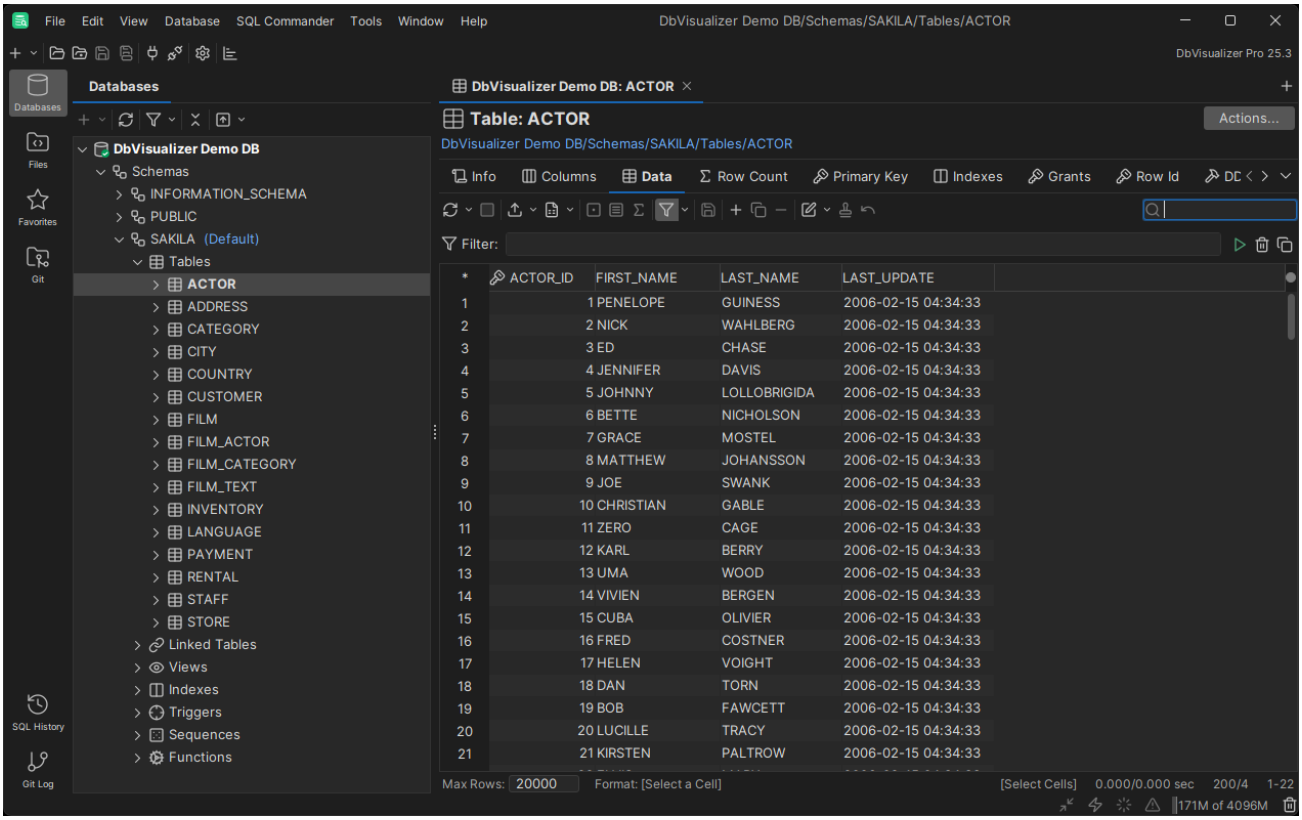
## Getting the Most Out of the GUI

DbVisualizer has a tab-based user interface that gives you a lot of control over the layout and how to work with your database objects. This section describes how you can open as many tabs as you need, arrange them to focus on what is important to you, and more.

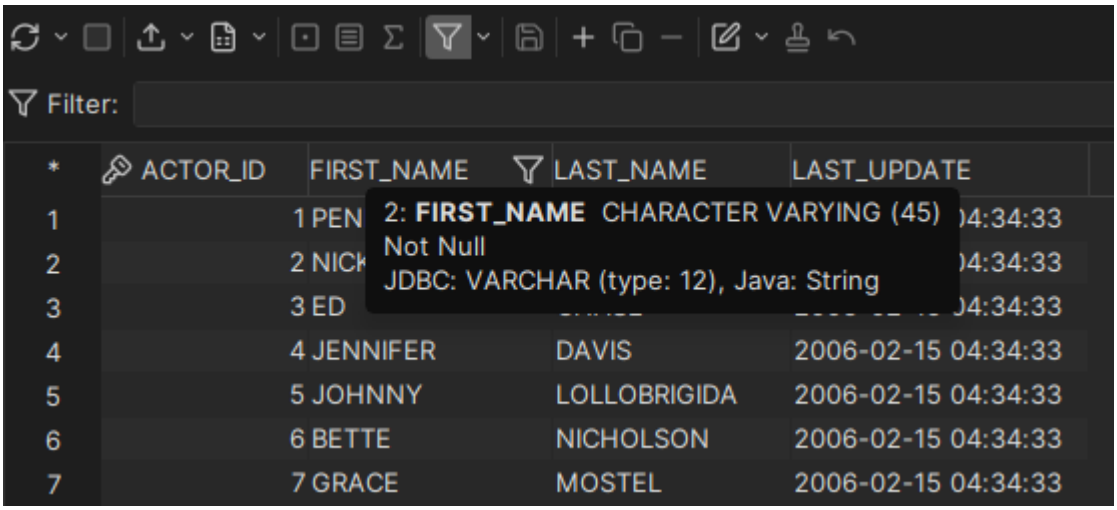

- Main Window Layout
- Tab Types
- Opening a Tab
- Pinning a Tab
- Closing a Tab
- Listing Open Tabs
- Maximizing and Minimizing a Tab
- Floating a Tab
- Rearranging Tabs
- Changing the Tab Label
- Selecting a Node for a Tab
- Preserving Tabs Between Sessions
- Using Tab Colors and Borders
- Changing the GUI Appearance
- Changing Keyboard Shortcuts
- Internationalization and Localization (i18N and L10N)
- Working with Files

# Main Window Layout

The main window in DbVisualizer contains a navigation area to the left and a work area to the right. The sidebar is used to switch between different navigation views, and objects such as database tables and scripts can be opened in the work area from the corresponding navigation tree.




Tooltips are used to provide more details about a component throughout the GUI. They are also used to express status information. An example is the grid column header tooltip that shows information about the column. To see a tooltip, let the mouse hover over an area of the user interface, e.g., a button or grid header. If there is a tooltip for the area, it will pop up in about a second.

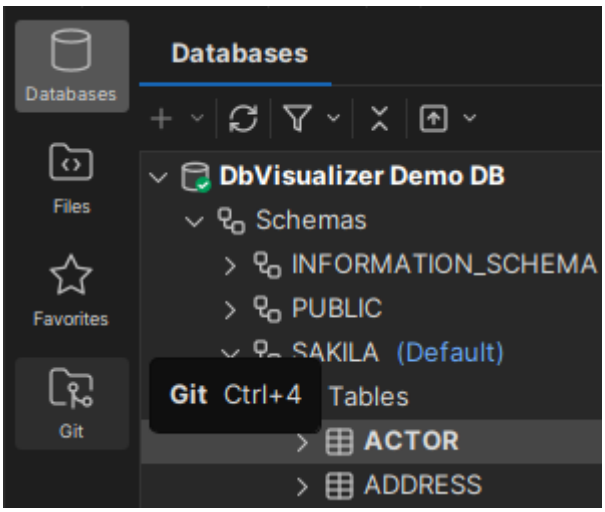



## Navigating between views

Using the sidebar, you can switch between views in the navigation area. The available views are Databases, Files, Favorites, and Git.

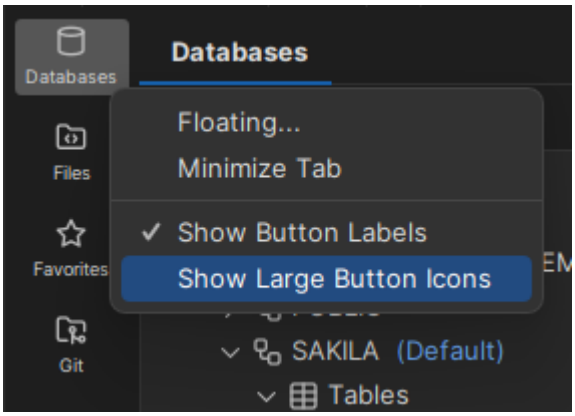
In the bottom part of the sidebar, there are also switches for SQL History and Git Log. These views are displayed at the bottom of the main window, and can be used together with the navigation and work areas.

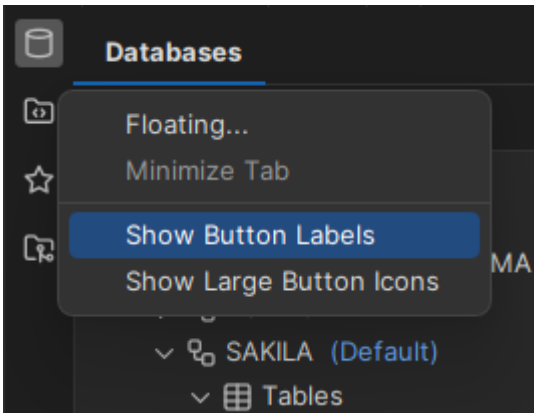
You can also toggle these views from the **View** menu or using keyboard shortcuts. To adjust the keyboard shortcuts, please see Changing Keyboard Shortcuts.



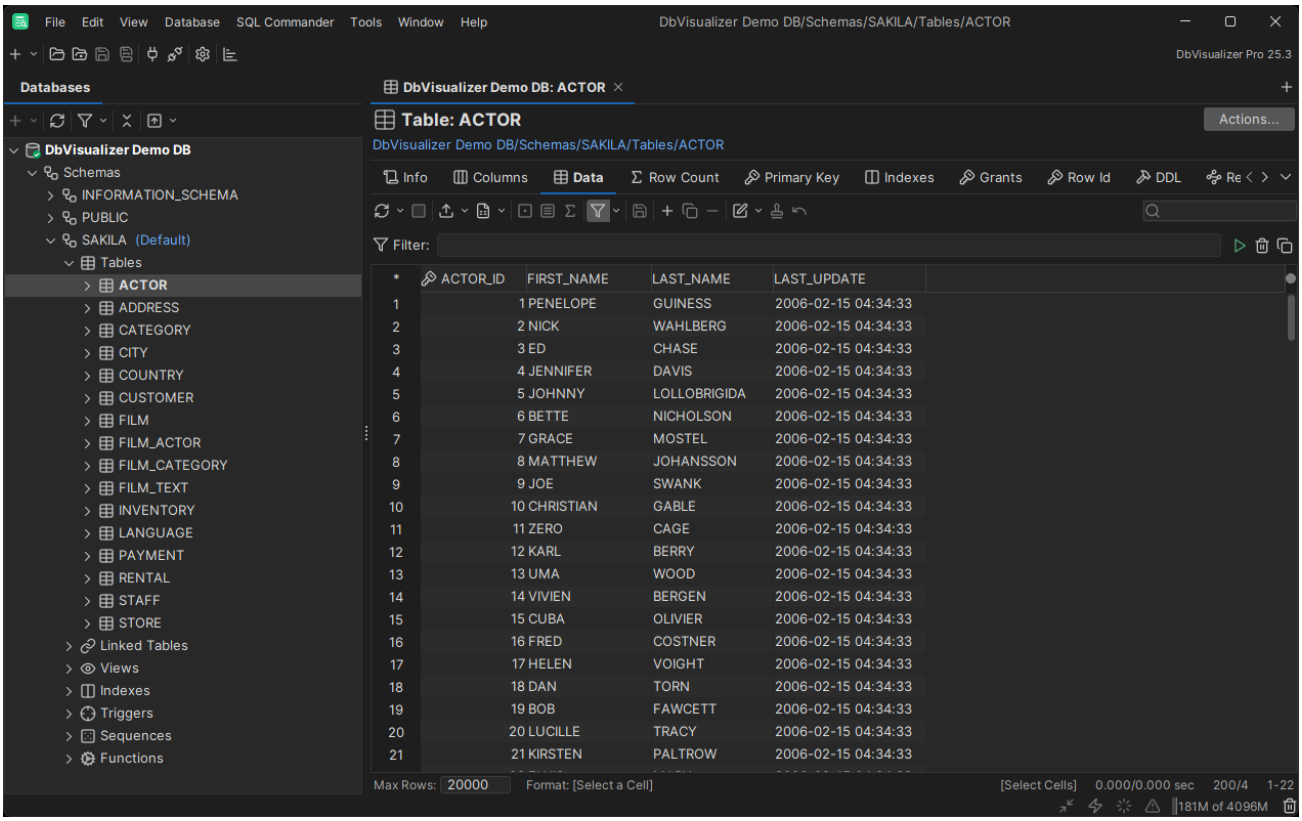

## Sidebar options

The sidebar can be configured to use large or small icons and to include or exclude text labels. This can be done using the sidebar context menu as well as under **Appearance** in **Tools → Tool Properties**.



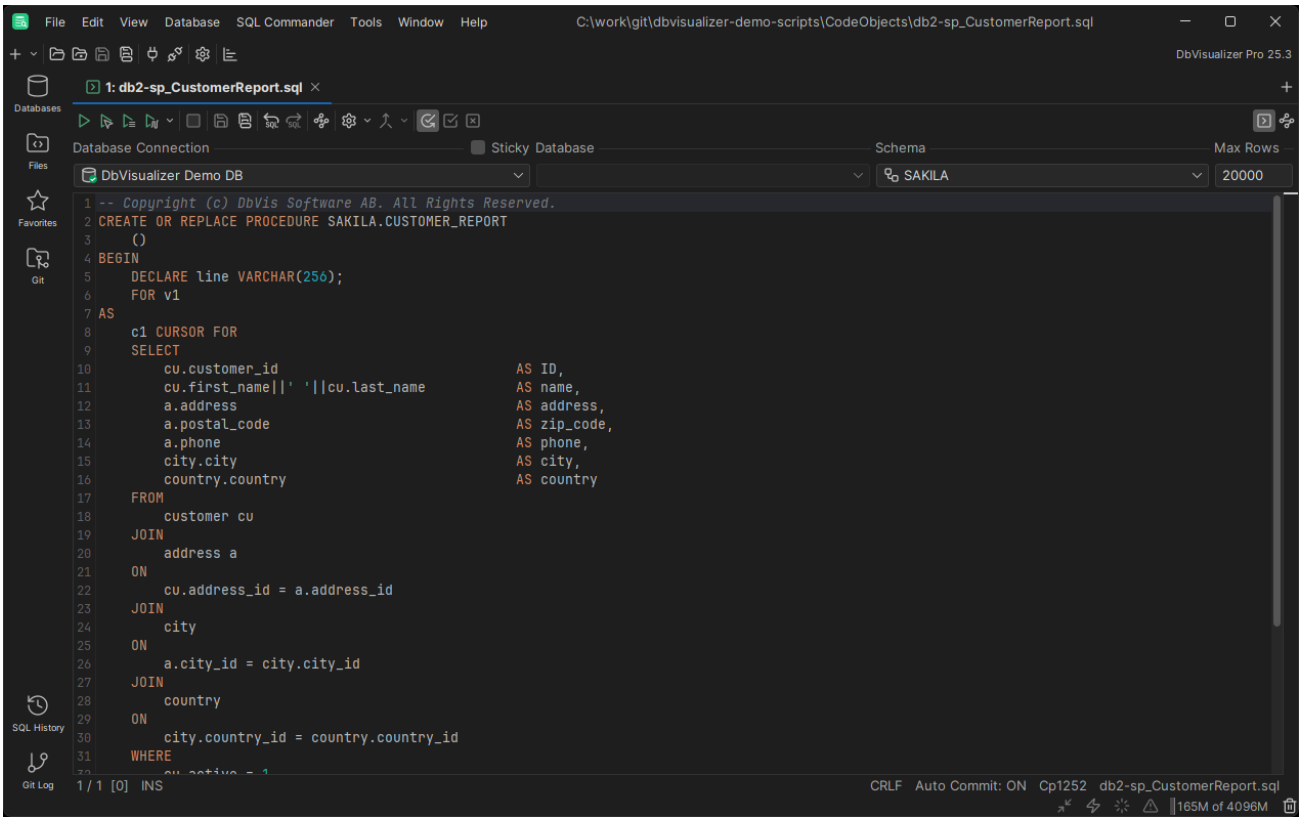

If you prefer to use the keyboard to switch between views and want to hive the sidebar altogether, you can do so using the **View** menu.




It is also possible to revert to the original DbVisualizer layout where views were presented as tabs and windows. To do so, see the **Appearance** tab in **Tools → Tool Properties**.

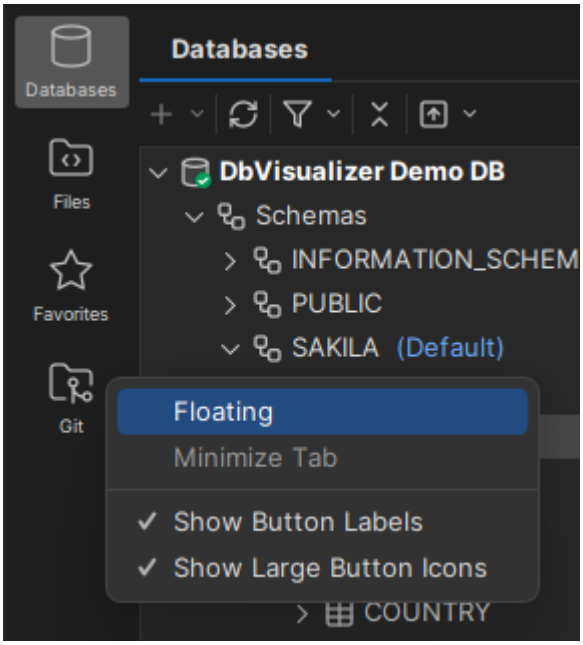
## Maximizing the work area

Sidebar buttons can also be used to close the current navigation view, thus maximizing the work area. This is particularly useful when you want to focus on a script you're working on.




## Undocking a sidebar panel

To move a sidebar panel to a separate window, right-click on the corresponding sidebar button and click the **Floating** option.

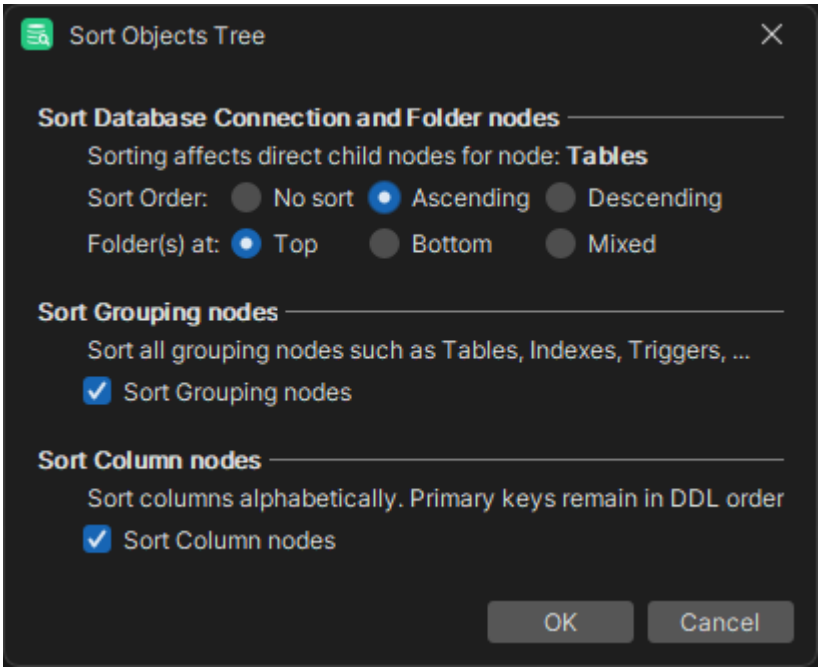


# Tab Types

There are three main types of tabs in DbVisualizer: navigation tabs, object view tabs and SQL Commander tabs.

## Navigation Tabs

The left part of the DbVisualizer window holds a navigation area with three tabs: **Databases**, **Files** and **Favorites**. They all contain an object tree where you can select the objects you want to work with and optionally filter and/or sort the nodes (right-click any node and select **Sort ...**).



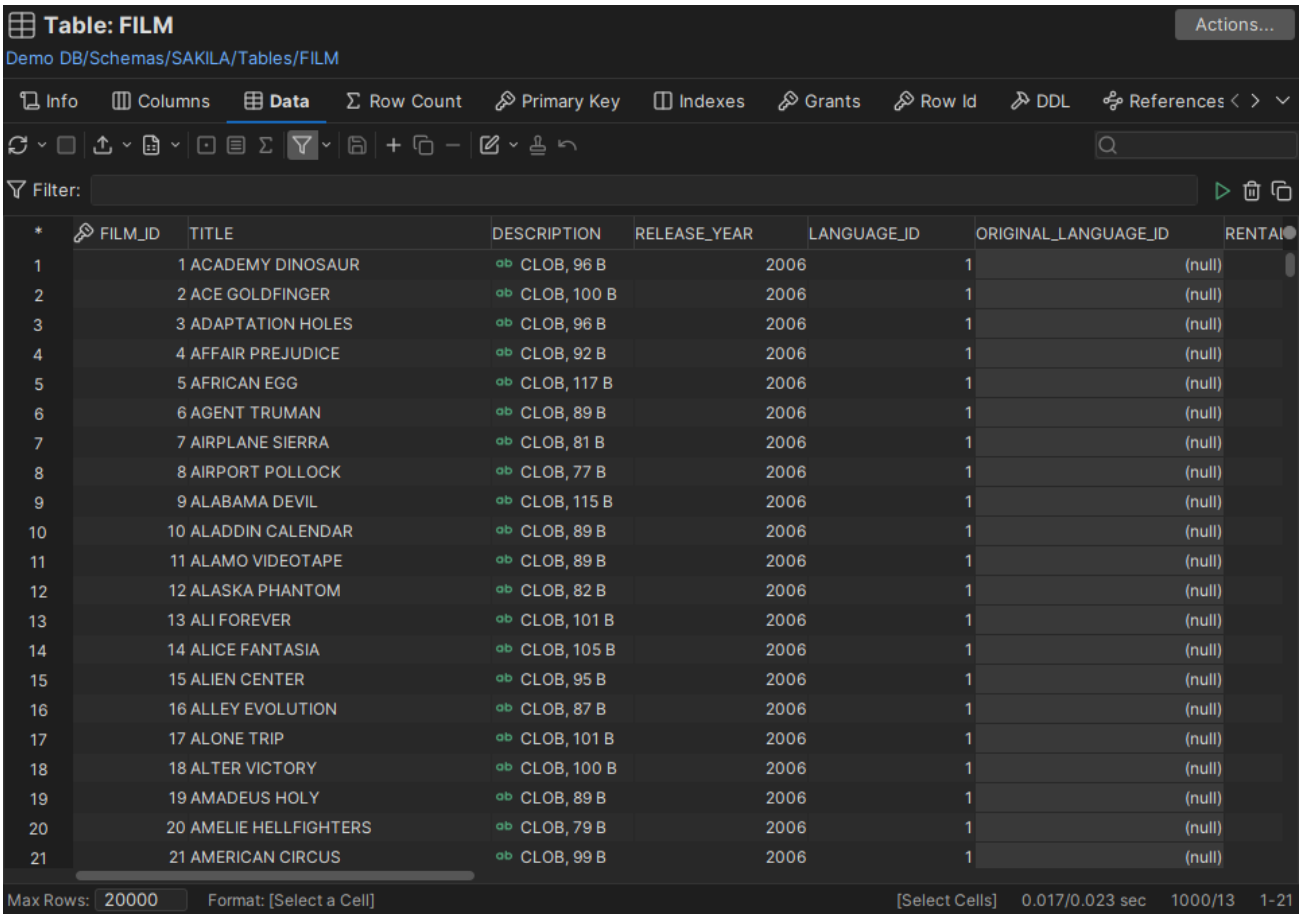

The **Databases** tab tree contains your database connections at the top and, when connected, the database objects they contain. If you have many database connections, you can also create folder objects in this tree to organize them.

The **Files** tab tree contains mounted script directories, see Using the Files Tab for details.

Finally, the **Favorites** tab tree contains objects that you want to have easy access to, either database objects (such as tables, views, or procedures) or scripts. For more details, see the Favorites page.

## Object View Tabs

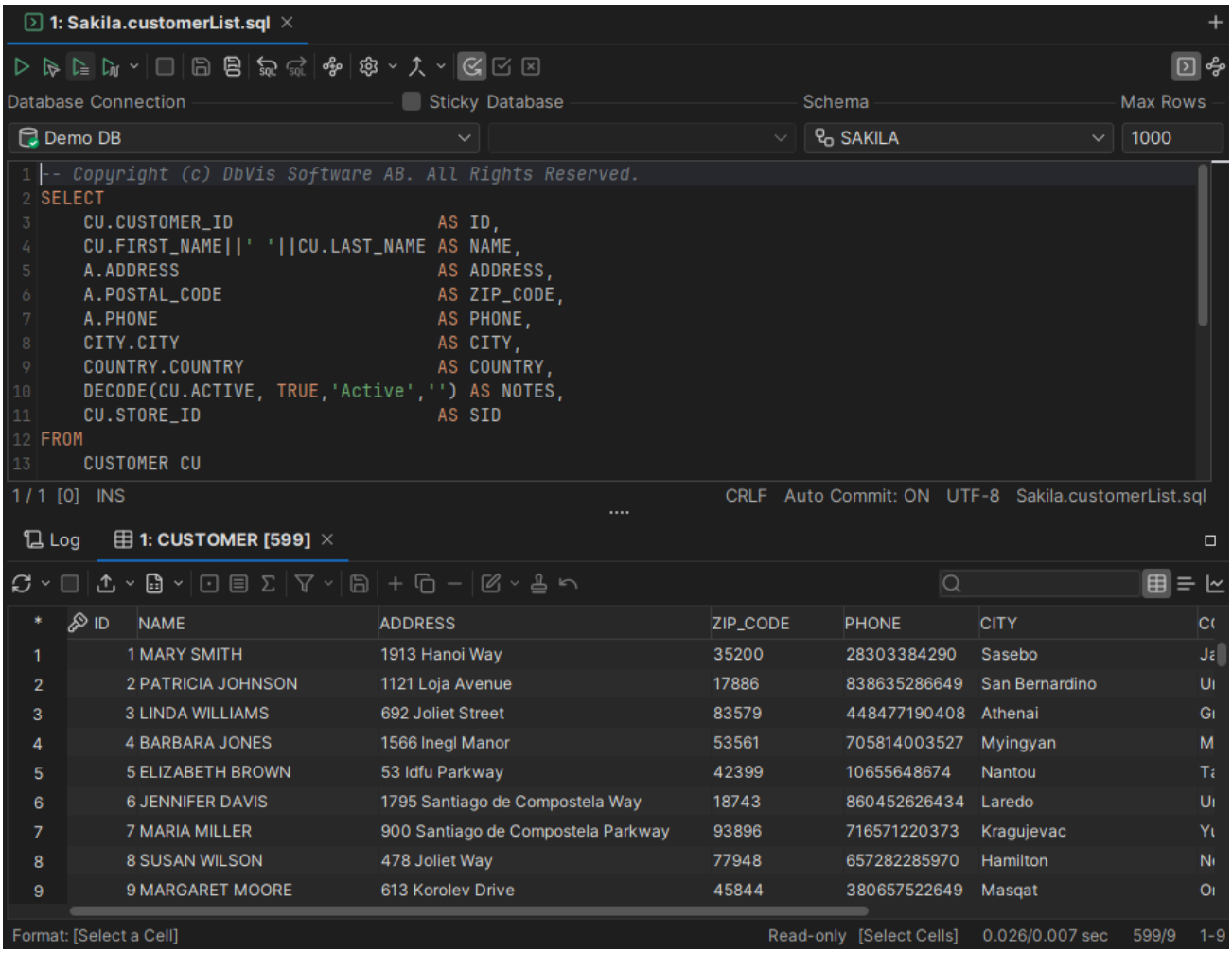
An **Object View** tab shows information about a database object, such as the data and DDL for a table, or the source code for a stored procedure. The different types of information are shown as sub tabs within the Object View tab.




| * | FILM_ID | TITLE | DESCRIPTION | RELEASE_YEAR | LANGUAGE_ID | ORIGINAL_LANGUAGE_ID | RENTAL |
|---|---|---|---|---|---|---|---|
| 1 | 1 | ACADEMY DINOSAUR | CLOB, 96 B | 2006 | 1 | (null) | |
| 2 | 2 | ACE GOLDFINGER | CLOB, 100 B | 2006 | 1 | (null) | |
| 3 | 3 | ADAPTATION HOLES | CLOB, 96 B | 2006 | 1 | (null) | |
| 4 | 4 | AFFAIR PREJUDICE | CLOB, 92 B | 2006 | 1 | (null) | |
| 5 | 5 | AFRICAN EGG | CLOB, 117 B | 2006 | 1 | (null) | |
| 6 | 6 | AGENT TRUMAN | CLOB, 89 B | 2006 | 1 | (null) | |
| 7 | 7 | AIRPLANE SIERRA | CLOB, 81 B | 2006 | 1 | (null) | |
| 8 | 8 | AIRPORT POLLOCK | CLOB, 77 B | 2006 | 1 | (null) | |
| 9 | 9 | ALABAMA DEVIL | CLOB, 115 B | 2006 | 1 | (null) | |
| 10 | 10 | ALADDIN CALENDAR | CLOB, 89 B | 2006 | 1 | (null) | |
| 11 | 11 | ALAMO VIDEOTAPE | CLOB, 89 B | 2006 | 1 | (null) | |
| 12 | 12 | ALASKA PHANTOM | CLOB, 82 B | 2006 | 1 | (null) | |
| 13 | 13 | ALI FOREVER | CLOB, 101 B | 2006 | 1 | (null) | |
| 14 | 14 | ALICE FANTASIA | CLOB, 105 B | 2006 | 1 | (null) | |
| 15 | 15 | ALIEN CENTER | CLOB, 95 B | 2006 | 1 | (null) | |
| 16 | 16 | ALLEY EVOLUTION | CLOB, 87 B | 2006 | 1 | (null) | |
| 17 | 17 | ALONE TRIP | CLOB, 101 B | 2006 | 1 | (null) | |
| 18 | 18 | ALTER VICTORY | CLOB, 100 B | 2006 | 1 | (null) | |
| 19 | 19 | AMADEUS HOLY | CLOB, 89 B | 2006 | 1 | (null) | |
| 20 | 20 | AMELIE HELLFIGHTERS | CLOB, 79 B | 2006 | 1 | (null) | |
| 21 | 21 | AMERICAN CIRCUS | CLOB, 99 B | 2006 | 1 | (null) | |

## SQL Commander Tabs

An SQL Commander tab contains an editor for editing SQL scripts, controls for executing the script and a results area with a **Log** tab and possibly **Result Set** tabs showing results from queries, and a **DMBS Output** tab for databases such as Oracle and Db2 LUW.


1: Sakila.customerList.sql
Database Connection
Sticky Database
Schema
Max Rows
Demo DB
SAKILA
1000
-- Copyright (c) DbVis Software AB. All Rights Reserved.
SELECT
CU.CUSTOMER_ID AS ID,
CU.FIRST_NAME||' '||CU.LAST_NAME AS NAME,
A.ADDRESS AS ADDRESS,
A.POSTAL_CODE AS ZIP_CODE,
A.PHONE AS PHONE,
CITY.CITY AS CITY,
COUNTRY.COUNTRY AS COUNTRY,
DECODE(CU.ACTIVE, TRUE,'Active','') AS NOTES,
CU.STORE_ID AS SID
FROM
CUSTOMER CU
1 / 1 [0] INS
CRLF Auto Commit: ON UTF-8 Sakila.customerList.sql
Log
1: CUSTOMER [599]
ID NAME ADDRESS ZIP_CODE PHONE CITY
1 1 MARY SMITH 1913 Hanoi Way 35200 28303384290 Sasebo
2 2 PATRICIA JOHNSON 1121 Loja Avenue 17886 838635286649 San Bernardino
3 3 LINDA WILLIAMS 692 Joliet Street 83579 448477190408 Athenai
4 4 BARBARA JONES 1566 Inegl Manor 53561 705814003527 Myingyan
5 5 ELIZABETH BROWN 53 Idfu Parkway 42399 10655648674 Nantou
6 6 JENNIFER DAVIS 1795 Santiago de Compostela Way 18743 860452626434 Laredo
7 7 MARIA MILLER 900 Santiago de Compostela Parkway 93896 716571220373 Kragujevac
8 8 SUSAN WILSON 478 Joliet Way 77948 657282285970 Hamilton
9 9 MARGARET MOORE 613 Korolev Drive 45844 380657522649 Masqat
Format: [Select a Cell]
Read-only [Select Cells] 0.026/0.007 sec 599/9 1-9

# Opening a Tab

## Database tree objects

You can open an object by double-clicking on the object node, or by pressing **Enter** with the node selected, in all navigation tab trees. This opens either an Object View tab or an SQL Commander tab, depending on the object type: database object or script.

By default, a database object is opened in an Object View tab that is "available," meaning it is not pinned, busy running a task, or contains pending edits. The current tab is chosen if it is available, otherwise any other available tab is used. If none of the tabs is available, a new tab is created for the object. You can change this behavior in the **General/Tabs** category in **Tools → Tool Properties**, so that a new tab is always used instead of using an available tab.

If a tab is already open for the object, it is made the active tab. An alternative to double-clicking a database object is to use the **Open in Tab** choice from the node's right-click menu. **Open in Tab** is also available in the **Open Object** drop-down menu button in the **Databases** tab toolbar.

If you want to open a database object in a new tab instead of an available tab, hold down the **Alt** key when you double-click the node, use the **Open in New Tab** choice from the node's right-click menu or from the **Open Object** drop-down menu button in the **Databases** tab toolbar. The drop-down button keeps the last choice as the default, so once you have selected **Open in New Tab** once, you only need to click the button to do the same.

Both **Open in Tab** and **Open in New Tab** can also be used when multiple nodes are selected in the tree to open multiple tabs in one go.

To replace the content in a specific **Object View** tab with information for another object, drag the node for the new object from the object tree and drop it on the **Object View** tab header.

The behaviour when double clicking a Database object can be configured in **Tools → Tool Properties and in General / Database Objects Tree**.

## Scripts

Scripts are always opened in a new **SQL Commander** tab unless the script is already opened in a tab. If so, that tab is activated instead. An alternative to double-clicking the node is to use **Open in SQL Commander** in the right-click menu or the corresponding button in the **Files** tab toolbar. The right-click menu also holds an **Open in SQL Commander and Execute** choice.

A new, empty SQL Commander tab is created by clicking the **Create SQL Commander** button in the main toolbar or using the corresponding **File** menu item. You can use this kind of **SQL Commander** tab for ad-hoc statement execution or to create a new script.

**SQL Commander** tabs can also be opened for files in the file system. Click the **Open File** button in the main toolbar or choose **File → Open** to open the file chooser window and select one of more files. Alternatively, you can drag files from your platform's file browser and drop them in the main toolbar area.

In addition see also Synchronizing object tab selection and selection in the tree which shows how to set up DbVisualizer to automatically select the database tree node when the corresponding object view tab is selected and vice versa.

# Pinning a Tab

To prevent an Object View tab to be reused for another object or to prevent any tab to be removed unless you explicitly asks for it, you can "pin" it. In the right-click menu for a tab you find a **Pin Tab** toggle to accomplish this. A red pin icon is added to the icon when pinned.

You can also click on the icon in the tab header as a shortcut for toggling between the "pinned" and "unpinned" states.

# Closing a Tab

A top level tab and a **Result Set** tab in an **SQL Commander** tab can be closed by clicking the cross to the right of the tab label or by clicking the tab header with the middle mouse button.

## Closing multiple tabs

There are more options available if you right-click on a tab header:

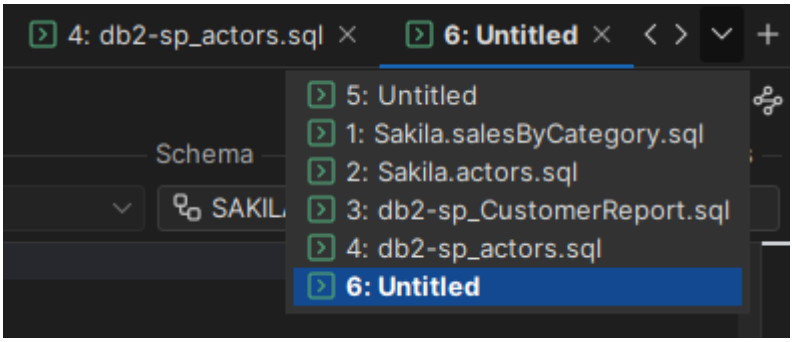
| Action | Description |
|---|---|
| **Close Tab** | Close the current tab, even if it is pinned. |
| **Close Other Tabs** | Close all unpinned tabs except the current tab. |
| **Close Tabs to the Right** | Close unpinned tabs to the right of the current tab. |
| **Close All Pinned Tabs** | Close all pinned tabs. |
| **Close All Tabs** | Close all unpinned tabs. |

## Hiding object view sub-tabs

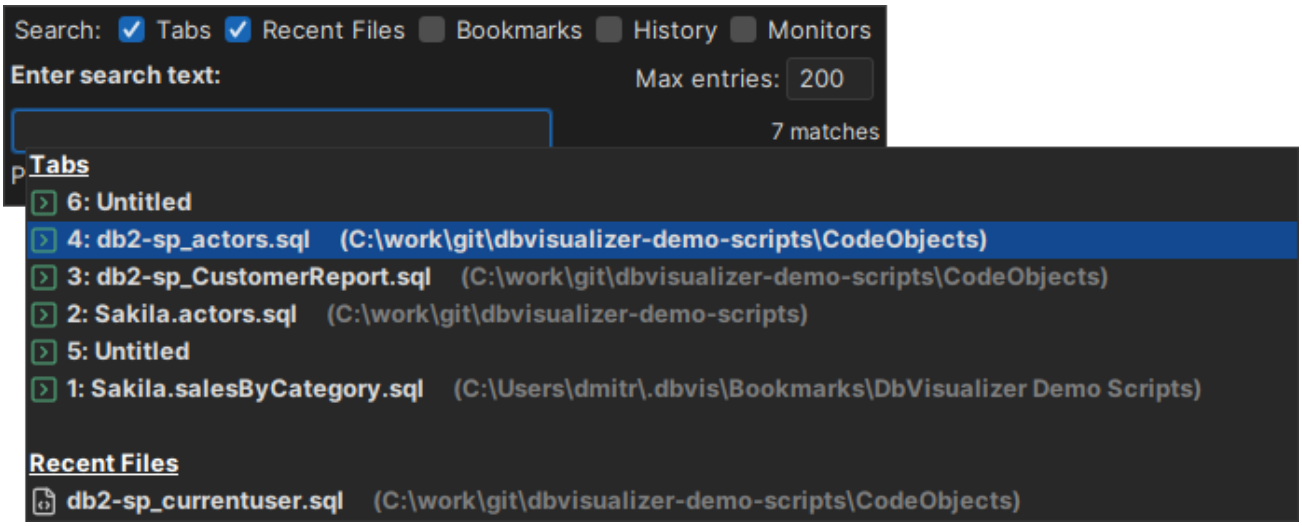
You can also hide object view sub-tabs that you are not interested in. Use the **Hide Tab** right-click menu choice to do so. To see the hidden tabs again, use **Restore Hidden Tabs** in the right-click menu.

# Listing Open Tabs

If you have many tabs opened, there may not be room enough to show them all at the same time. In this case, you can scroll through them using the left and right arrow buttons that appear to the right of the tabs. To bring up a list of all open tabs, click the down arrow icon instead.



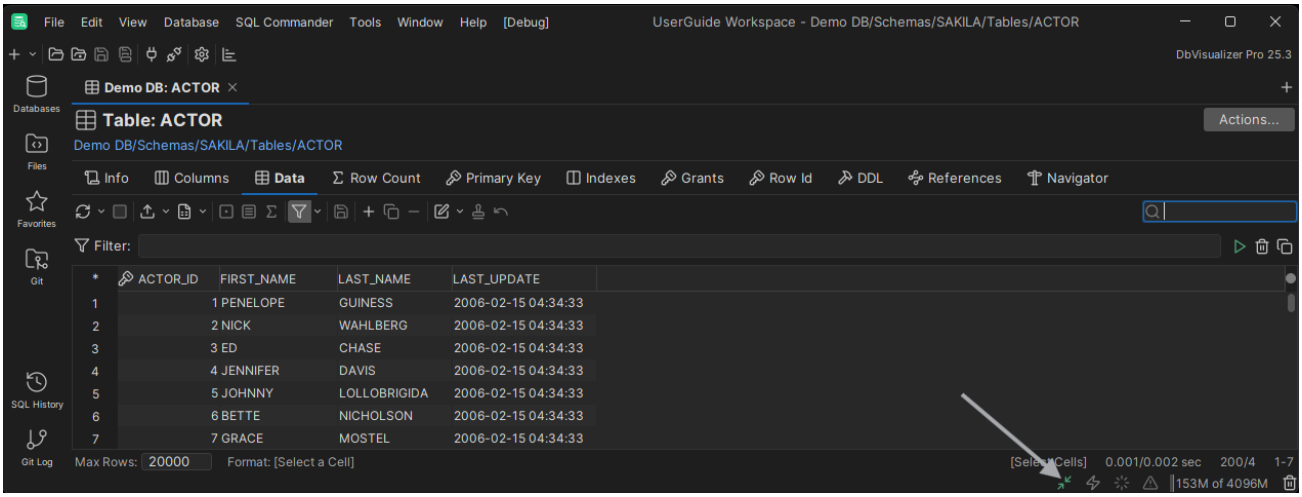

A more powerful alternative is to use the **Quick Load** feature. It is available in the main toolbar, in the File menu, or via the corresponding keyboard shortcut. Among other things, it lists the open tabs, and most recently used are displayed first. You can either select the tab directly or enter a part of the tab name to narrow the search.



# Maximizing and Minimizing a Tab

The three Navigation tabs can be minimized either by double-clicking on the tab header or using **Minimize Tab** in the tab's right-click menu. Clicking on a minimized tab brings it back to its regular place and size again.

All other tabs can be maximized by double-clicking on the tab header or using **Maximize Tab** in the tab's right-click menu. When you maximize a tab, the Navigation tabs are minimized to make as much room as possible available and the tab you maximized fills all available space. Double-clicking on the the tab again restores all tabs back to their original size. The latter can also be accomplished by deselecting the **Maximize Tab** right-click item.




An icon in the main status bar indicates when a tab is maximized. An alternative for restoring the original size of all tabs is to double-click on this icon.

# Floating a Tab

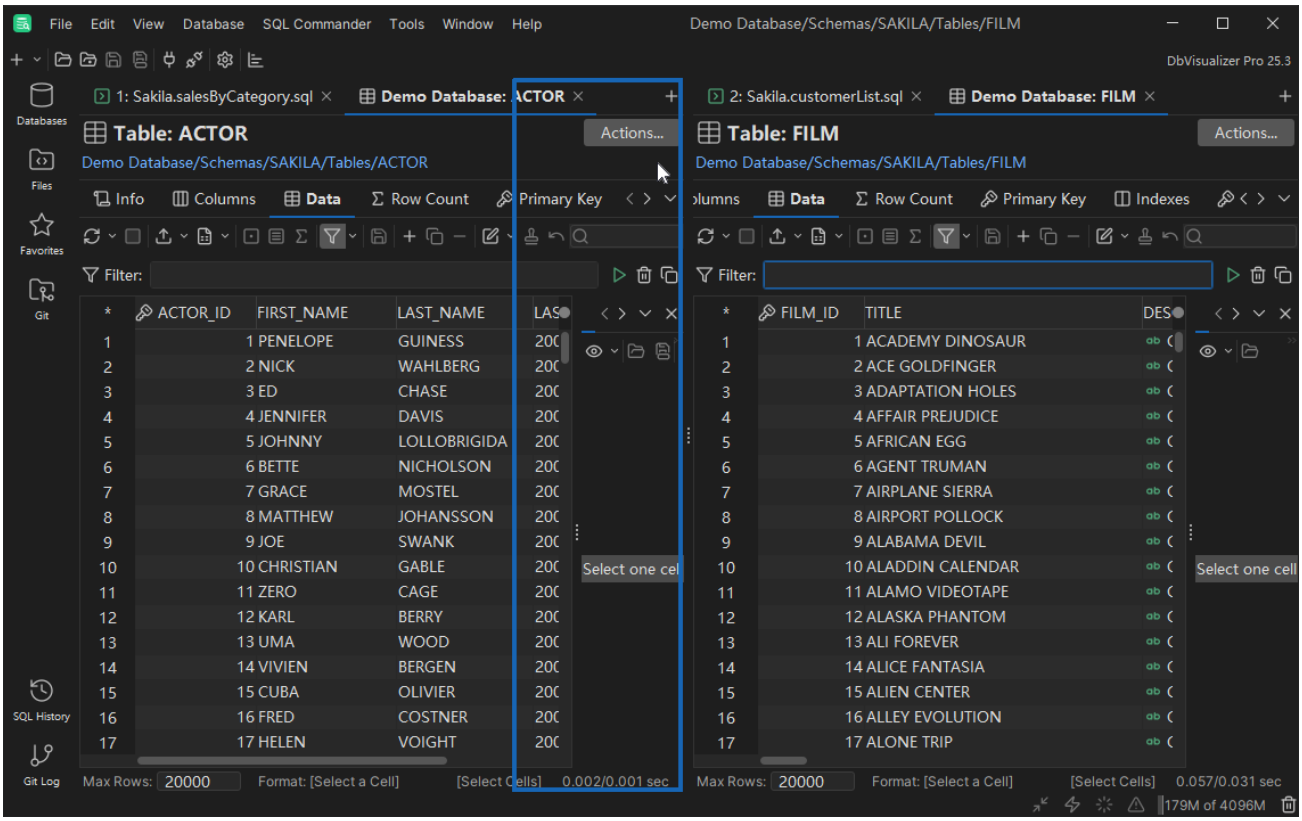
Sometimes it is handy to break up the user interface in multiple freestanding windows. Every tab in DbVisualizer can be placed in a separate window by "floating" the tab. Use the **Floating** menu choice in the tab header right-click menu to float the tab in a separate window. An alternative is to drag the tab outside of the window. To dock the tab with the original location, right-click the tab and deselect **Floating**.

Closing a top level floating window will for **Object View** and **SQL Commander** tabs ask whether to really close the tab as it is then closed permanently and will not be restored in the DbVisualizer window. Closing an Object View sub tab, will hide the tab. To restore it, right-click another sub tab and select **Restore Hidden Tabs**.

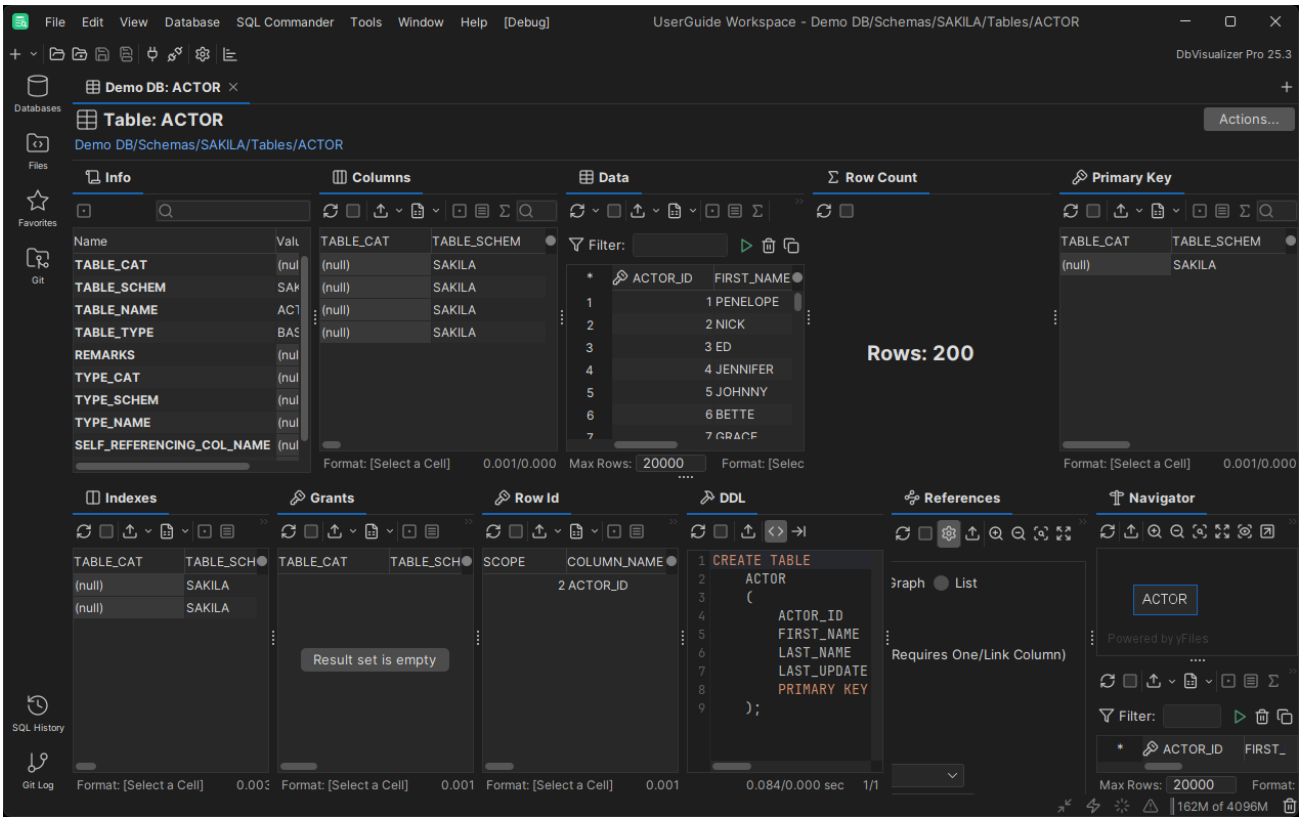
# Rearranging Tabs

You can move the tabs around by drag and drop. This allows you to see the content of multiple tabs at the same time

When you drag a tab, an outline of the tab borders shows what will happen when you drop it: place it in above, below or next to another tab, or simply move it to another location among its siblings.




The tab header right-click menu also has a couple of choices for arranging all tabs at the same level as either "tiled" (the content of all tabs visible side-by-side) or "collapsed" (only the content of the active tab visible).




You can save your rearranged layout of an Object View tab so that it is applied for all objects of the the same type for the same database type. In the sub tab header right-click menu, just select **Save as Default Layout**. To restore the default layout, use **Reset to Factory Layout**.

### Only in DbVisualizer Pro

This feature is only available in the DbVisualizer Pro edition.

# Changing the Tab Label

The tab labels are set based on a pattern that you can change in **Tools → Tool Properties**, in the General/Tabs category. The pattern can also be changed by selecting the **Modify Tab Labelling** right-click menu for **Object View**, **SQL Commander**, and **Result Set** tabs.

You can also manually change the label for a single **Object View**, **SQL Commander**, and **Result Set** tabs, using the **Rename** menu choice in the tab right-click menu.

You can select one of the predefined patterns or create your own by editing the pattern. The variables available for these patterns are:

| Variable | Available For | Description |
|---|---|---|
| **${connectionname}** | All | Connection name |
| **${index}** | All | A unique index for the tab in the tab group |
| **${userid}** | All | Userid used for the connection |
| **${filename}** | SQL Commander Tabs | Script filename, or "Untitled" if no file is loaded. The **nofile** option can be used to change "Untitled" to something else, e.g. **${filename\|nofile=No File}**. Set it to blank to only show a label when a file is loaded, i.e. **${filename\|nofile=}** |
| **${longfilename}** | SQL Commander Tabs | Absolute path for the script, or "Untitled" if no file is loaded. The **nofile** option can be used to show something else, see the **${filename}** variable for details |
| **${objectname}** | Object View Tabs | Object name |
| **${objecttype}** | Object View Tabs | Object type, e.g. Table, View etc. |
| **${catalog}** | Object View Tabs | The database (if any) for the currently open object |
| **${schema}** | Object View Tabs | The schema (if any) for the currently open object |
| **${editorindex}** | Result Set Tabs | The index for the parent SQL Commander tab |
| **${rows}** | Result Set Tabs | Number of rows in the result set |
| **${sql}** | Result Set Tabs | Part of the SQL statement that produced the result set |
| **${table}** | Result Set Tabs | Name of the table (first if more than one) the result set comes from |
| **${time}** | Result Set Tabs | Time when the result set was produced |
| **${vendor}** | Result Set Tabs | Database vendor name |

# Selecting a Node for a Tab

To quickly navigate to and select the node in the **Databases** tab tree that an **Object View** tab belongs to, you can click on the object path in the Object View header area. Alternatively, you can use **Select in Databases Tab** in the tab right-click menu.

Similarly, you can navigate to the object represented by a Favorite object using its **Select Target Object** right-click menu choice. It selects the object in the **Databases** or **Files** tab, depending on the Favorite type.

For an **SQL Commander** tab that is associated with a connection, you can use **Select in Databases Tab** in the tab right-click menu to select the connection node.

It is also possible to automatically select the Node whenever its object view tab is clicked. This is done in **Tools → Tool Properties** under the **General / Database Objects Tree** category.

An alternative is by selecting the **Autoscroll FROM Object View tab** in the **Open selected object(s)** drop down in the toolbar of the **Databases** tab tree.

# Preserving Tabs Open Between Sessions

If you often work with the same objects and a few scripts, you can ensure that the Object View and SQL Commander tabs for these objects remain open between DbVisualizer sessions.

1. Open **Tools → Tool Properties**,
2. Select the **Tabs** category,
3. Enable one or both of **Preserve SQL Commander tabs between Sessions** and **Preserve Object View tabs between Sessions**,
4. Click **Apply** or **OK** to apply the new settings.

This feature is enabled by default for SQL Commander tabs but not for Object View tabs.

The content of the SQL Commander tabs is saved at regular intervals so when you restart DbVisualizer, the content is the same as where you left off.

For Object View tabs, you can also enable **Preserve Object View tabs at Disconnect**. By default, Object View tabs for objects that belong to a connection are closed when it is disconnected.

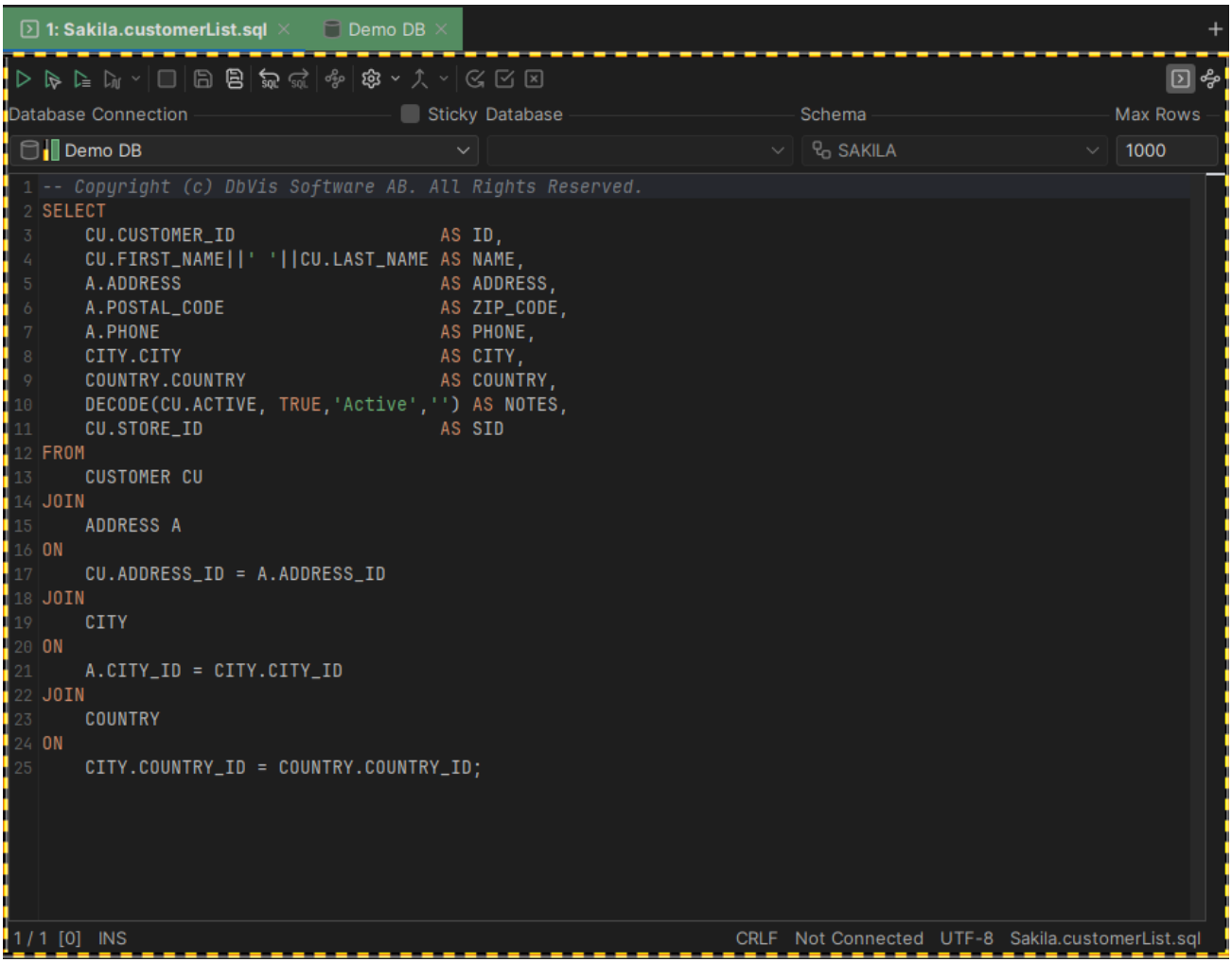
# Using Tab Colors and Borders

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

If you like to distinguish tabs and other GUI components for a specific connection from those of others, you can specify a tab background color and/or border to use for a connection. If you set these in the **Tools → Tool Properties** under the **Database / <database type> / Color and Border** category, they apply to all connections with the corresponding database type. Therefore these properties are typically set in the **Properties** tab for a specific connection instead.

For the border, you can either select one of the predefined styles or specify a small image file to use for the border.

The selected color and border are also shown in the connection tab node in the Databases tab and in the Database Connection list in the SQL Commander.


1: Sakila.customerList.sql
Demo DB
Database Connection
Sticky Database
Schema
Max Rows
Demo DB
SAKILA
1000
-- Copyright (c) DbVis Software AB. All Rights Reserved.
SELECT
CU.CUSTOMER_ID AS ID,
CU.FIRST_NAME||' '||CU.LAST_NAME AS NAME,
A.ADDRESS AS ADDRESS,
A.POSTAL_CODE AS ZIP_CODE,
A.PHONE AS PHONE,
CITY.CITY AS CITY,
COUNTRY.COUNTRY AS COUNTRY,
DECODE(CU.ACTIVE, TRUE,'Active','') AS NOTES,
CU.STORE_ID AS SID
FROM
CUSTOMER CU
JOIN
ADDRESS A
ON
CU.ADDRESS_ID = A.ADDRESS_ID
JOIN
CITY
ON
A.CITY_ID = CITY.CITY_ID
JOIN
COUNTRY
ON
CITY.COUNTRY_ID = COUNTRY.COUNTRY_ID;
1 / 1 [0] INS
CRLF Not Connected UTF-8 Sakila.customerList.sql

# Changing the GUI Appearance

To change how the DbVisualizer GUI is displayed:

1. Open **Tools → Tool Properties**,
2. Select the **Appearance** category under the **General** tab.

Under **Appearance**, you can choose the **theme**, adjust the size and use of icons, set the use of the **Window Title Bar** and **Reset Layouts**. For themes, you can also choose to sync the theme **Automatic** from the underlying operating system.

In the subcategories **Fonts, Editor Styles** and **Colors** you can adjust the fonts used for different parts of the GUI and the colors used to highlight grid elements. Color choices are saved for the currently used theme.

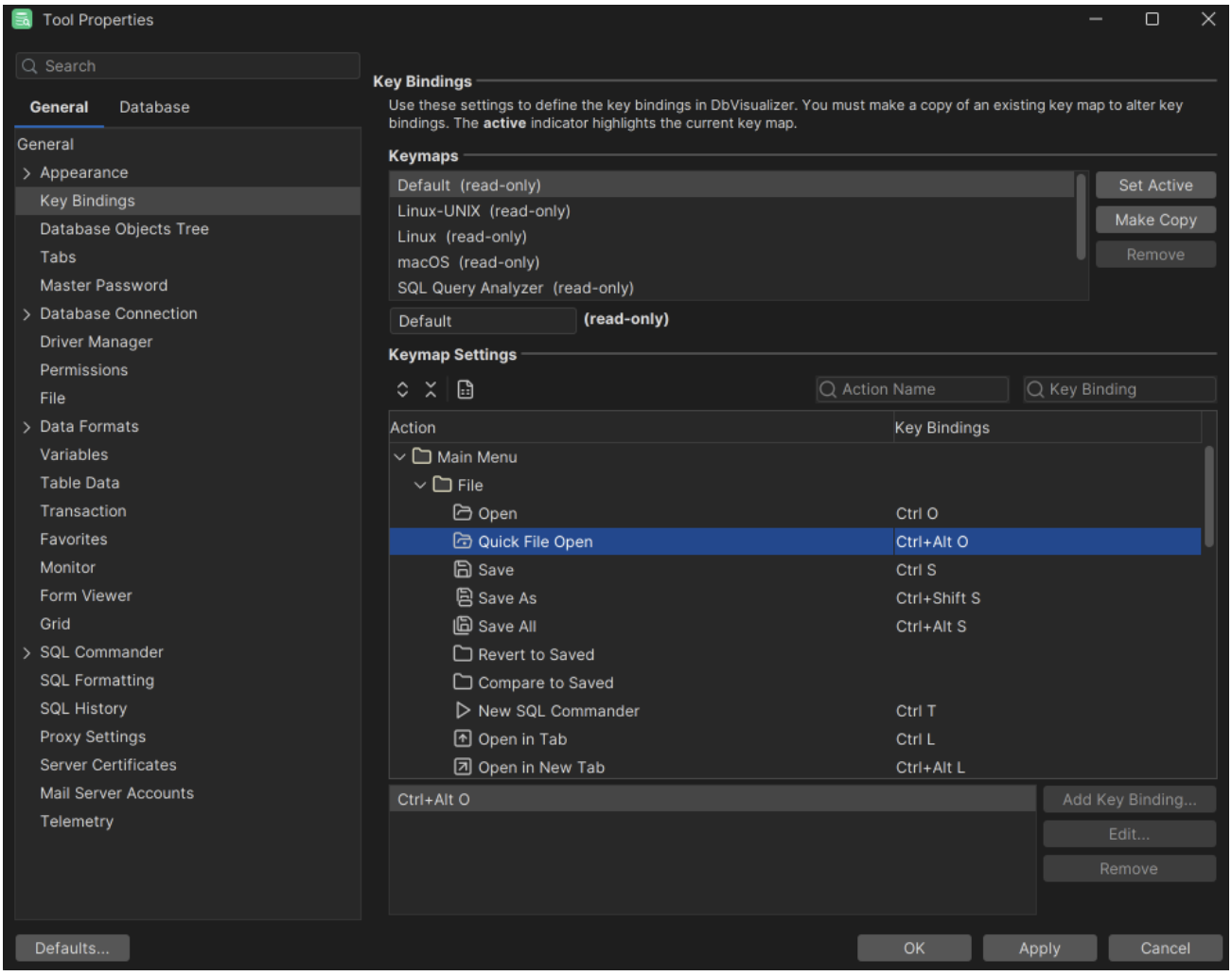
The main **View** menu also contains a number of toggle controls for showing or hiding GUI elements, such as toolbars and status bars.

# Changing Keyboard Shortcuts

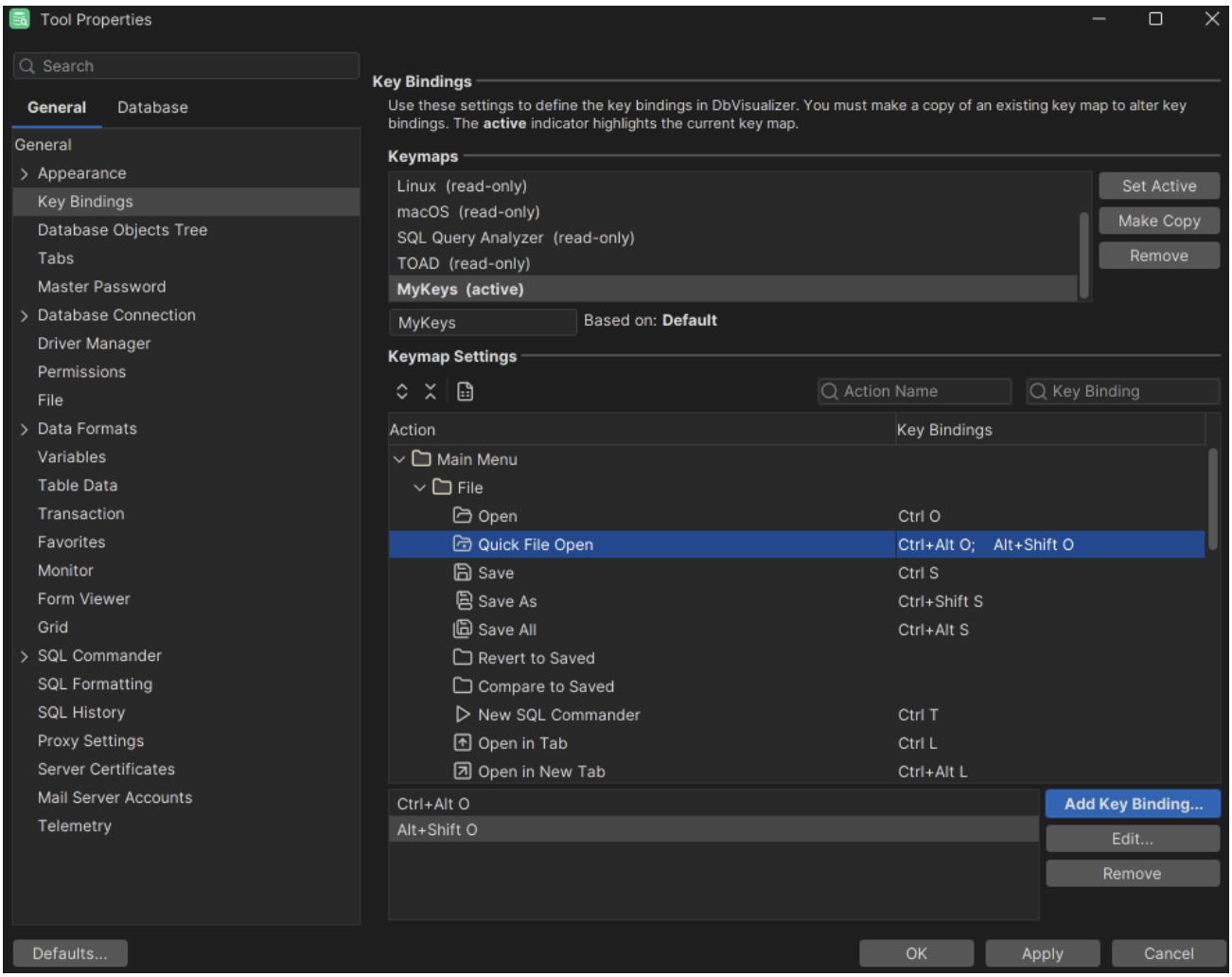
You can define key bindings for almost all operations and editor commands in DbVisualizer. Key bindings are defined in **Tools > Tool Properties** under the General / K**ey Bindings** category. Key bindings are grouped in **Key Maps**, each with an action list organized in folders; the **Editor Commands** folder lists all actions available in the SQL Commander editor and their current key bindings (in the **Key Bindings** list), the **Main Menu** folder contains subfolders, each representing a main window menu and other folders group feature specific actions, such as actions to control the references graph, form editor, etc.

DbVisualizer includes a set of predefined keymaps targeted for the supported operating systems. These keymaps cannot be deleted or modified. To customize key bindings, copy an existing keymap and make your changes.

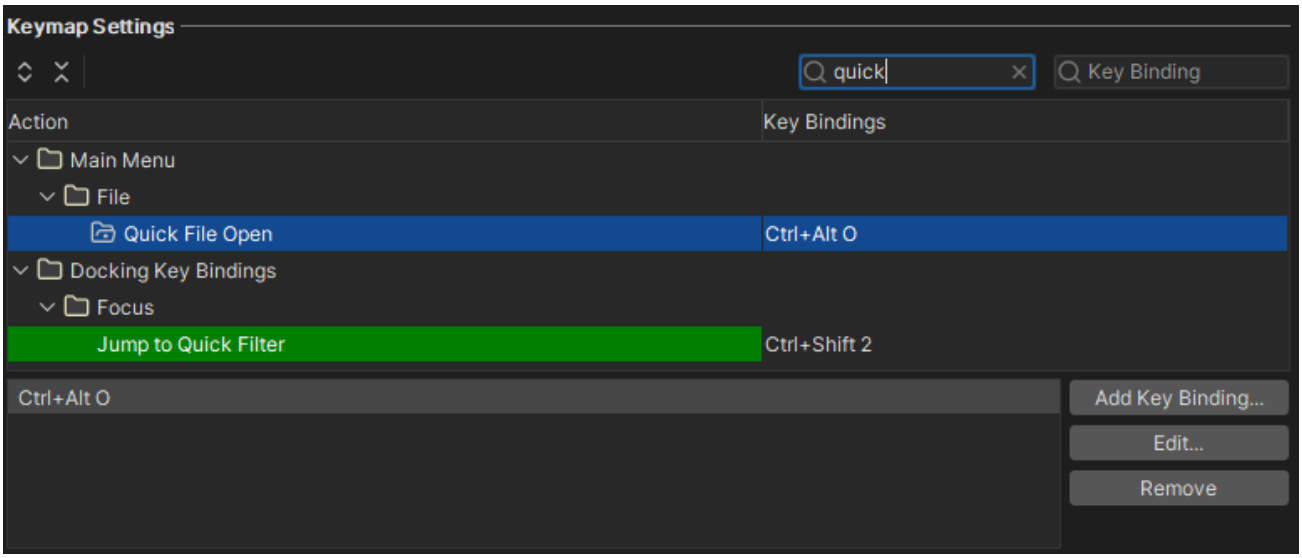
For instance, the **Main Menu** folder contains actions in subfolders, each representing a main window menu, and the **Editor Commands** folder lists all actions available in the SQL editor. For each action, the key binding(s) are defined in the **Key Bindings** column.



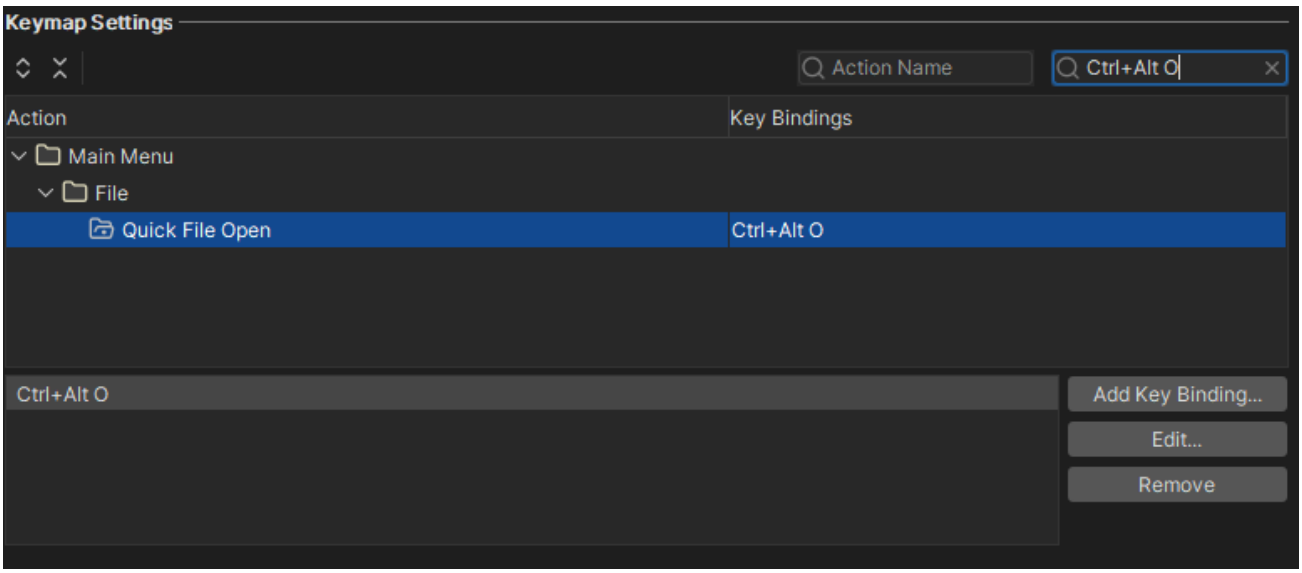
All user-defined keymaps are stored in your $HOME/.dbvis/config140/keymaps directory. A keymap file contains only the differences between the copied keymap and the current. To create a new keymap, select the map you want to copy and click the **Make Copy** button. The name of the copied keymap is suffixed with **_copy and is activated automatically**. Set a name on the new keymap if you like. The newly created keymap now has the exact same key bindings as the parent keymap.

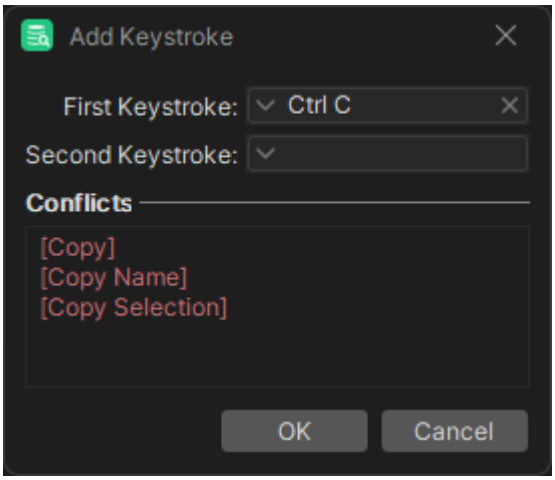



To modify the key bindings for an action, select the action from the action list. The current key bindings are listed in the **Key Bindings** list, and you can search for mappings by action or key binding.

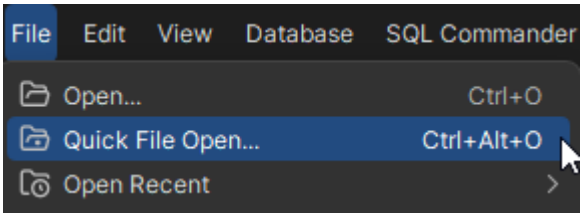



The modifier keys **Shift**, **Alt**, **Ctrl** and **Command** can be used to define the key binding. The keystroke dialog controls whether a key binding is already assigned somewhere else. If there is a conflict with another binding, the **Conflicts** area shows the names of the actions that are conflicting.






Menu items and tooltips show the first defined key binding in the list.



# Internationalization and Localization (i18N and L10N)

DbVisualizer is currently available in US English and supports user data in more or less any language or character encoding; DbVisualizer leverages the character sets supported by the Java VM, the fonts supported by the operating system, and the character encoding capabilities of the database to read, write, and render user data. You can also set your own preferences for date and time formats (see Changing the Data Display Format).

## Fonts and Character Sets

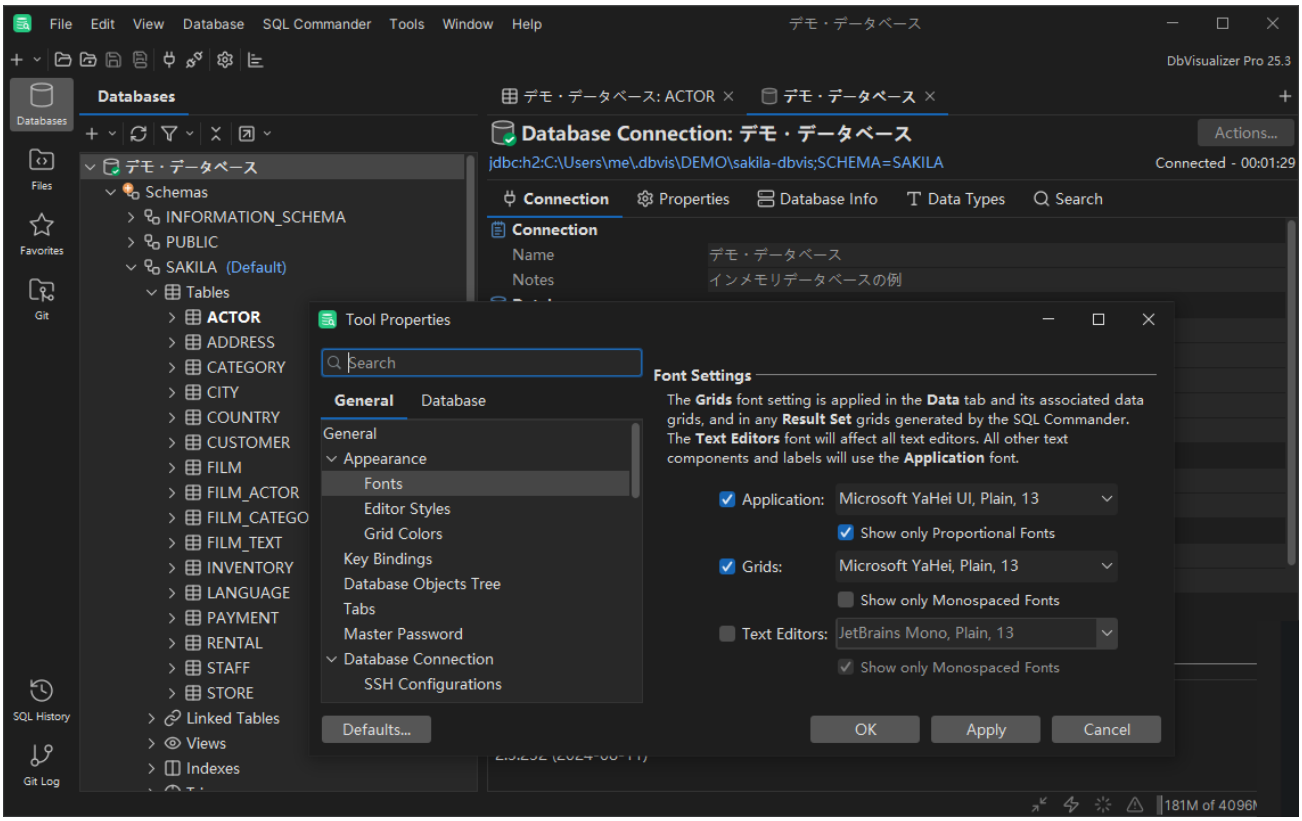
Depending on which platform you run (Windows, Unix/Linux, macOS) and what character set you want to use, you may have to change the font settings.

DbVisualizer offers three settings for fonts:

- **Application:** controls the font for the application components such as titles, buttons, menus, etc. The font size also controls the size of all tool icons and images.
- **Grids:** controls the font for GUI components where a proportional font is usually preferred (see Changing the GUI Appearance).
- **Text Editors:** controls the font for editors, log files, and other components where a monospaced font is usually preferred (see Editing SQL Scripts - Font Settings).

**Example:**

If you run Microsoft Windows and want to name your database connections using simplified Chinese (or any other language not supported by the default font), you will have to change the font for **Grids** and/or the **Application** to one that supports this character set. In this example, we use Microsoft YaHei UI for the application and Microsoft YaHei for the grids.

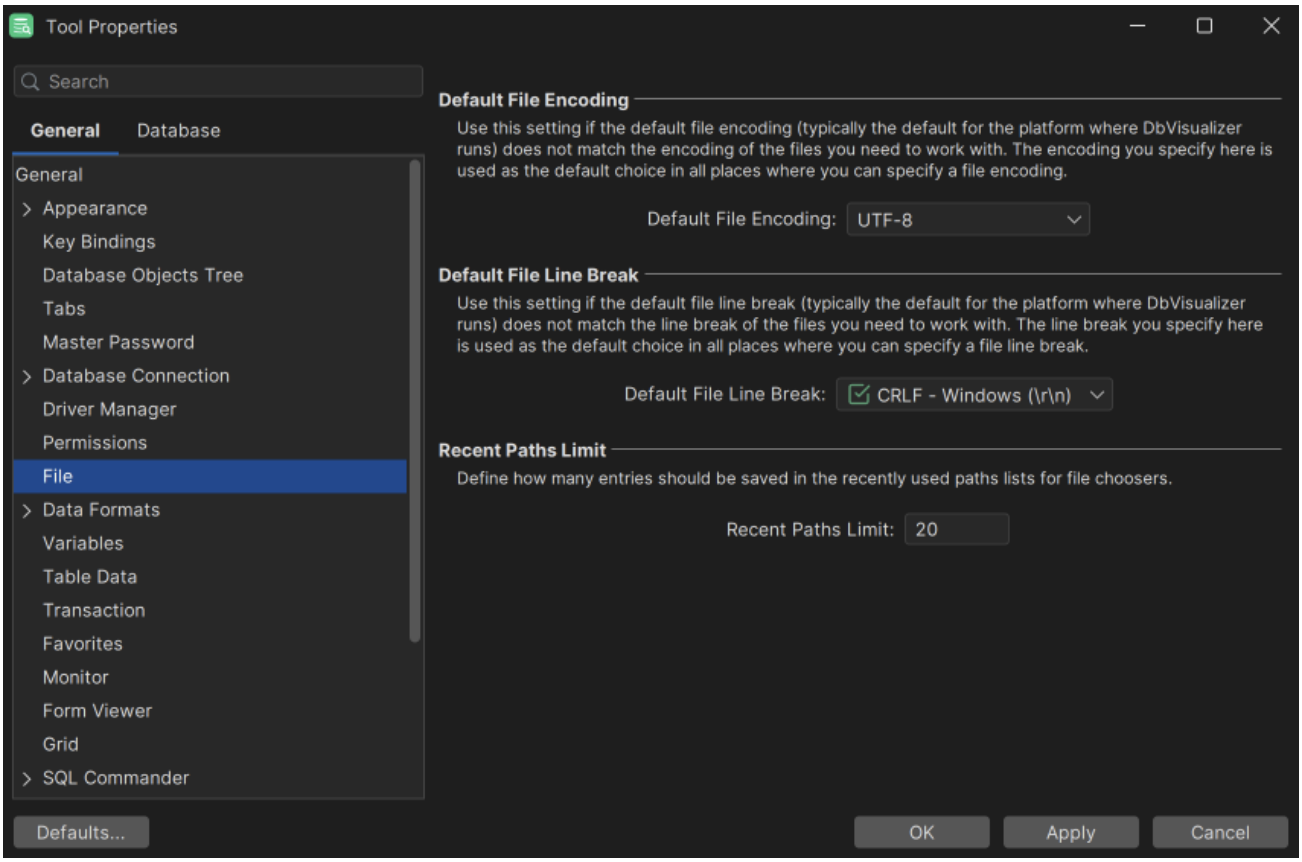



## Encoding

Even if the font is capable of rendering the characters, data may be scrambled in DbVisualizer if the encoding between the JDBC driver and the database is wrong.

> **When text results fetched from the database are scrambled there are a few areas that needs to be verified:**
>
> - The font selected in DbVisualizer must be capable to show the characters.
> - The encoding used to pass information from the database to the JDBC driver must match. Encoding depends on the capabilities of the JDBC driver. Check the driver documentation what options are available. Once determined, encoding changes are most often applied as driver properties in DbVisualizer.

You may also need to specify the encoding of text in binary data types, either for a specific connection or for all all connections of a specific database type. In this example we specify **UTF-8** encoding for all **SQL Server** connections:


Tool Properties
Search
General
Database
General
Appearance
Key Bindings
Database Objects Tree
Tabs
Master Password
Database Connection
Driver Manager
Permissions
File
Data Formats
Variables
Table Data
Transaction
Favorites
Monitor
Form Viewer
Grid
SQL Commander
Default File Encoding
Use this setting if the default file encoding (typically the default for the platform where DbVisualizer runs) does not match the encoding of the files you need to work with. The encoding you specify here is used as the default choice in all places where you can specify a file encoding.
Default File Encoding: UTF-8
Default File Line Break
Use this setting if the default file line break (typically the default for the platform where DbVisualizer runs) does not match the line break of the files you need to work with. The line break you specify here is used as the default choice in all places where you can specify a file line break.
Default File Line Break: CRLF - Windows (\r\n)
Recent Paths Limit
Define how many entries should be saved in the recently used paths lists for file choosers.
Recent Paths Limit: 20
Defaults...
OK
Apply
Cancel

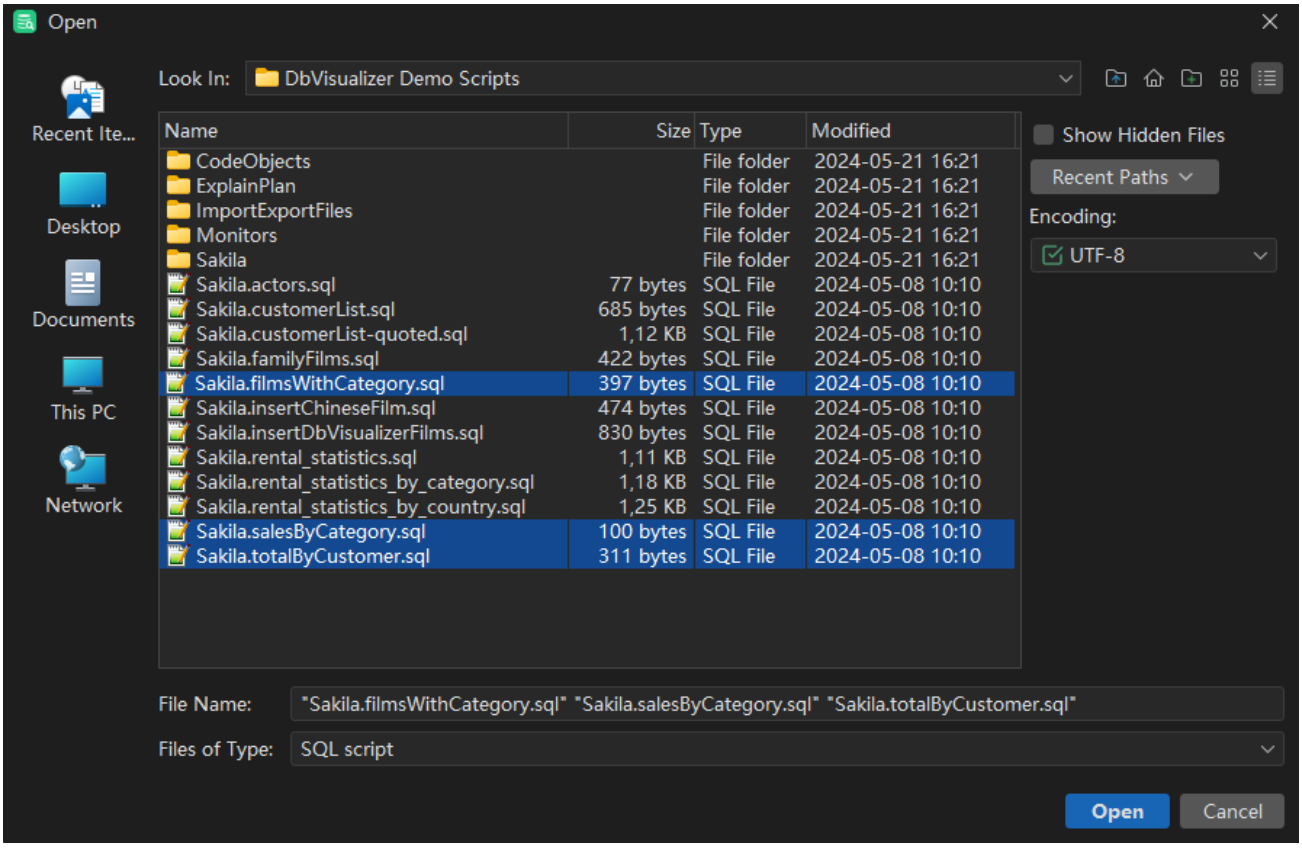
# Working with Files

Generic tasks such as working with files are available throughout DbVisualizer. These are used to open, select, and save files and folders.

> The file chooser will when invoked, in some Windows environments block DbVisualizer for a few seconds. Check the following article how to workaround the issue XXX

## File Chooser window

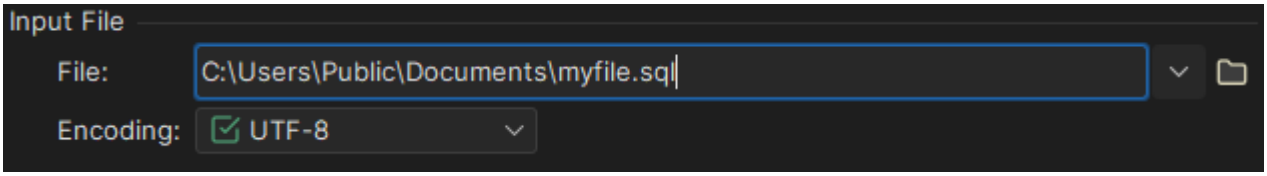
The file chooser window is a standard file chooser dialog allowing to navigate the file system and select files. The functionality and appearance may differ based on in which context it is opened. This example shows the file chooser with the left vertical button bar used to quickly navigate locations in the Windows file system. This button bar is only available in Windows. Apart from that the window looks the same in all OS'es. The **Recent Paths** drop down shows a list of recently selected files which makes it simple to select previously used files and folders. Encoding is only visible when loading or saving files and is by default set to match the encoding in the operating system.



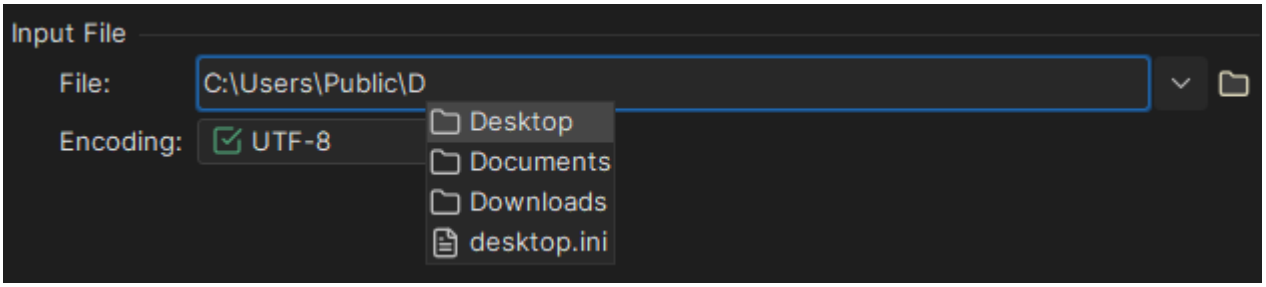
The file chooser allows dropping a file or folder path from other tools on it and will navigate to the path of the dropped file or folder.

## File Chooser field

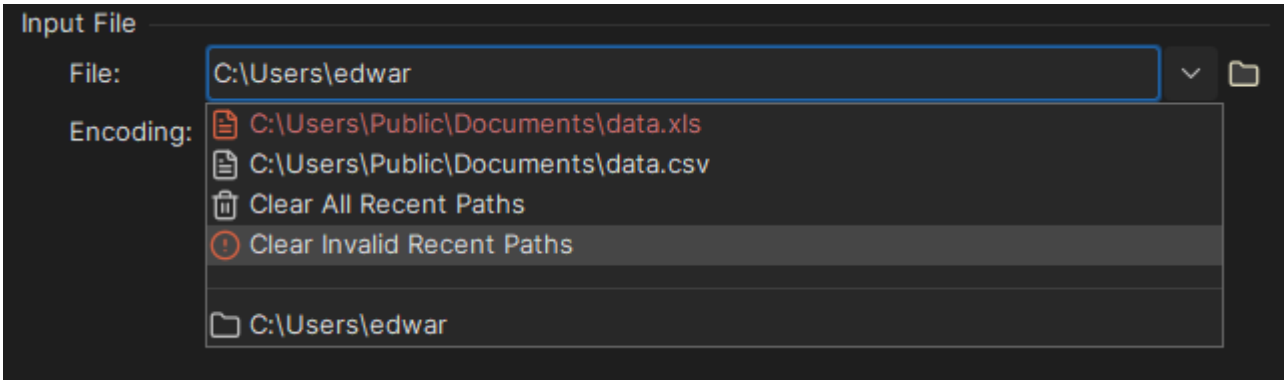
The file chooser field is a simple but powerful inline path component used to specify paths for files and folders. The file chooser may also be launched using the yellow button to the right of the field. The file chooser field is displayed in various dialogs, for example when importing table data.




The field allows auto completion using Ctrl+space (Cmd+space on macOS). Specify a letter or a few and the auto complete from a drop-down:




The rightmost drop-down, contains the list of recently used paths. Selecting one will put it in the file chooser field:




Any red path indicates these do not exist or are not available, select **Clear Invalid Recent Paths** to remove them.

The file chooser field supports dropping a path on it just like the file chooser window does.

### Using tilde ("~") in the File Chooser field

Using the tilde ("~") character as prefix in file paths represents the user's home folder. Consider the home for a user being `/Users/ben`, if then entering `~/Documents/project/SQL-Files` it will expand to `/Users/ben/Documents/project/SQL-Files.`

# Managing Database Objects

## Managing Database Objects

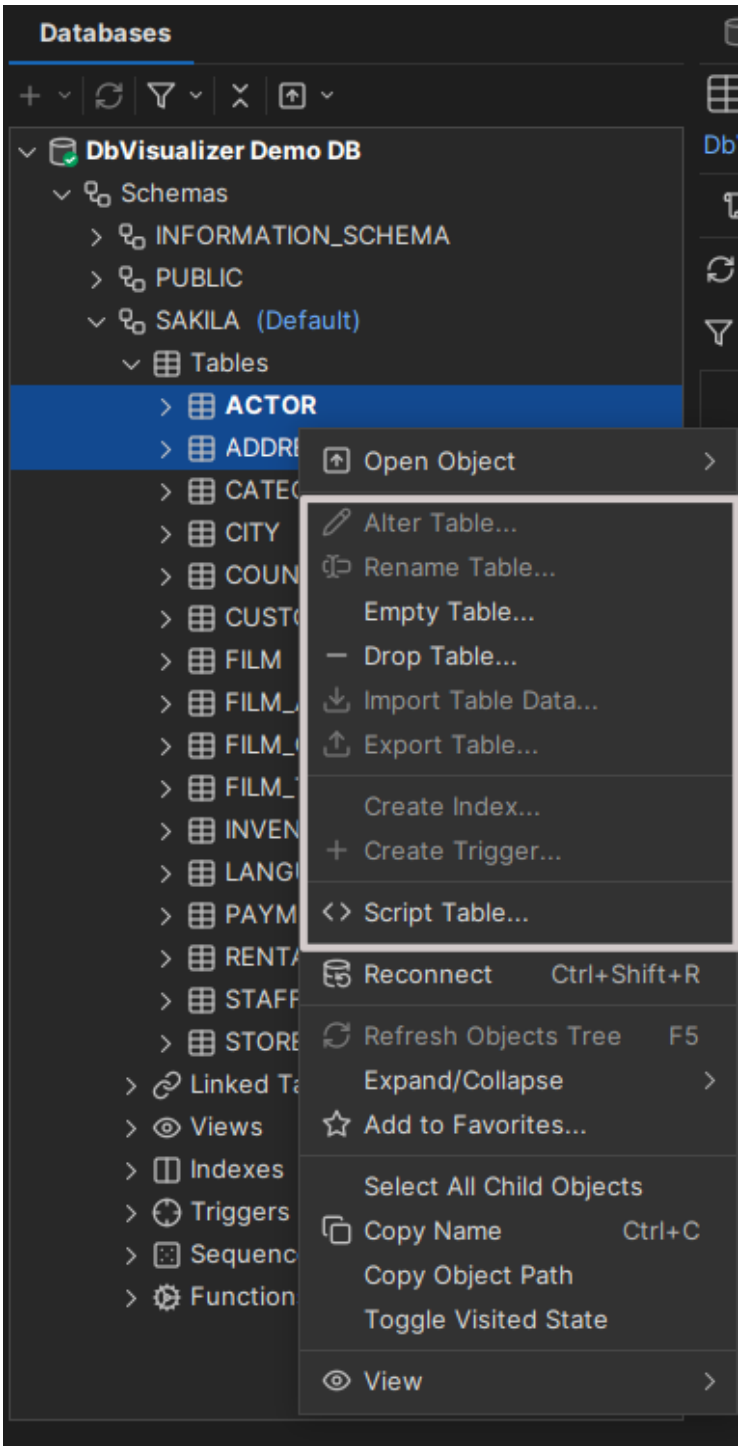

- Opening a Database Object
- Perform Actions on Multiple Database Objects
- Filtering Database Objects

# Opening a Database Object

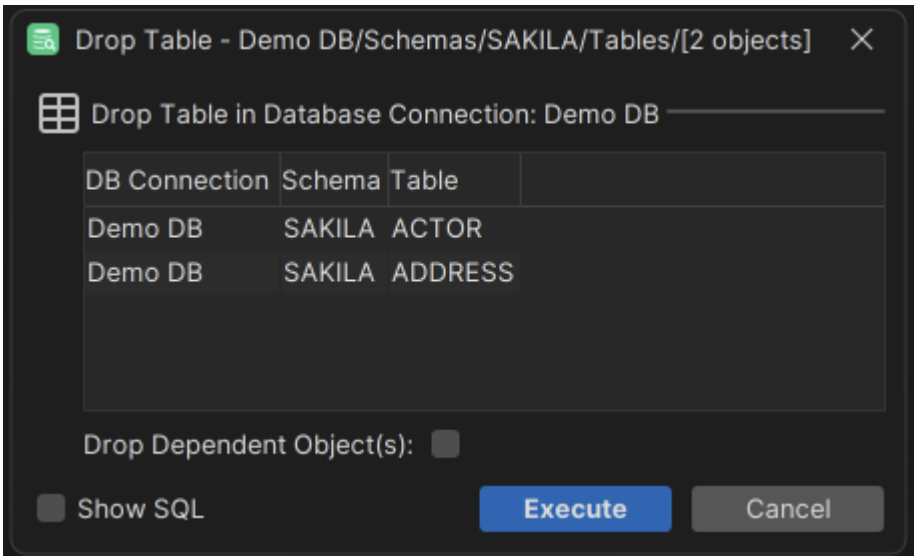
To open an **Object View** tab with the details for a database connection or for a specific database object, the simplest way is to double-click the object node in the **Databases** tab. For more information about opening tabs, check the Opening a Tab section.

# Perform Actions on Multiple Database Objects

The actions menu for a database object in the **Databases** tab consists of actions that are available for the selected object type. Some of the operations are common for any currently selected object while others are only available for the current object. Sometimes it is convenient to run a single action on multiple nodes at once. To do this, select the nodes (make sure all are of the same object type) and then right-click in the object list. The menu will now highlight those actions that are valid to use.




Choosing for example the **Drop Table** action shows the following window in which the drop can then be performed on multiple tables.


Drop Table - Demo DB/Schemas/SAKILA/Tables/[2 objects]
Drop Table in Database Connection: Demo DB
DB Connection
Schema
Table
Demo DB
SAKILA
ACTOR
Demo DB
SAKILA
ADDRESS
Drop Dependent Object(s):
Show SQL
Execute
Cancel

# Filtering Database Objects

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

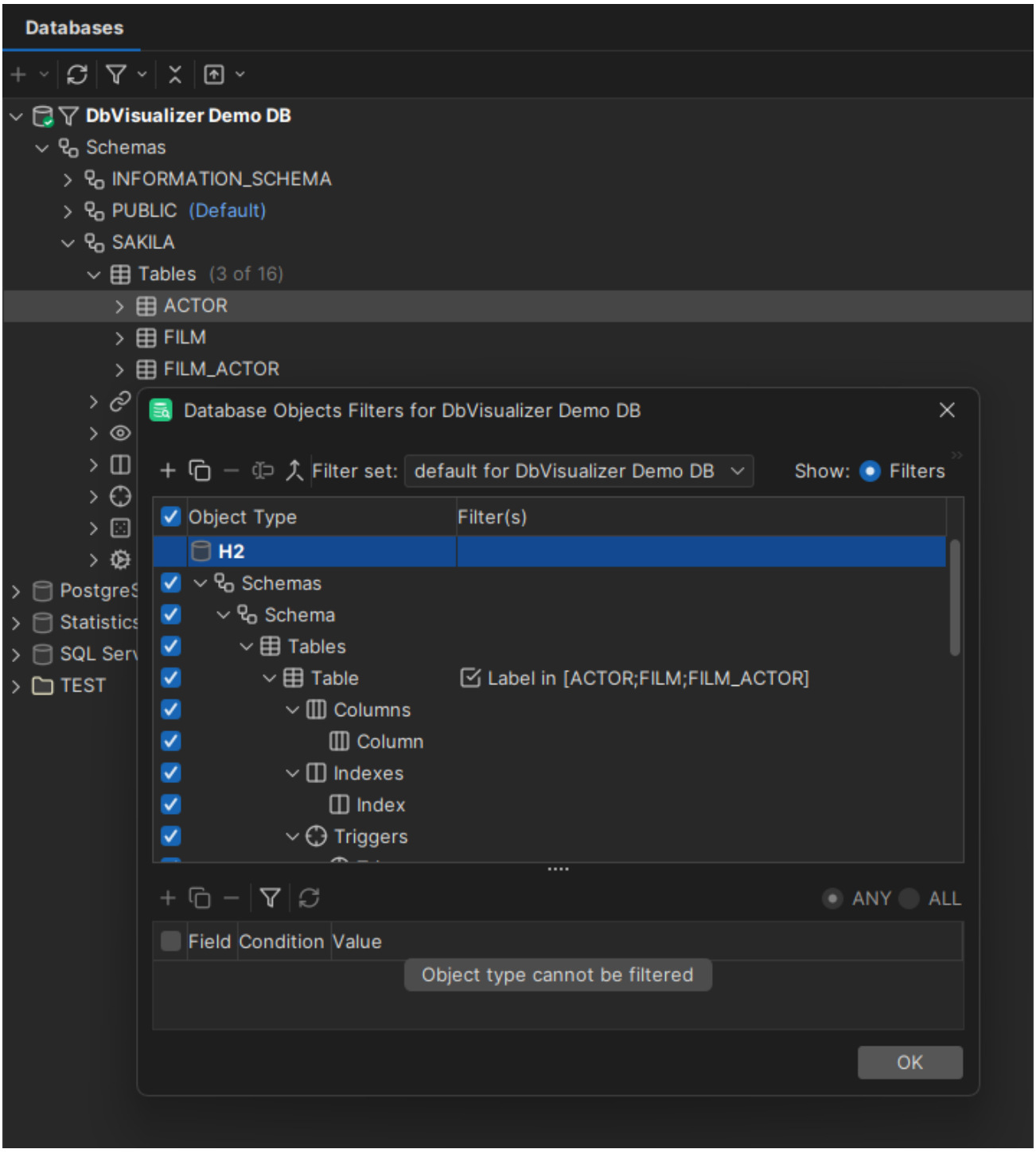
Managing what database objects are listed in the **Databases** tab is done at two levels:

1. **Objects Filtering** Defines what individual object nodes are listed for a specific object type. This allows filtering on for example Table objects so that only tables matching a condition are listed.
2. **Object Type Visibility** Defines what object types such as views, tables, indexes, procedures, etc. are listed. Every database in DbVisualizer supports all sorts of database objects. Having the ability to hide some object types makes it easier to locate the database objects that are of primary interest.

The **Filter Editor** is used to manage object type visibility, object filters, and labels. The Filter Editor is opened from one of:

- **Database** main menu and the **Database Objects Filters → Open Filter Editor**
- Right-click somewhere in the **Databases tab** and choose **Database Objects Filters → Open Filter Editor**
- Click the drop-down button on the funnel icon in the **Databases tab** toolbar and choose **Open Filter Editor**.

Managing filters and opening the filter editor requires that a node is selected in the Databases tab tree and that the related database connection is connected.




The upper list shows all available object types for the actual database connection. The leading **check mark** controls whether the object type should be visible in the Databases tab. Read more in the Object Type Visibility section. The **Filter(s)** column shows any filters defined for each object type. A **green check mark symbol** indicate that at least one filter is active while a **yellow cone symbol** indicate that all filters for the object type are deactivated.

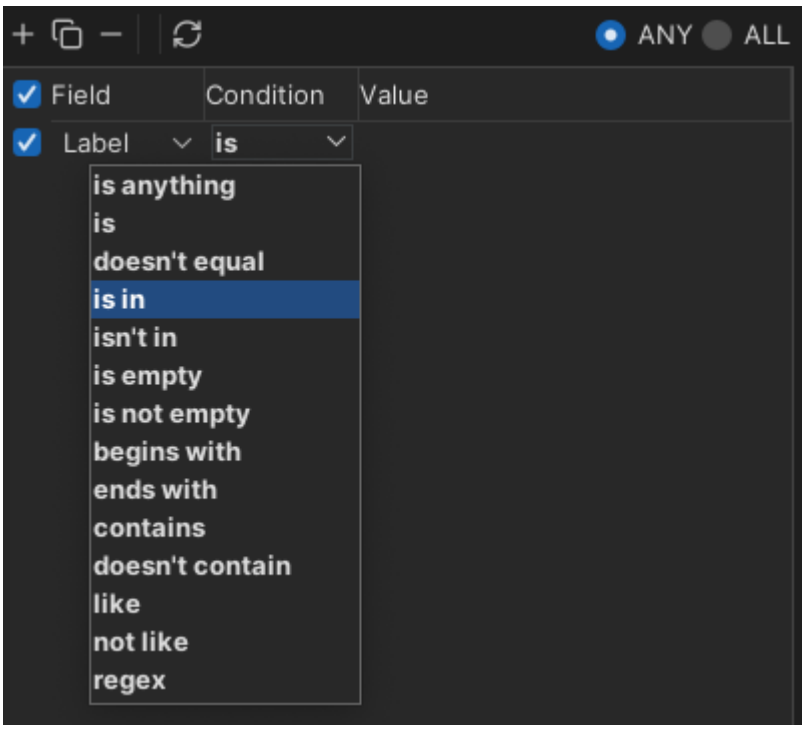
Selecting an object type enables the objects filtering area in the lower part of the window. Here individual filters are defined for an object type.

# Object Filtering

Object filtering can be made on any database object (**Table, Function, Procedure, User**) except for grouping objects such as **Tables, Functions, Procedures, Users**. Grouping objects in DbVisualizer are often labeled with the related object type name in plural.

> Object Filters for Database and Schema objects also apply to the corresponding drop-down lists in SQL Commander tabs, but only after a reconnect.

To setup a filter, select the object type in the objects list and in the filter area, click the button with a plus sign to insert a new row.




A filter entry consists of a field (e.g **Label),** a condition and a value to match against. Click the **condition** field to select the condition to use. In the **Value** field, enter the value that should be matched. For multi value conditions, such as **is in** and **isn't in,** the list of values are separated with a semicolon or edited in separate window that is opened by pressing the right-most icon in the input field.

Each individual filter can be deactivated using the **check mark**. Uncheck it and the corresponding filter will not be used unless reactivated. The currently defined filters are listed in the upper object type list for each object type. The leading symbol shows either a green check mark which indicates that some of its filters are active or a yellow cone symbol that shows that no filter is active.

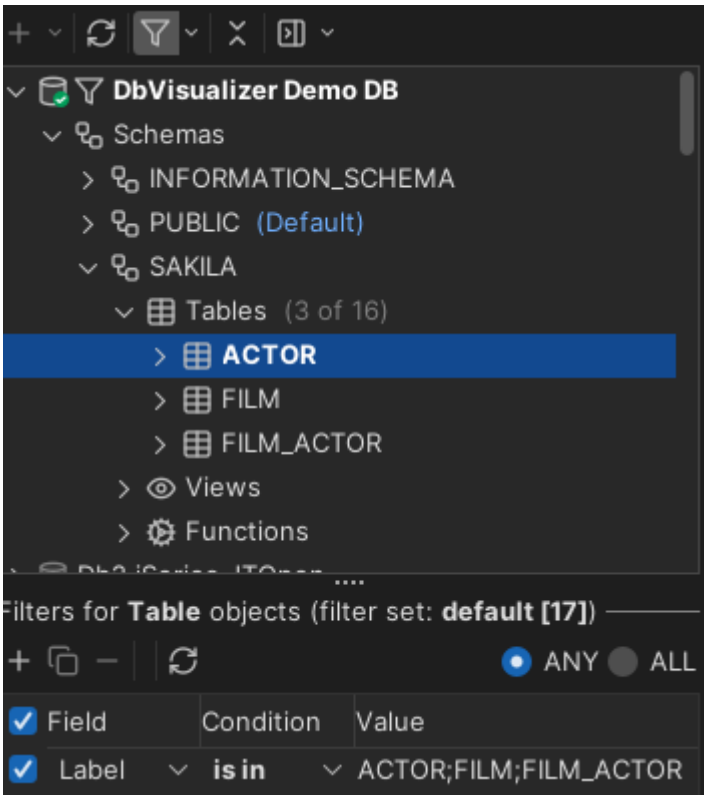
You can define more than one filter. Just click the green plus or yellow star (to duplicate an existing row) buttons. If you have more than one active filter, you must also select if the filter should match **Any** or **All** filter entries.

Filter entries (rows) can be moved up/down to arrange the filters, just select one ore more rows and use the popup menu **Move Up**/**Move Down.** You can also use **Drag&Drop** to move the rows.

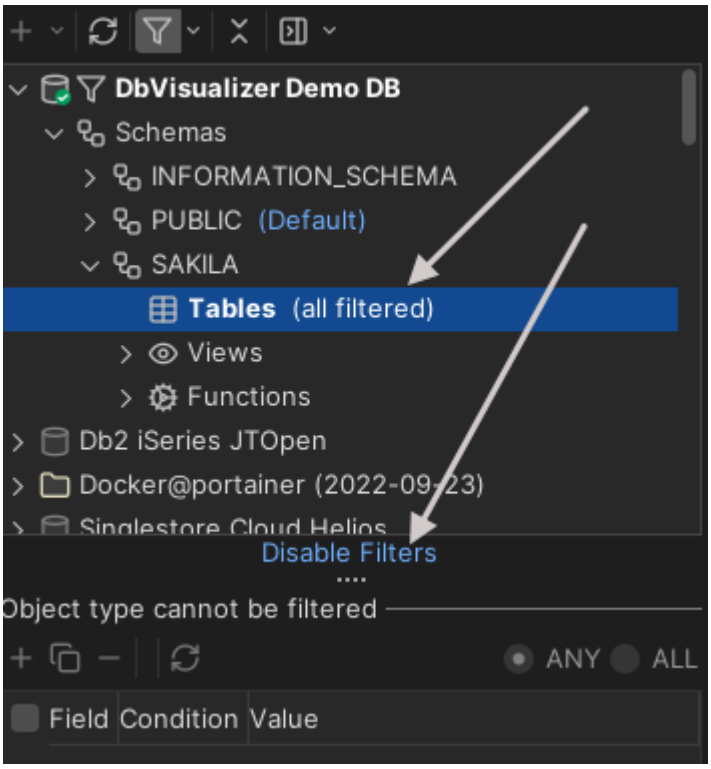
## Inline Objects Filtering

The filter area can also be displayed just below the objects tree in the **Databases** tab. This is convenient as you can then quickly manage and verify the effect of certain filters. To toggle the display of the filtering area, either click the left part of the funnel icon in the **Databases** tab toolbar or click the right menu arrow and select **Database Objects Filter → Show/Hide Filter Area**.

> Managing filters and opening the filter editor requires that a node is selected in the Databases tab tree and that the related database connection is connected.



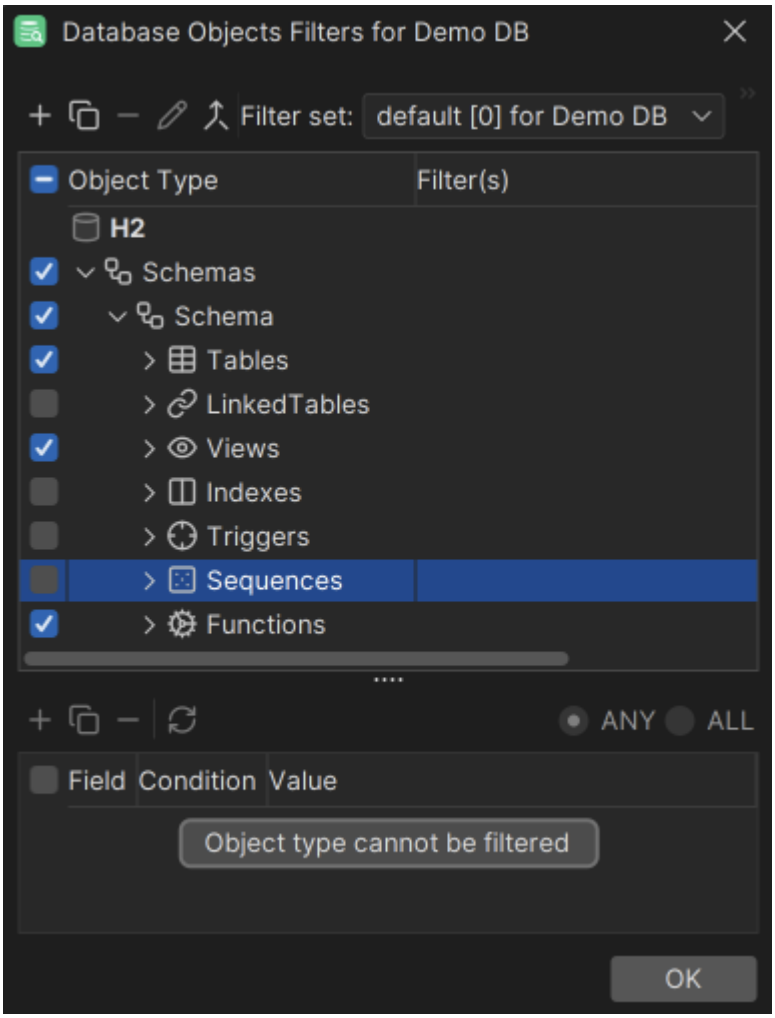

In this example, the filter area is displayed with the filters for the **Tables** object type listed. When filters have been modified, you need to manually apply the filters using one of the **reload buttons** in the filter area toolbar or in the Databases tab toolbar. If a filter results in all object nodes being filtered, "**(all filtered)**" is displayed next to the parent object node in the tree. Since the filtered object type is now invisible as a result of the current filter, you need to disable the filter to refine it. Do this by selecting the parent node with the "**(all filtered)**" label and then click the "Disable Filters" link just below the objects tree:




Clicking "Disable Filters**"** deactivates all filters for the object type and you are now able to modify the filter to get the matches you want.

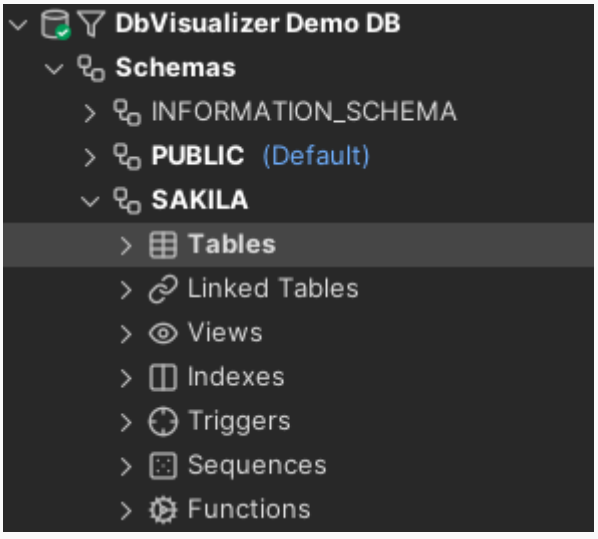
## Object Type Visibility

Object type visibility is the functionality used to define what object types should be visible in the **Databases** tab. For some databases, the tree of object types can be really long and many objects are rarely used or of minor interest. By hiding object types, the tree is compressed to only show what you are really interested in. To control the visibility of object types, open the **Filter Editor**.

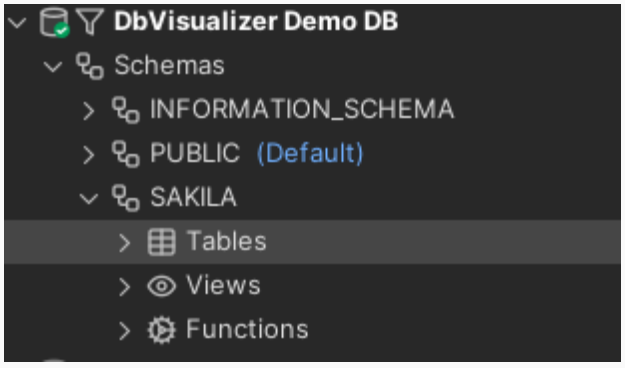



The previous image shows the object types available under a **Schema** in H2. As you can see, there are plenty of them (sub object types are collapsed in the example for better illustration). The following example shows the database tree in the Databases tab before and after the visibility has been set based on the previous screenshot.

**All Schema objects displayed**

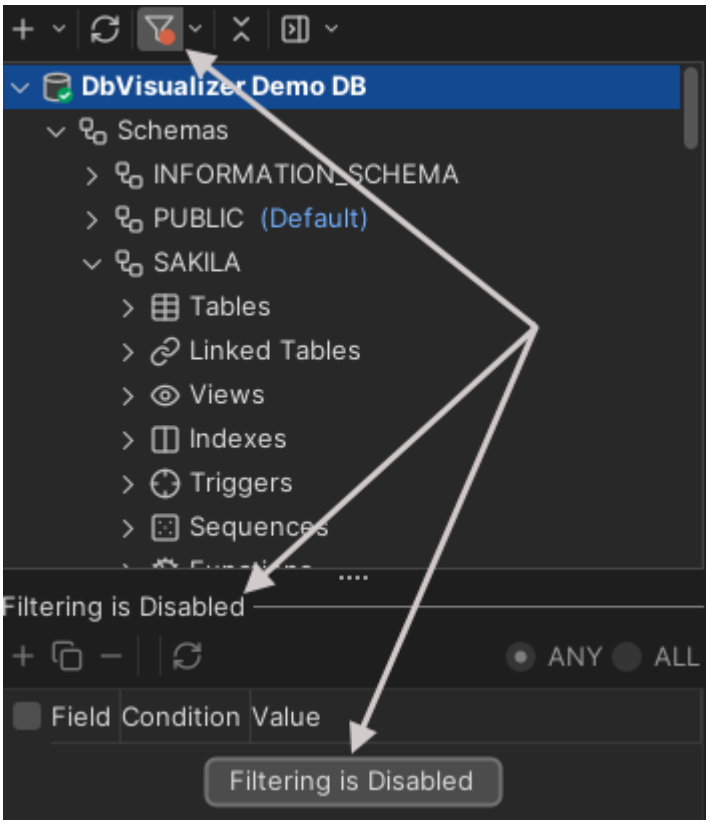



**With hidden Schema objects**



## Temporarily Disable Filtering

While browsing the objects in the Databases tab, it may be convenient to quickly toggle between the standard non filtered view and the filtered view of objects and types. This is easily accomplished with the **Disable Filtering** action in the funnel drop-down menu. While filtering is disabled, the funnel symbol in the toolbar shows a red indicator and text in the inline filtering area shows the current status.




While filtering is disabled it is not possible to manage filters at all and the related actions are disabled.

> Disabling filters is applied per database connection. This means that if you disable the current filter set on a MySQL connection it wont affect any filters defined for other database connections.

## Filter Sets

When you apply an object filter or hide an object type, that configuration is saved in a **Filter Set** that is saved between sessions. For every database connection, there is always a **default filter set**. If you are happy with the basic filtering capability you can stop reading here. If you however are interested in having multiple filter sets that can optionally be shared between multiple database connections and easily be switched between, keep on reading.

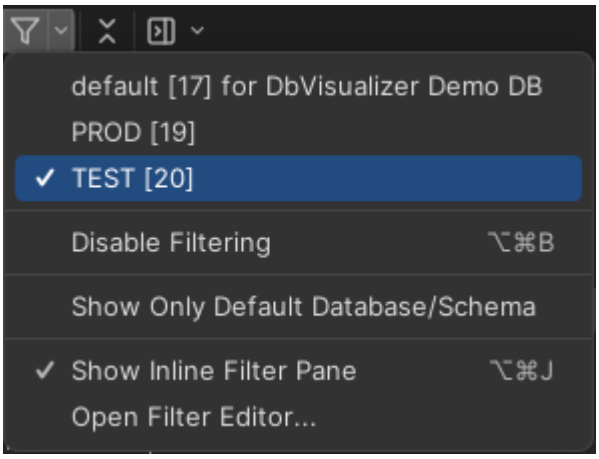
Filter sets are managed in the filter editor. Here you can create and delete filter sets and merge from another. Filter sets are either associated with a specific database connection, its name is then **default for <database connection name>** or a custom filter set that can be enabled for many connections, named as you like. A custom filter set is always associated with the database type currently being used for your database connection. The latter means that you are able to share filters sets that are all associated with the same database type. If you create a filter set for MySQL then this will never show up if you are working with an Oracle database.

The Filter Set drop-down in the filter editor shows the currently used filter set, and when clicked, the available filter sets that you may switch to.

The merge functionality (the blue merge arrow button) can merge not only from custom filter sets but also from the default filter sets associated with other database connections with the same database type.

### Switching Filter Set

In the filter editor, you switch filter set by selecting one from the Filter Set drop-down. In the Databases tab, there is the funnel symbol which when clicked is used to toggle the display of the inline filter area. Clicking on the funnel drop-down symbol opens a menu:



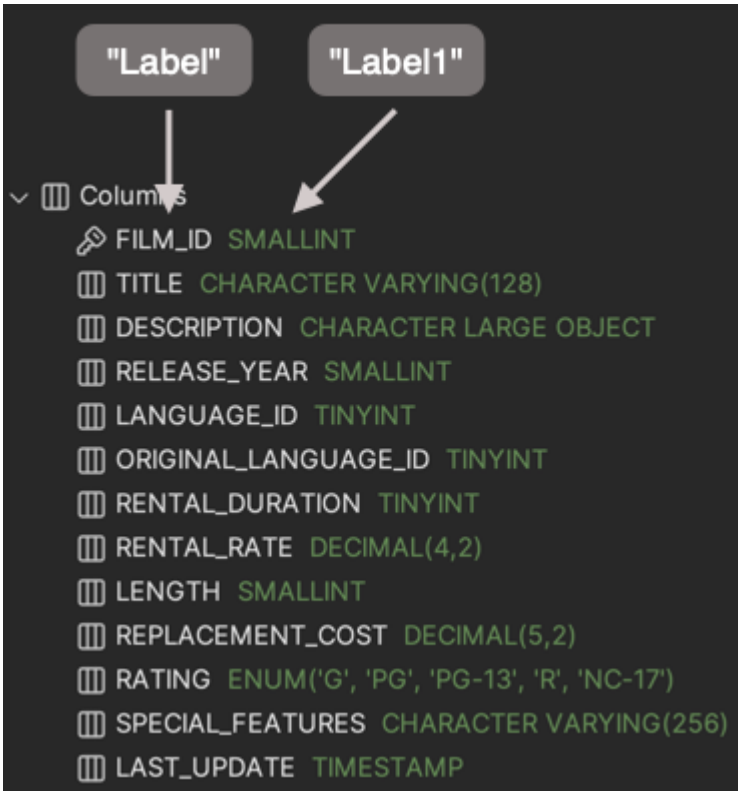

At the top of the menu, the default filter set for the database connection is displayed first with any custom filter sets below it. An entry that is check marked indicates that it is active.

## Show Only Default Database/Schema filter

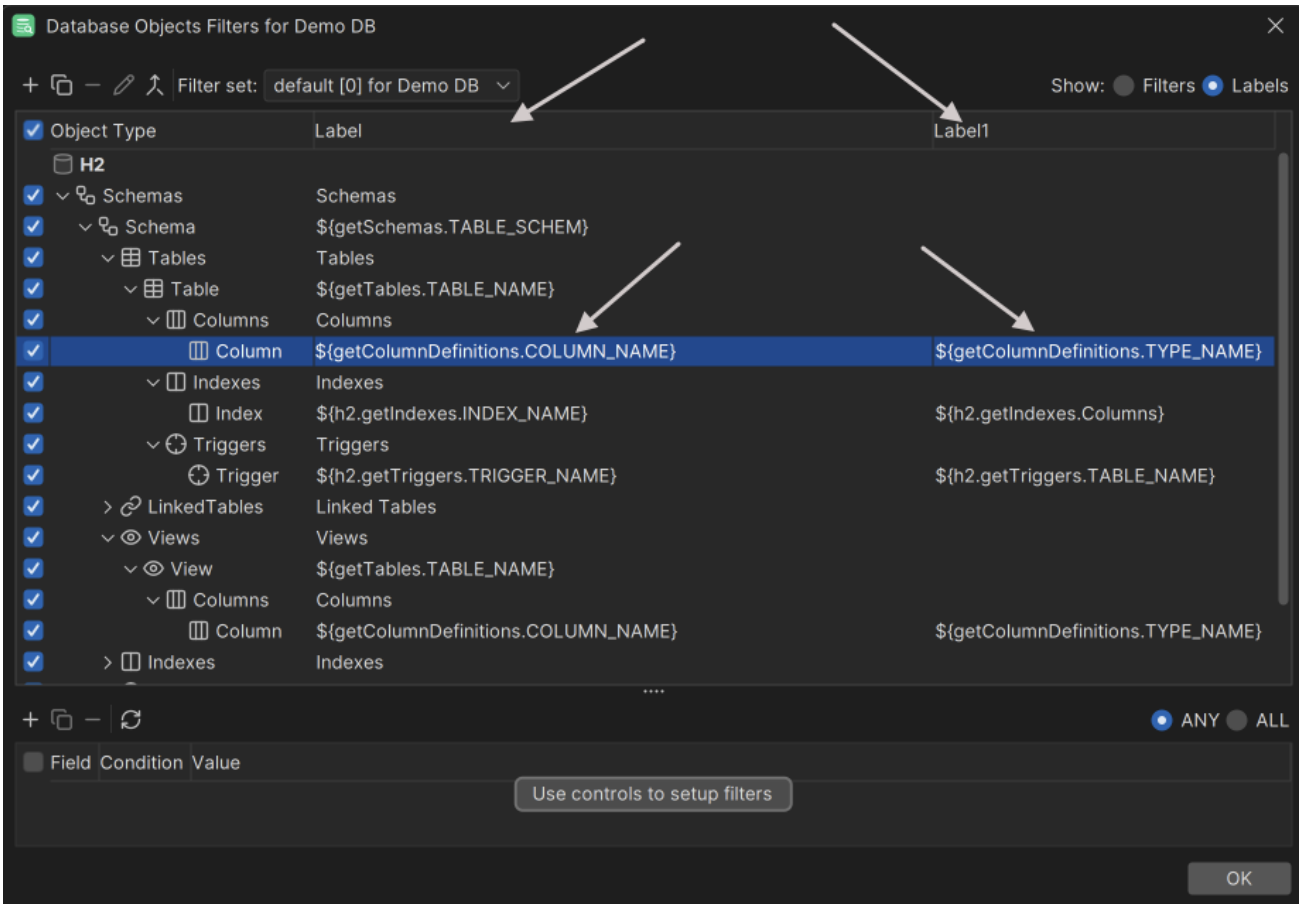
There is a special filter used to filter any database and schema objects to show only the default for the session. It is listed as Show Only Default Database/Schema in the filter menu. When selected, a special filter is applied on the corresponding Database and/or Schema object type and the effect is what its name implies.

## Labels

The label displayed for an object in the **Databases** tab is composed from variables associated with the database object. These labels can be modified in the Filter Editor (**Database Objects Filters → Open Filter Editor** in the **Database** main menu). Here is an example how the object label look for a column object type in the **Databases** tab:




In the Filter Editor click the **Labels** button in the top right corner of the window. Two columns, **label** and **label1** now shows how each object type should be labelled. The selected row for the **Column** object type shows that the first **label** should be rendered with the value from `${getColumnDefinitions.COLUMN_NAME}` and the **label1** is `${getColumnDefinitions.TYPE_NAME}`.




The variables used in the **Label** column should rarely be altered. **Label1** on the other hand can be modified to show for example the column comment (if any), i.e. `${getColumnDefinitions.COMMENT}`. For the Column object type and its `getColumnDefinition` variable scope these are available:

- COLUMN_NAME
- SIZE
- SCALE
- TYPE_NAME
- NULLS
- AUTO_INCREMENT
- ENCODING
- COMMENT
- IS_INVISIBLE
- COLUMN_DEFAULT
- IS_PRIMARY_KEY

Note that the variables are not documented as these are all database specific and obtained dynamically in the database profile for each database.

# Working with Tables

## Working with Tables

DbVisualizer provides many ways to work with tables.


- Creating a Table
- Altering a Table
- Creating a Trigger
- Creating an Index
- Viewing Table Data
- Editing Table Data
- Working with Document Data
- Working with Binary and BLOB Data
- Working with Large Text/CLOB Data
- Using Max Rows and Max Chars for a Table
- Changing the Data Display Format
- Exporting a Table
- Importing Table Data
- Comparing Tables
- Viewing Table Relationships
- Navigating Table Relationships
- Viewing the Table DDL
- Filtering Tables in the Tree
- Showing Row Count in the Tree
- Using Permissions for Table Data Editing
- Scripting a Table
- Managing Table and Column Comments

# Creating a Table

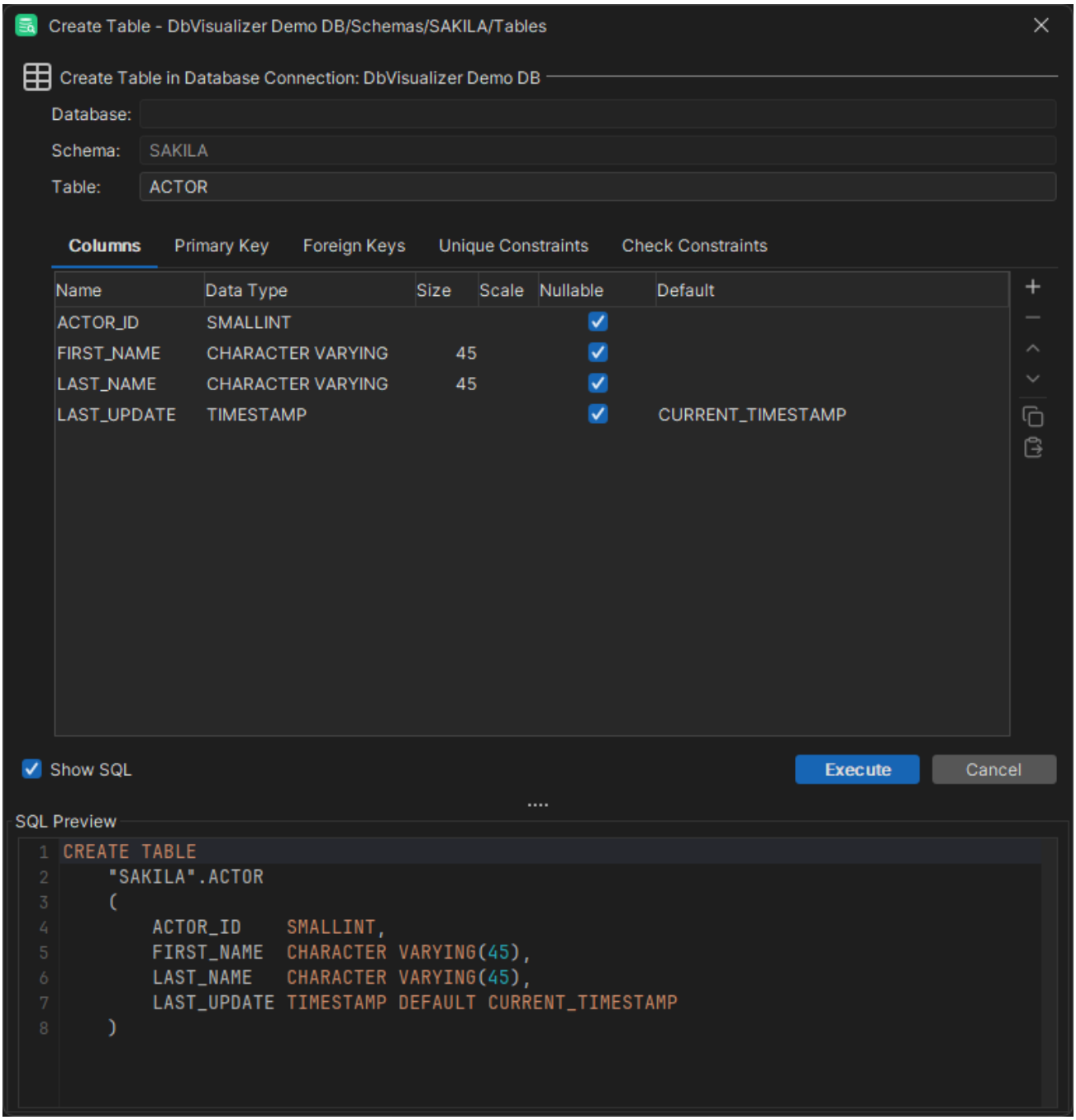
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The Create Table dialog helps you create a table without writing SQL.

## Opening the Create Table Dialog

To create a new table:

1. Expand nodes in the tree under the connection node in the **Databases** tab tree until you reach the **Tables** node,
2. Select the **Tables** node and open the **Create Table** dialog from the right-click menu.

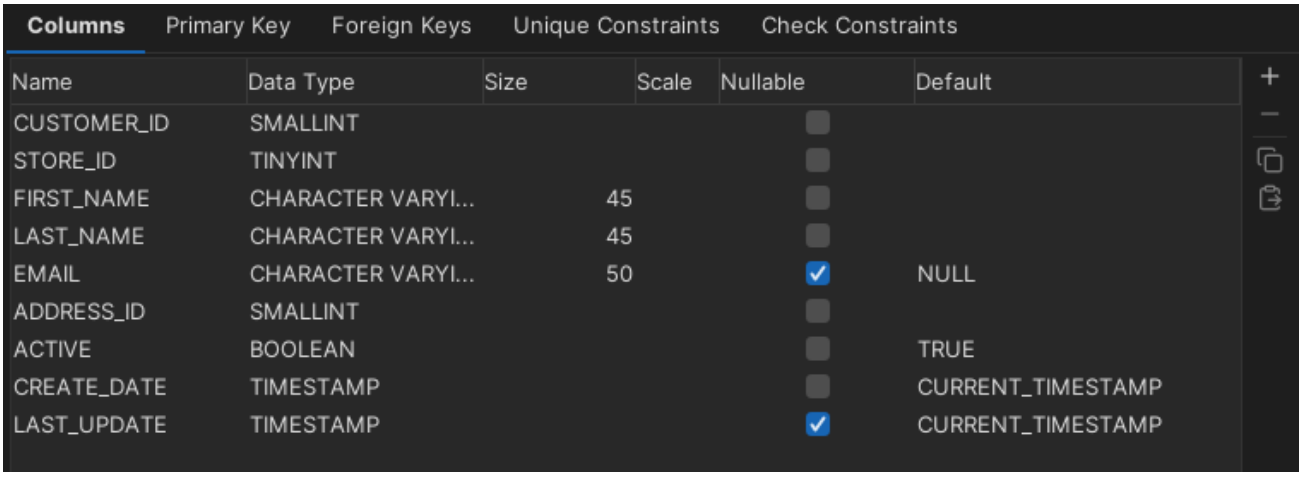



The **Create Table** dialog is organized in three areas from the top:

- General Table Info Specifies the owning database connection, database and/or schema. These are picked up from the selection in the tree when the dialog is opened. Table name is set to a default name that you should change to the real table name.
- Table Details A number of tabs where you specify information about the columns and, optionally, various constraints. The **Columns**, **Primary Key** and **Foreign Key** tabs are available for all databases. The remaining tabs are database-specific and depends on the features supported by the database engine.
- SQL Preview The SQL previewer shows the SQL statement for creating the table based on your input.

## Columns Tab

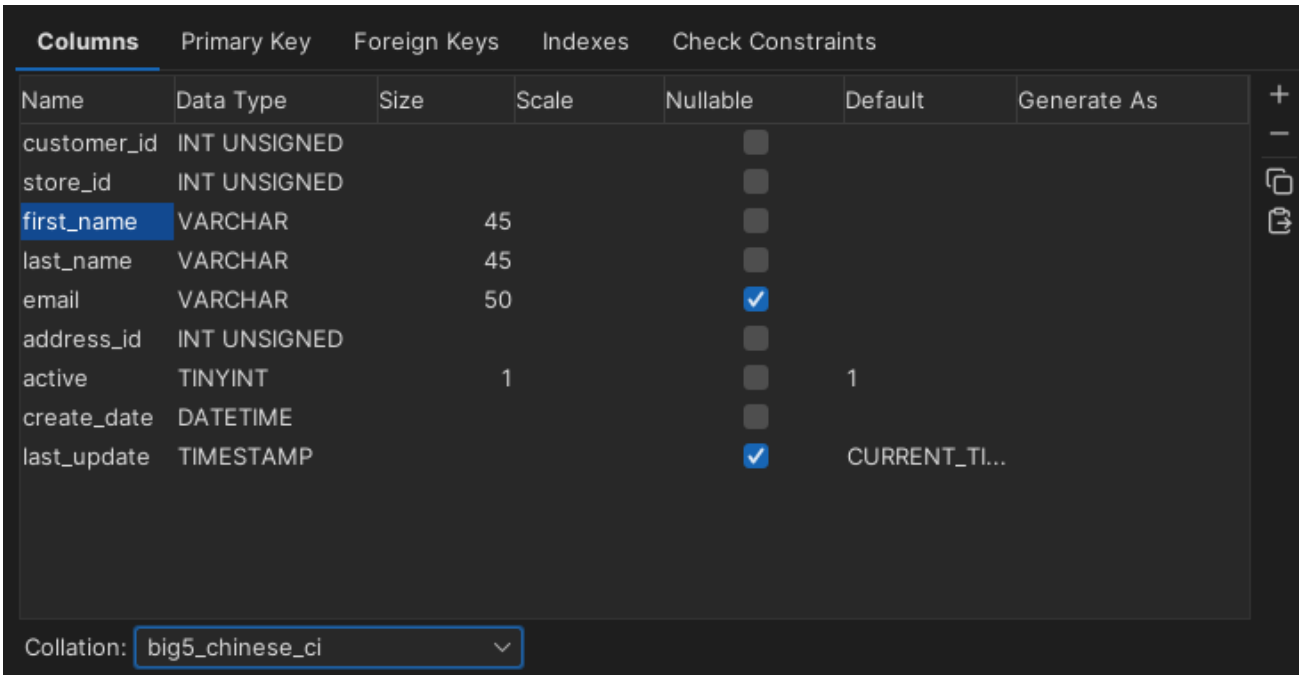
The **Columns** tab lists all table columns along with their attributes. The actual columns of the Columns tab is dependant on database. Additional columns may be shown for other databases. An example is Oracle and MariaDB which supports invisible columns and thus a column for setting this is shown.



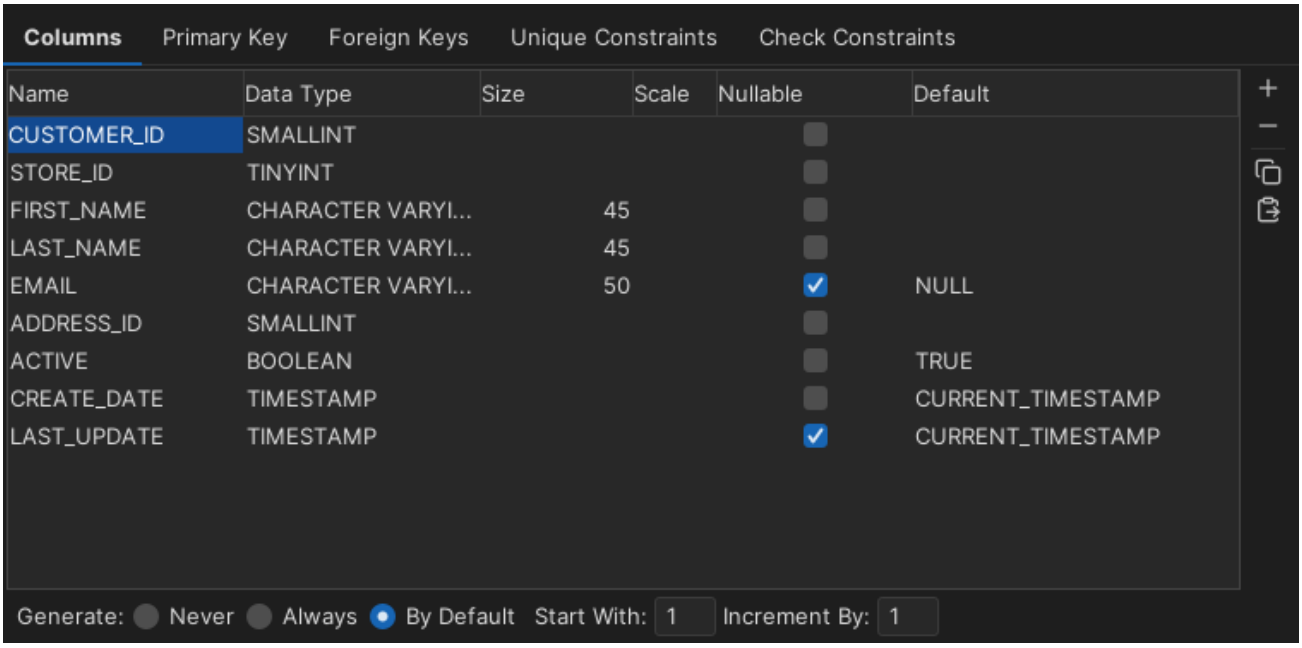

To add a column:

1. Click the **Add** button,
2. Enter the name of the column in the first field and select a data type from a drop down list in the second field, or start typing the data type name to find it and select it with the **Enter** key. The list contains the names of all data types the database supports,
3. For some data types, such as character types, you may also specify a size, i.e., the maximal length of the value. For others, like the decimal types, you can may specify both a size and a scale (the maximal number of decimals),
4. In the last two fields, specify if the table is nullable and a default value to use for rows inserted into the table without specifying a value for the column.

You may find additional fields depending on the features supported by the database you are working with and the data type for the current column. The **Collation** field is shown for character columns if the database supports the declaration of a collation for textual data.




Settings for generated fields are shown if the database supports automatically inserted values, typically to insert the next available sequence number in a numeric column.




The **Create/Alter Table** dialog uses database metadata to try to enable only the fields that apply to the selected data type, but please note that it is not always possible. For instance, there is no metadata available to tell if a data type requires, or allows, a size. If you don't enter a required attribute or enter an attribute that is unsupported for a data type, you will get an error message when you click **Execute** to create/alter the table.

To remove a column:

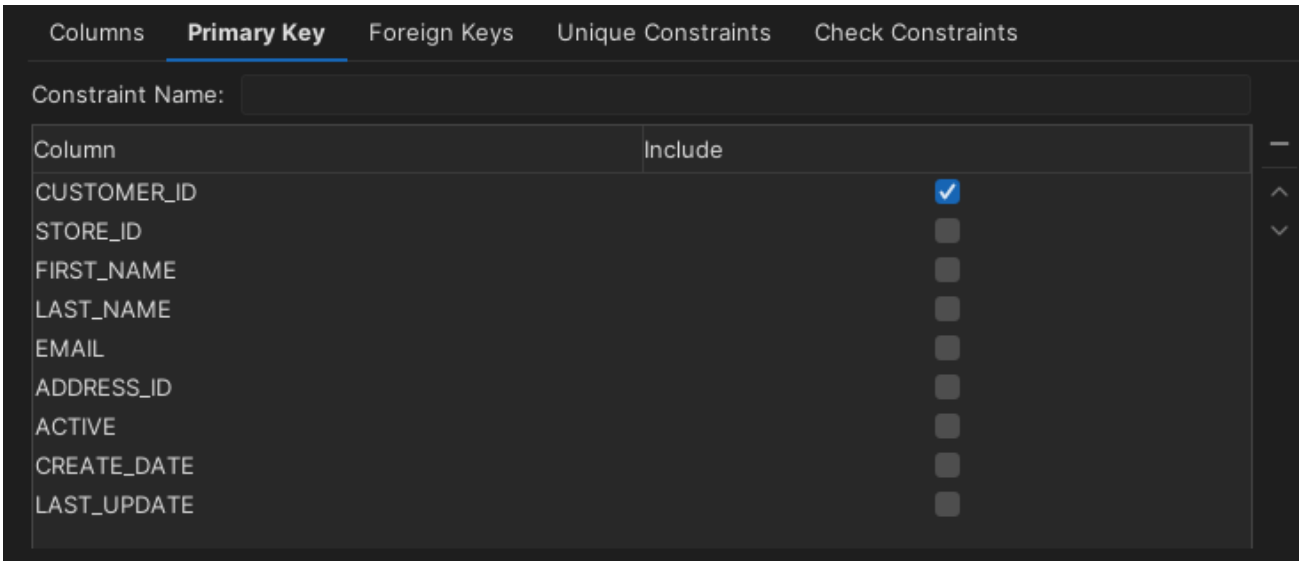
1. Select a cell in the column row,
2. Click the **Remove** button.

To move a column to another location (Only supported for Create Table):

1. Select a cell in the column row,
2. Click the **Up** or **Down** buttons.

## Primary Key Tab

The **Primary Key** tab contains information about an optional primary key for the table. A primary key is a column, or a combination of columns, that uniquely identifies a row in a table.

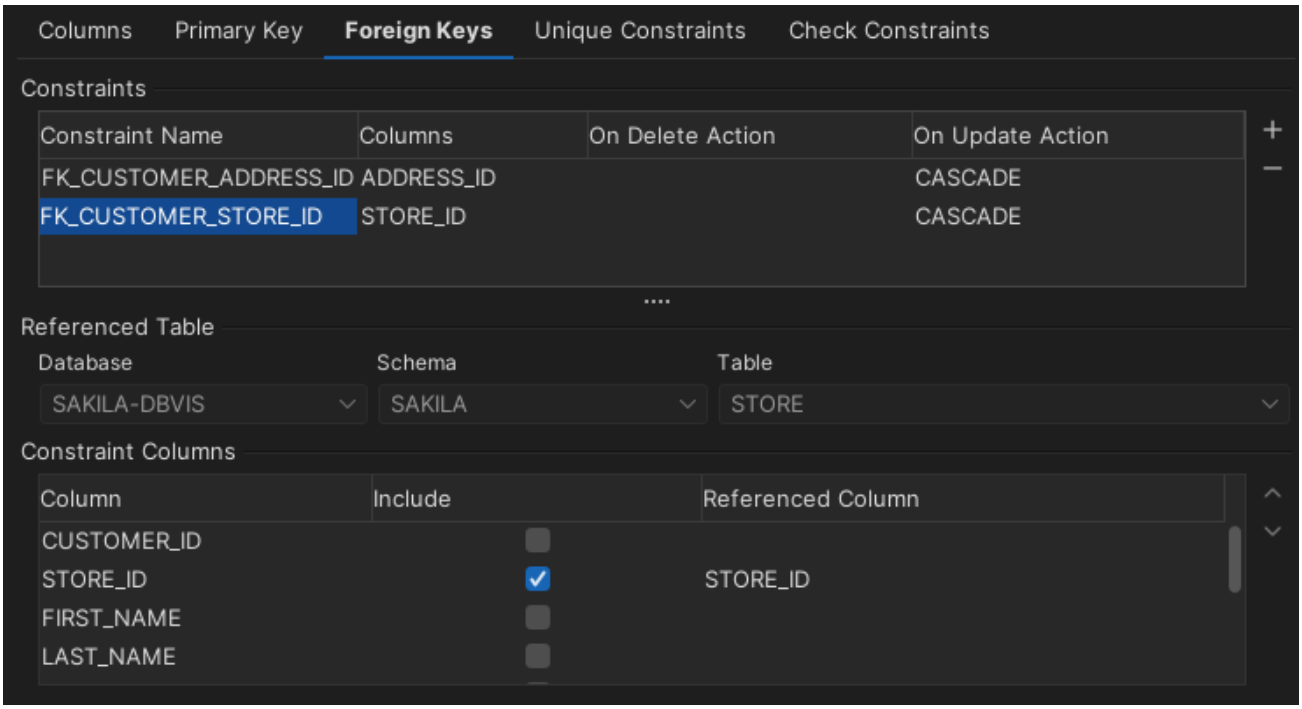



To declare a Primary Key:

1. Optionally enter a constraint name for the primary key constraint in the **Constraint Name** field,
2. Select the column(s) to be part of the primary key by clicking the checkboxes in the **Include** field in the columns list.

## Foreign Keys Tab

In the **Foreign Keys** tab, you can declare one or more foreign keys for the table. A foreign key is a column, or a combination of columns, that refer to the primary key of another table. Foreign keys are used by the database to enforce integrity, i.e., that there is a row in the referenced table with a primary key that matches the foreign key value when a new row is inserted or updated, and you can optionally declare rules for what to do when a referenced primary key is removed or updated in the referenced table.




The tab has the following sections:

- A list of foreign keys,
- Controls for selecting the table the currently selected foreign key refers to, including the database (catalog) and/or schema for the table,
- A list of all columns for the table being created/altered.

To declare a new foreign key constraint:

1. Click the **Add** button next to the list of foreign keys,
2. Optionally enter a name for the foreign key in the first field in the list,
3. Select **On Delete** and **On Update** actions from the pull-down menus. The pull-down lists include all actions that the database support, typically CASCADE, RESTRICT, NO ACTION and SET NULL. The **Columns** field is read-only and gets its value automatically when you select which columns to include in the key later,
4. Use the **Referenced Table** controls to select the table that the foreign key refers to,
5. Check the **Include** checkbox for all columns in the column list that should be part of the foreign key and then select the corresponding column in the referenced table from the pull-down menu in the **Referenced Column** field.

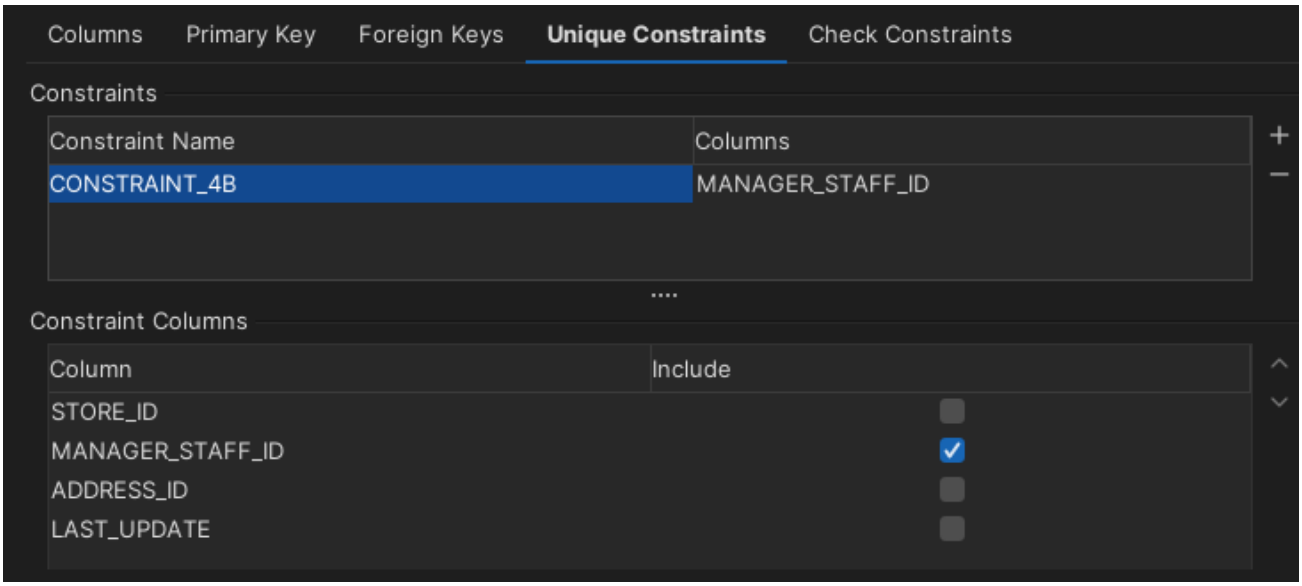
You can change the column order for the key with the **Up** and **Down** buttons.

To remove an existing foreign key:

1. Select the foreign key in the list in the top section,
2. Click the **Remove** button.

## Unique Constraints Tab

The **Unique Constraints** tab is only available for databases that support this constraint type. A unique constraint declares that the columns in the constraint must have unique values in the table.




The top portion of the tab holds a list of all unique constraints, and the lower portion holds a list of all table columns.

To create a constraint:

1. Click the **Add** button,
2. Optionally enter a constraint name in the **Constraint Name** field. The **Columns** field in the constraints list is read-only, filled automatically as you include columns in the constraint,
3. Select the columns to be part of the constraint by clicking the checkboxes in the Include field in the columns list.

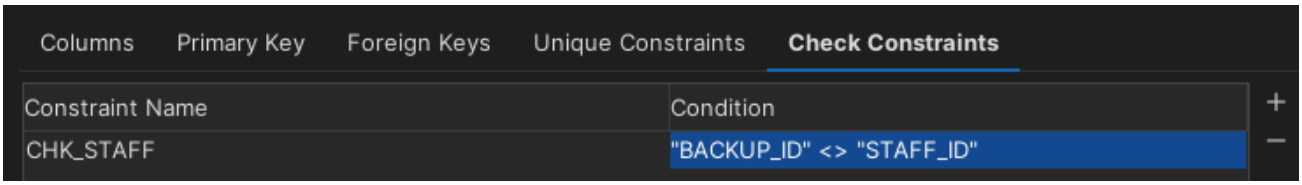
You can change the column order for the constraint with the **Up** and **Down** buttons.

To remove an existing constraint:

1. Select the constraint in the list in the top section,
2. Click the **Remove** button.

## Check Constraints Tab

The **Check Constraints** tab is only available for databases that support this constraint type. A check constraint declares that a column value fulfills a certain condition when a row is inserted or updated. Some databases uses check constraints to enforce nullability rules, so when you alter a table, you may see auto-generated check constraints for columns that you marked as not allowing null values in the **Columns** tab.




To create a check constraint:

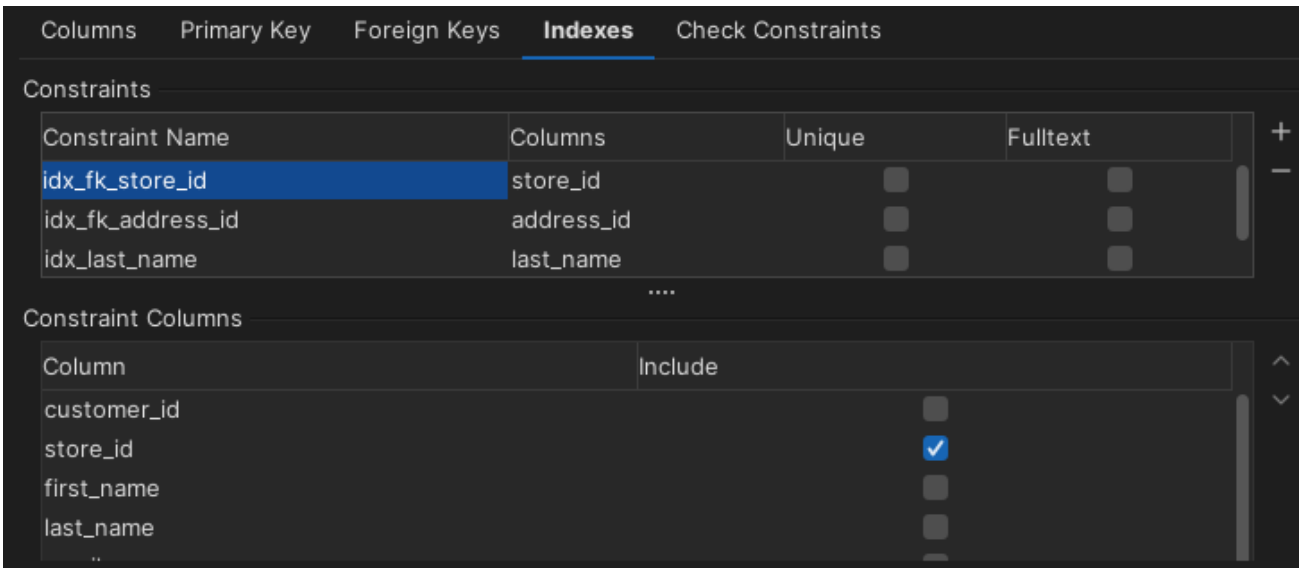
1. Click the **Add** button,
2. Optionally enter a constraint name in the **Constraint Name** field.
3. Enter the condition for the column in the **Condition** field. You can use the same type of conditions as you use in a SELECT WHERE clause.

To remove an existing constraint:

1. Select the constraint in the list,
2. Click the **Remove** button.

## Indexes Tab

The **Indexes** tab is only used for the MySQL database, as a replacement for the **Unique Constraints** tab. The reason is that for MySQL, the CREATE TABLE statement can be used to declare both unique and non-unique indexes. MySQL also does not make a clear distinction between a unique constraint (a rule, most often enforced and implemented as an index by the database) and a unique index (primarily a database structure for speeding up queries, with the side-effect of ensuring unique column values), as most other databases do.




The top portion of the tab holds a list of all indexes, and the lower portion holds a list of all table columns.

To create an index:

1. Click the **Add** button,
2. Optionally enter a name in the **Constraint Name** field. The **Columns** field in the constraints list is read-only, filled automatically as you include columns in the constraint,
3. If you want the index columns to have unique values for all rows in the table, click the checkbox in the **Unique** field,
4. Select the columns to be part of the index by clicking the checkboxes in the **Include** field in the columns list.

You can change the column order for the index with the **Up** and **Down** buttons.

To remove an existing index:

1. Select the index in the list in the top section,
2. Click the **Remove** button.

## SQL Preview

The **SQL Preview** area is updated automatically to match the edits made in the assistant. To show the SQL Preview area check the **Show SQL** checkbox. The preview is read only, but you can copy the SQL to the SQL Commander and flip between formatted and unformatted views using the corresponding choices in the preview area right-click menu.

## Execute

When you are satisfied with the table declaration, click the **Execute** button to create it.

# Altering a Table

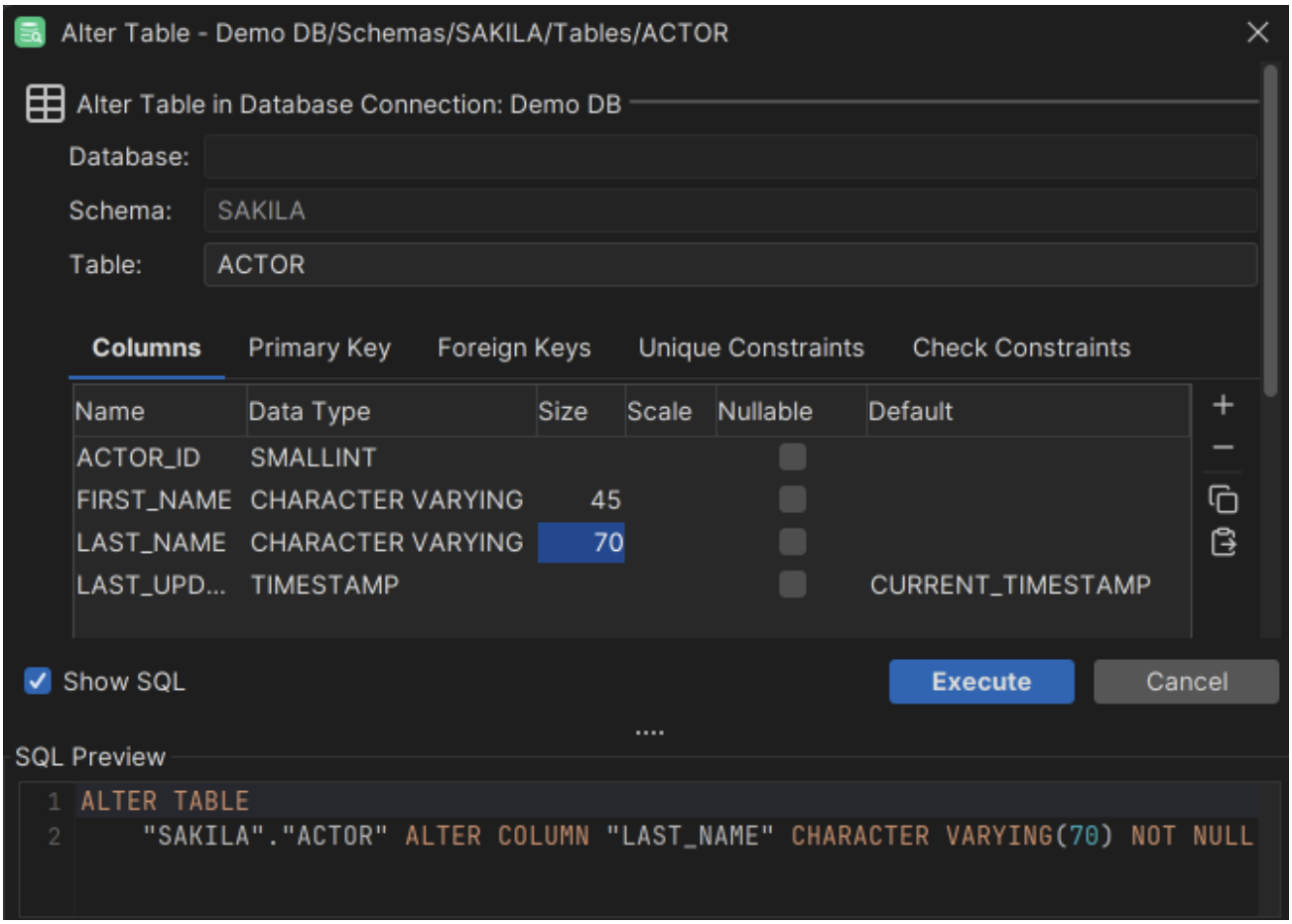
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The Alter Table dialog helps you alter a table without writing SQL.

## Opening the Alter Table Dialog

To alter an existing table:

1. Locate the table node in the **Databases** tab tree,
2. Select the table node and open the **Alter Table** dialog from the right-click menu.



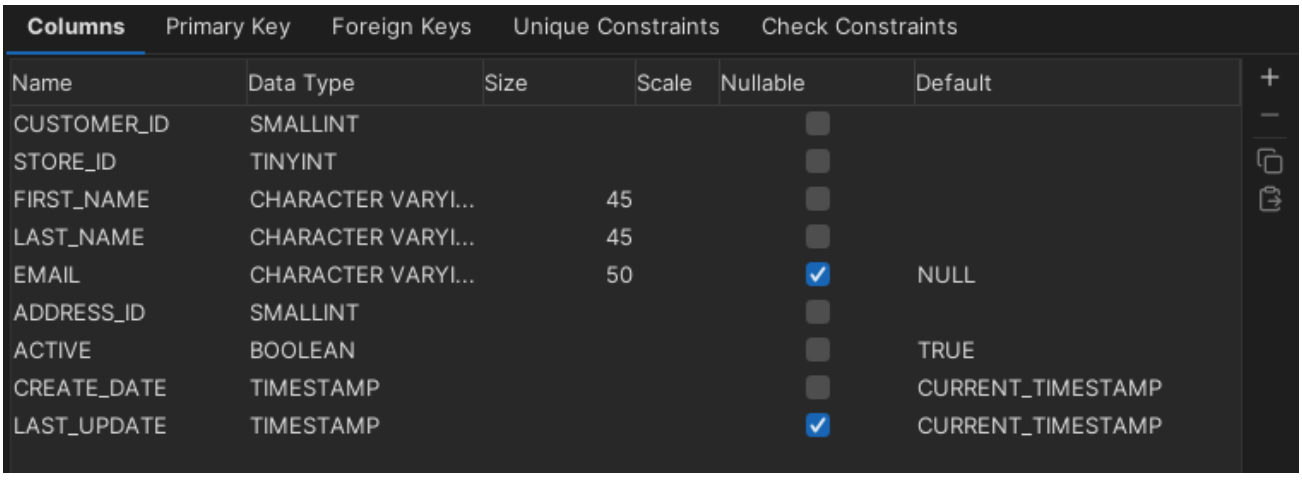
The **Alter Table** dialog is organized in three areas from the top:

- General Table Info Specifies the owning database connection, database and/or schema, and table name. These are picked up from the selection in the tree when the assistant is started.
- Table Details A number of tabs where you specify information about the columns and, optionally, various constraints. The **Columns**, **Primary Key** and **Foreign Key** tabs are available for all databases. The remaining tabs are database-specific and depends on the features supported by the database engine.
- SQL Preview The SQL previewer shows the SQL statements for altering the table based on your input.

> The controls, such as the fields, pull-down menus and buttons, in the assistant are only enabled if the ALTER TABLE statement for the database holding the table provides a way to alter the corresponding table attribute. For instance, for a database that only allows the size of a VARCHAR column to be altered, the **Size** field in the **Columns** tab is disabled for all columns with other data types. If you find that you can not make the change you want, it is because the ALTER TABLE statement does not allow that change to be made.

## Columns Tab

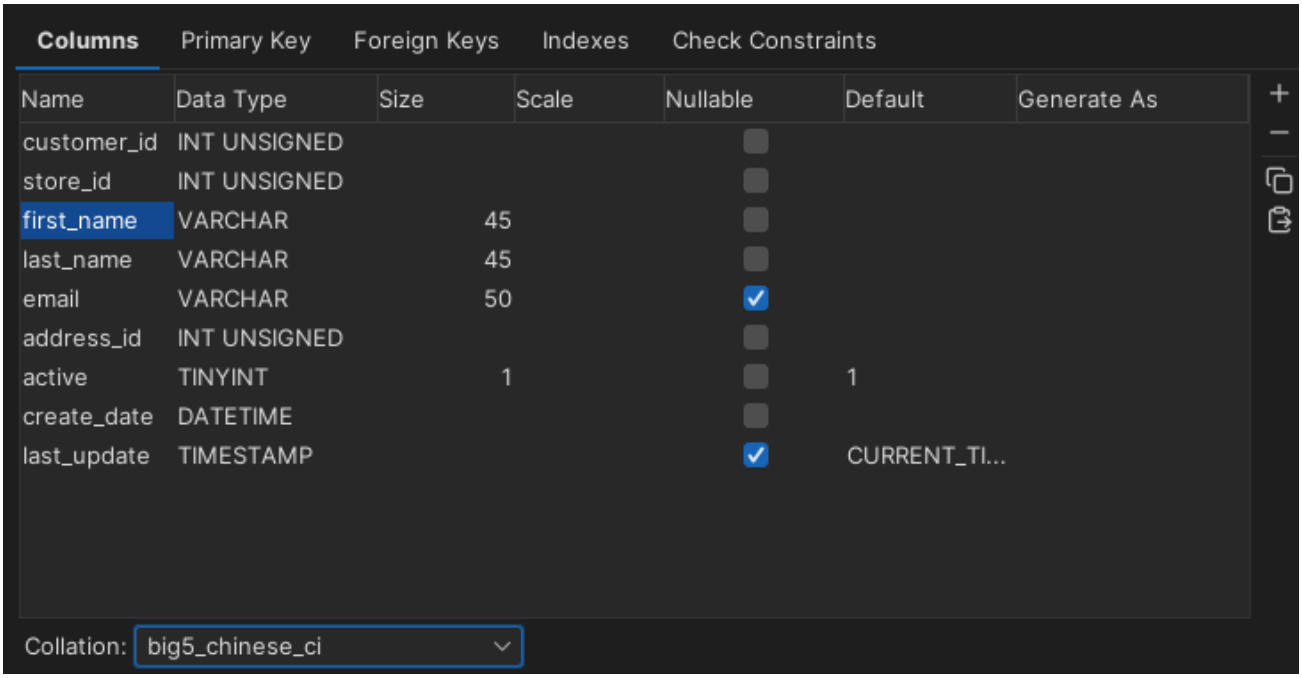
The **Columns** tab lists all table columns along with their attributes. The actual columns of the Columns tab is dependant on database. Additional columns may be shown for other databases. An example is Oracle and MariaDB which supports invisible columns and thus a column for setting this is shown.



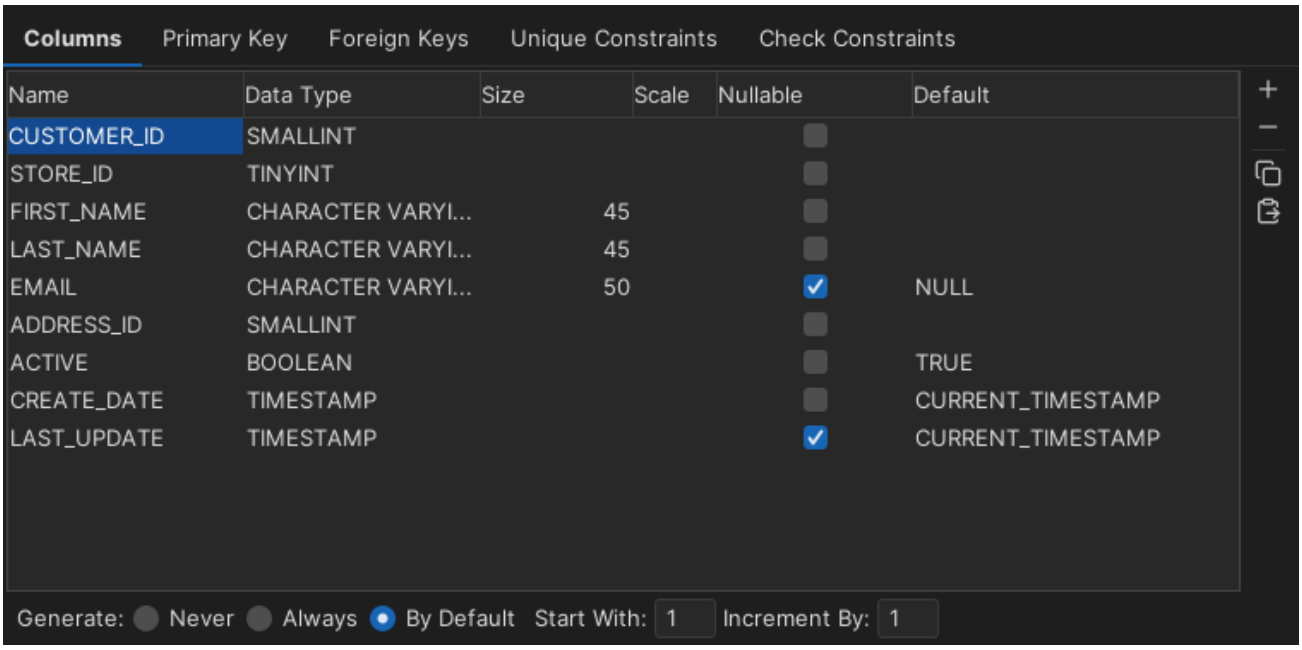
To add a column:

1. Click the **Add** button,
2. Enter the name of the column in the first field and select a data type from a drop down list in the second field, or start typing the data type name to find it and select it with the **Enter** key. The list contains the names of all data types the database supports,
3. For some data types, such as character types, you may also specify a size, i.e., the maximal length of the value. For others, like the decimal types, you can may specify both a size and a scale (the maximal number of decimals),
4. In the last two fields, specify if the table is nullable and a default value to use for rows inserted into the table without specifying a value for the column.

You may find additional fields depending on the features supported by the database you are working with and the data type for the current column. The **Collation** field is shown for character columns if the database supports the declaration of a collation for textual data.




Settings for generated fields are shown if the database supports automatically inserted values, typically to insert the next available sequence number in a numeric column.




The **Create/Alter Table** dialog uses database metadata to try to enable only the fields that apply to the selected data type, but please note that it is not always possible. For instance, there is no metadata available to tell if a data type requires, or allows, a size. If you don't enter a required attribute or enter an attribute that is unsupported for a data type, you will get an error message when you click **Execute** to create/alter the table.

To remove a column:

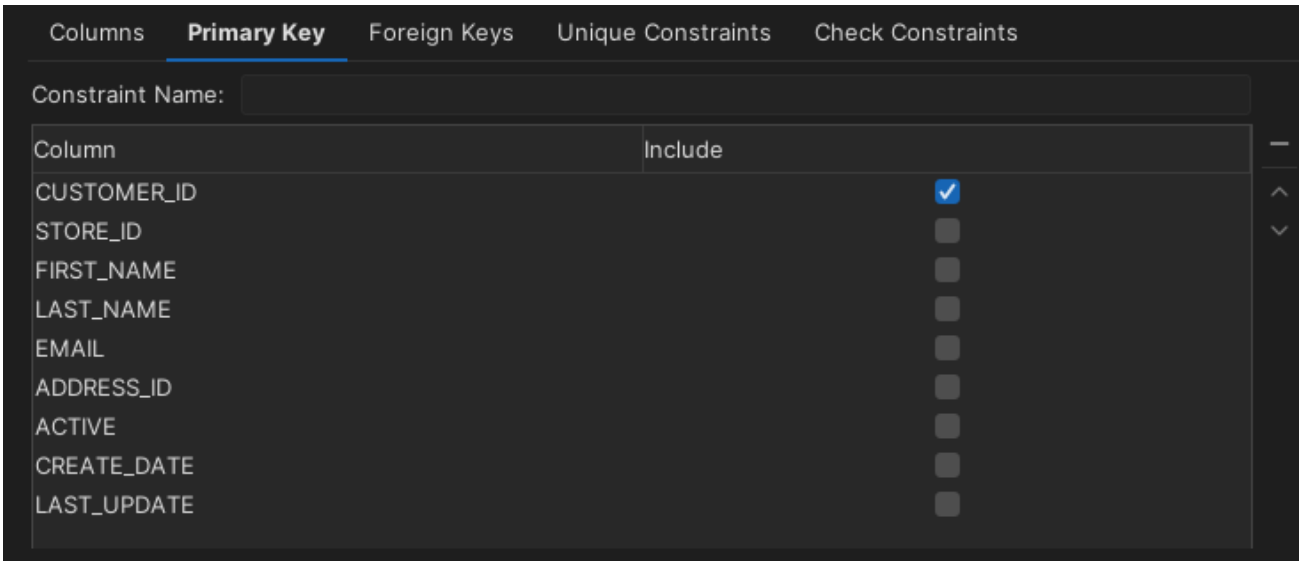
1. Select a cell in the column row,
2. Click the **Remove** button.

To move a column to another location (Only supported for Create Table):

1. Select a cell in the column row,
2. Click the **Up** or **Down** buttons.

## Primary Key Tab

The **Primary Key** tab contains information about an optional primary key for the table. A primary key is a column, or a combination of columns, that uniquely identifies a row in a table.

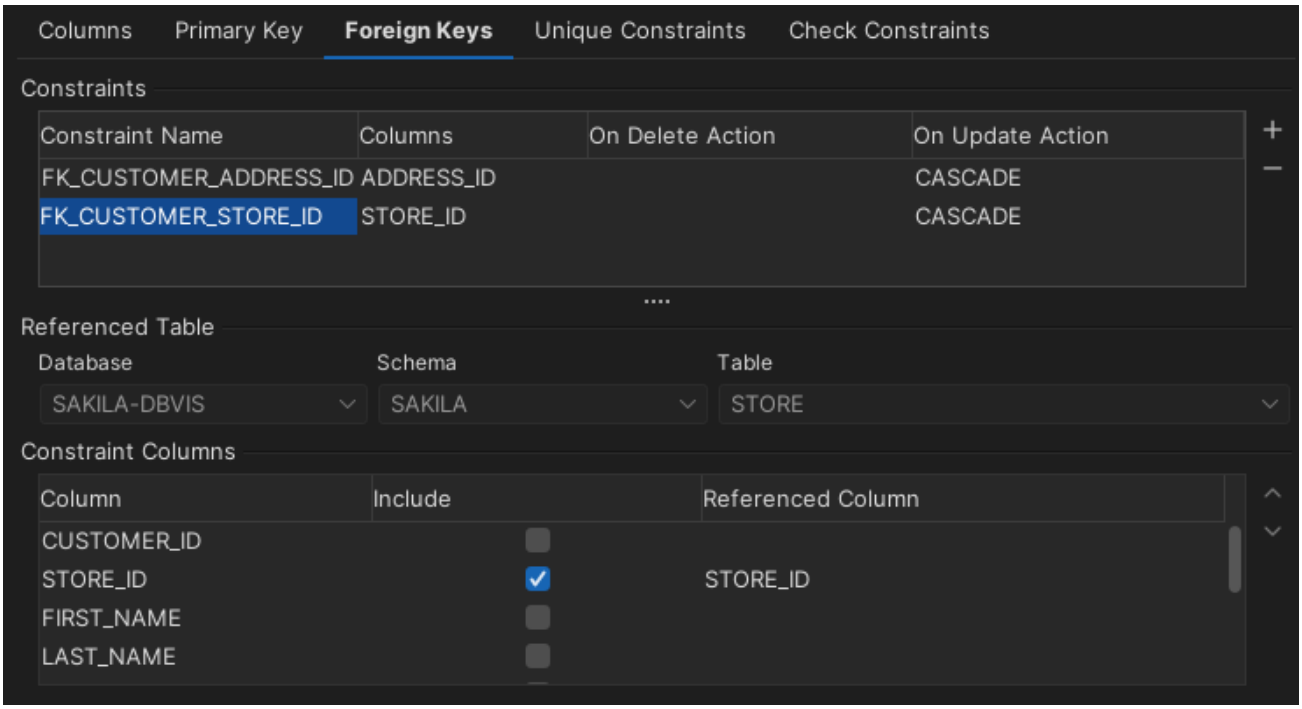



To declare a Primary Key:

1. Optionally enter a constraint name for the primary key constraint in the **Constraint Name** field,
2. Select the column(s) to be part of the primary key by clicking the checkboxes in the **Include** field in the columns list.

## Foreign Keys Tab

In the **Foreign Keys** tab, you can declare one or more foreign keys for the table. A foreign key is a column, or a combination of columns, that refer to the primary key of another table. Foreign keys are used by the database to enforce integrity, i.e., that there is a row in the referenced table with a primary key that matches the foreign key value when a new row is inserted or updated, and you can optionally declare rules for what to do when a referenced primary key is removed or updated in the referenced table.




The tab has the following sections:

- A list of foreign keys,
- Controls for selecting the table the currently selected foreign key refers to, including the database (catalog) and/or schema for the table,
- A list of all columns for the table being created/altered.

To declare a new foreign key constraint:

1. Click the **Add** button next to the list of foreign keys,
2. Optionally enter a name for the foreign key in the first field in the list,
3. Select **On Delete** and **On Update** actions from the pull-down menus. The pull-down lists include all actions that the database support, typically CASCADE, RESTRICT, NO ACTION and SET NULL. The **Columns** field is read-only and gets its value automatically when you select which columns to include in the key later,
4. Use the **Referenced Table** controls to select the table that the foreign key refers to,
5. Check the **Include** checkbox for all columns in the column list that should be part of the foreign key and then select the corresponding column in the referenced table from the pull-down menu in the **Referenced Column** field.

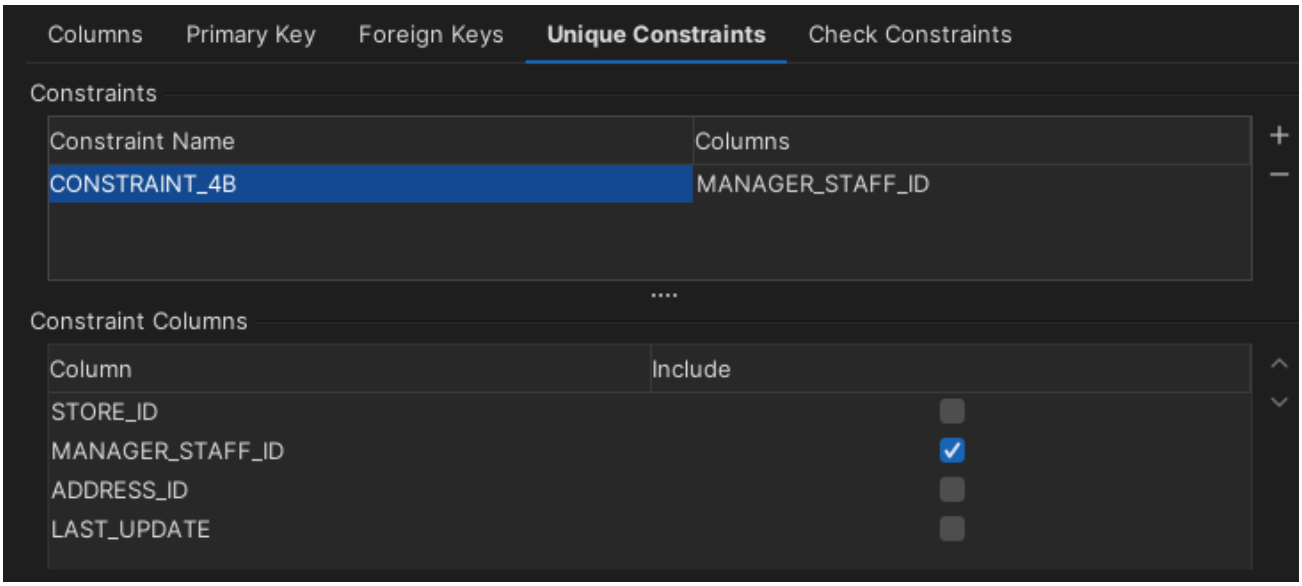
You can change the column order for the key with the **Up** and **Down** buttons.

To remove an existing foreign key:

1. Select the foreign key in the list in the top section,
2. Click the **Remove** button.

## Unique Constraints Tab

The **Unique Constraints** tab is only available for databases that support this constraint type. A unique constraint declares that the columns in the constraint must have unique values in the table.




The top portion of the tab holds a list of all unique constraints, and the lower portion holds a list of all table columns.

To create a constraint:

1. Click the **Add** button,
2. Optionally enter a constraint name in the **Constraint Name** field. The **Columns** field in the constraints list is read-only, filled automatically as you include columns in the constraint,
3. Select the columns to be part of the constraint by clicking the checkboxes in the Include field in the columns list.

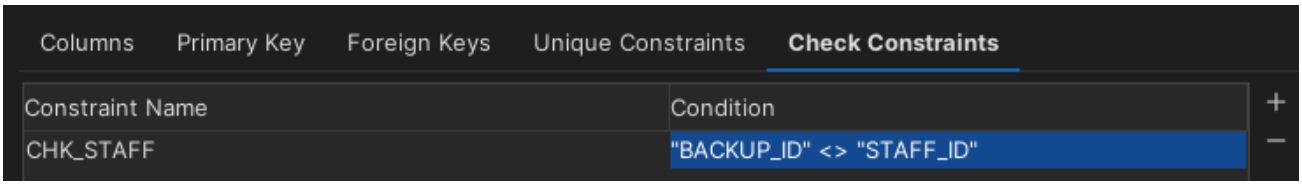
You can change the column order for the constraint with the **Up** and **Down** buttons.

To remove an existing constraint:

1. Select the constraint in the list in the top section,
2. Click the **Remove** button.

## Check Constraints Tab

The **Check Constraints** tab is only available for databases that support this constraint type. A check constraint declares that a column value fulfills a certain condition when a row is inserted or updated. Some databases uses check constraints to enforce nullability rules, so when you alter a table, you may see auto-generated check constraints for columns that you marked as not allowing null values in the **Columns** tab.




To create a check constraint:

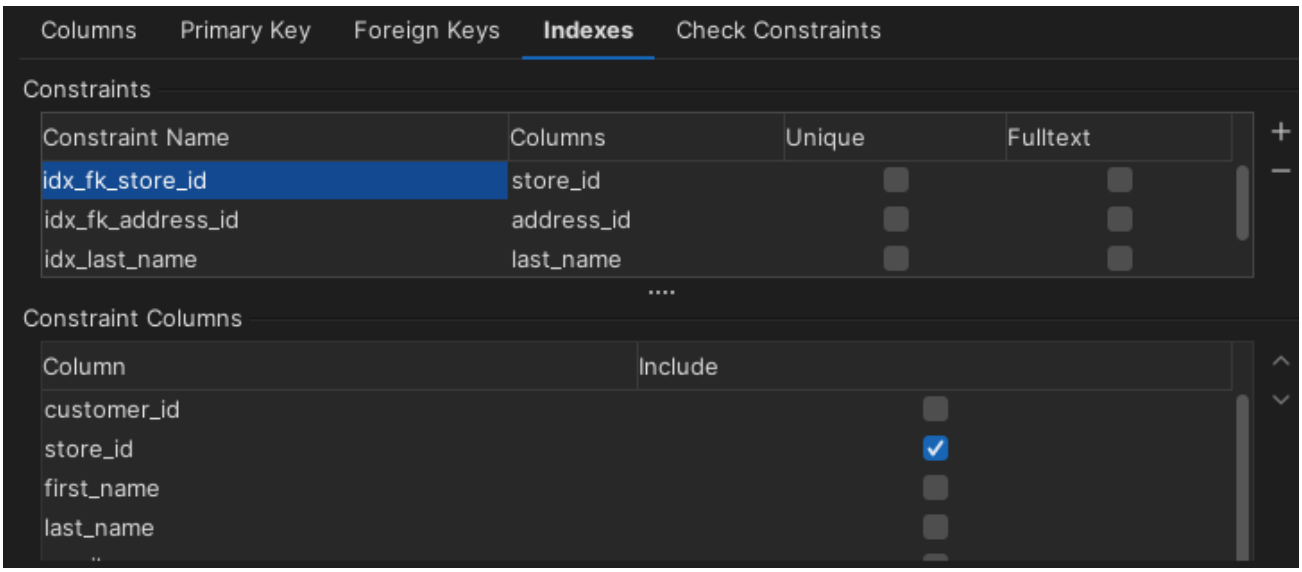
1. Click the **Add** button,
2. Optionally enter a constraint name in the **Constraint Name** field.
3. Enter the condition for the column in the **Condition** field. You can use the same type of conditions as you use in a SELECT WHERE clause.

To remove an existing constraint:

1. Select the constraint in the list,
2. Click the **Remove** button.

## Indexes Tab

The **Indexes** tab is only used for the MySQL database, as a replacement for the **Unique Constraints** tab. The reason is that for MySQL, the CREATE TABLE statement can be used to declare both unique and non-unique indexes. MySQL also does not make a clear distinction between a unique constraint (a rule, most often enforced and implemented as an index by the database) and a unique index (primarily a database structure for speeding up queries, with the side-effect of ensuring unique column values), as most other databases do.




The top portion of the tab holds a list of all indexes, and the lower portion holds a list of all table columns.

To create an index:

1. Click the **Add** button,
2. Optionally enter a name in the **Constraint Name** field. The **Columns** field in the constraints list is read-only, filled automatically as you include columns in the constraint,
3. If you want the index columns to have unique values for all rows in the table, click the checkbox in the **Unique** field,
4. Select the columns to be part of the index by clicking the checkboxes in the **Include** field in the columns list.

You can change the column order for the index with the **Up** and **Down** buttons.

To remove an existing index:

1. Select the index in the list in the top section,
2. Click the **Remove** button.

## SQL Preview

The **SQL Preview** area is updated automatically to match the edits made in the assistant. To show the SQL Preview area check the **Show SQL** checkbox. The preview is read only, but you can copy the SQL to the SQL Commander and flip between formatted and unformatted views using the corresponding choices in the preview area right-click menu.

## Execute

When you are satisfied with the alterations, click the **Execute** button to create it.

# Creating a Trigger

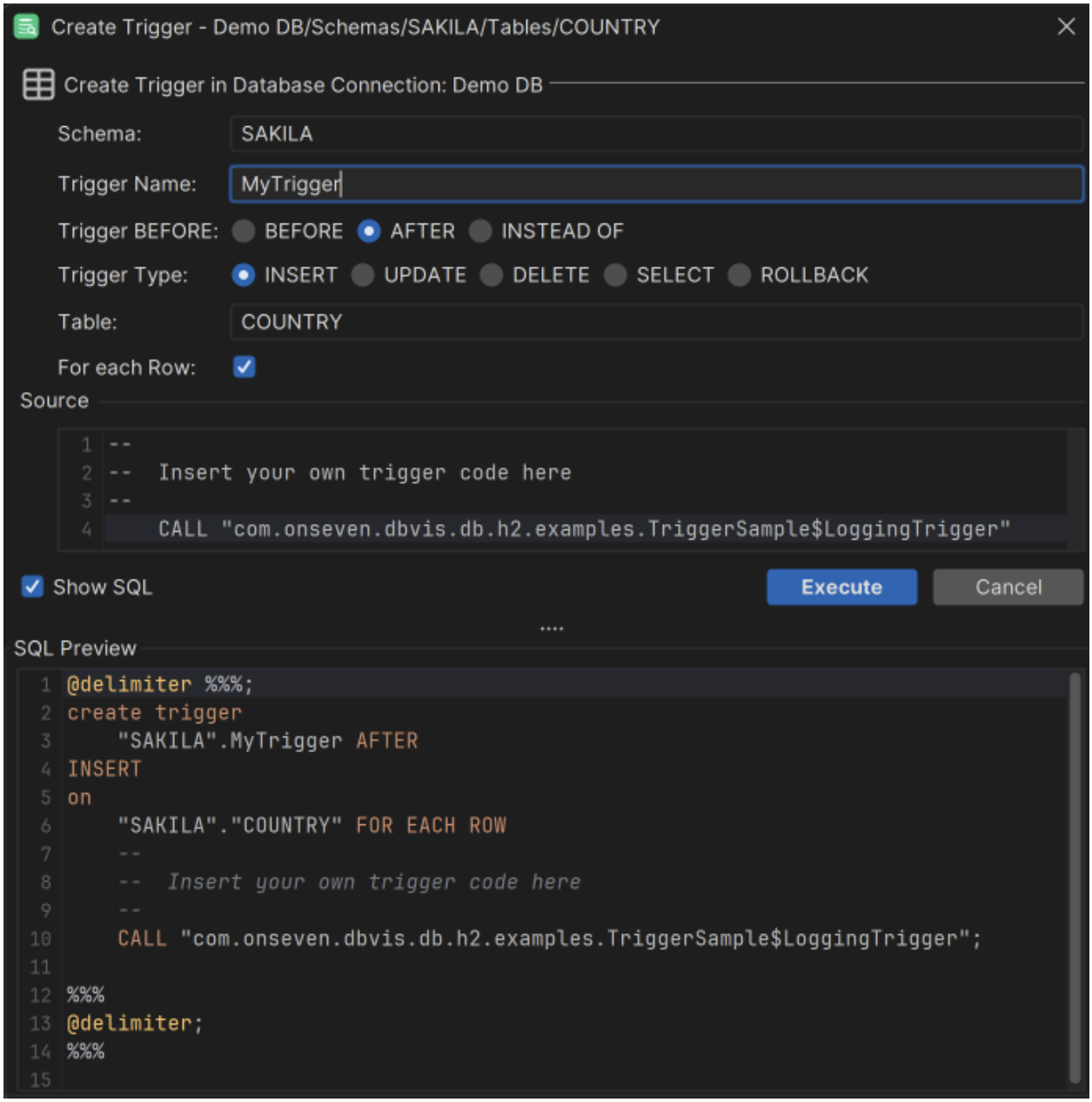
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The Create Trigger dialog helps you create a trigger for a table.

## Opening the Create Trigger Dialog

To create a trigger for a table:

1. Locate the table in the **Databases** tab tree,
2. Select the table node and open the **Create Trigger** dialog from the right-click menu,




3. Enter the required info in the fields, e.g. trigger name. The fields are database dependent so the figure is just an example,
4. The **Source** area contains stub code that you can later edit in the Trigger Editor. For most databases you can leave it as is, but for some databases, you must adjust the stub code to match your database objects.
5. Click the **Execute** button to create the trigger

## Trigger Editor

To edit the trigger code:

1. Expand the **Trigger** node for the table in the Databases tab tree,
2. Double-click the trigger node to open its **Object View** tab,
3. Open the **Trigger Editor** tab and edit the code in the editor,
4. Click the **Save** toolbar button to save (and for some databases, compile) the trigger.

If the database reports any errors, the location of the errors are highlighted with curly red underlines in the editor for most databases. Hovering the mouse over such an underline shows the error message.

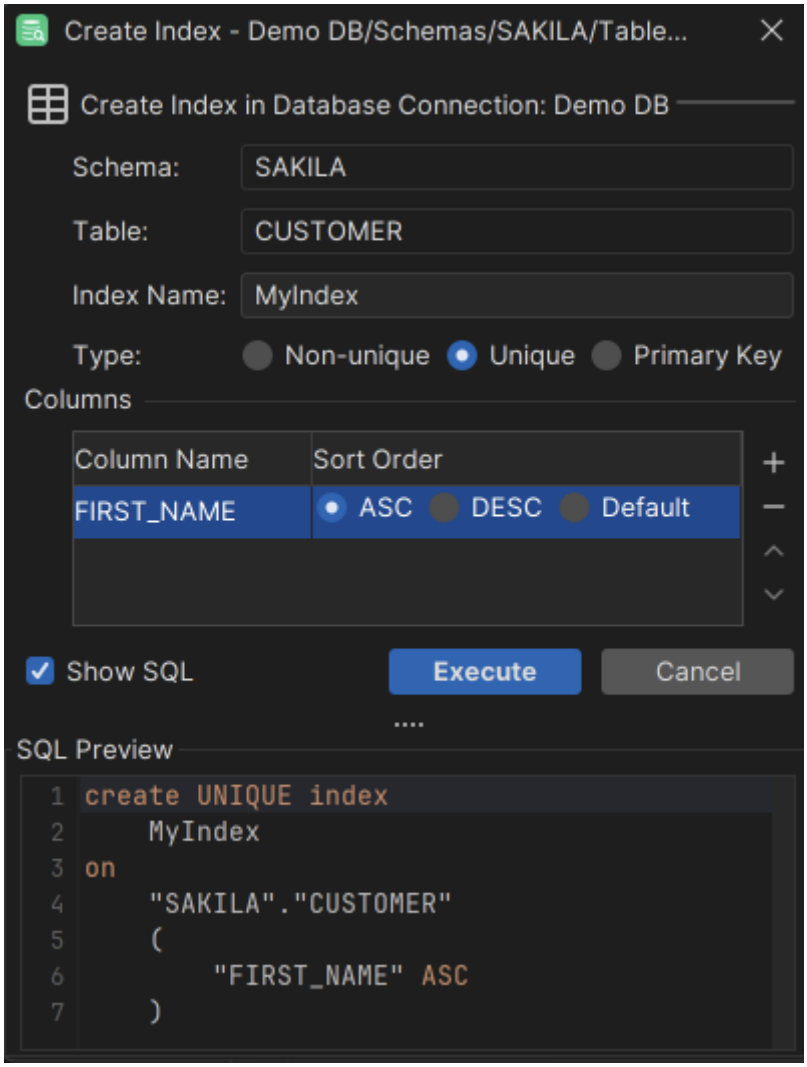
The **Log** tab in the results area also lists all errors. Clicking on the icon next to an error message selects the corresponding line and positions the caret at the error location, if the database reports error locations.

# Creating an Index

The Create Index dialog helps you create an index for a table without writing SQL.

To create an index for a table:

1. Locate the table in the **Databases** tab tree,
2. Select the table node and open the **Create Index** dialog from the right-click menu,




3. Enter the required info in the fields, e.g. index name. The fields are database dependent so for some databases there are additional fields compared to the figure,
4. Click the **Add** button in the **Columns** area to add an index column,
5. Select the column to index from the **Column Name** drop down list,
6. Select the sort order for the index column from the **Sort Order** radio buttons,
7. Click the **Execute** button to create the index.

To remove an index column:

1. Select the column row,
2. Click the **Remove** button.

To move an index column

1. Select the column row,
2. Click the **Up** and **Down** button to move it.

# Viewing Table Data

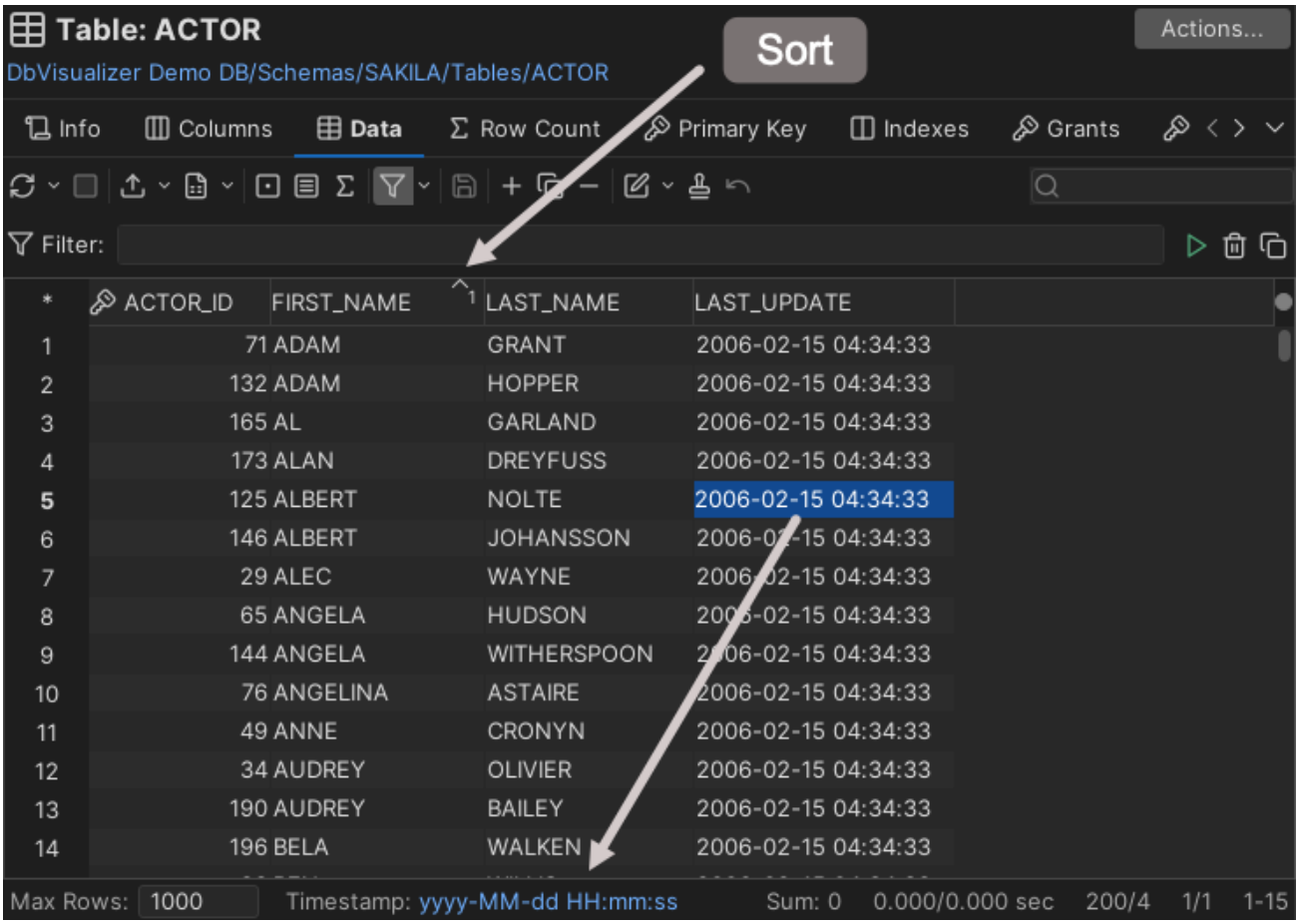
> Some features and screenshots in this section are for the DbVisualizer Pro edition.

A table's data can be viewed in various ways in the **Data** tab in its **Object View** tab. For information about how to **edit** table data in DbVisualizer, please check Editing Table Data.

## Opening the Data tab

To open the Data tab for a table:

1. Locate the table in the **Databases** tab,
2. Double-click the table node to open its **Object View** tab,
3. Open the **Data** sub tab.



Each column width is automatically resized to match the column width, including the column header, by default. You can disable this behavior in the the Tool Properties dialog, in the **Grid** category under the General tab.

If **Auto Resize Column Widths** is enabled, the **Max Column Width** setting can be used to limit the column width so that an extremely wide column does not take up all space.

In the same Tool Properties category, you can also disable **Show Grid Row Header**, i.e. the row number shown to the left of the data rows, for read-only grids such as the Data tab in the DbVisualizer Free edition and result sets from joined tables.

The column headers correspond to the column names by default, but you can specify in the Tool Properties dialog, in the Grid category under the General tab, that you like to use the column alias instead. This is mostly useful for grids representing SQL Commander result sets, but may also be useful in the Data tab grid for some databases.

The **Data** tab contains a number of features for locating and focusing on just the data of interest as described in the following sections.

## Sorting

You can sort the data grid based on the values in one or more columns:

1. Click on a column header to sort the grid in ascending order on the values in that column, indicated by an up-arrow in the column header.
2. Click the same column header again to sort in descending order, indicated by a down-arrow in the column header.
3. Click a third time to show the data in the order it was received from the database. This removes the sort indicator.

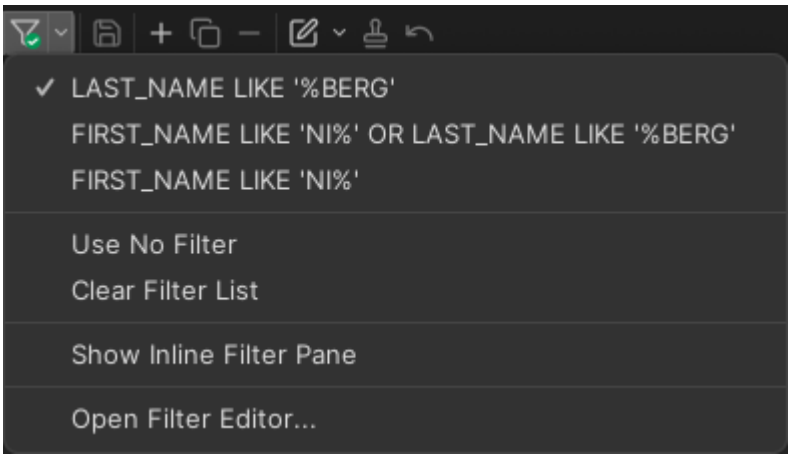
To sort on more than one column, Ctrl-click (keep the Ctrl key pressed when clicking) on additional columns. The grid is then sorted on the values in the first column you clicked on (indicated with a 1 next to the arrow), and then all rows with the same value in the first column are sorted on the values in the second sort column (indicated with a 2 next to the arrow), and so on.

## Formatting

Note that when a cell is selected, the format pattern used to display the cell value is visible at bottom of the grid. By clicking the blue link, the Data Formats section of **Tool Properties** will be opened.

## Where Filter

You can use the filter capability in the **Data** tab to limit the number of rows shown in the grid, using the same syntax as for an SQL WHERE clause. The Filter menu button in the grid toolbar contains all operations related to using a filter.

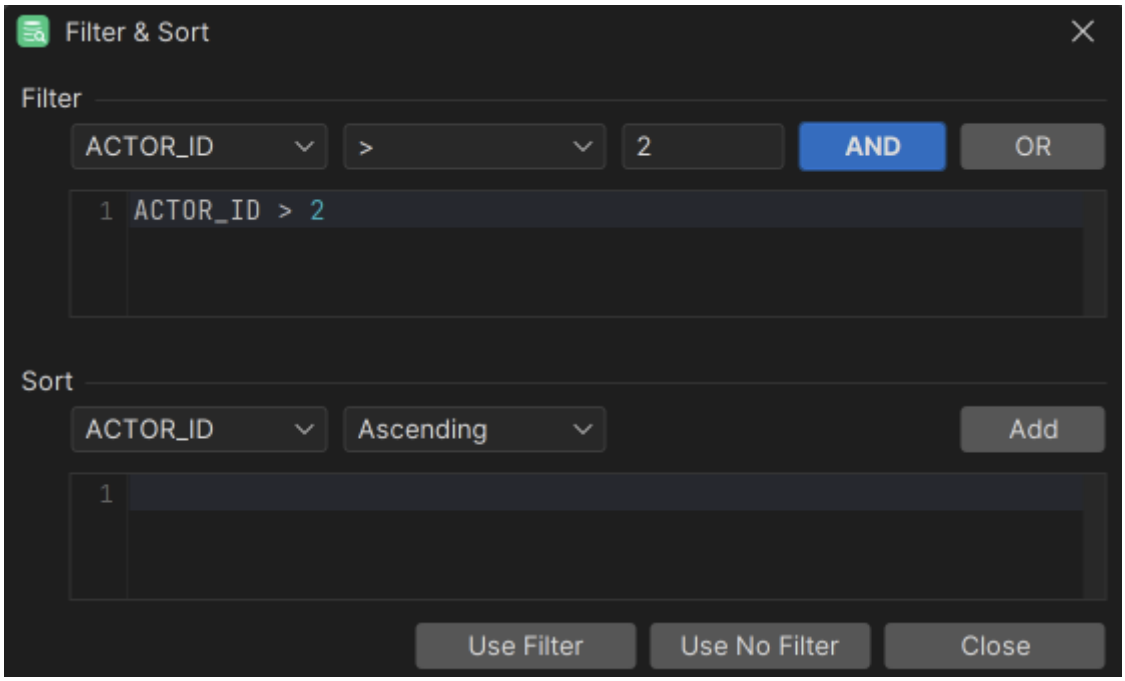



The top entries in the menu are previously used filters for the table, if any. The checkbox is selected for the filter that is currently in use. The filters are saved between DbVisualizer sessions, and you can toggle between them by selecting them from the menu. The maximum number of filters to save is specified in the Tool Properties dialog, in the **Table Data** category under the General tab.

You use the **Use No Filter** choice to disable all filters for the table, and the **Clear Filter List** to permanently remove all filters for the table.

To create a new filter:

1. Select **Open Filter Editor** to launch the **Filter & Sort Configuration** dialog,

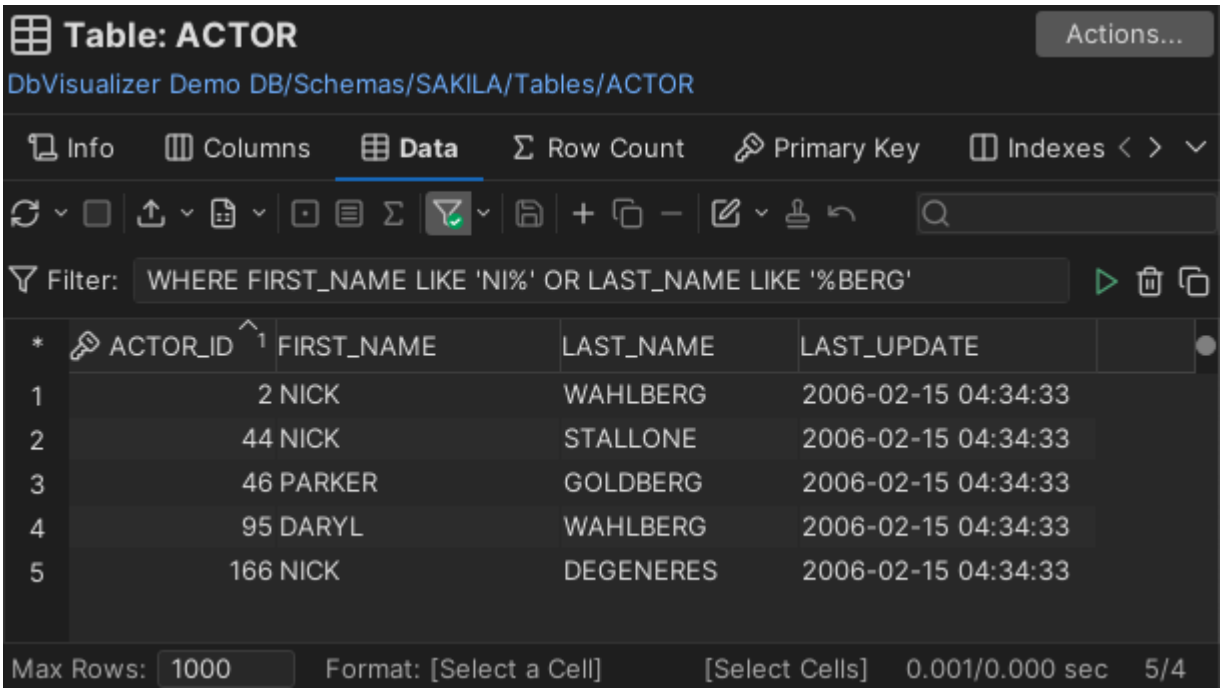



2. Select a column name, an operation and the value for the condition using the controls in the **Filter** area,
3. Add the condition to the filter by clicking the **AND** or **OR** button, and create additional conditions in the same way if needed,
4. Click the **Use Filter** button to apply the filter, save it and close the dialog, or close the dialog without applying and saving the filter by clicking the **Close** button.

You can use **Ctrl-Enter** while editing the value field to reload the grid with that single condition applied, or in the editor to reload the grid based on all filter conditions created so far.

The **Sort** area is similar to the **Filter** area. You can select column names and sort order from the two lists, and click the **Add** button to add the sort criteria for the single column to the complete criteria.

If you often need to tweak the filter conditions and want a more compact user interface, you can use the inline filter view. Use the **Show/Hide Filter Area** choice in the **Filter** menu to toggle the visibility of the inline filter.




## Column Filter

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The **Column Filter** acts on the data that is already loaded in the grid, as opposed to a Where Filter which is used to limit the number of rows fetched from the database. With a **Column Filter**, you can easily list only those rows having a column value matching the **Column Filter.** By customizing the filter it is possible to add more complex filters such as only listing rows where the column value does not contain a certain value or substring.

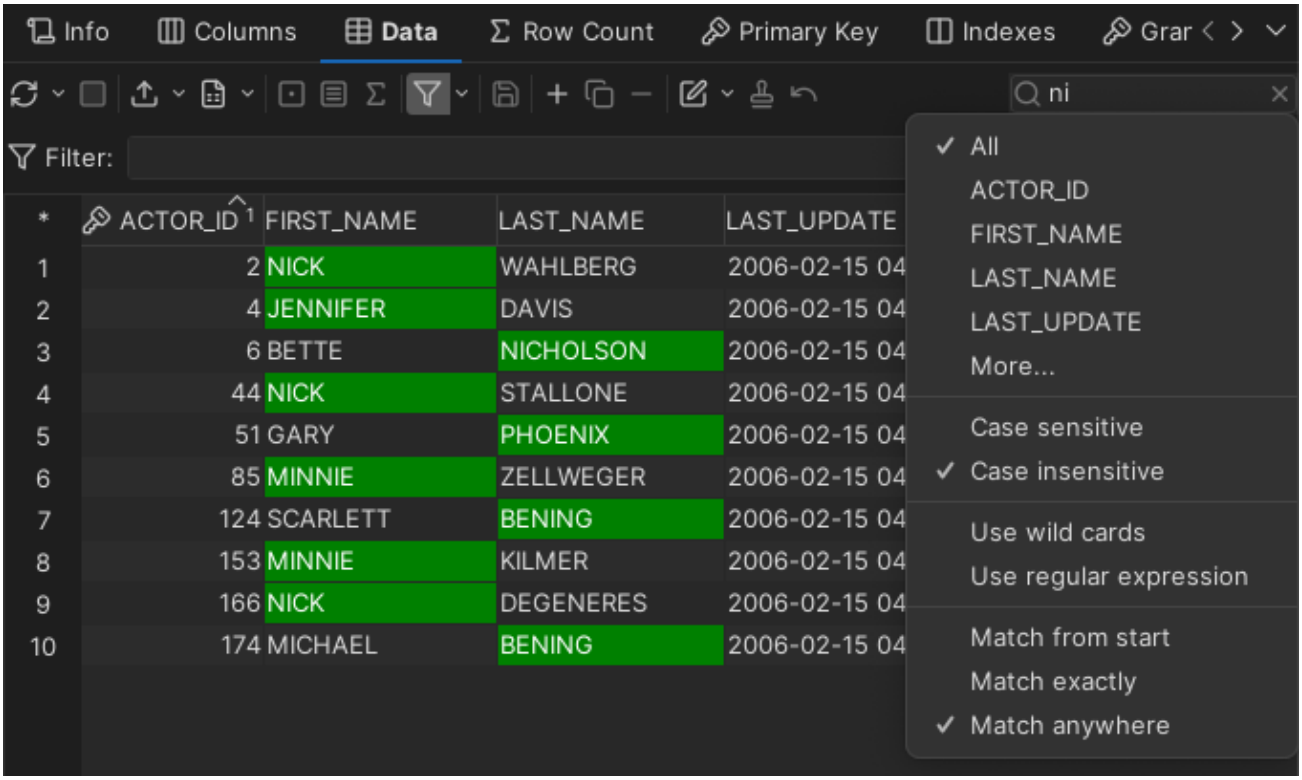
A **Column Filter is added** by clicking the right part of the column header. A column having a Column Filter is indicated by the presence of a filter icon in the column header.

It is possible to to add **Column Filters** to multiple columns. For cases where you don't care which column has a specific value we recommend the Quick Filter

## Quick Filter

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The **Quick Filter** acts on the data that is already loaded in the grid, as opposed to a Where Filter which is used to limit the number of rows fetched from the database. With a **Quick Filter**, you can easily list only those rows in the grid that match the entered search string.




Use the Quick Filter pull-down menu (click on the magnifying glass) to choose if the filter should match cells in all columns or just one selected column, case or case insensitive matching, and where in the cell the value must match.

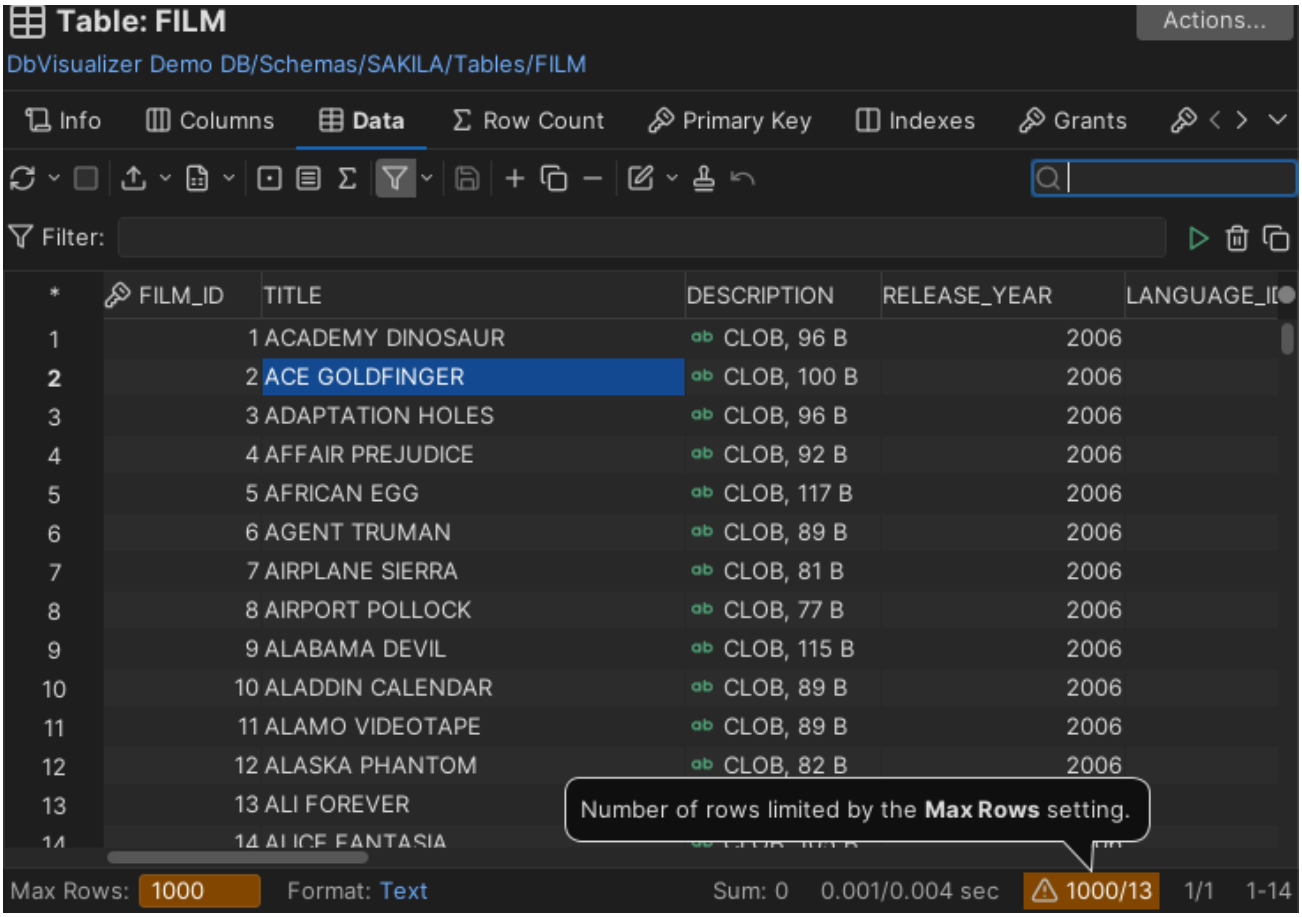
For the **Use wild cards** option the following characters have special meaning:

`?` - The question mark indicates there is zero or one of the preceding element. For example, colou?r matches both "color" and "colour". `*` - The asterisk indicates there are zero or more of the preceding element. For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on. `+` - The plus sign indicates that there is one or more of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".

## Max Rows and Max Chars settings

### Max Rows setting

DbVisualizer by default limits the number of rows shown in the Data tab to 20,000 rows to conserve memory. The following is displayed when this limit is reached.




If the max rows limit prevents you from seeing the data of interest, you should first consider:

1. Using a Where Filter to only retrieve the rows of interest,
2. Exporting the table to a file.

If you really need to look at more than 20,000 rows, you can change the value in the **Max Rows** field in the grid status bar. Use a value of 0 or -1 to get all rows, or a specific number (e.g., 5000) to set a new limit.

MAX ROWS AT FIRST DISPLAY

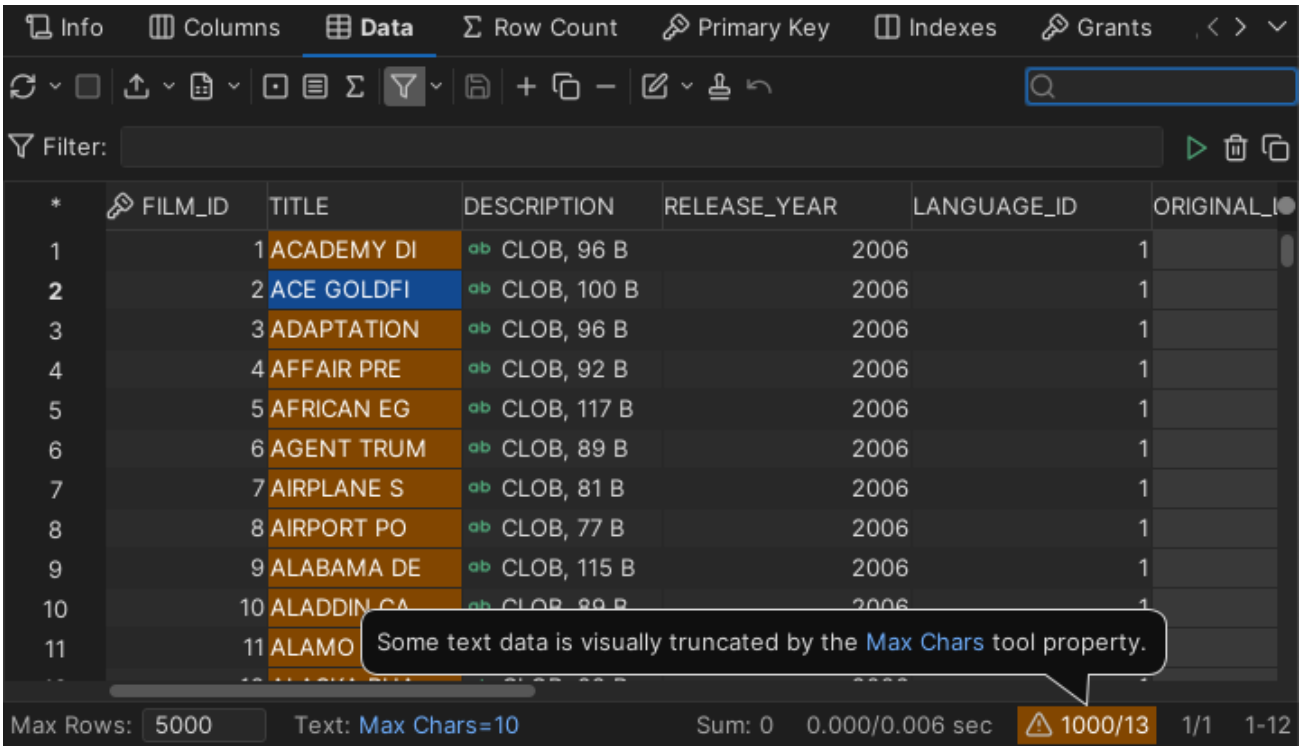
By default, opening the Data tab for a table loads all rows, unless there is a **Max Rows** limit. If you have very large tables and don't want to risk memory issues if you accidentally open the Data tab and have no Max Rows limit, you can specify a **Max Rows at First Display** limit. You do this in the Tool Properties dialog, under the **General** / **Table Data** category.

The default is -1, which means no limit. If you set it to a positive number, only the specified number of rows are loaded when the Data tab is first opened for a table. To load more rows, up to the Max Rows setting in the Data tab, click the **Reload** button in the Data tab toolbar.

## Max Chars setting

Text values presented in the Data tab grid may be very long, potentially consuming lots of memory and affecting performance. By default, DbVisualizer visually truncates text values at 1,√000 characters, and when it happens, there

is an indicator below the grid. The **Max Chars** setting can be modified in the **Tool Properties** / **Data Formats** category or by clicking on any of the **Max Chars** links in the screenshot below.

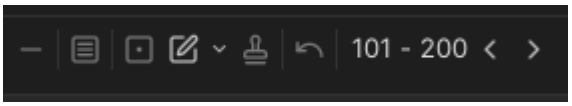



Along with the highlighted field, a warning pops up close to the field. You can disable this behavior in the Tool Properties dialog under the **General** / **Grid** category.

> To view the full text of a truncated text value, double-click on it to open in the Value viewer

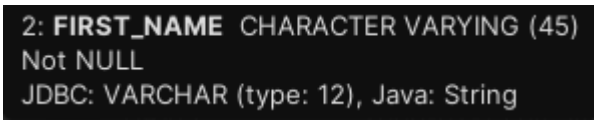
## Pagination

For some databases, DbVisualizer supports navigating the data set page by page. When this is possible, a paging section is added to the toolbar allowing to navigate between the pages. The page size is determined by the **Max Rows** setting.



## Column Header Tooltips

The column header tooltip shows data type information about the column. To see the tooltip, let the mouse hover over the column header. The tooltip pops up in about a second.

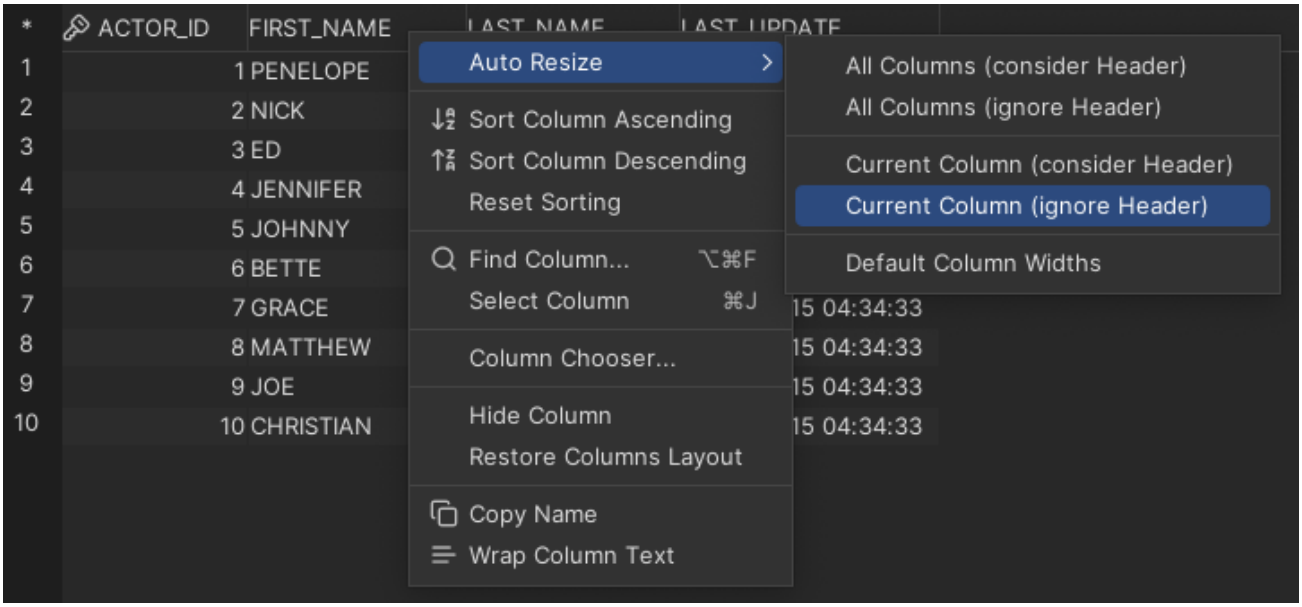



## Highlight Primary Key Columns

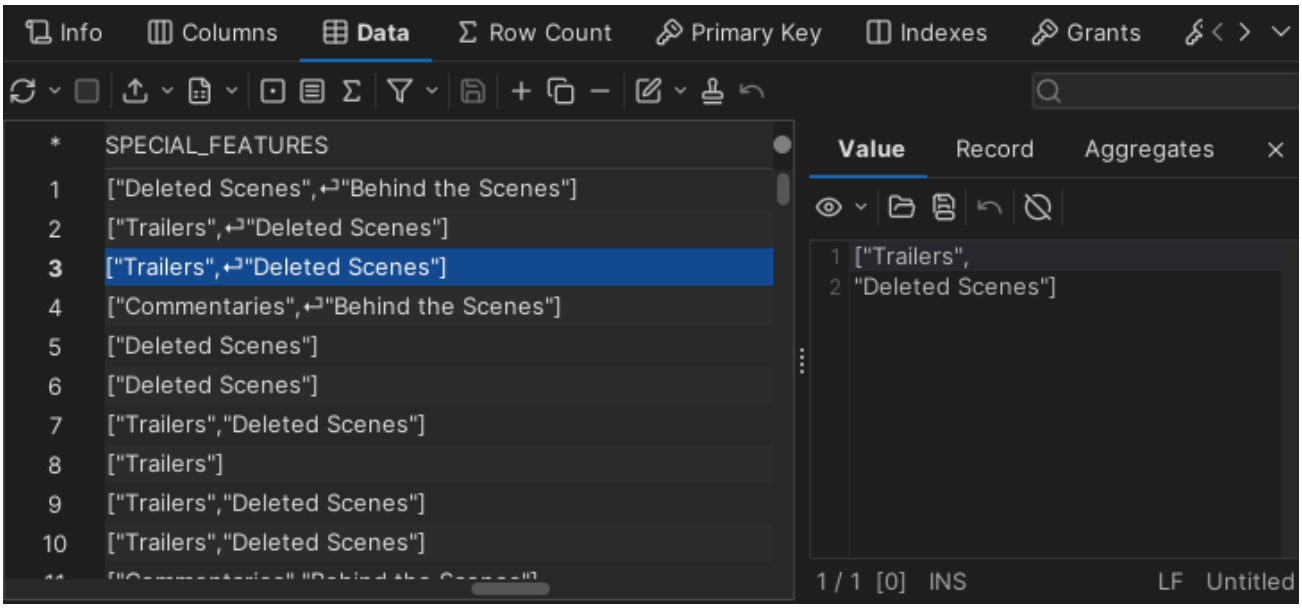
By default, a Primary Key column is shown with an icon in the column header. You can disable this in the Tool Properties dialog, under the **General** / **Table Data** category.

## Auto Resize Columns and Wrap Column Lines

The column header right-click menu contains a number of options for automatic resizing of column widths. The same menu allows you to choose to wrap long text lines in columns. Line wrap is not available for CLOB or Binary data types.




If a column contains line endings and wrapping is not turned on for that column, line endings will be represented by a symbol within the cell.

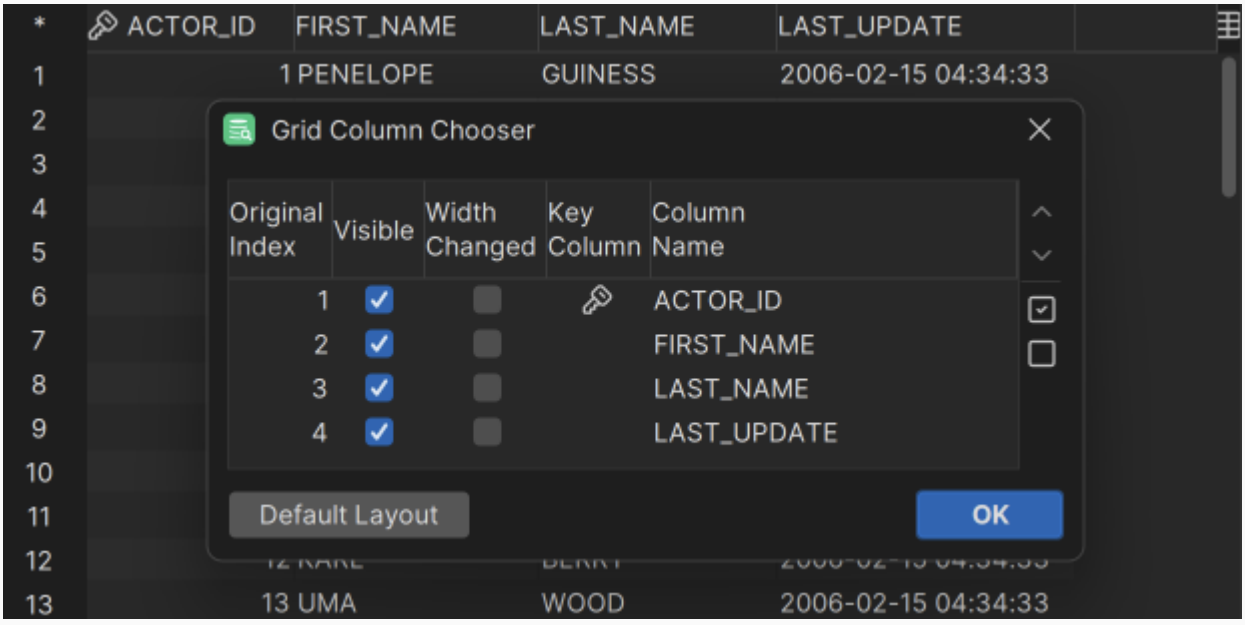



To show the full text of a wrapped text value, double-click on it to open in the Value viewer.

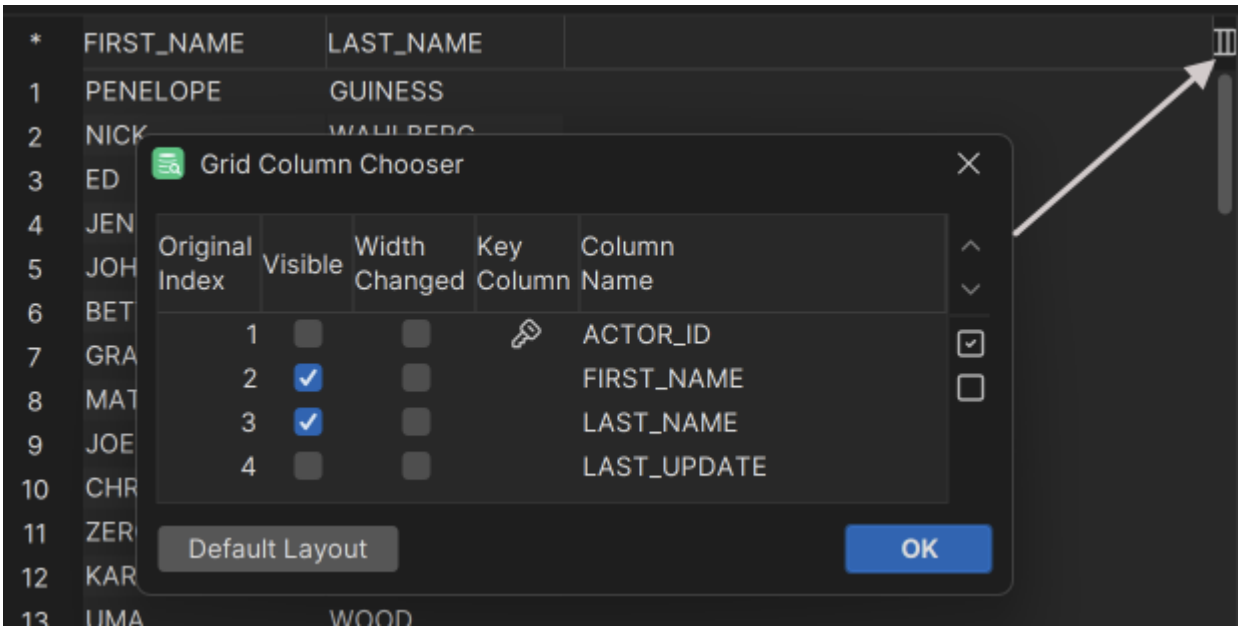
## Show Only Some Columns

The **Grid Column Chooser** dialog controls which columns you want to appear in a grid. Open the dialog by the right-click menu on the column header or the button above the vertical scrollbar in the grid. See the screenshot below.

**Grid Column Chooser with the default layout. The icon in the top right corner of the grid indicates the standard layout.**




Grid Column Chooser with a **custom layout**. The icon in the top right corner of the grid indicates that the layout is changed.




The **Grid Column Chooser** dialog shows all columns that are available in the grid. The checkmark in front of a column name indicates that the column is visible in the grid, while an unchecked box indicates that it is excluded from the grid. Click the checkmark to change the visibility of a column. You can change the visibility for all columns at once using the two visibility buttons in the dialog.

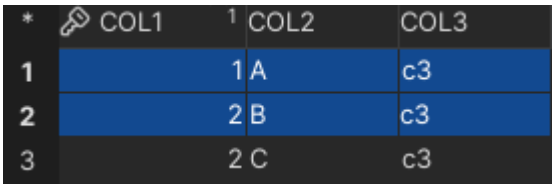
The order of the columns can also be adjusted in this dialog. Just select one or several rows and use the **Up** and **Down** buttons to move the rows up (left in the grid) or down (right in the grid). You can also use **Drag&Drop** to move the rows.

If you want to revert your changes, you can click on the **Default Layout** button to reset the grid, i.e., making all columns visible and putting them in their default locations.

> ✎ Modifications of column visibility, size, and order are saved between invocations of DbVisualizer for the **Log** tab and all grids in the various **Object View** tabs except for the **Data** tab.
>
> If you modify the column visibility in the **Data** tab, the changes persist throughout the session. For instance, if you remove the **ACTOR_ID** column in the **Data** tab for the table Actor, the **ACTOR_ID** column remains excluded when you reload the table or come back to the **Data** tab for that table later in the same session. You must manually make it visible again to bring it back. The changes are, however, reset when you restart the application. If you modify the column visibility in the **Log** tab in the SQL Commander, the changes will affect the Log tab immediately and other SQL Commander Log tabs the next time they are opened or refreshed by running a query. Any custom column setups are saved individually for the Log tab in SQL Commander, Actions, Export (all but export grid), Import, and Procedure Editor.

## Right-Click Menu Operations

The right-click menu for the grid contains a lot of operations for working with the data without changing it. In addition to the common select, copy, and print operations, some operations that may require a bit of an explanation are described below. Examples are based on the following simple grid that contains 9 cells on 3 rows with 3 columns, where the first two rows are selected:



| * | COL1 | 1 | COL2 | COL3 |
|---|---|---|---|---|
| 1 | 1 | | A | c3 |
| 2 | 2 | | B | c3 |
| 3 | 2 | | C | c3 |

| Operation | Description |
| --- | --- |
| **Copy Selection** | Copy all selected cells onto the system clipboard. Example:<br>`1 A c3`<br>`2 B c3` |
| **Copy Selection with Column Header** | Copy all selected cells including column header onto the system clipboard. Example:<br>`col1 col2 col3`<br>`1 A c3`<br>`2 B c3` |
| **Copy Selection as Formatted Text** | Copy all selected cells including column header in fixed width columns onto the system clipboard. Example:<br>`col1 col2 col3`<br>`---- ---- ----`<br>`1 A c3`<br>`2 B c3` |
| **Copy Selection as Comma List** | Copy all selected cells onto the system clipboard, formatted as Comma Separated Values (CSV), one row for each column. Example:<br>`1, 2`<br>`A, B`<br>`c3, c3` |
| **Copy Selection as IN Clause** | Copy all selected cells onto the system clipboard, formatted as an IN clause. Example:<br>`(col1 IN (1, 2))`<br>`and (col2 IN ('A', 'B'))`<br>`and (col3 IN ('c3'))` |
| **Copy Selection as IN Clause with AND (ALL)** | Copy all selected cells onto the system clipboard, formatted as a clause to select rows where ALL selected values match . Example:<br>`(col1 = 1 and col2 = 'A' and col3 = 'c3')`<br>`or (col1 = 2 and col2 = 'B' and col3 = 'c3')`<br>This is different from **Copy Selection as IN Clause with OR(ANY)** since the clause matches all column values on each row; it will not select a row with column values `1, B, c3` . |
| **Copy Selection as IN Clause with OR (ANY)** | Copy all selected cells onto the system clipboard, formatted as a clause to select rows where ANY of the selected values match. Example:<br>`(col1 IN (1, 2))`<br>`and (col2 IN ('A', 'B'))`<br>`and (col3 = 'c3')`<br>This is similar to **Copy Selection as IN Clause,** but is slightly more efficient in some situations; in this example, `col3` is matched using equals ( `=` ) rather than `IN` since the selected rows |

| Operation | Description |
| --- | --- |
| | have the same value in this column. This is different from **Copy Selection as IN Clause with AND (ALL)** since the clause matches any column value on each row; it will select a row with column values `1, B, c3`. |
| **Copy Selection as HTML Table** | Copy all selected cells onto the system clipboard, formatted as **HTML** code. Paste this into an HTML-capable editor to get a nicely formatted table. |
| **Copy Selection as JIRA Table** | Copy all selected cells onto the system clipboard, formatted as markup code suitable for **JIRA** and similar tools. Paste this into a suitable editor to get a nicely formatted table. |
| **Save Selected Cell** | Save the value of the selected cell to a file, selected with a file chooser dialog |
| **Compare** | Compare the data in this grid to the data in other open grids. |
| **Compare Selected Cells** | Compare the data in the two cells as text |
| **Reset Grid** | Reset the visual state of the grid (does not affect the column settings or the actual data in the grid). |
| **Value Viewer** | Display the value in the value viewer. Read more in Value Viewer. |
| **Record Viewer** | Display all data for the selected row in the record viewer. Read more in Record Viewer. |
| **Aggregates Viewer** | Displays aggregation data for the current selection. Read more in Aggregates Viewer. |
| **Show in Navigator** | Open the Navigator tab with the current selections and sorting. |
| **Describe Data** | Show detailed information about the columns in the grid. |
| **Generate Filter & Sort** | The operations in this submenu help you create common Where Filters. |
| **Extract Filter & Sort as SELECT** | Creates a SELECT statement based on the visible columns, filters and sorting in the grid to the clipboard. |
| **Create Row Count Data Monitor** | Creates a monitor for tracking the row count in the table over time. |
| **Create Row Count DIff Data Monitor** | Creates a monitor for tracking the number of added or removed rows in the table over time. |

## Script to SQL Commander

There are also a set of operations for generating SQL statements based on the current selection. Choosing any of these creates the appropriate SQL and then switches the view to a new **SQL Commander** tab. You must use these

operations to edit table data in the DbVisualizer Free edition. With the DbVisualizer Pro edition, you can instead use inline and form based editing.

| Operation | SQL Example |
| --- | --- |
| **Script: SELECT ALL** | `SELECT * FROM COUNTRY;` |
| **Script: SELECT ALL WHERE** | `select * FROM SAKILA.COUNTRY where COUNTRY = 'Brazil'` |
| **Script: SELECT ALL WITH FILTER** | `select * from SAKILA.COUNTRY where COUNTRY_ID = 15 // If this is the filter, see above` |
| **Script: INSERT INTO TABLE** | `insert into SAKILA.COUNTRY (COUNTRY_ID, COUNTRY) values (15, 'Brazil')` |
| **Script: INSERT COPY INTO TABLE** | `insert into SAKILA.COUNTRY (COUNTRY_ID, COUNTRY) values (15, 'Brazil')` |
| **Script: UPDATE WHERE** | `update SAKILA.COUNTRY set COUNTRY_ID = 15, COUNTRY = 'Brazil' where COUNTRY = 'Brazil'` |
| **Script: DELETE WHERE** | `delete from SAKILA.COUNTRY where COUNTRY = 'Brazil'` |

You can generate SQL with either static values as they appear in the grid, or with DbVisualizer variables. A variable is essentially a placeholder for a value in an SQL statement. When the statement is executed, DbVisualizer locates all variables and presents them in a dialog where you can enter or modify values for the variables. DbVisualizer replaces the variable placeholders with the new values before executing the statement.

Variables are used in the generated SQL statements by default. You can disable the **Include Variables in SQL** setting in the Tool Properties dialog, in the **Table Data** category under the General tab, to use literal values are instead.

Here is an example of the SQL generated for **Script: SELECT ALL WHERE** with the **Include Variables in SQL** setting enabled, assuming the table is named SAKILA.COUNTRY and has a column named COUNTRY with the value 'Brazil' on the selected row:

```
SELECT * FROM COUNTRY WHERE COUNTRY = ${COUNTRY (where)||Brazil||String||where
ds=50 dt=VARCHAR}$;
```

And here is the same example with the **Include Variables in SQL** setting disabled:

```
SELECT * FROM COUNTRY WHERE COUNTRY = 'Brazil';
```

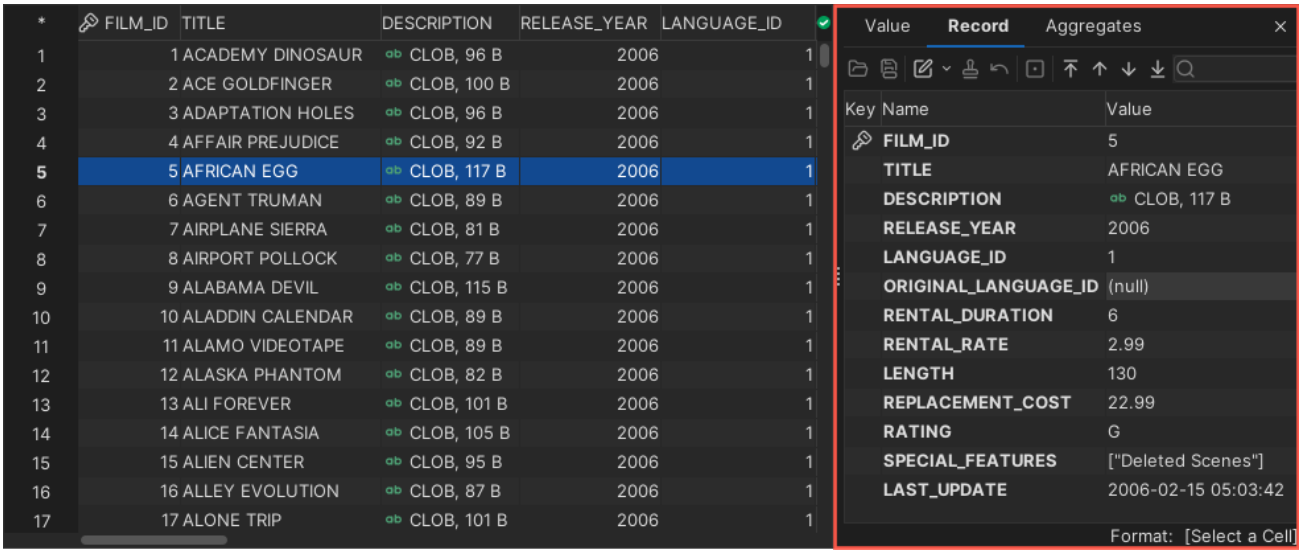
## Creating Monitors

A monitor in DbVisualizer is an SQL query executed at a specified frequency so you can track changes in data over time. The result can be viewed either as a grid or a graph. The right-click menu for the grid in the Data tab contains operations for creating two common types of monitors for the table: a **Row Count Data** monitor or a **Row Count Diff Data** monitor. The first tracks the number of rows in the table over time and the second tracks the number of added or removed rows over time. Please read more about monitors in Monitoring Data Changes.

## Grid viewers

Data in grids can be presented in three different viewers:

- **Value viewer**
  A viewer for a single selected cell in the grid. Based on the data type, the viewer may be a text, image, XML, JSON, serialized Java or a hex viewer.
- **Record viewer**
  The record viewer shows all column values in the selected grid row in a transposed form view with all columns as rows.
- **Aggregates viewer**
  Shows various aggregation metrics for the current selection in the grid.




The default position of these viewers is in a tabbed layout side-by-side with the grid. In Tool Properties under the **General / Grid / Grid Viewers** category, the layout can be changed to instead show these in separate windows rather than next to the grid.

## Value viewer

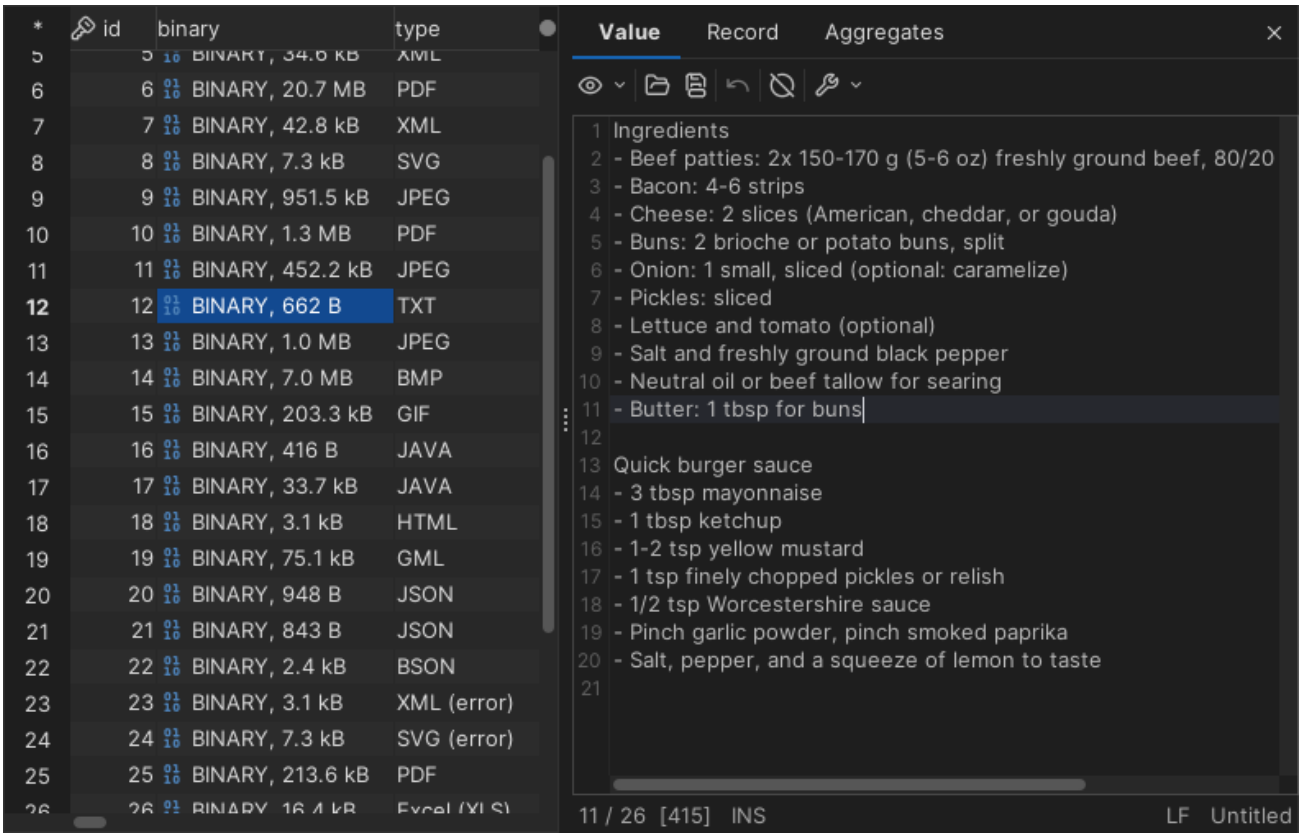
The **Value viewer** shows the selected grid cell value in a content-type-specific viewer. These are the available viewers:

- Text
- XML
- JSON
- PDF
- Image (GIF, JPEG, PNG, TIFF, BMP, SVG)
- Serialized Java Object
- HEX
- Date, Time, and Timestamp (with choosers when editing)

The value viewer is opened with one of:

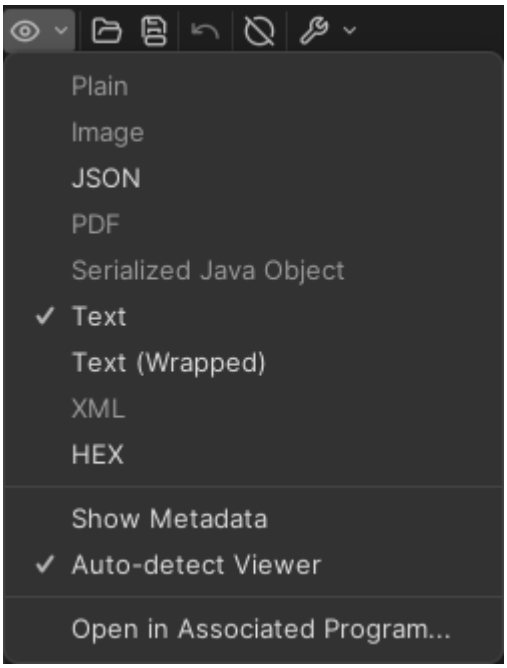
- Click the grid toolbar button: ⊡
- Choose **Value Viewer** in the grid right-click menu
- Double-click on a CLOB or binary cell
- Double-click on a text cell having newline or tab character symbols
- Double-click on a cell that is truncated due to the **Max Chars** setting in Tool Properties under the **General / Data Formats** category

The following is an example of what the Value viewer to the right shows:




In this example, a binary cell is selected in the grid. The value viewer then detects the type of content and shows the matching viewer, in this case a text editor.

The left-most eye icon shows when clicked a list of all viewers:



Only viewers that are supported by the selected cell are enabled. The **Show Metadata** entry shows a pane at the bottom of the value viewer with type information for the cell. The **Auto-detect Viewer** entry will, when selected, automatically show, for example, XML and JSON in a graphical viewer rather than in a text editor.

The value viewer supports dropping files on it.

Check the **General / Grid / Grid Viewers** category in Tool Properties for settings related to the value viewer. In particular, you can opt to use the text editor font in the value viewer, which may be helpful if you often work with text values that contain code.

EDITING IN THE VALUE VIEWER

If editing is supported, edits made in the grid are directly propagated to the value viewer and vice versa.

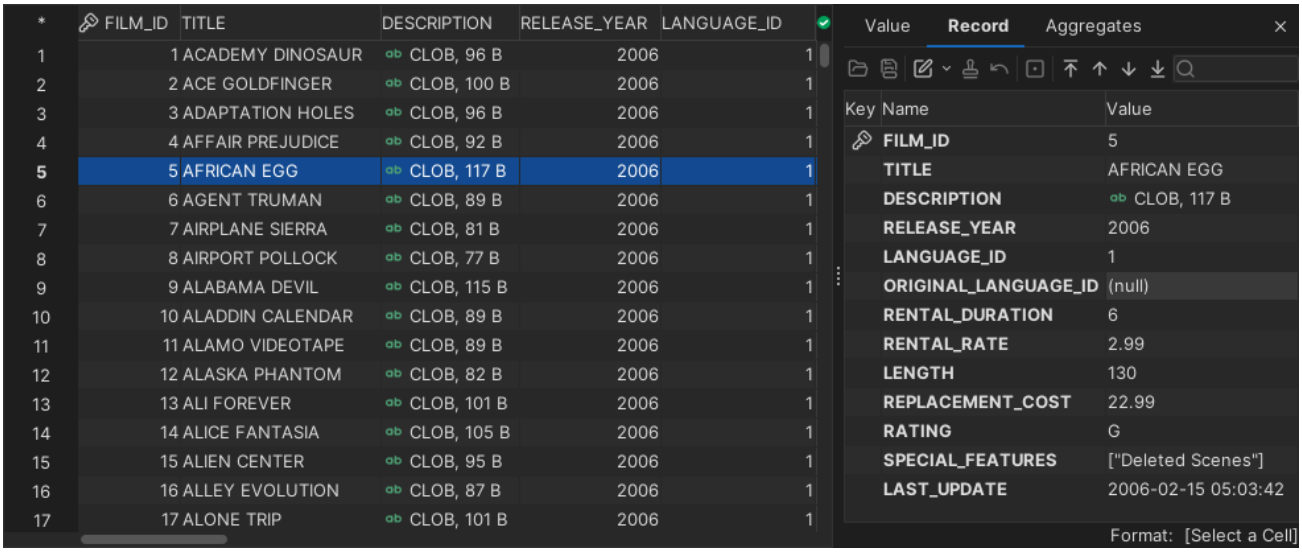
## Record viewer

The **Record viewer** shows the selected grid row in a transposed ("rotated") view with all the original columns as rows. This is a convenient viewer for grids having a lot of columns.

The record viewer is opened with one of:

- Click the grid toolbar button: ☰
- Choose **Record Viewer** in the grid right-click menu
- Double-click the row number to the left of the grid data

The following is an example of what the Record viewer to the right shows:




The **Key** field shows an icon for primary key columns, and the **Name** field corresponds to the column name in the grid. The **Value** field shows the value for each column. For CLOB and binary data types, the value field shows an icon and the size of the data. To see the data for these values, double-click on the cell to open the value viewer in a separate window or click the ⊡ in the toolbar. The format of the values can be adjusted in Tool Properties under the **General / Data Formats** category.

The navigation buttons in the toolbar allow you to navigate between the rows in the grid and show the corresponding row in the record viewer.

For detailed information about each column, hover a **Name** field to show database data type, nullability, and more.

Check the **General / Grid / Grid Viewers** category in Tool Properties for settings related to the record viewer.

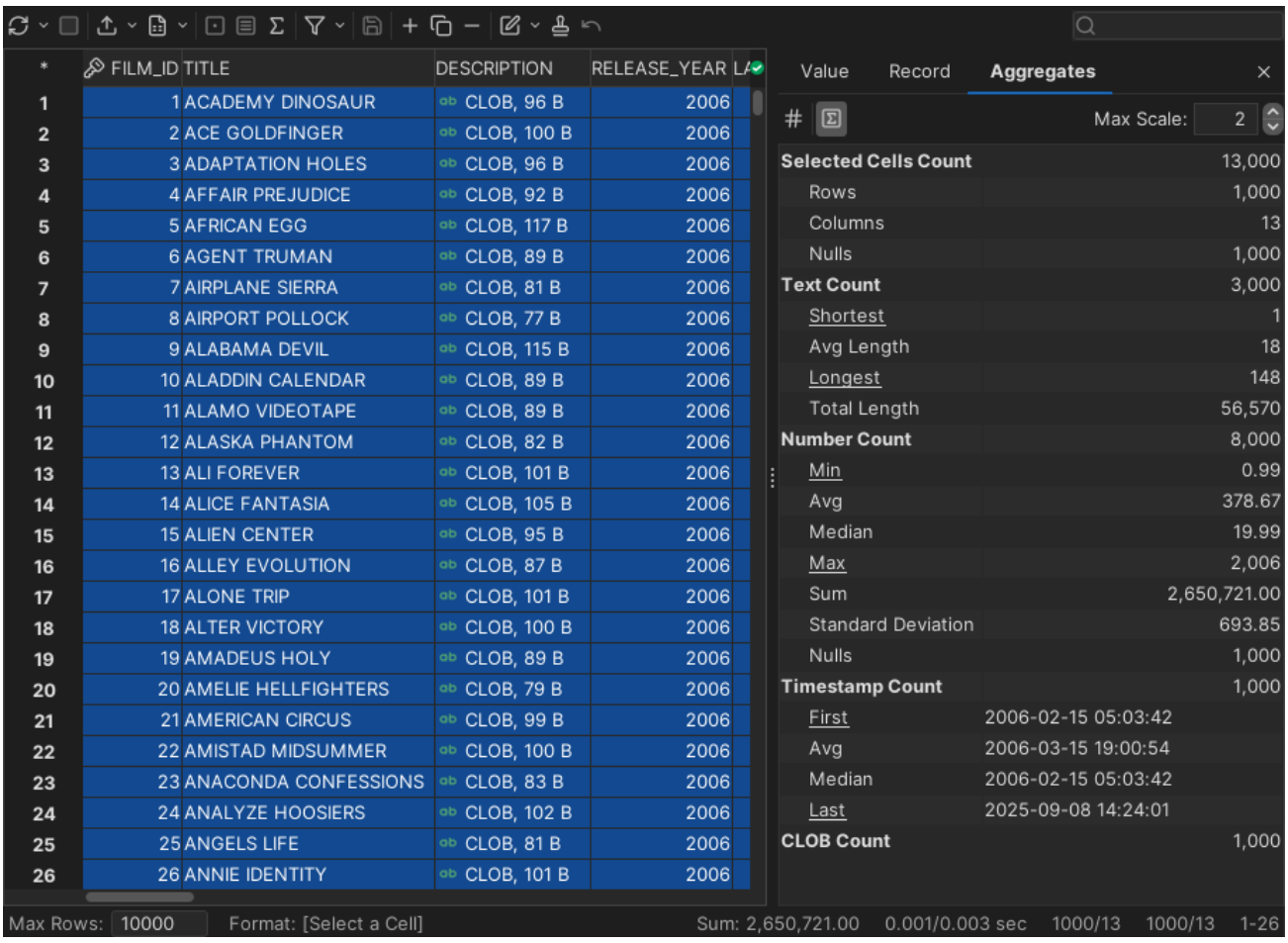
EDITING IN THE RECORD VIEWER

If editing is supported, edits made in the grid are directly propagated to the record viewer and vice versa. Edits are indicated with the same colors as in the grid.

## Aggregates viewer

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

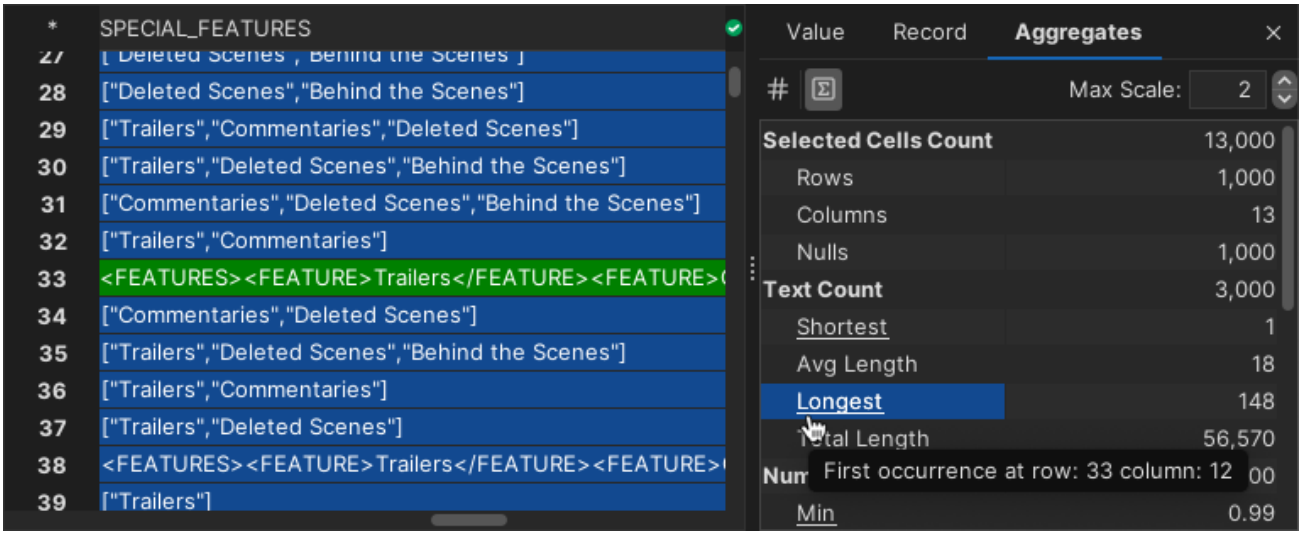
The **Aggregates viewer** presents aggregation data organized per data type for the current selection in a grid. It provides information about cells holding numbers, text, date/time information and more. The following is an example of what it shows:



The aggregate viewer runs in the background whenever it is visible on screen, and the selection changes in the source grid. The buttons in the toolbar:

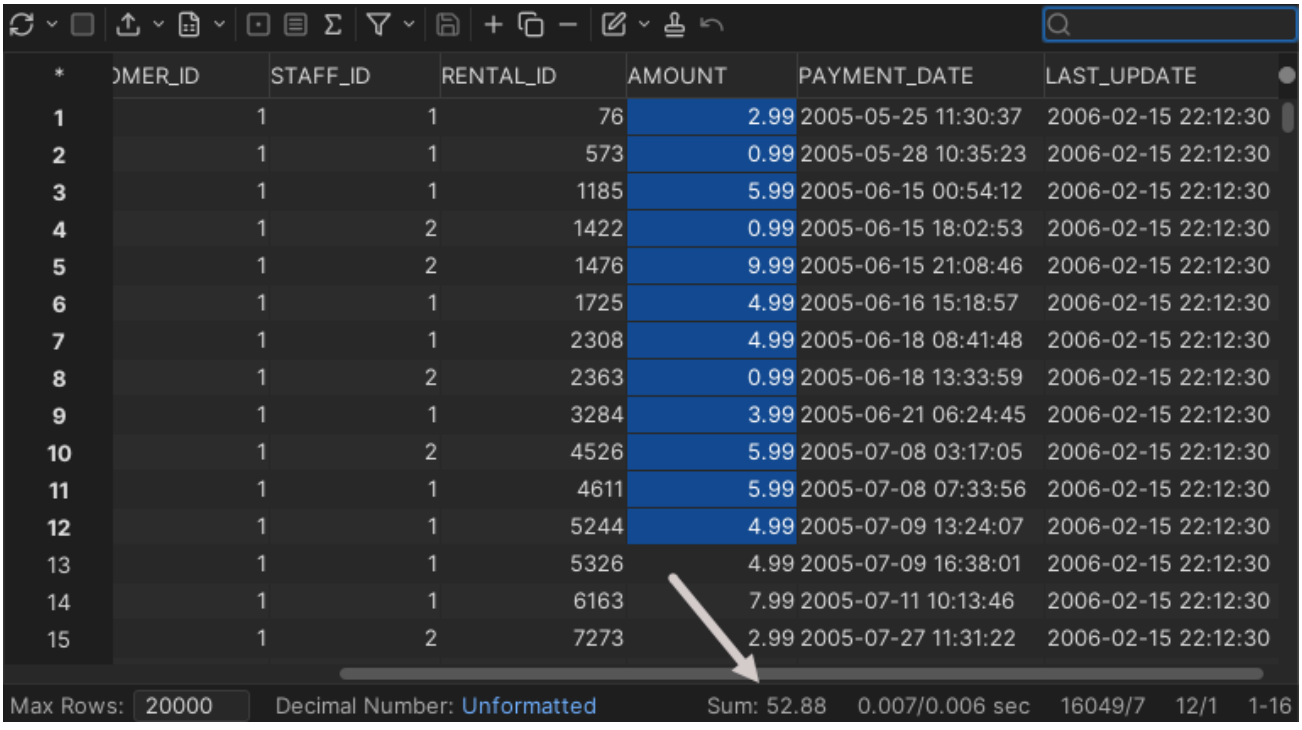
# - When selected, numbers in text data types will be calculated as numbers
Σ - When selected, the compact aggregation status entry will be displayed below the source grid.

Clicking a label with an underline highlights the corresponding cell in the source grid.

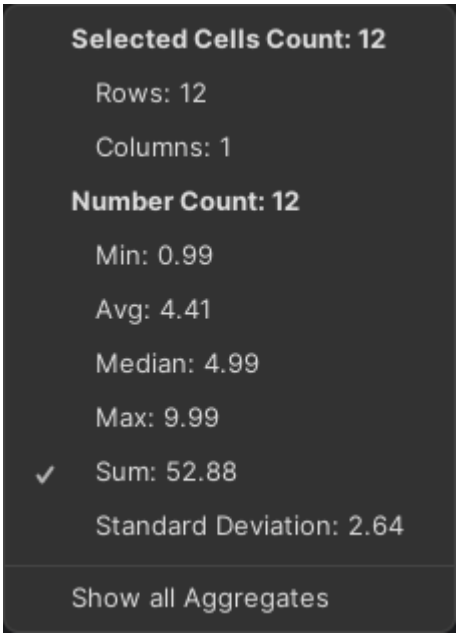



AGGREGATION STATUS BAR

The aggregation status bar can show up to three metrics for the current selection. It is updated even when the aggregation viewer is not displayed. The default is to show the summary of numbers in the selected cells.



Click the status bar entry to choose up to three metrics and to show their current values:




The **Show all Aggregates** item will open the aggregate viewer, and there the aggregation status bar can be turned off.

## Document data

DbVisualizer offers additional features for document data grids. Check Working with Document Data for more information.

# Editing Table Data

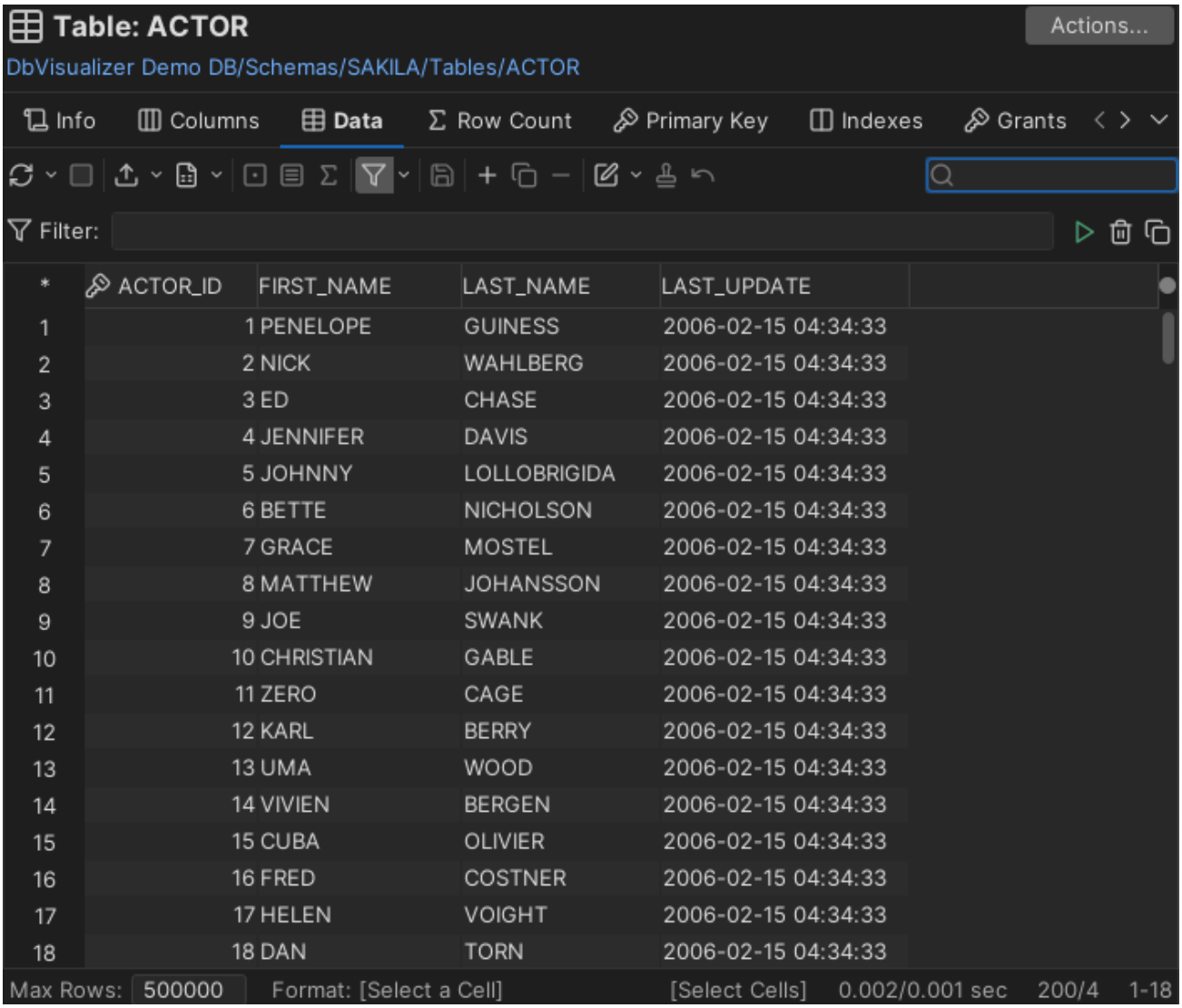
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

With the DbVisualizer Pro edition, you can edit table data in the **Data** tab grid; just click a cell value and edit. Edits are saved in a single database transaction which ensures that all or no changes are committed. The editing feature supports saving binary and large text data and it even presents common data formats in their respective viewers, such as image viewer, PDF, XML, HEX, etc.

## Opening the Data tab

To open the **Data** tab for a table:

1. Locate the table in the **Databases** tab tree,
2. Double-click the table node to open its **Object View** tab,
3. Open the **Data** sub tab.




Each column width is automatically resized to match the column width, including the column header, by default. You can disable this behavior in the the Tool Properties dialog, in the **Grid** category under the General tab.

If **Auto Resize Column Widths** is enabled, the **Max Column Width** setting can be used to limit the column width so that an extremely wide column does not take up all space.

## Editing Data in the Grid

To edit a column value:

1. Select the column cell,
2. Type the new value, or double click to edit the current value,
3. Click the **Save** toolbar button to update the database.

You can also use the **Set Selected Cells** drop down menu to set a number of column values to things like null or the current date or time.

To add a new row:

1. Select the row above where you want to insert the new row,
2. Click the **Add Row** toolbar button,
3. Enter values for the columns,
4. Click the **Save** toolbar button to update the database.

To duplicate a row:

1. Select the row you want to duplicate,
2. Click the **Duplicate Row** toolbar button,
3. Edit at least the key column(s) value(s),
4. Click the **Save** toolbar button to update the database.

To delete one or more rows:

1. Select the rows to delete,
2. Click the **Delete Rows** toolbar button,
3. Click the **Save** toolbar button to update the database.

If you change your mind, you easily can undo edits:

1. Select the cell(s) you want to revert,
2. Click the **Undo** toolbar button.

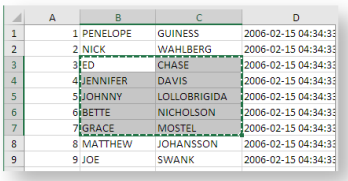
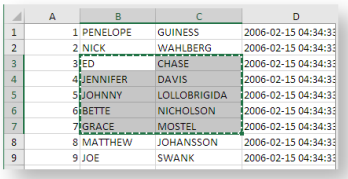
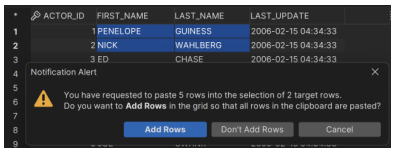
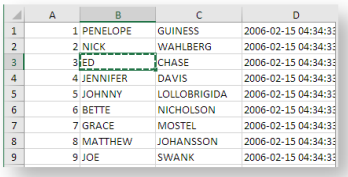
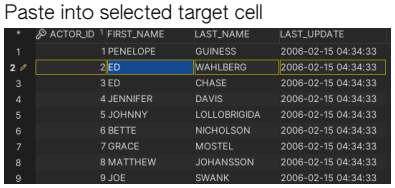
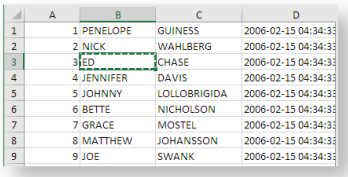
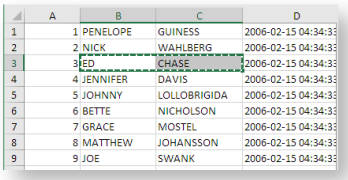
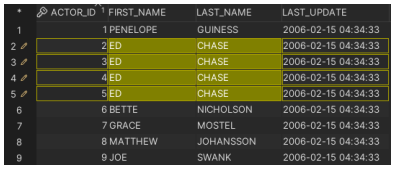
Reverting all cells in a row that are marked as **Insert** or **Duplicate** removes the complete row from the grid while a **Delete** marked row is cleared from its delete state. Undoing updated cells simply reverts the changes to the original values.

## Copy/Paste

You can copy selected cell values with the **Copy Selection** right-click menu choice or the corresponding key binding (**Ctrl-C** or **Command-C** by default). The data on the clipboard may then be pasted either into DbVisualizer or any external application. The column and newline delimiter used for copy and paste operations in the grid editor are defined by the **Copy Grid Cells in CSV Format** settings in the **Grid** category in the Tool Properties dialog, under the General tab. The default setting are sufficient for most uses.

The grid editor supports pasting data from the major spreadsheet applications, such as Excel and OpenOffice. The grid editor supports pasting single data as well as block of data. Copy/paste of binary data is transparent between

grids or in the same grid. Binary files may also be copied in an external application and pasted in a cell in DbVisualizer (target cell must be a binary type).

| Copy from spreadsheet | | Paste into DbVisualizer grid |
|---|---|---|
| A block of cells is copied<br>  | → | The block is pasted into the selected region<br>  |
| A block of cells is copied<br>  | → | The block cannot be pasted into a different number of target cells<br>  |
| A single cell is copied<br>  | → | Paste into selected target cell<br>  |
| A single cell is copied<br>  | → | Paste and fill the single column target selection<br>  |
| Multiple cells in a single row is copied<br>  | → | Paste and fill the target selection<br>  |

## Updates and Deletes Must Match Only One Table Row

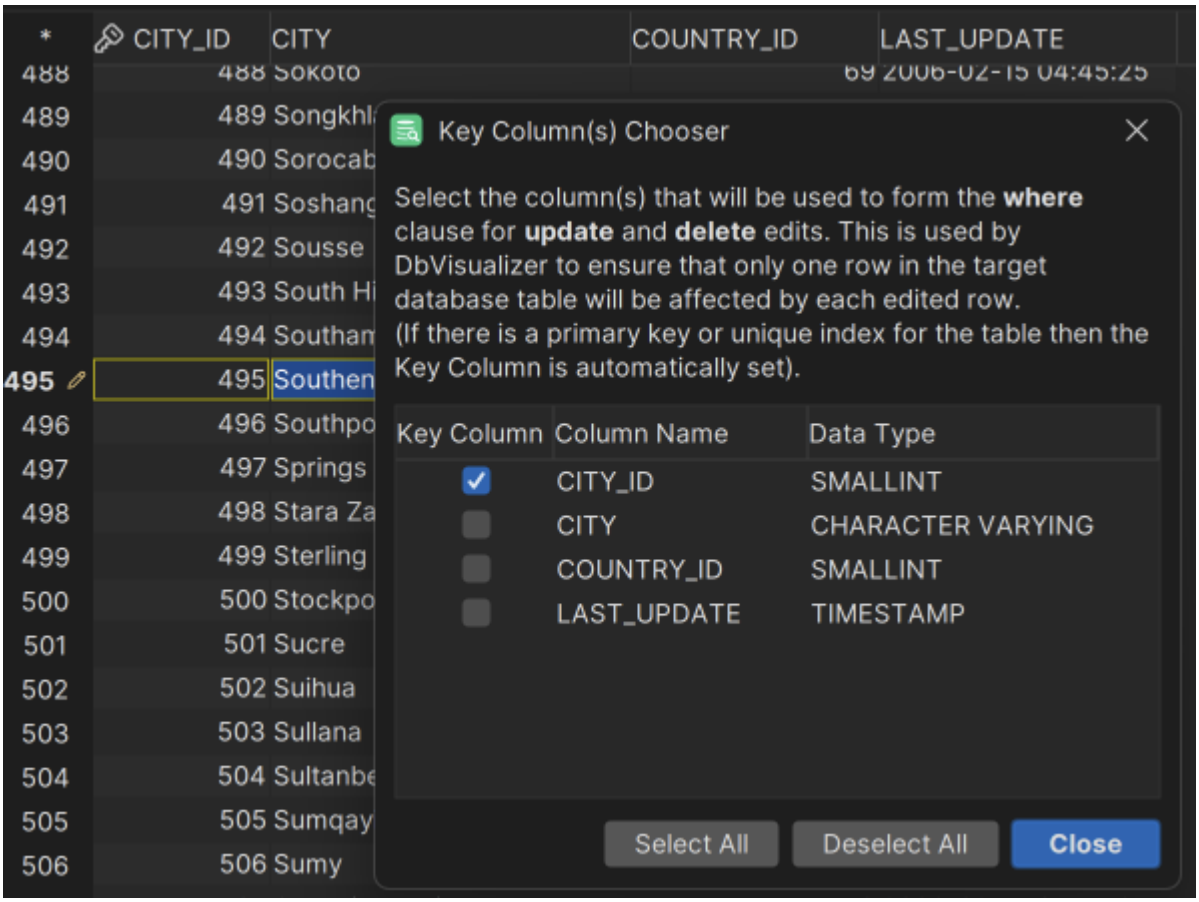
When you update or delete rows, DbVisualizer ensures that only one row in the table will be affected. This is to prevent changes in one row to silently affect data in other rows. DbVisualizer uses the following strategies to determine the uniqueness of the edited row:

1. Primary Key,
2. Unique Index,
3. Manually Selected Columns.

The Primary Key concept is widely used in databases to uniquely identify the key columns in tables. If the table has a primary key, DbVisualizer uses it. There are situations when primary keys are not supported by a database or when primary keys are supported but not used. If no primary key is defined, DbVisualizer checks if there is a unique index. If there are several unique indexes, DbVisualizer picks one of them. If there is no primary key or any unique indexes defined for the table, you need to manually choose what columns to use. The **Key Column Chooser** is automatically displayed if the key columns can't be determined automatically.

## Key Column(s) Chooser

Normally database tables have a primary key or at least one unique index. If this is the case, editing is no problem. If there is no way to uniquely identify rows in the table, you need to manually define what columns DbVisualizer should use. While saving the changes, DbVisualizer checks that there is a way to identify unique rows in the table. If it cannot be accomplished, the following window is displayed.




The key column chooser can also be manually opened via the **Edit Table Data → Key Column Chooser** right-click menu choice.

If the database request to save the edits cannot uniquely identify the single row that should be changed, an error dialog is displayed and the editing state is kept for that row in the grid editor.

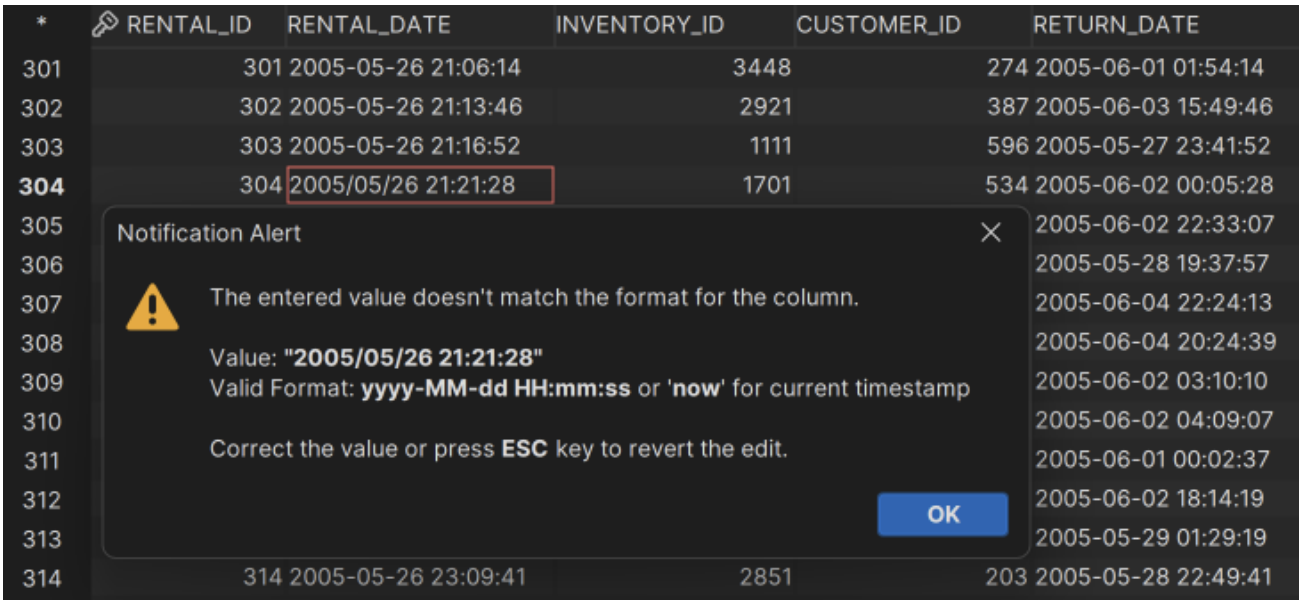
## Editing Multiple Rows

The grid editor supports editing multiple rows and saving all changes in one database transaction. Edited rows are indicated with an icon in the row header:

- Cell(s) in the row has been edited
- \+ Row is new
- ☆ Row is duplicated from another row
- ⌫ Row is marked for deletion (edit is not allowed)

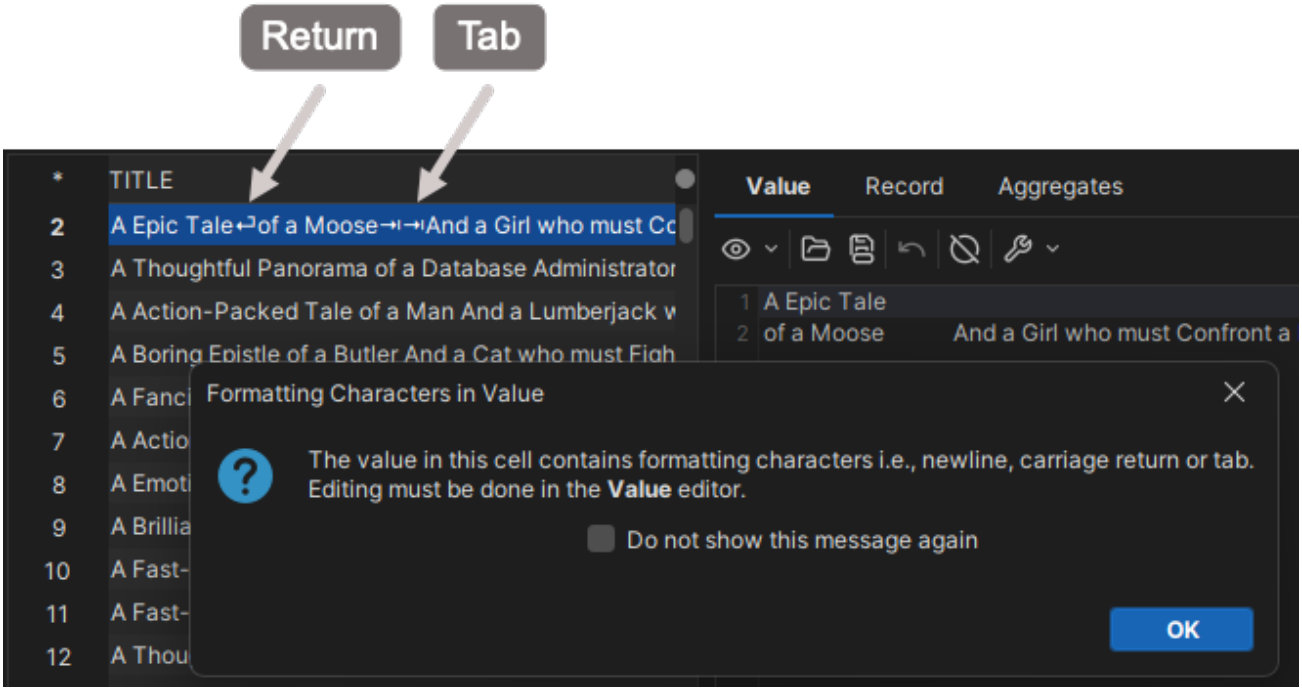
## Data Type checking

When leaving an edited cell, the new value is validated with the data type for the column. If there is an error, the following dialog is displayed.




## New Line and Carriage Return

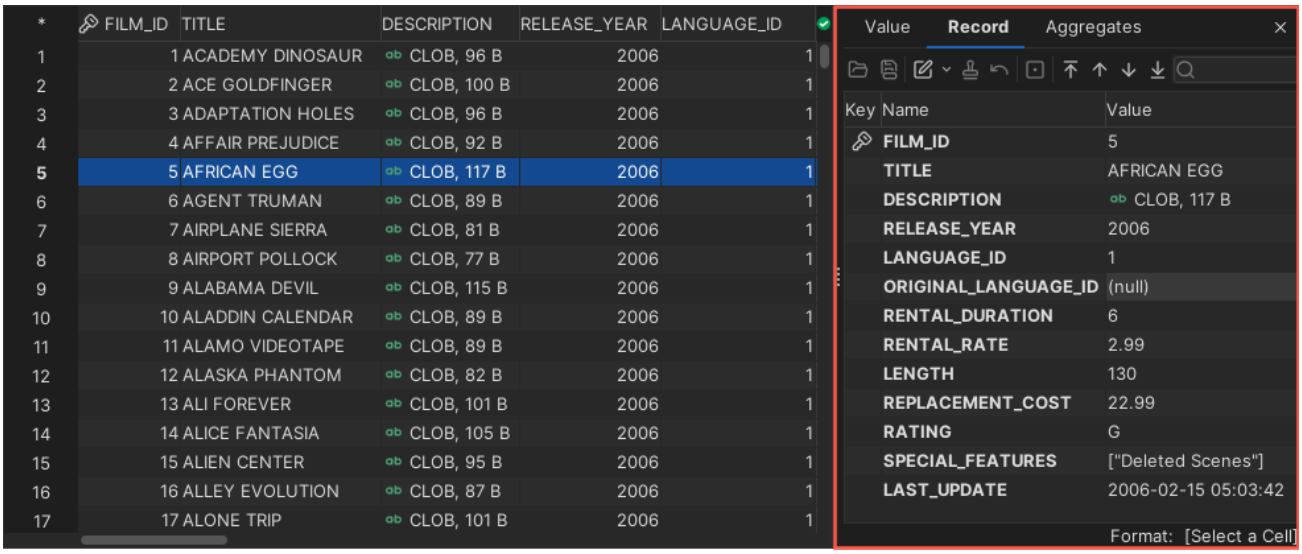
If a cell in the grid or record editor contains new line, carriage return or tab characters, these are visually represented in the grid, but an information dialog will be displayed whenever you try to edit such a value:



# Grid viewers

Data in grids can be presented in three different viewers:

- **Value viewer**
  A viewer for a single selected cell in the grid. Based on the data type, the viewer may be a text, image, XML, JSON, serialized Java or a hex viewer.
- **Record viewer**
  The record viewer shows all column values in the selected grid row in a transposed form view with all columns as rows.
- **Aggregates viewer**
  Shows various aggregation metrics for the current selection in the grid.




The default position of these viewers is in a tabbed layout side-by-side with the grid. In Tool Properties under the **General / Grid / Grid Viewers** category, the layout can be changed to instead show these in separate windows rather than next to the grid.

## Value viewer

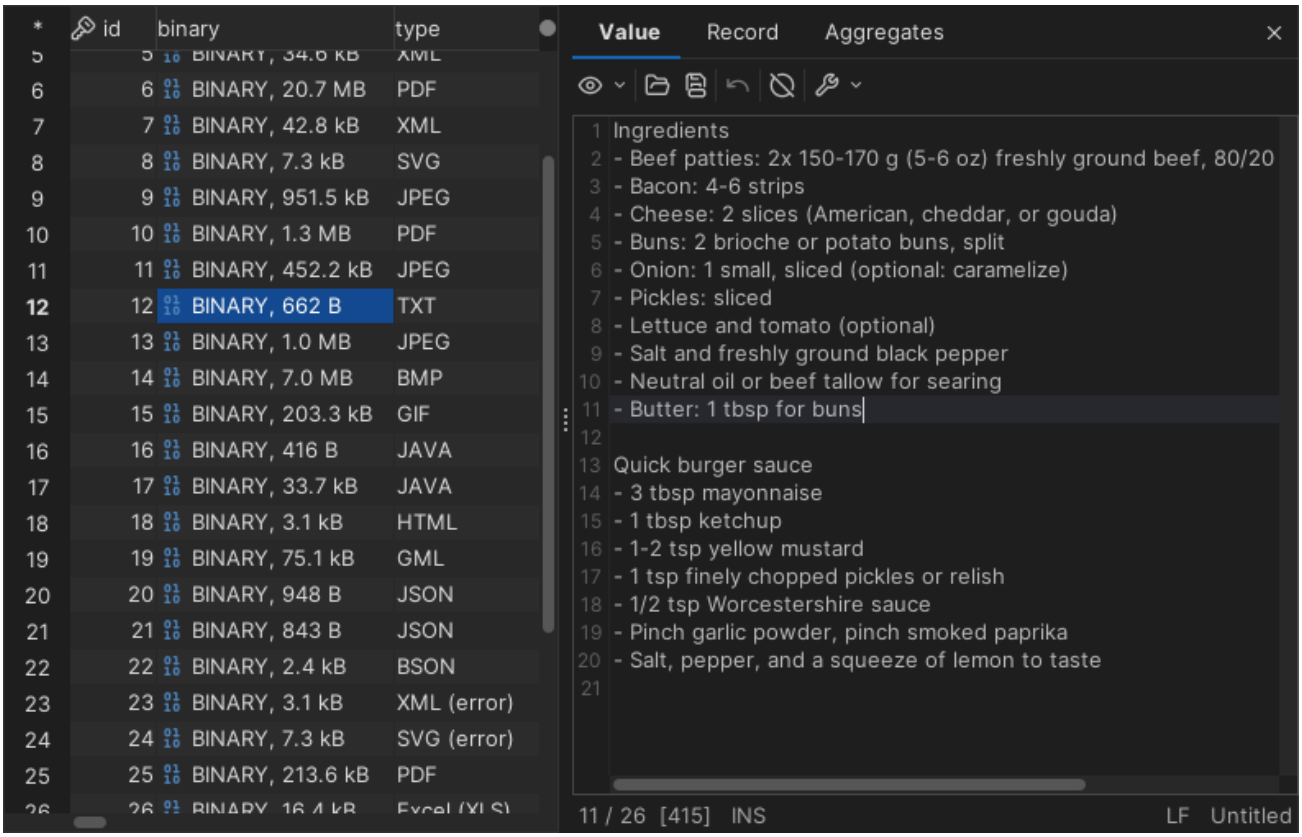
The **Value viewer** shows the selected grid cell value in a content-type-specific viewer. These are the available viewers:

- Text
- XML
- JSON
- PDF
- Image (GIF, JPEG, PNG, TIFF, BMP, SVG)
- Serialized Java Object
- HEX
- Date, Time, and Timestamp (with choosers when editing)

The value viewer is opened with one of:

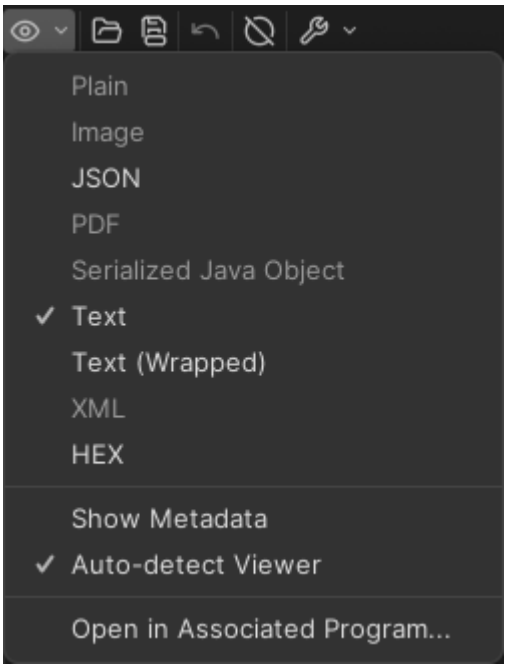
- Click the grid toolbar button: ⊡
- Choose **Value Viewer** in the grid right-click menu
- Double-click on a CLOB or binary cell
- Double-click on a text cell having newline or tab character symbols
- Double-click on a cell that is truncated due to the **Max Chars** setting in Tool Properties under the **General / Data Formats** category

The following is an example of what the Value viewer to the right shows:




In this example, a binary cell is selected in the grid. The value viewer then detects the type of content and shows the matching viewer, in this case a text editor.

The left-most eye icon shows when clicked a list of all viewers:



Only viewers that are supported by the selected cell are enabled. The **Show Metadata** entry shows a pane at the bottom of the value viewer with type information for the cell. The **Auto-detect Viewer** entry will, when selected, automatically show, for example, XML and JSON in a graphical viewer rather than in a text editor.

The value viewer supports dropping files on it.

Check the **General / Grid / Grid Viewers** category in Tool Properties for settings related to the value viewer. In particular, you can opt to use the text editor font in the value viewer, which may be helpful if you often work with text values that contain code.

EDITING IN THE VALUE VIEWER

If editing is supported, edits made in the grid are directly propagated to the value viewer and vice versa.

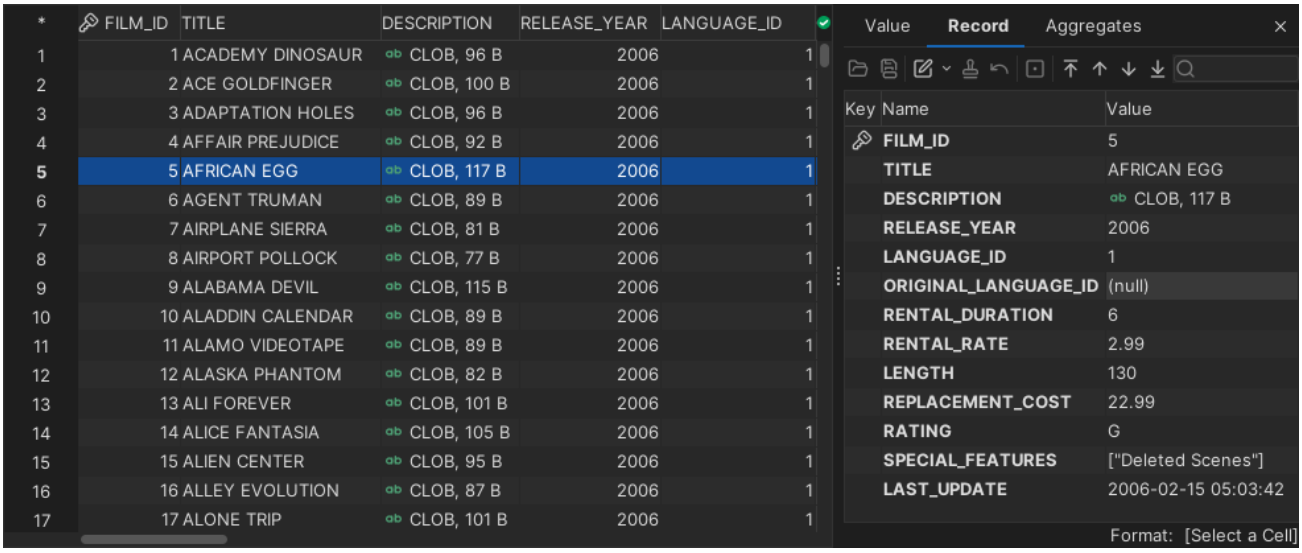
## Record viewer

The **Record viewer** shows the selected grid row in a transposed ("rotated") view with all the original columns as rows. This is a convenient viewer for grids having a lot of columns.

The record viewer is opened with one of:

- Click the grid toolbar button: ☰
- Choose **Record Viewer** in the grid right-click menu
- Double-click the row number to the left of the grid data

The following is an example of what the Record viewer to the right shows:




The **Key** field shows an icon for primary key columns, and the **Name** field corresponds to the column name in the grid. The **Value** field shows the value for each column. For CLOB and binary data types, the value field shows an icon and the size of the data. To see the data for these values, double-click on the cell to open the value viewer in a separate window or click the ⊡ in the toolbar. The format of the values can be adjusted in Tool Properties under the **General / Data Formats** category.

The navigation buttons in the toolbar allow you to navigate between the rows in the grid and show the corresponding row in the record viewer.

For detailed information about each column, hover a **Name** field to show database data type, nullability, and more.

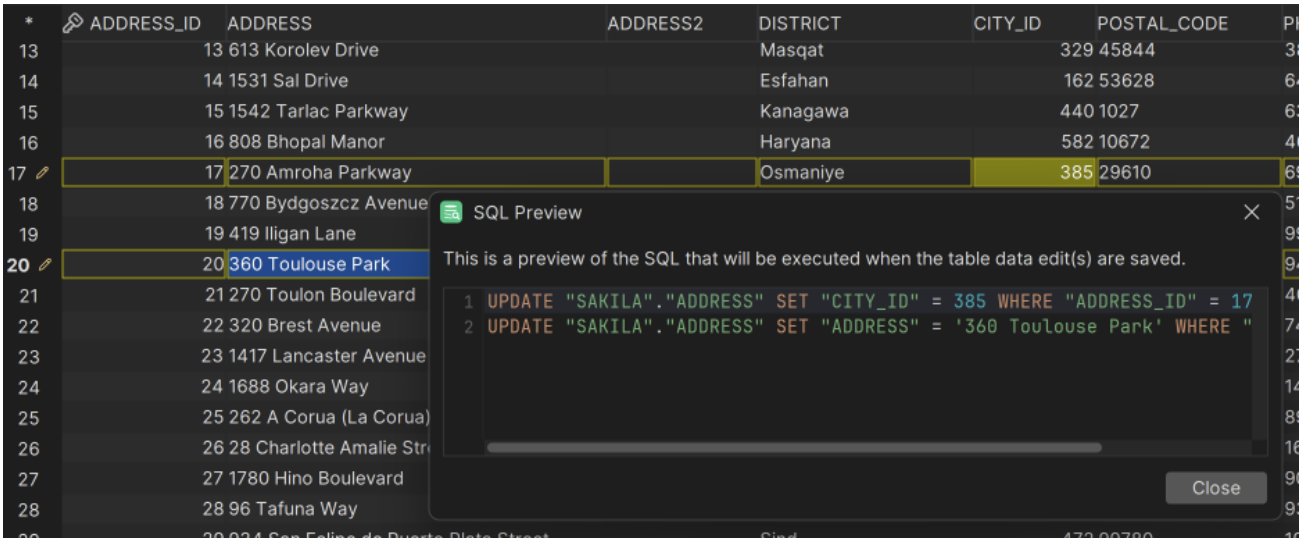
Check the **General / Grid / Grid Viewers** category in Tool Properties for settings related to the record viewer.

EDITING IN THE RECORD VIEWER

If editing is supported, edits made in the grid are directly propagated to the record viewer and vice versa. Edits are indicated with the same colors as in the grid.

## Preview Changes

You may preview the SQL statements that will be executed when choosing to **Save** the edits via the **Edit Table Data → SQL Preview** right-click menu choice.




> The listed SQL statements may not be 100% identical to what is sent to the database, as the save process uses variable binding to pass values to the database.

## View and edit Binary/BLOB and CLOB Data

Due to the nature of binary/BLOB and CLOB data, values of these types can only be fully modified and viewed in the value viewer (there is partial support in the Record viewer to view image data and to load from a file).

In the grid, Binary/BLOB and CLOB data is by default presented by an icon and the size of the value. You can select another presentation format in the **Tools Properties** dialog, in the **General/Data Formats** category under the General tab. Selecting **By Value** or **Image Preview** results in performance penalties and the memory consumption

increases dramatically. The **Image Preview** for BLOB presents a downscaled image directly in the grid if the BLOB is in one of the supported image formats. You can set the image preview size.

In the same Tool Properties category, you can also specify how to handle **Copy/Paste and Drag and Drop** when pasting binary data in a target component that doesn't support binary data.

Editing binary data can be done by importing from a file or via the text editor in the value viewer. You can also copy the file in the operating system's file browser and paste it into a BLOB/CLOB cell.

Binary data in DbVisualizer is the generic term for several common binary database types:

- LONGVARBINARY
- BINARY
- VARBINARY
- BLOB

## Document data

DbVisualizer offers additional features for document data grids. Check Working with Document Data for more information.

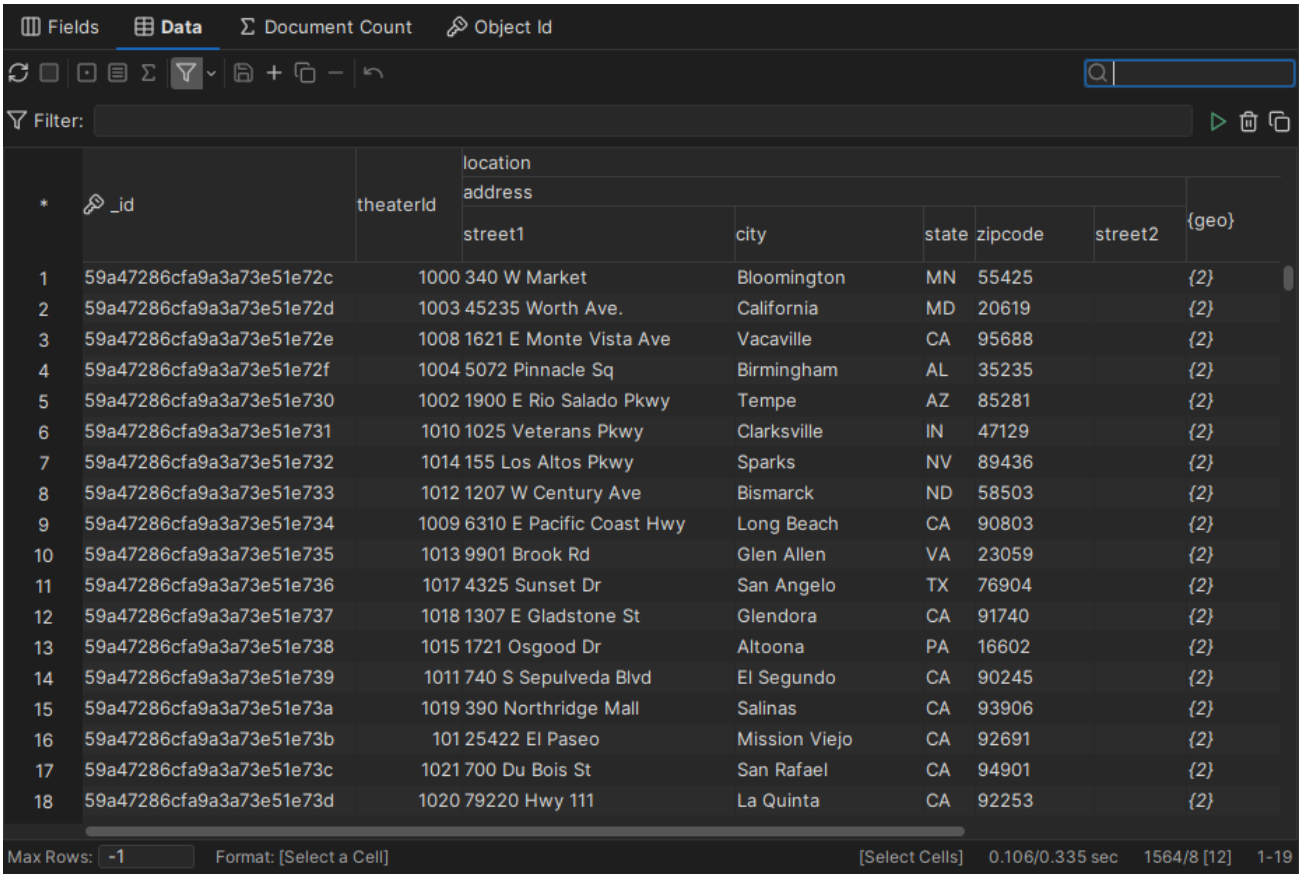
# Working with Document Data

To offer better support for complex nested data, DbVisualizer includes special functionality for document-based databases.

> This is currently supported for MongoDB only.

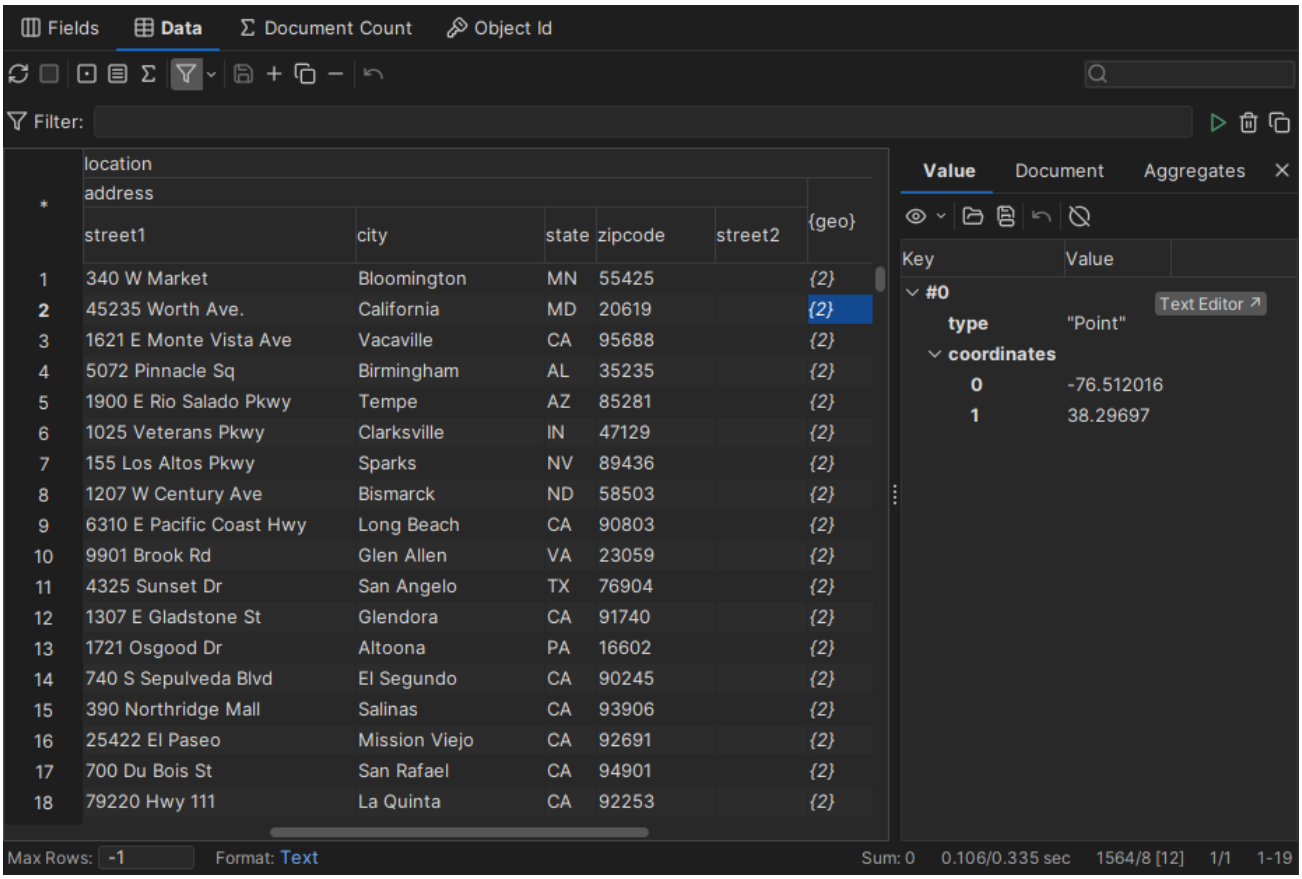
## Nested column headers

When working with documents, a regular two-dimensional grid is not always suitable. Instead, the data is presented in a special grid view that uses expandable column header groups based on the structure of the displayed document data.




| * | _id | theaterId | location | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | address | | | | | |
| | | | street1 | city | state | zipcode | street2 | {geo} |
| 1 | 59a47286cfa9a3a73e51e72c | 1000 | 340 W Market | Bloomington | MN | 55425 | | {2} |
| 2 | 59a47286cfa9a3a73e51e72d | 1003 | 45235 Worth Ave. | California | MD | 20619 | | {2} |
| 3 | 59a47286cfa9a3a73e51e72e | 1008 | 1621 E Monte Vista Ave | Vacaville | CA | 95688 | | {2} |
| 4 | 59a47286cfa9a3a73e51e72f | 1004 | 5072 Pinnacle Sq | Birmingham | AL | 35235 | | {2} |
| 5 | 59a47286cfa9a3a73e51e730 | 1002 | 1900 E Rio Salado Pkwy | Tempe | AZ | 85281 | | {2} |
| 6 | 59a47286cfa9a3a73e51e731 | 1010 | 1025 Veterans Pkwy | Clarksville | IN | 47129 | | {2} |
| 7 | 59a47286cfa9a3a73e51e732 | 1014 | 155 Los Altos Pkwy | Sparks | NV | 89436 | | {2} |
| 8 | 59a47286cfa9a3a73e51e733 | 1012 | 1207 W Century Ave | Bismarck | ND | 58503 | | {2} |
| 9 | 59a47286cfa9a3a73e51e734 | 1009 | 6310 E Pacific Coast Hwy | Long Beach | CA | 90803 | | {2} |
| 10 | 59a47286cfa9a3a73e51e735 | 1013 | 9901 Brook Rd | Glen Allen | VA | 23059 | | {2} |
| 11 | 59a47286cfa9a3a73e51e736 | 1017 | 4325 Sunset Dr | San Angelo | TX | 76904 | | {2} |
| 12 | 59a47286cfa9a3a73e51e737 | 1018 | 1307 E Gladstone St | Glendora | CA | 91740 | | {2} |
| 13 | 59a47286cfa9a3a73e51e738 | 1015 | 1721 Osgood Dr | Altoona | PA | 16602 | | {2} |
| 14 | 59a47286cfa9a3a73e51e739 | 1011 | 740 S Sepulveda Blvd | El Segundo | CA | 90245 | | {2} |
| 15 | 59a47286cfa9a3a73e51e73a | 1019 | 390 Northridge Mall | Salinas | CA | 93906 | | {2} |
| 16 | 59a47286cfa9a3a73e51e73b | 101 | 25422 El Paseo | Mission Viejo | CA | 92691 | | {2} |
| 17 | 59a47286cfa9a3a73e51e73c | 1021 | 700 Du Bois St | San Rafael | CA | 94901 | | {2} |
| 18 | 59a47286cfa9a3a73e51e73d | 1020 | 79220 Hwy 111 | La Quinta | CA | 92253 | | {2} |

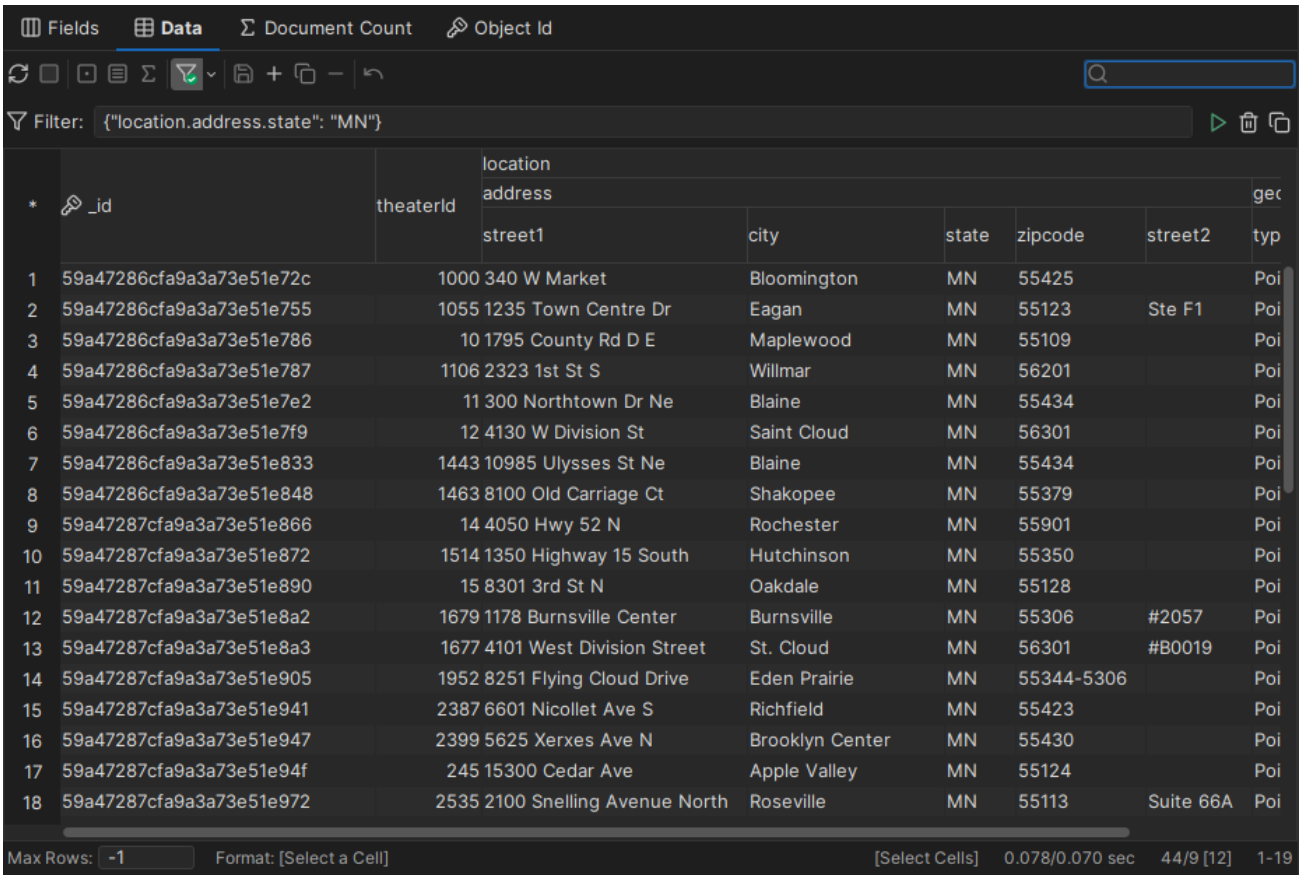
You can expand/collapse headers to focus on the data you are interested in (either by double-clicking the header or via the header's context menu). When a header group is collapsed, the corresponding data is also collapsed in the grid. Use the **Value viewer** to see the expanded value.




## Filtering

Similarly to normal grids, data in a document grid can be filtered. The **quick filter** can be used for full text filtering after the data has been fetched from the database.

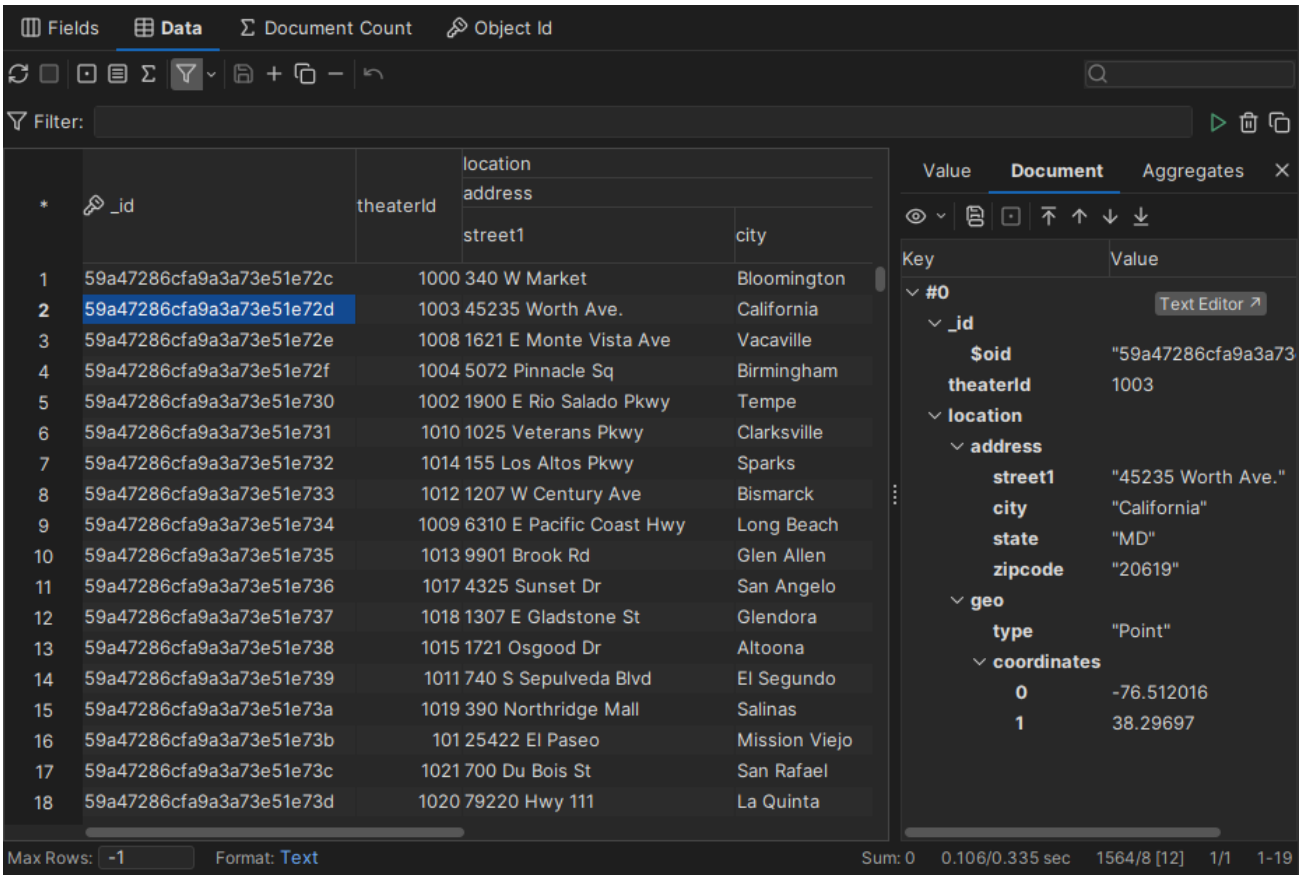
The **where filter** is instead included in the underlying database query, and can be used to control which subset of the data is fetched from the database. In the case of MongoDB, the MongoDB Shell syntax is used here.




| * | _id | theaterId | location | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | address | | | | | geo |
| | | | street1 | city | state | zipcode | street2 | typ |
| 1 | 59a47286cfa9a3a73e51e72c | 1000 | 340 W Market | Bloomington | MN | 55425 | | Poi |
| 2 | 59a47286cfa9a3a73e51e755 | 1055 | 1235 Town Centre Dr | Eagan | MN | 55123 | Ste F1 | Poi |
| 3 | 59a47286cfa9a3a73e51e786 | 10 | 1795 County Rd D E | Maplewood | MN | 55109 | | Poi |
| 4 | 59a47286cfa9a3a73e51e787 | 1106 | 2323 1st St S | Willmar | MN | 56201 | | Poi |
| 5 | 59a47286cfa9a3a73e51e7e2 | 11 | 300 Northtown Dr Ne | Blaine | MN | 55434 | | Poi |
| 6 | 59a47286cfa9a3a73e51e7f9 | 12 | 4130 W Division St | Saint Cloud | MN | 56301 | | Poi |
| 7 | 59a47286cfa9a3a73e51e833 | 1443 | 10985 Ulysses St Ne | Blaine | MN | 55434 | | Poi |
| 8 | 59a47286cfa9a3a73e51e848 | 1463 | 8100 Old Carriage Ct | Shakopee | MN | 55379 | | Poi |
| 9 | 59a47287cfa9a3a73e51e866 | 14 | 4050 Hwy 52 N | Rochester | MN | 55901 | | Poi |
| 10 | 59a47287cfa9a3a73e51e872 | 1514 | 1350 Highway 15 South | Hutchinson | MN | 55350 | | Poi |
| 11 | 59a47287cfa9a3a73e51e890 | 15 | 8301 3rd St N | Oakdale | MN | 55128 | | Poi |
| 12 | 59a47287cfa9a3a73e51e8a2 | 1679 | 1178 Burnsville Center | Burnsville | MN | 55306 | #2057 | Poi |
| 13 | 59a47287cfa9a3a73e51e8a3 | 1677 | 4101 West Division Street | St. Cloud | MN | 56301 | #B0019 | Poi |
| 14 | 59a47287cfa9a3a73e51e905 | 1952 | 8251 Flying Cloud Drive | Eden Prairie | MN | 55344-5306 | | Poi |
| 15 | 59a47287cfa9a3a73e51e941 | 2387 | 6601 Nicollet Ave S | Richfield | MN | 55423 | | Poi |
| 16 | 59a47287cfa9a3a73e51e947 | 2399 | 5625 Xerxes Ave N | Brooklyn Center | MN | 55430 | | Poi |
| 17 | 59a47287cfa9a3a73e51e94f | 245 | 15300 Cedar Ave | Apple Valley | MN | 55124 | | Poi |
| 18 | 59a47287cfa9a3a73e51e972 | 2535 | 2100 Snelling Avenue North | Roseville | MN | 55113 | Suite 66A | Poi |

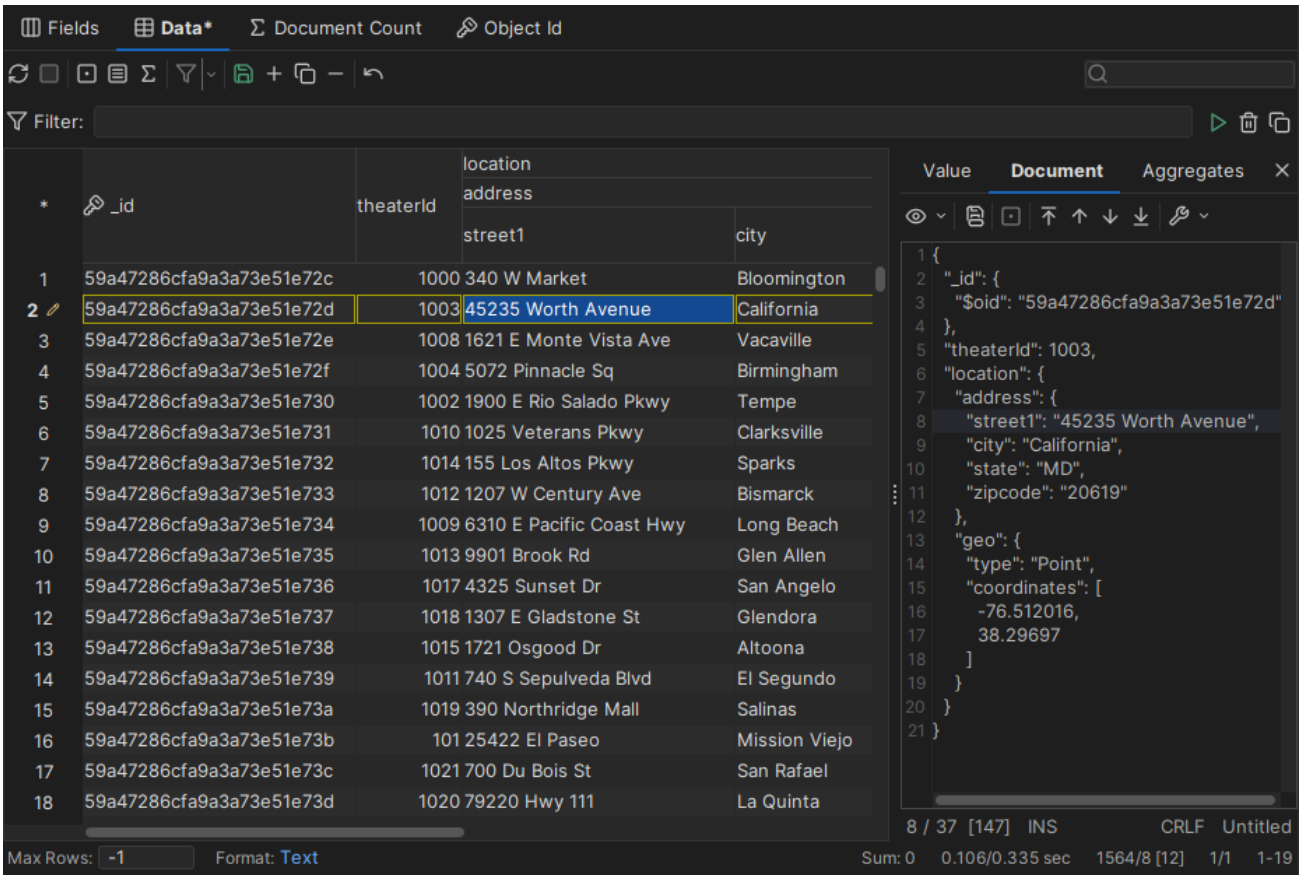
## Document viewer

Instead of the familiar Record viewer, the **Document viewer** is used for documents. It shows the JSON representation of the underlying document. Depending on the way you want to interact with the data, you can switch between the graphical JSON view and the text view.




## Editing

When editing is supported, edits can be made inline in the grid, in the Value tab, or in the Document tab. Changes made in the grid are directly propagated to the other tabs and vice versa. Uncommitted changes are color-marked in the grid.




| * | _id | theaterId | location address street1 | city |
|---|---|---|---|---|
| 1 | 59a47286cfa9a3a73e51e72c | 1000 | 340 W Market | Bloomington |
| 2 | 59a47286cfa9a3a73e51e72d | 1003 | 45235 Worth Avenue | California |
| 3 | 59a47286cfa9a3a73e51e72e | 1008 | 1621 E Monte Vista Ave | Vacaville |
| 4 | 59a47286cfa9a3a73e51e72f | 1004 | 5072 Pinnacle Sq | Birmingham |
| 5 | 59a47286cfa9a3a73e51e730 | 1002 | 1900 E Rio Salado Pkwy | Tempe |
| 6 | 59a47286cfa9a3a73e51e731 | 1010 | 1025 Veterans Pkwy | Clarksville |
| 7 | 59a47286cfa9a3a73e51e732 | 1014 | 155 Los Altos Pkwy | Sparks |
| 8 | 59a47286cfa9a3a73e51e733 | 1012 | 1207 W Century Ave | Bismarck |
| 9 | 59a47286cfa9a3a73e51e734 | 1009 | 6310 E Pacific Coast Hwy | Long Beach |
| 10 | 59a47286cfa9a3a73e51e735 | 1013 | 9901 Brook Rd | Glen Allen |
| 11 | 59a47286cfa9a3a73e51e736 | 1017 | 4325 Sunset Dr | San Angelo |
| 12 | 59a47286cfa9a3a73e51e737 | 1018 | 1307 E Gladstone St | Glendora |
| 13 | 59a47286cfa9a3a73e51e738 | 1015 | 1721 Osgood Dr | Altoona |
| 14 | 59a47286cfa9a3a73e51e739 | 1011 | 740 S Sepulveda Blvd | El Segundo |
| 15 | 59a47286cfa9a3a73e51e73a | 1019 | 390 Northridge Mall | Salinas |
| 16 | 59a47286cfa9a3a73e51e73b | 101 | 25422 El Paseo | Mission Viejo |
| 17 | 59a47286cfa9a3a73e51e73c | 1021 | 700 Du Bois St | San Rafael |
| 18 | 59a47286cfa9a3a73e51e73d | 1020 | 79220 Hwy 111 | La Quinta |


```
{
  "_id": {
    "$oid": "59a47286cfa9a3a73e51e72d"
  },
  "theaterId": 1003,
  "location": {
    "address": {
      "street1": "45235 Worth Avenue",
      "city": "California",
      "state": "MD",
      "zipcode": "20619"
    },
    "geo": {
      "type": "Point",
      "coordinates": [
        -76.512016,
        38.29697
      ]
    }
  }
}
```


For collapsed cells, the corresponding compound value is displayed in the Value tab, which typically means that it will be displayed as a JSON object. To be able to edit the value, you will need to switch to the text view. The same goes for the Document tab. To be able to edit the document, switch to the text view.

## Add document

To add a document to a collection, use the table editor. If you want to create a document from JSON, if the collection is empty or the document has a different structure than the rest of the collection, use the **Add Document** action (right-click on the collection node in the database objects tree or use the Actions menu in the Data tab).

## Remove document

To remove a document visually, use the table editor. For mass operations or other advanced cases, use scripting.

## Edit document

Values can be edited inline in the grid, using the Value tab or the Document tab. Any changes are synced between the different views.

### Add column

To add a column to a single document, use the editor in the Document tab. After the grid is reloaded, the new column will become visible in the grid and you will be able to edit it visually for other documents.

### Remove column

To remove a column completely from a single document, use the editor in the Document tab. To remove a column from the whole collection, use scripting.

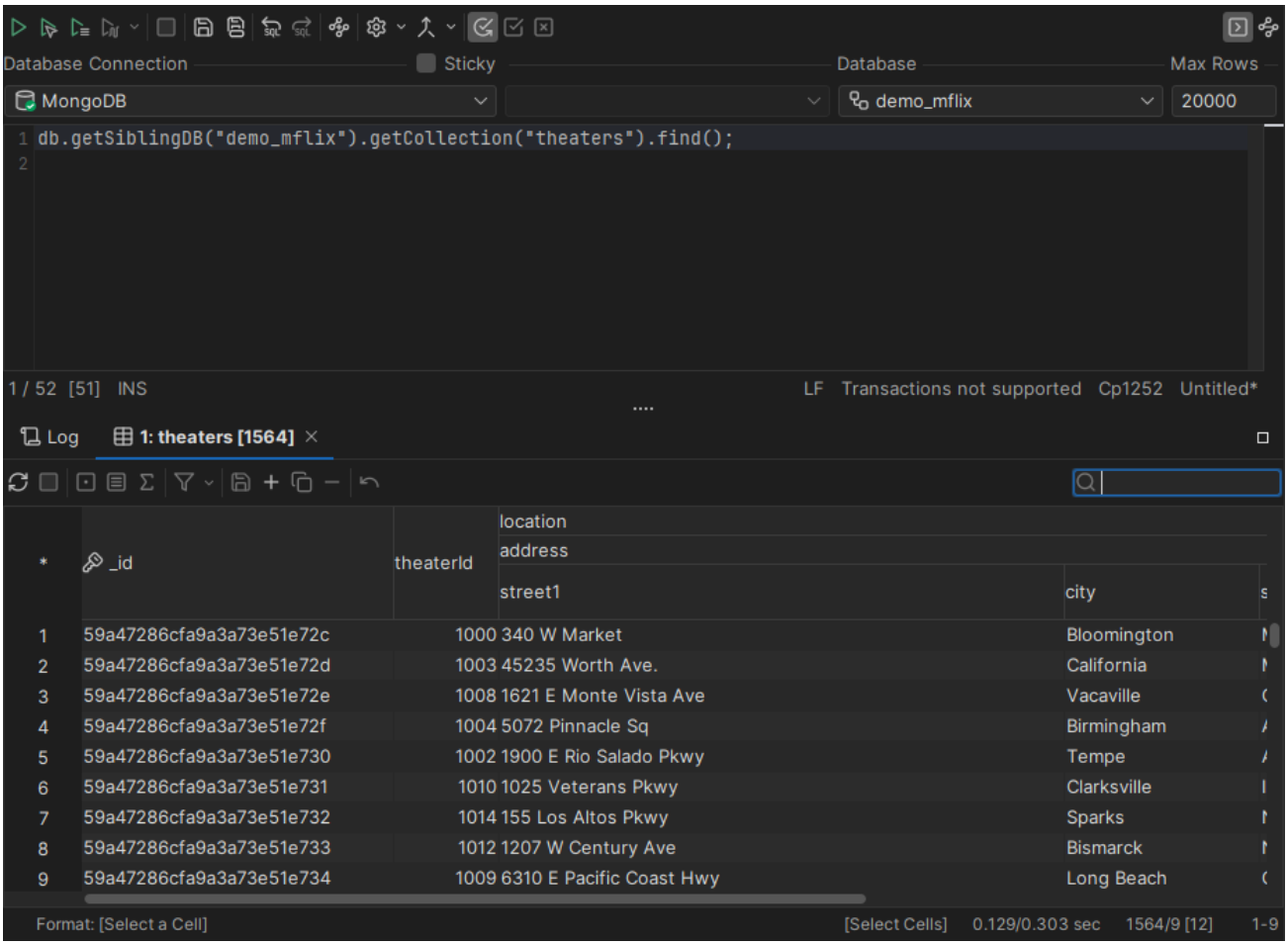
### Advanced cases

For scenarios where the table editor and the editor capabilities in the Value tab and the Document tab aren't enough, use SQL Commander to execute scripts instead.

## Scripting

As with SQL databases, scripting can be used to fetch data and perform other operations. When fetching data, the result set will be displayed in a table that has the same properties as described above.

In the case of MongoDB, the MongoDB Shell syntax is used.




| * | _id | theaterId | location address street1 | city |
|---|---|---|---|---|
| 1 | 59a47286cfa9a3a73e51e72c | 1000 | 340 W Market | Bloomington |
| 2 | 59a47286cfa9a3a73e51e72d | 1003 | 45235 Worth Ave. | California |
| 3 | 59a47286cfa9a3a73e51e72e | 1008 | 1621 E Monte Vista Ave | Vacaville |
| 4 | 59a47286cfa9a3a73e51e72f | 1004 | 5072 Pinnacle Sq | Birmingham |
| 5 | 59a47286cfa9a3a73e51e730 | 1002 | 1900 E Rio Salado Pkwy | Tempe |
| 6 | 59a47286cfa9a3a73e51e731 | 1010 | 1025 Veterans Pkwy | Clarksville |
| 7 | 59a47286cfa9a3a73e51e732 | 1014 | 155 Los Altos Pkwy | Sparks |
| 8 | 59a47286cfa9a3a73e51e733 | 1012 | 1207 W Century Ave | Bismarck |
| 9 | 59a47286cfa9a3a73e51e734 | 1009 | 6310 E Pacific Coast Hwy | Long Beach |

# Working with Binary and BLOB Data

DbVisualizer provides special support for working with Binary/BLOB data in a number of areas, such as:

- Viewing and Editing Binary/BLOB data,
- Exporting Binary/BLOB data,
- Importing Binary/BLOB data

# Working with Large Text and CLOB Data

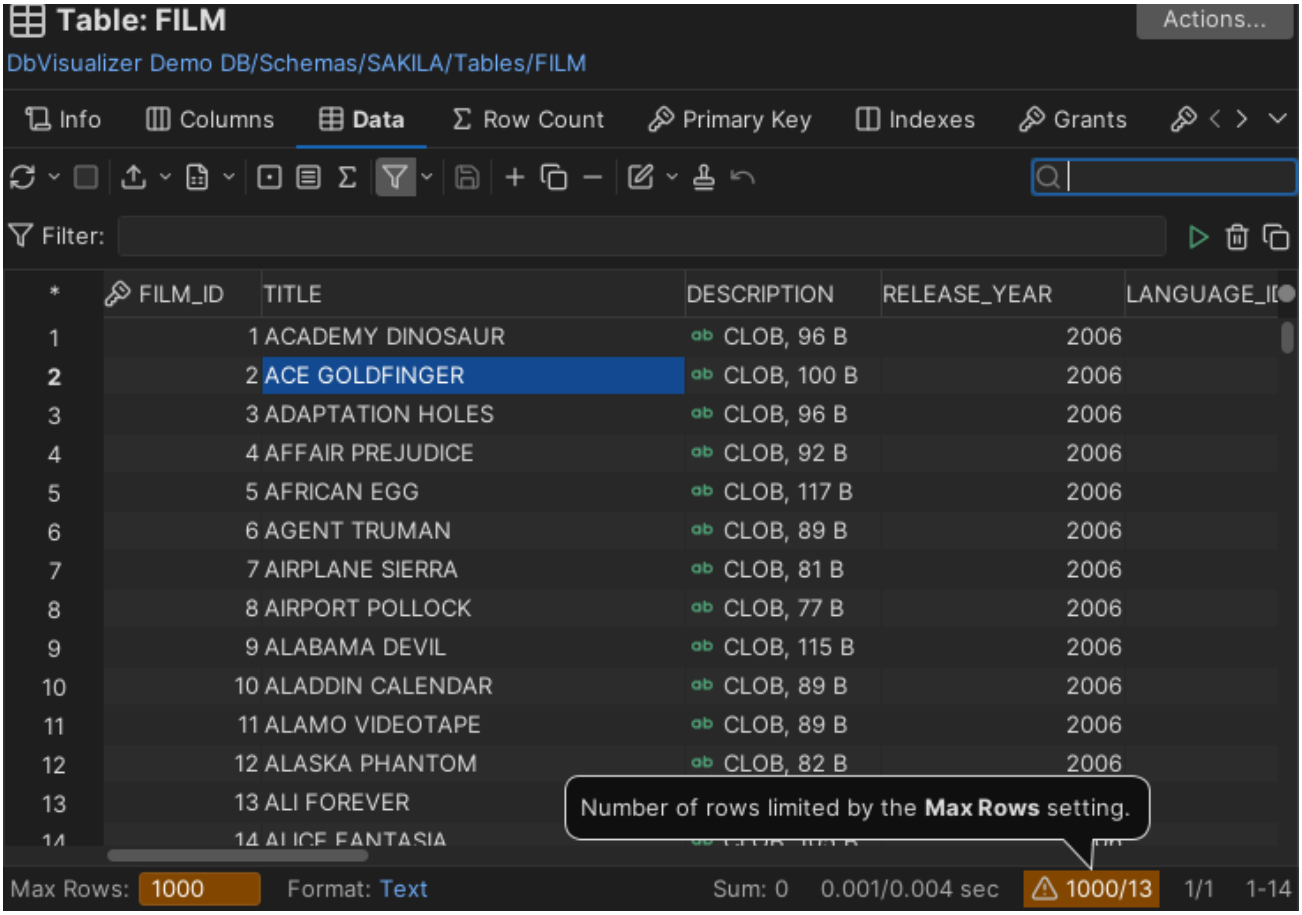
DbVisualizer provides special support for working with Large Text/CLOB data in a number of areas, such as:

- Viewing and Editing Large Text/CLOB data,
- Exporting Large Text/CLOB data,
- Importing Large Text/CLOB data

# Using Max Rows and Max Chars in the Data tab

## Max Rows setting

DbVisualizer by default limits the number of rows shown in the Data tab to 20,000 rows to conserve memory. The following is displayed when this limit is reached.




If the max rows limit prevents you from seeing the data of interest, you should first consider:

1. Using a Where Filter to only retrieve the rows of interest,
2. Exporting the table to a file.

If you really need to look at more than 20,000 rows, you can change the value in the **Max Rows** field in the grid status bar. Use a value of 0 or -1 to get all rows, or a specific number (e.g., 5000) to set a new limit.

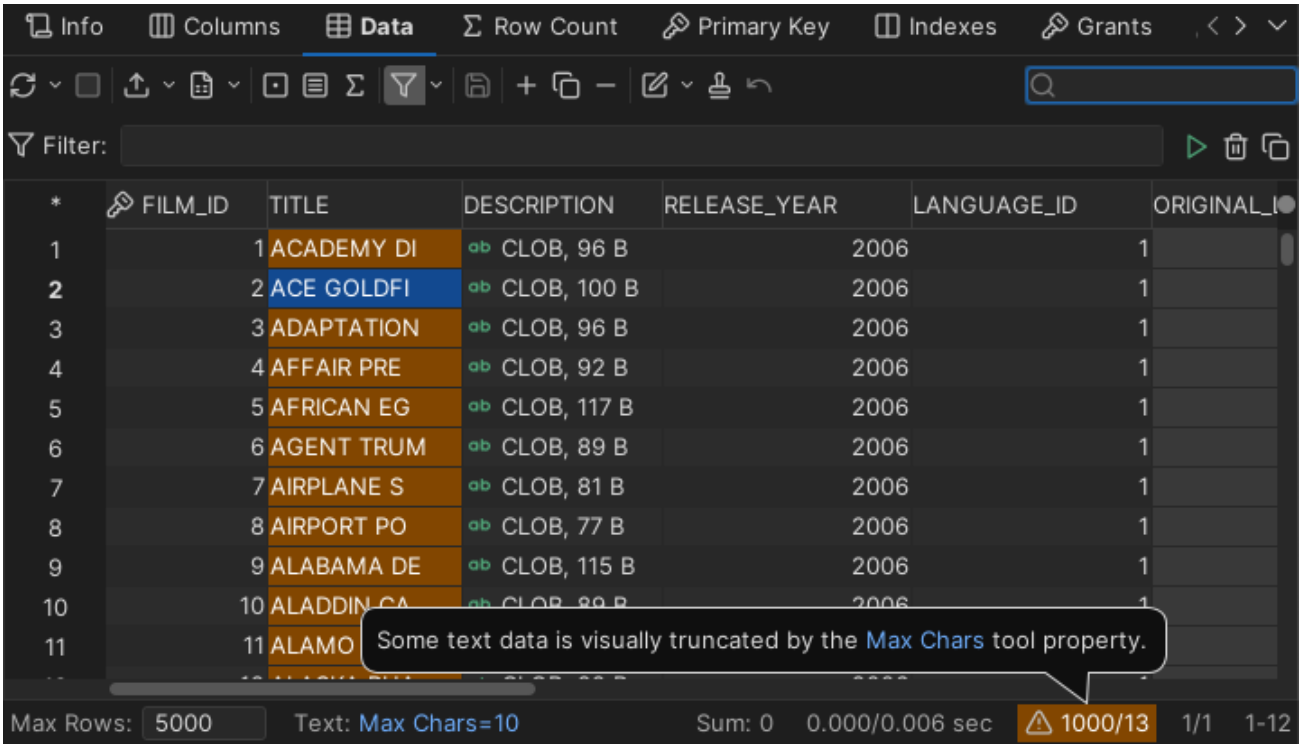
### Max Rows at First Display

By default, opening the Data tab for a table loads all rows, unless there is a **Max Rows** limit. If you have very large tables and don't want to risk memory issues if you accidentally open the Data tab and have no Max Rows limit, you can specify a **Max Rows at First Display** limit. You do this in the Tool Properties dialog, under the **General** / **Table Data** category.

The default is -1, which means no limit. If you set it to a positive number, only the specified number of rows are loaded when the Data tab is first opened for a table. To load more rows, up to the Max Rows setting in the Data tab, click the **Reload** button in the Data tab toolbar.

## Max Chars setting

Text values presented in the Data tab grid may be very long, potentially consuming lots of memory and affecting performance. By default, DbVisualizer visually truncates text values at 1,000 characters, and when it happens, there is an indicator below the grid. The **Max Chars** setting can be modified in the **Tool Properties** / **Data Formats** category or by clicking on any of the **Max Chars** links in the screenshot below.




Along with the highlighted field, a warning pops up close to the field. You can disable this behavior in the Tool Properties dialog under the **General** / **Grid** category.

> To view the full text of a truncated text value, double-click on it to open in the Value viewer

# Changing the Data Display Format

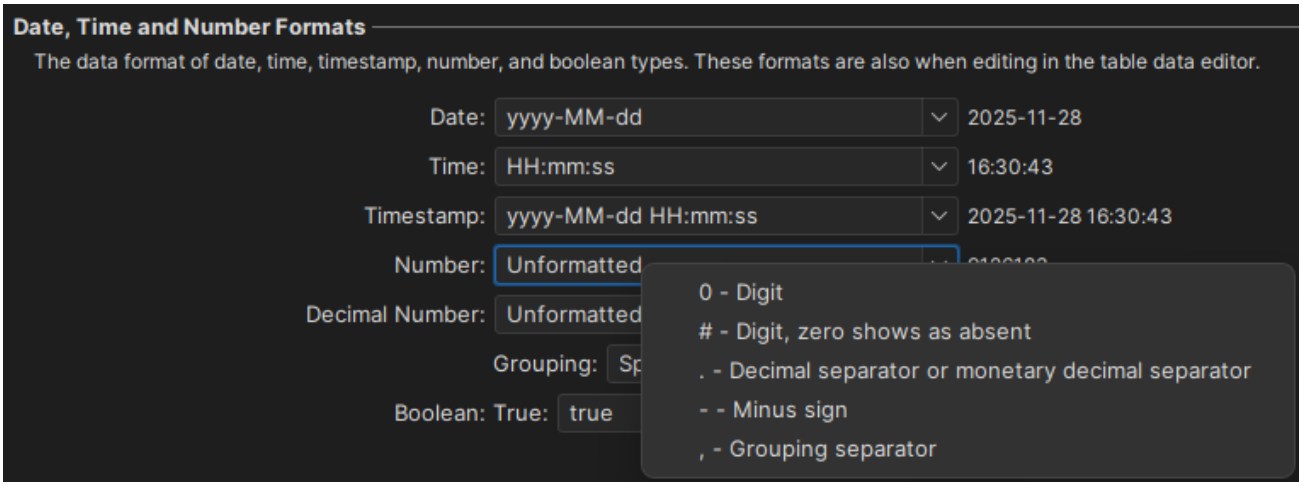
Some data, like numeric and date/time data, can be displayed in many different ways.

To define how to display and enter data in grids and forms in DbVisualizer:

1. Open **Tools → Tool Properties**,
2. Select the **General / Data Formats** node under the **General** tab,
3. Select or enter your preferred format for the different data types.

## Date, Time and Timestamp formats

The lists for date, time and timestamp format contain collections of standard formats. If these formats are not suitable, you can enter your own format in the appropriate field. The tokens used to define the format are listed in the right-click menu when the field has focus and a sample is shown after the field.




The complete documentation for these tokens is available at the following web page: SimpleDateFormat.

## Number formats

The lists for number and decimal number contain collections of standard formats. If these formats are not suitable, you can enter your own format in the appropriate field. The tokens used to define the format are listed in the right-click menu when the field has focus, and complete documentation for these tokens is available at the following web page: DecimalFormat.

The **Unformatted** format for numbers and decimal number allows no grouping character. For Unformatted and decimal numbers a dot (".") is used as the decimal number separator.

# Exporting a Table

## Exporting a Table

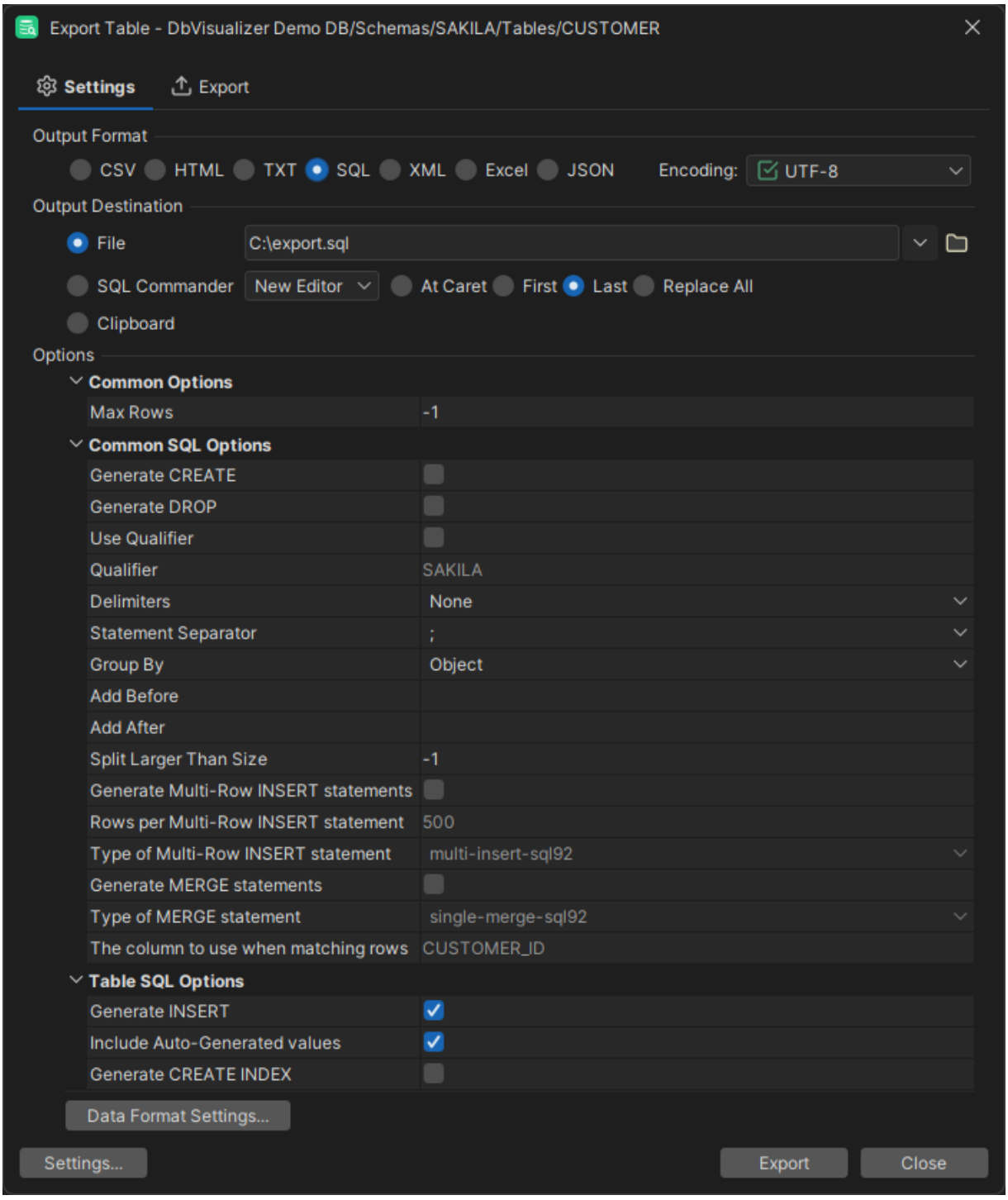
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

You can export an individual table using the Export Table assistant.

To export a table:

1. Select a table node in the **Databases** tab tree,
2. Open the **Export Table** dialog from the right-click menu,
3. Select an **Output Format**, **Output Destination**, and **Options**,
4. Click **Export**.

Below is an example of what the Export Table window looks like. There may be different options based on what database type the table is exported from.




## Output Format

You can export tables in one of these formats: **CSV**, **HTML**, **SQL**, **TXT**, **XML**, **Excel**, or **JSON**.

For the **SQL** and **XML** formats, you can choose to export the DDL and the table data; the other formats only export table data. For the SQL format, also see Exporting to SQL.

You can control whether to use delimited identifiers and/or qualified names by default in the DDL and INSERT statements generated for the SQL format, and you can override the defaults in the Export dialog for a single export operation.

**Excel** export do not support formatted date, time or timestamp. To get the correct date format in Excel you either have to export dates as text (check the **Export Date/Time as Text**) or set the format explicitly in Excel.

## Output Destination

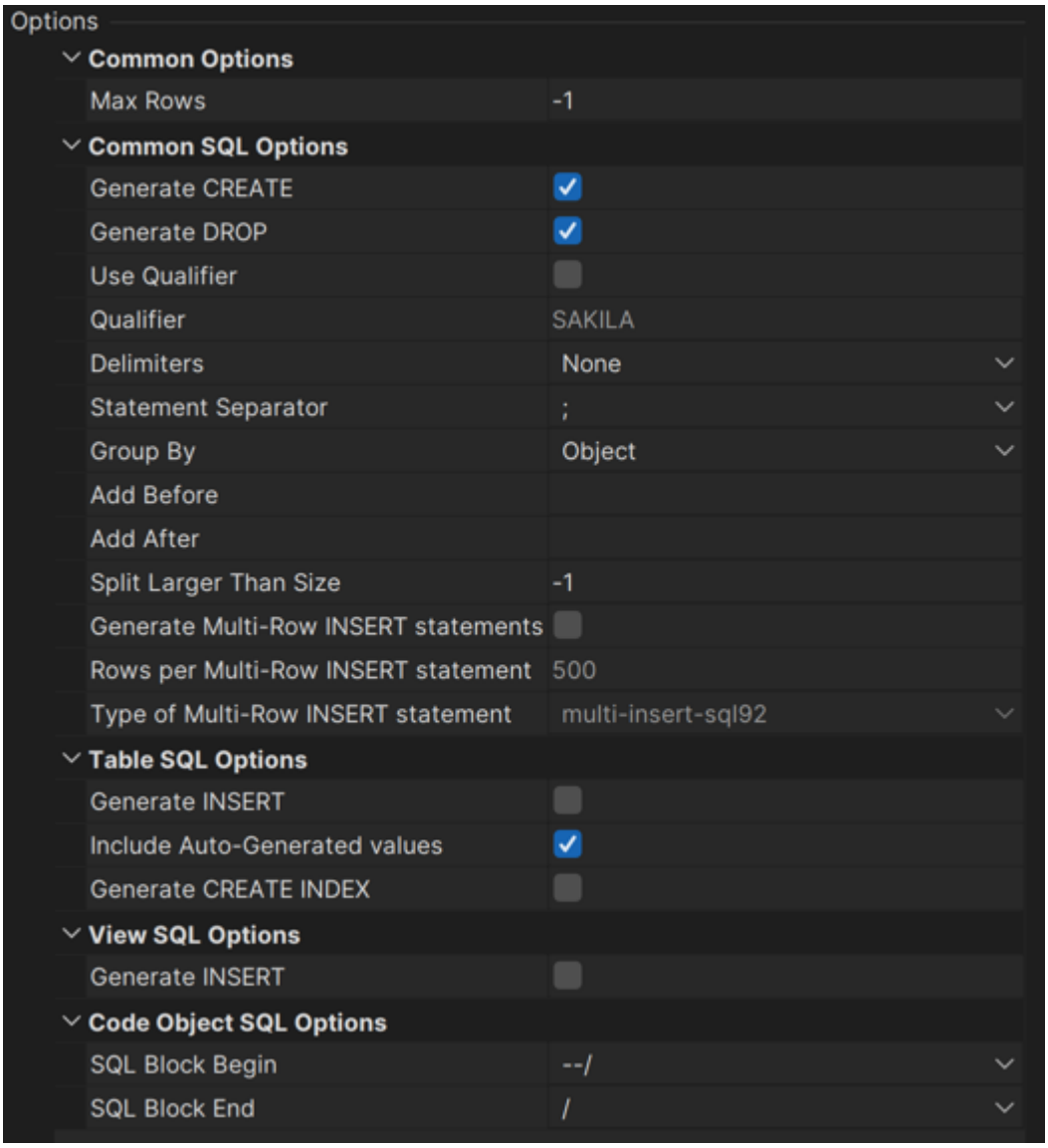
The destination can be one of:

- a file,
- an open or new SQL Commander tab, with options for where in an open SQL Commander to insert the result,
- the system clipboard.

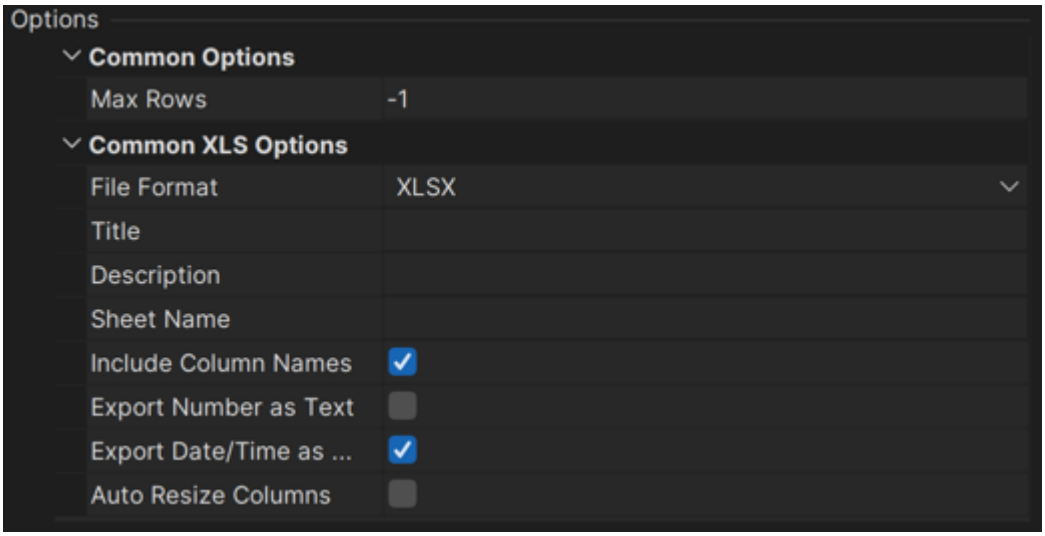
## Options

The **Options** section contains options common to all Output Formats at the top, followed by options for the selected format.

Example of the options for SQL:




Example of the options for Excel:




For the **SQL** and **XML** formats, you can choose to export the DDL, the DDL for indexes for a table and the table data: as INSERT statements for the SQL statement or in one of three XML formats.

For the **Excel** format, you can choose to export table data as either in the **XLSX** (default) or the legacy **XLS** format.

Most formats also let you specify other options, such as delimiters, title and descriptions. Just select an Output Format to see which options are available. All options are described in the context of the @export command, as the Export dialog is just a GUI for the command.

You can adjust the **Data Formats** specifically for the exported table data. By default, the formats defined in **Tool Properties** are used, but sometimes you need to export dates and numbers in a different format because you intend to import the data into a different type of database.

> If you are exporting table data in the **SQL** format from one database type (e.g. Oracle) to import it in a database of a different type (e.g. PostgreSQL) by executing the generated script, you need to be aware of differences in the literal formats for Date, Time and Timestamp data. If you connect to the other database using a JDBC client like DbVisualizer, you can select the **JDBC escape** format for these data format. This generates literals that the JDBC driver converts into a format the target database can interpret.

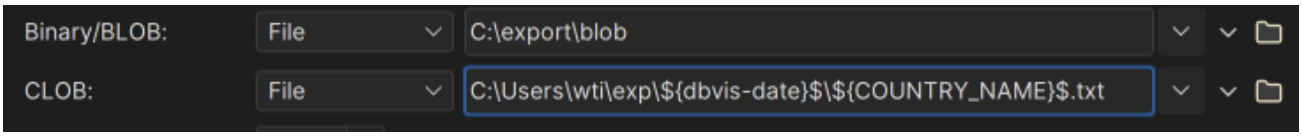
In the **Data Format Settings** dialog you can also specify how to quote text data and how to handle quotes within the text value.

## Using Variables in Fields

You can use some of the pre-defined DbVisualizer variables ( `${dbvis-date}$` , `${dbvis-time}$` , `${dbvis-timestamp}$` , `${dbvis-connection}$` , `${dbvis-database-type}$` and `${dbvis-object}$` ) in all fields that hold free text (e.g. title and description fields) and as part of the file name field.

## Exporting Binary/BLOB and CLOB Data

You can use the export assistant to export Binary/BLOB and CLOB data. You enable this by choosing **File** as the data format for **Binary/BLOB** and/or **CLOB** data. Optionally, you can specify the directory or filename pattern for the data files. If you do not specify a directory or filename pattern, the operating system's default directory for temporary files (e.g. C:\TEMP or /tmp) is used.




A pattern is a path with fixed and variable parts, where the variable parts are expressed as DbVisualizer variables, e.g. C:\Users\wti\exp\${dbvis-date}$\${COUNTRY_NAME}$.txt. You can select variables to insert at the current caret position in the path field from the dropdown. The variables that can be used are the predefined DbVisualizer variables plus variables for each column in the table, e.g. `${COUNTRY_NAME}$` in the example above. Another special variable that can be helpful here is `${dbvis-column-name}$`. If a table has multiple BLOB or CLOB columns, you can use it in the pattern to export the columns to separate files, e.g. C:\Users\wti\exp\${dbvis-column-name}$.txt. All variables that can be used are listed in the menu that is displayed when you click the blue arrow to the right of the field. You can select a variable from the menu to insert it in the pattern field at the caret position.

The data for each individual value of this type is then exported to a separate file and a DbVisualizer variable referencing the file is inserted in the main export file.

**Example:**

Assume you have a table with multiple pictures and want them exported in individual files, for instance a **BOOK** table with the columns **ISBN** and pictures of **FRONT** and **BACK**:

```
ISBN       FRONT                BACK
---------- -------------------- --------------------
0345391802 BINARY, 4,998 Bytes BINARY, 4,998 Bytes
0345391810 BINARY, 4,998 Bytes BINARY, 4,998 Bytes
```

If you specify the output as `D:\tmp\${ISBN}$-${dbvis-column-name}$.png`, you will get the following image files:

- D:\tmp\0345391802-FRONT.png
- D:\tmp\0345391802-BACK.png
- D:\tmp\0345391810-FRONT.png
- D:\tmp\0345391810-BACK.png

## Saving And Loading Settings

If you often use the same settings, you can save them as the default settings for this assistant. If you use a number of common settings, you can save them to individual files that you can load as needed. Use the **Settings** drop-down button menu to accomplish this:

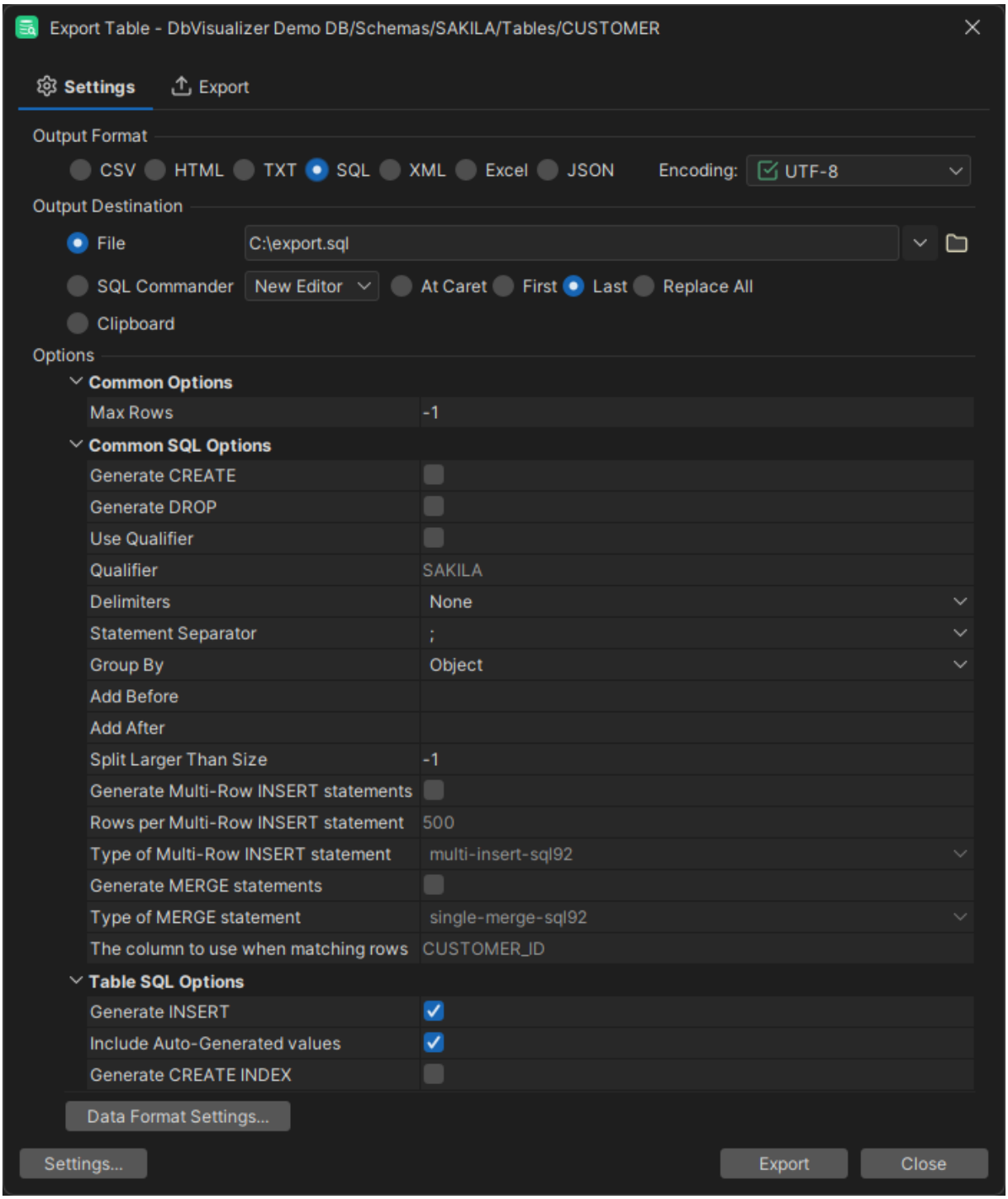
- **Save as Default Settings** Saves all format settings as default. These are then loaded automatically when open an Export Schema dialog
- **Use Default Settings** Use this choice to initialize the settings with default values
- **Remove Default Settings** Removes the saved defaults and restores the regular defaults
- **Load…** Use this choice to open the file chooser dialog, in which you can select a settings file
- **Save As…** Use this choice to save the settings to a file
- **Copy Settings to Clipboard** Copies the settings to the system clipboard
- **Copy Settings to Clipboard** Use this choice to copy all settings to the system clipboard. These can then be pasted into the SQL Commander to define the settings for the @export editor commands.

## Other Ways to Export Table Data

- Export all or selected tables with the Export Schema assistant
- Export a subset of the table data with the @export command
- Export query results by exporting the grid with the query results.

# Exporting to SQL

Compared to other formats, exporting to SQL has a lot more configurability. What follows is a brief explanation of each option.



## Table Export

COMMON OPTIONS

- **Max Rows** - How many rows to export. **-1** means all rows.

COMMON SQL OPTIONS

- **Generate CREATE** - Generates **CREATE** statements. Do not enable this option if the objects already exists in the target database.
- **Generate DROP** - If **Generate CREATE** is selected, also generates **DROP** statements first.
- **Use Qualifier** - Uses the **Qualifier** in the next step as a prefix to the object name in all generated statements.
- **Qualifier** - The qualifier to use to prefix objects. For example: **SAKILA** will result in **INSERT INTO SAKILA.\<object>**.
- **Delimiters** - Object name delimiters to use in all statements. For example **[]** will result in **[\<object>]**.
- **Statement Separator** - The statement separator, which will be pasted at the end of each statement.
- **Group By** - Generate all statements for one object at a time, or generate each statement for all objects together. This setting is only relevant when exporting more than one object. In that case, grouping by **statement** type will put all the statements together (**DROP**,**DROP**,**CREATE**..), whereas **Object** will cause the grouping by table (**DROP CREATE INSERT**, **DROP**...) instead.
- **Add Before** - Custom SQL that will be added **before** the generated SQL.
- **Add After** - Custom SQL that will be added **after** the generated SQL.
- **Split Larger Than Size** - Split results larger than this (size [g|G|m|M|k|K]) into multiple files, or **-1** to not split. Example: **4k** means split the file if it exceeds 4 kibibytes (4096 bytes).
- **Generate Multi-Row INSERT statements** - Instead of having one insert per row, combine all inserts into one big statement.
- **Type of Multi-Row INSERT statement** - Different databases have different syntaxes for multi-row inserts. **INSERT INTO ... VALUES (),(),()** might not work for all target databases. Choose the type according to the target database.
- **Generate MERGE statements** - Instead of generating **INSERT** statements, **MERGE** statements can be generated instead. This can be useful the target already has partial data.
- **Type of MERGE statement** - Different databases have different syntaxes for **MERGE** statements. Choose the type according to the target database.
- **The column to use when matching rows** - When using **MERGE**, a column needs to be matched against in order to know when a row is a duplicate or not. Choose the column(s) (usually the primary key(s)) which uniquely identifies each row.

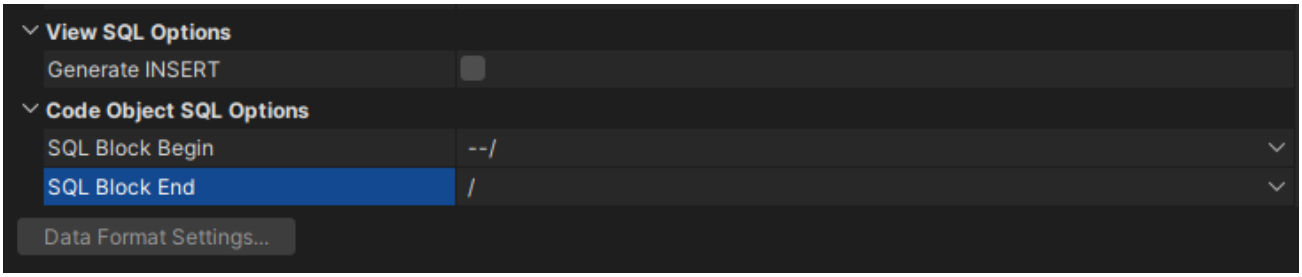
TABLE SQL OPTIONS

These options are specific to exporting one or more tables.

- **Generate INSERT** - If unchecked, does not generate any statements to create rows with data, i.e. **INSERT** (or **MERGE**) statements.
- **Include Auto-Generated values** - Include Auto-Generated values (AUTO_INCREMENT, and GENERATED ALWAYS or BY DEFAULT) in the INSERT statement.
- **Generate CREATE INDEX** - Generate **CREATE INDEX** statements for existing indexes for the selected tables.

## Schema Export

When exporting a schema, additional options are available.




VIEW SQL OPTIONS

- **Generate INSERT** - If unchecked, does not generate any statements to create rows for the selected views.

CODE OBJECT SQL OPTIONS

- **SQL Block Begin** - The delimiter to use before the start of an SQL Block, e.g., **CREATE PROCEDURE**
- **SQL Block End** - The delimiter to use after the end of an SQL Block, e.g., **CREATE PROCEDURE**

# Importing Table Data

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

You can import data using the Import Table Data wizard.

You can import data from a file into an existing table or to a new table. The import source can be either a CSV file or an Excel file ( `.xls` or `.xlsx` ). The steps are almost identical:

1. Select the table node for the table you want to import to, or the **Tables** node if you are importing to a new table, in the **Databases** tab tree,
2. Open the **Import Table Data** wizard from the right-click menu,
3. Specify the input file on the first wizard page (CSV or Excel file),
4. [Excel only]: If the input file is an Excel file, you are asked to choose the Excel sheet to import on the next page.
5. Specify file format and other options,
6. Specify data formats and the data type per column,
7. Adjust details about the destination table,
8. Click **Import** on the last page.

> Instead of choosing Import Table Data from the right-click menu, you can **drag and drop a file** from the operating system's file manager on the Tables node or a table node.

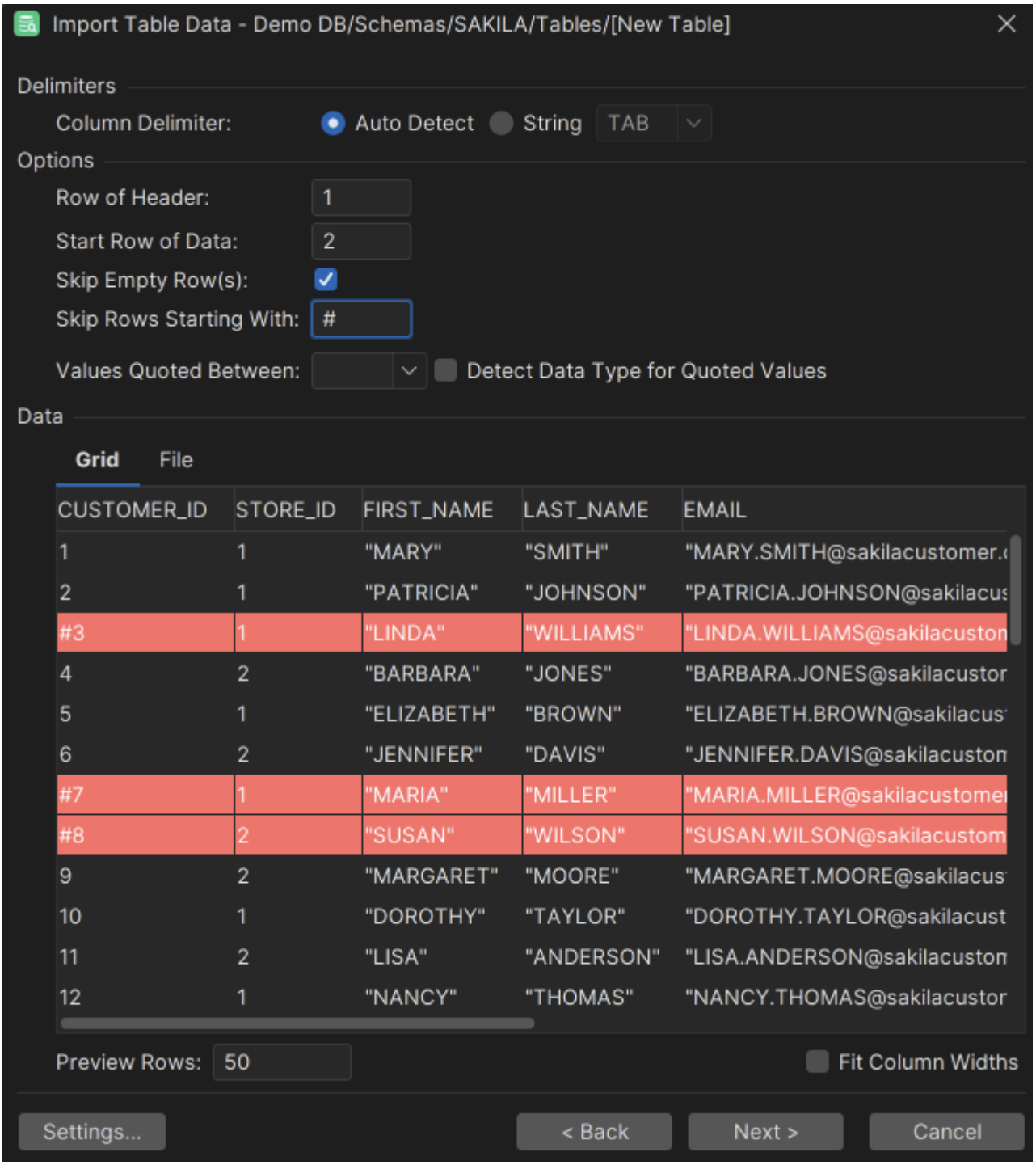
How many INSERT statements to execute during the import process before committing the changes can be specified in the **Properties** tab for the connection, in the **Transaction** category.

## Input File Format and Other Options

On the File Format page, you specify what and how the data in the source file should be imported. This includes specifying what row to start the import from and if empty rows should be skipped.

DbVisualizer supports import of CSV and Excel files (both the `.xslx` and the legacy `.xsl` file formats).

CSV format page



In the **Delimiters** section, define the character that separates the columns in the file. If you enable **Auto Detect**, DbVisualizer tries to auto detect which delimiter is used. Examples of auto detected delimiters are:

- comma ","
- tab "TAB"
- semicolon ";"
- percent "%"
- pipe "|"
- Using Unicode Code Points such as `\u2656`.

> You can specify any character sequence as a delimiter, but it must not contain more than four characters.

You can use the **Options** area to further specify how to read the input file. For instance, you can skip rows that start with a specific string (e.g. comments) or strip quotes from text data. If you leave **Values Quoted Between** empty, data will be imported as is (which may cause problems if you have text strings that include the column delimiter).

The **Data** section at the bottom of the page shows a preview of the parsed data in the **Grid** tab and the original source file in the **File** tab. If a row in the Grid tab is red, it indicates that the row will be ignored during the import process. This happens if any of the **Options** settings result in rows not being qualified.

If the checkbox **Detect Data Type for Quoted Values** is checked the data type is also detected for that value. E.g a value "1" will be detected as a Number and not a String.

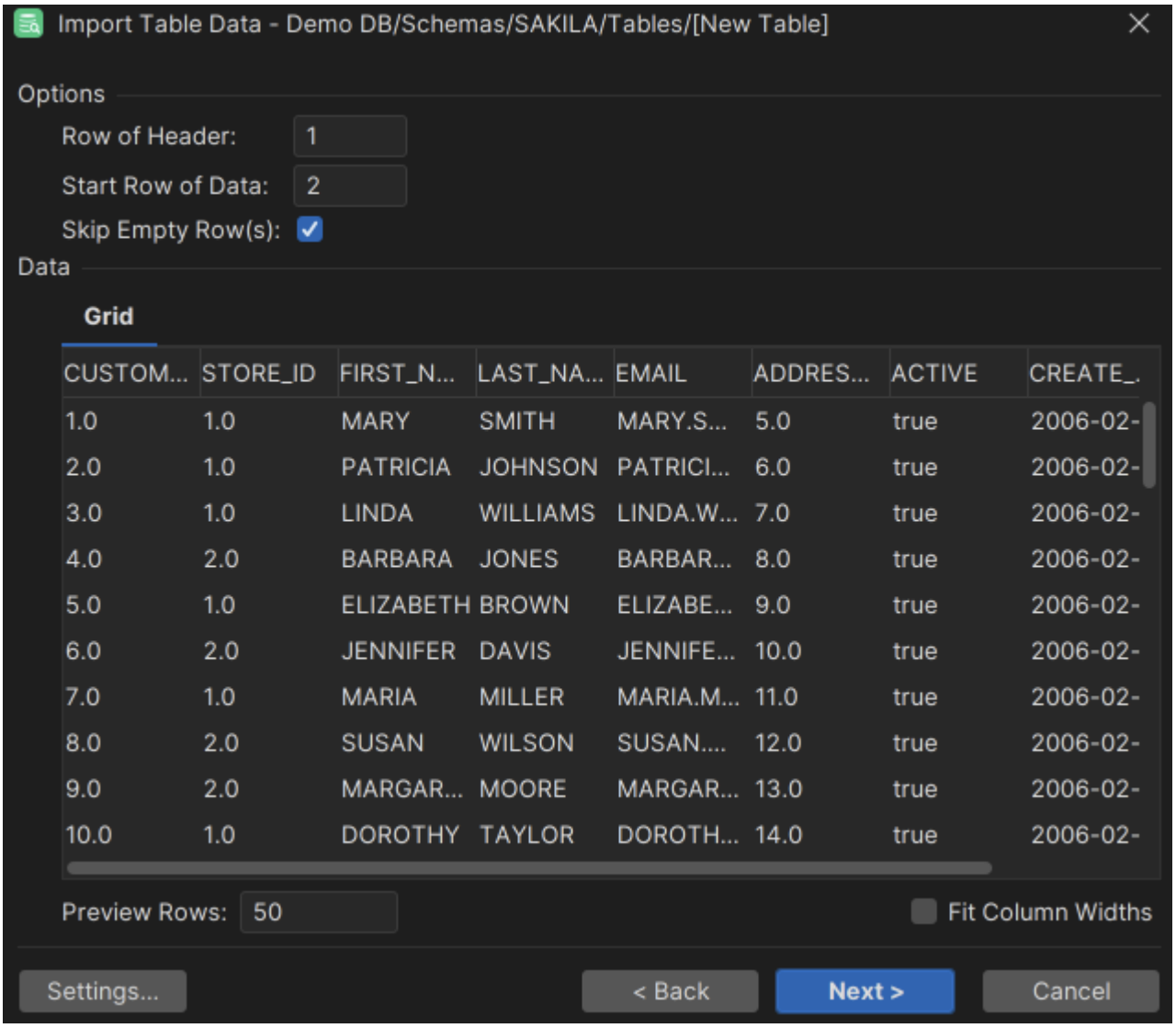
## Excel format page

The Excel format page is very much like the CSV format page.

As Excel is from start organized in columns and rows the **Column delimiter** setting is not applicable to Excel files. The **Skip Rows Starting With** and the **Text Quoted Between** options are also not supported for Excel.

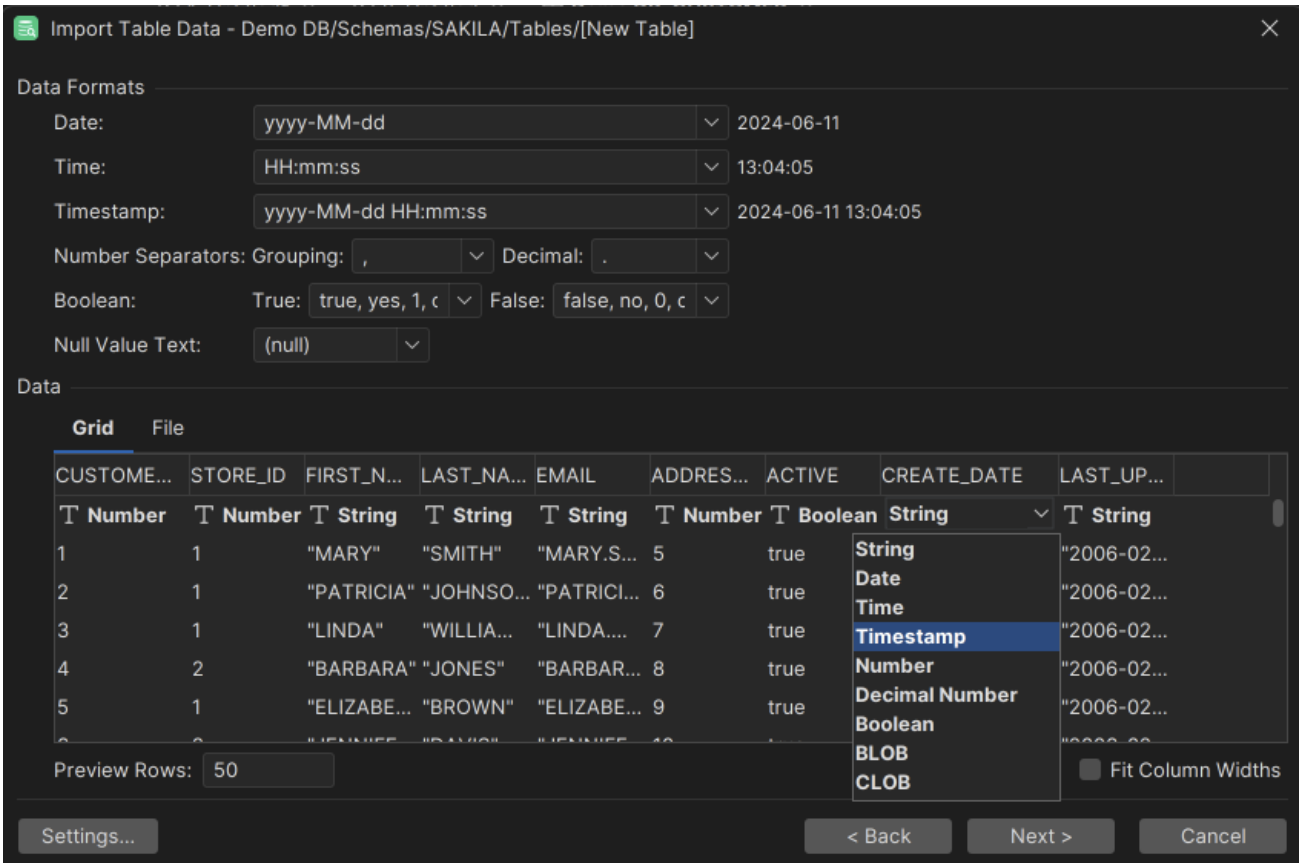
As shown in the snapshot below there is no **File** tab for Excel files.

The **Grid** tab shows a preview of the data, just as in the CSV case.




## Data Formats and Data Type Per Column

The Data Formats page is used to define formats for some data types. The first row in the preview grid contains a data type drop-down lists. DbVisualizer tries to determine the data type for each column by looking at the value for the number of rows specified as **Preview Rows**. If this data type is incorrect for a column, use the drop-down lists to select the appropriate type.



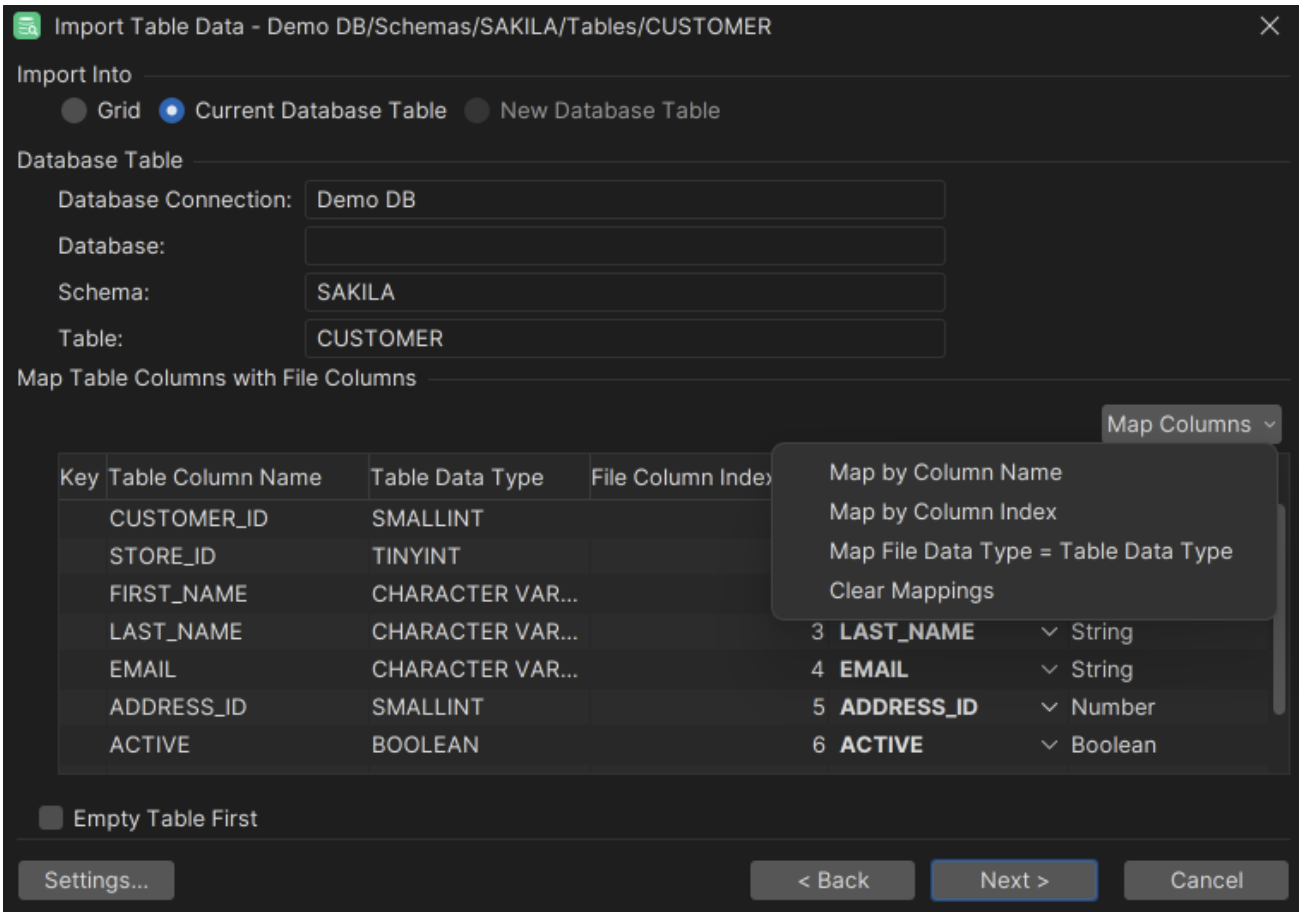

If you need to change the data type for a number of columns, e.g. set them all to String, you can Copy/Paste the data type. First change it for one of the columns using the drop-down, select and copy that new data type value and then select the data type for all other columns and use paste to change them all at once. If you make a mistake, you can change the **Preview Rows** value to let DbVisualizer determine the types again.

If you import to an existing table, there is yet another way to adjust the data types for the file columns, described in the next section.

## Matching Columns and Data Types for an Existing Table

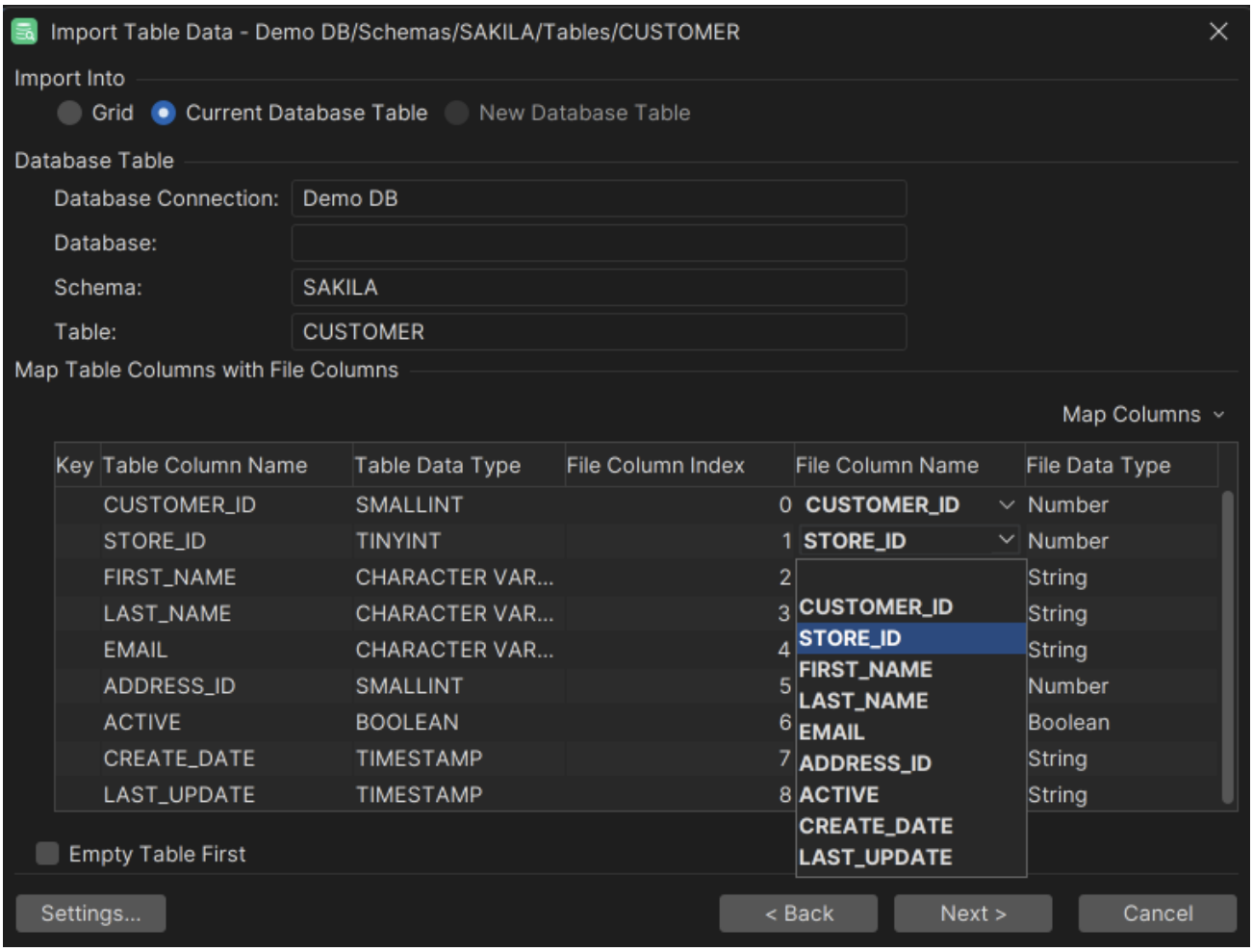
When you are importing to an existing table, the Import Destination page provides two options: **Grid** and **Current Database Table**. You can use the **Grid** choice to import the data into a grid that is presented in its own window in DbVisualizer if you just want to just process the data in some way without saving it in the database.

When the **Current Database Table** choice is selected, the page shows information about the table into which the data will be imported in the **Map Table Columns with File Columns** grid shows the columns in the selected database table and the columns in the source file.




DbVisualizer automatically associates the columns in the source file with the columns in the target table in the order they appear. If the columns appear in a different order in the file than in the table, but they are named the same, you can use the auto-mapping menu in the upper right corner of the **Map Table Columns with File Columns** grid to automatically map the columns by name. **Map by Column Name** and **Map by Column Index** do exactly what it sounds like. **Map File Data Type = Table Data Type** sets the **File Data Type** for each column to the type of the corresponding table column.

If the column names are different between the file and the table and also appear in different order, you can manually map them using the drop-down lists in the **File Column Name** field. Choose the empty choice in the columns drop-down to ignore the column during import.




You can use copy/paste of the values in the **File Column Name** and **File Data Type** fields to quickly fill the selection of cells instead of manually selecting the correct data in the drop-downs.

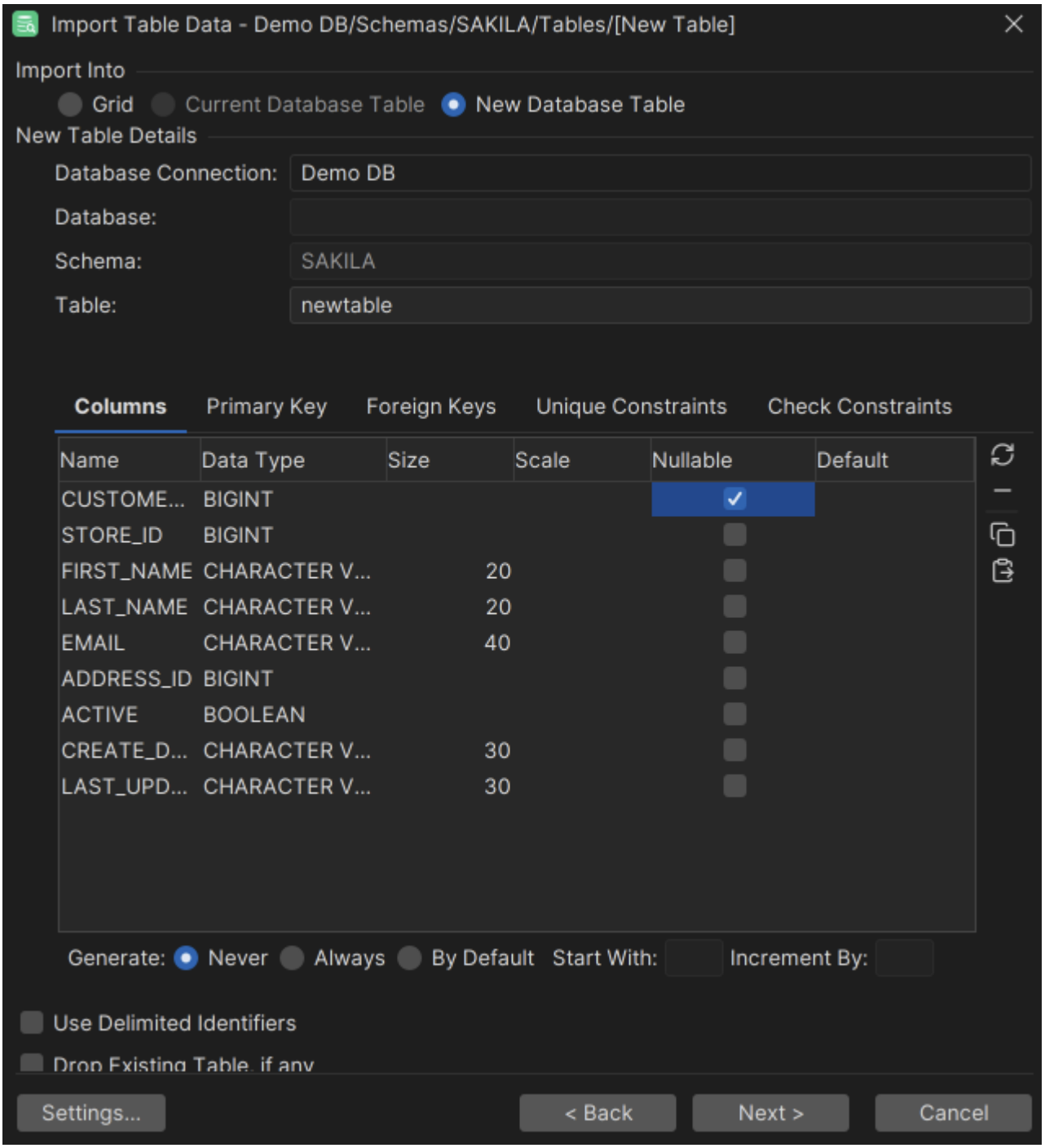
There are two checkboxes at the bottom of the page:

- **Use Delimited Identifiers**: check this if you want the SQL statements for importing the table to use delimited identifiers; in other words, if you want to use table and column names with special characters, mixed case, or anything else that requires delimited (quoted) identifiers.
- **Empty Table First**: check this if you want to clear/empty the table before import. Choose between **Truncate** or **Delete**. Truncate is faster as it usually will clear the data without occupying the rollback management in the database.

## Adjusting Table Declaration for a New Table

When you import into a new table, the Import Destination page provides two options: **Grid** and **New Database Table**.

- Use the **Grid** choice to import the data into a grid if you just want to just process the data in some way without saving it in the database. The data is presented in its own window in DbVisualizer and can be formatted, filtered, viewed and exported again.
- Use the **New Database Table** choice to import into a table; you are presented with a field for the table name and a number of tabs for column and constraint declarations. The **Columns** tab is filled out based on the source data and the data types from the Data Formats page.




Note that it is not always possible to find a database specific type for the data format specified on the Data Format page. You must then pick the correct type from the **Data Type** drop-down menu. The size for string column types may also need to be adjusted. By default, the size is set to the maximum number of characters found for the column in the number of rows specified as **Preview Rows**, adjusted up to the next power of ten. You can ignore certain columns by removing them in the **Columns** tab. **Keys** and other constraints can be created using the other tabs.

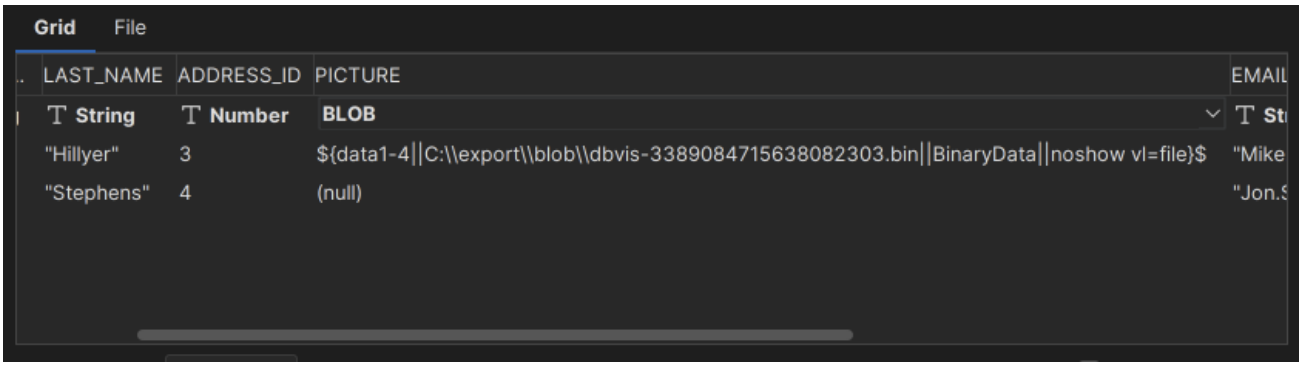
You can go back to the Data Format page and increase the **Preview Rows** value if you believe that it will help DbVisualizer to pick better defaults. If you do so, you need to click the **Reload** button when you come back to this page to rescan the source data and get new default values.

If you make a mistake or if the import fails and you have to go back and make adjustments before you import again, make sure you enable **Drop Existing Table, if any**. It is disabled by default to prevent you from accidentally dropping an existing table when you intend to import to a new table, but if the import fails, the new table may already have been created so it needs to be dropped before a new table with your adjusted input can be created.

There is also a **Use Delimited Identifiers** checkbox. Check this box if you want the SQL statements for importing the table to use delimited identifiers; in other words, if you want to use table and column names with special characters, mixed case, or anything else that requires delimited (quoted) identifiers.

## Importing Binary/BLOB and CLOB Data (CSV and SQL Only)

If you have exported data to a CSV file using DbVisualizer, use the Import Table Data feature to import it. On the Data Format page, ensure that the format for the source file column is set to **BLOB** or **CLOB**.




If you have exported Binary/BLOB and CLOB data as an SQL script, you just run the script in the SQL Commander to import it. When the SQL Commander encounters a variable that refers to a file, it reads the file and inserts the content as the column value.

## Running the import

The last wizard page contains some basic settings for the import.

- **Import All rows/Import:** Used to limit the number of rows to import
- **Single Inserts:** If checked, each row will be sent to the database and inserted one-by-one. The **commit interval** specifies how often a commit should be performed.
- **Batch Import:** If checked the import will be performed in batches. This may improve performance considerably making the import faster. Note that different databases/drivers may have different level of support for batch import. The **rows per batch/commit** setting controls how large batches should be sent to the database. There will be a commit for each batch.
- **Keep Window After import:** If checked the Import window will remain open after import. If the import fails the window will always remain open.
- **Stop On Error:** If checked the import will stop if an error occurs.

## Saving And Loading Settings

If you often use the same settings, you can save them as the default settings for this assistant. If you use a number of common settings, you can save them to individual files that you can load as needed. Use the **Settings** drop-down button menu to accomplish this:

- **Save as Default Settings** Saves all format settings as default. These are then loaded automatically when open an Export Schema dialog
- **Use Default Settings** Use this choice to initialize the settings with default values
- **Remove Default Settings** Removes the saved defaults and restores the regular defaults
- **Load…** Use this choice to open the file chooser dialog, in which you can select a settings file
- **Save As…** Use this choice to save the settings to a file
- **Copy Settings to Clipboard** Copies the settings to the system clipboard

## Other Ways to Import Table Data

If you have a script containing INSERT statements for all data, you can execute it in the SQL Commander.

There is also the @import client-side command that is executed in the SQL Commander. With it you can define the import setup using commands only and let it run both in the DbVisualizer GUI or in the command-line interface for DbVisualizer, `dbviscmd`.

The special LOAD DATA option for MySQL, MariaDB and SingleStore is available in DbVisualizer, and it enables fast import from a local or server file. To open the Load Table Data dialog, right-click on a table and select **Import Table Data->Load Table Data**. This option requires special driver property setting and may also require privileges on the server.

## Known limitations

- Excel files cannot contain CLOB/BLOB type of data (e.g. images etc). Cells with this kind of data are imported as empty.
- There is a size limitation when importing Excel files with the `.xls` filename extension. The size limitation is roughly 20 megabytes, depending on your configuration and how much memory is used for other things. Increasing DbVisualizer max memory may allow you to import larger files.

# Comparing Tables

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

You can compare different aspects of a table to other tables and/or result set grids.

For instance, to compare the DDL for a table to the DDL for another table:

1. Open the **DDL** tab for the table,
2. Open the **DDL** tab for the other table,
3. Select **Compare** from the right-click menu in one of the **DDL** tabs to compare their text content.

To compare the table data to the data of another table or a result set:

1. Open the **Data** tab for the table,
2. Open the **Data** tab for another table or execute an SQL query to open a result set tab,
3. Select **Compare** from the right-click menu in one of the tabs to compare their grid content.

You can do the same for all the other Object View sub tabs containing a grid, such as the **Primary Key** or **Columns** tab.

# Viewing Table Relationships

Use the References graph to see how a table is related to other tables through Foreign Keys. This is done by selecting the **References** sub tab of the Database object view, as further described below

You can view relations to/from one or more tables in different graph layouts: **Hierarchical**, **Organic**, **Orthogonal**, or **Circular**.Layout settings can be changed in the **Graph Control** area, which is shown or hidden with the settings toggle button in the toolbar. For instance, you can select how much information to include for each table in the graph: just the **Table Name**, the **Primary Key** column(s) or all **Columns**, which links to include, and whether or not to highlight links for selected tables. Settings vary slightly depending on whether you are viewing relations for a single table or multiple tables, and some settings (e.g. **Links to Columns**) are only available in **Hierarchical** layout.

If you choose the **Hierarchical** layout and select **Links to Columns** and **Highlight Links**, the graph will for all selected table nodes color code links and columns to indicate usage:

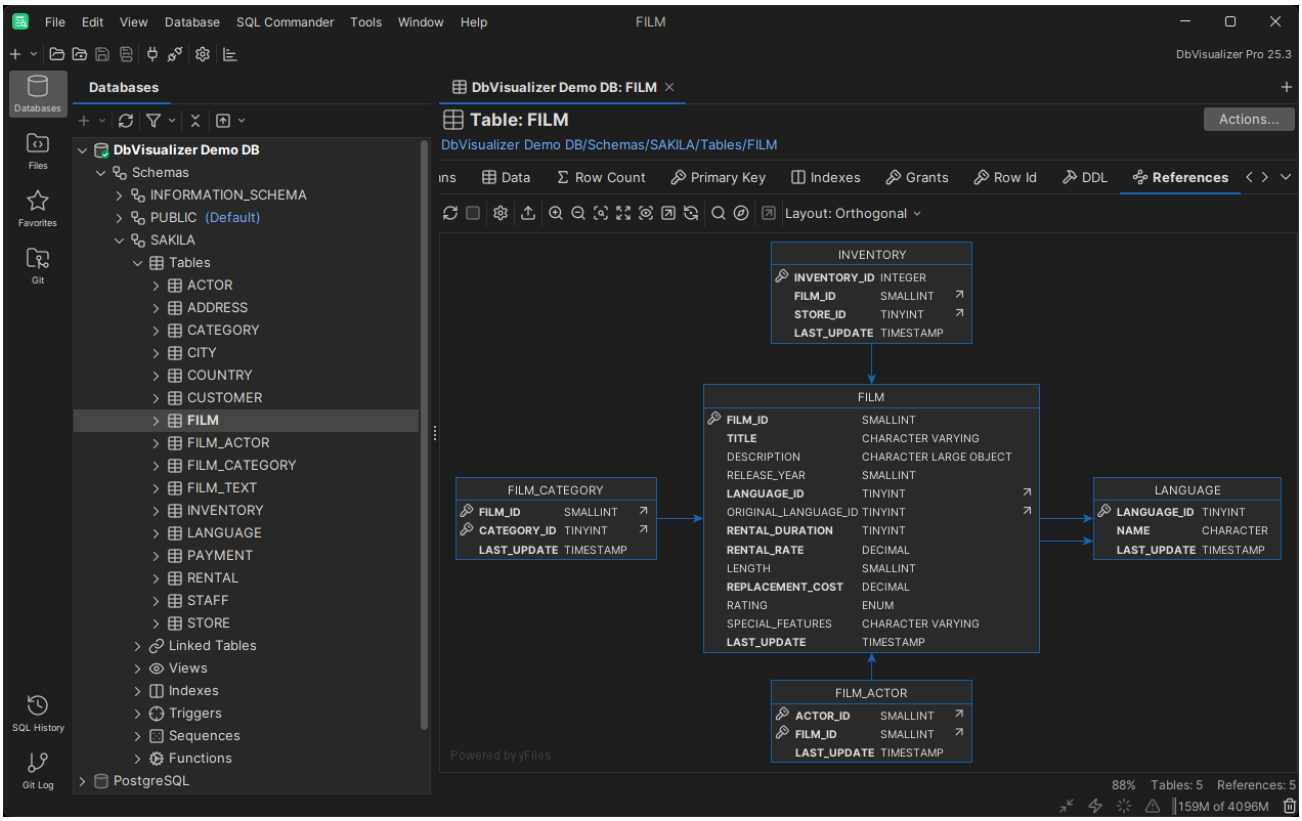
- foreign key columns and outbound links are green
- primary key columns and inbound links are red
- bidirectional columns (used for both inbound and outbound links) and bidirectional links (where both source and target tables are selected) are orange

The graph can be **Exported** to a file in **JPG**, **GIF**, **PNG**, **SVG**, **PDF** or **EMF**. It can also be saved s a **GML** (Graph Modeling Language) file that you can then open in the yEd tool from yWorks for further manipulation. The **GML** format is saved using **Save As** in the right-click menu. You can also control whether the table names should be qualified with the schema/catalog in the graph.

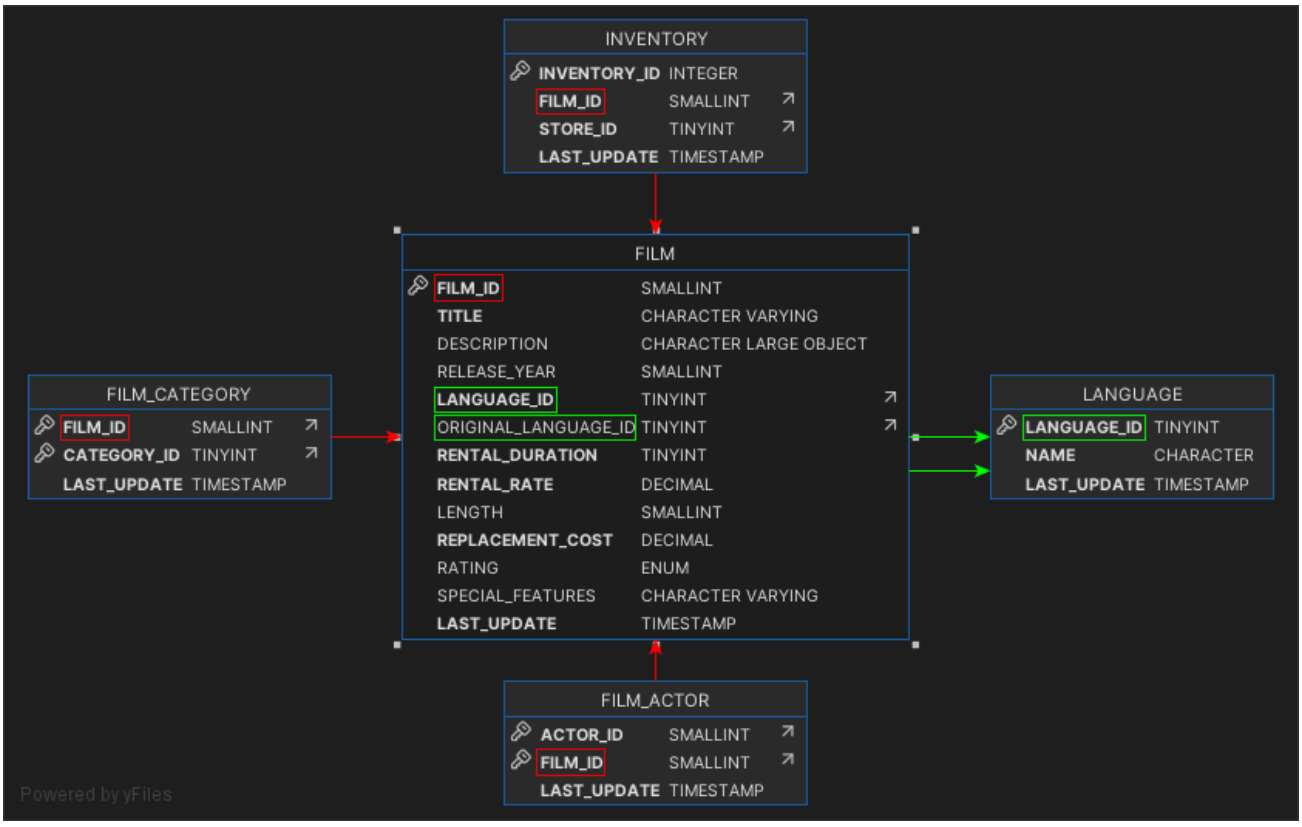
When opened on a single table, the graph will only show links to/from this table. You can choose which links to show: **Inbound**, **Outbound**, **All**, or **None**.

You can also view entity relationships to get an overview of multiple tables.

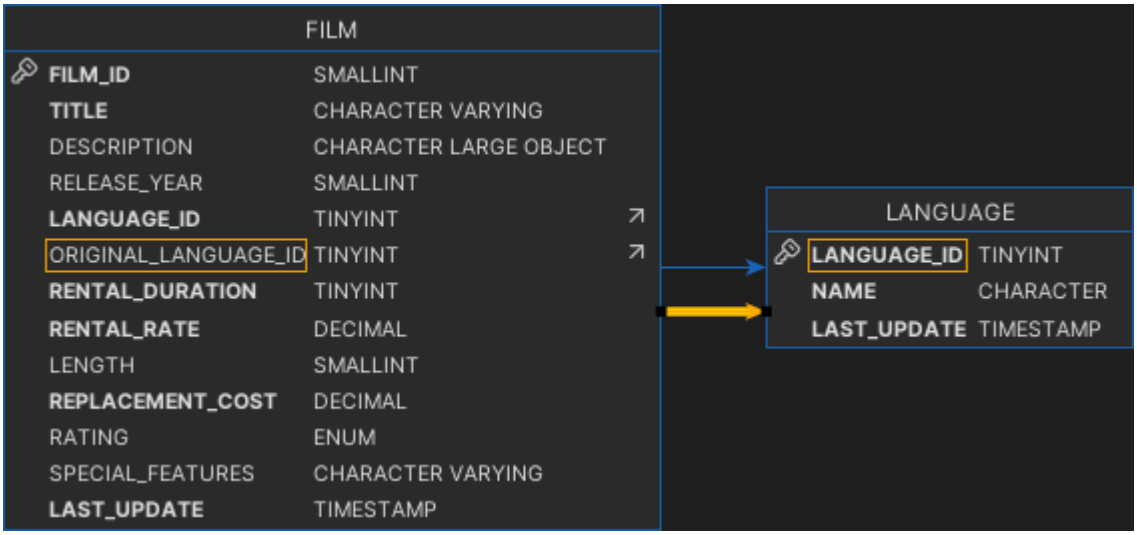
1. Locate the desired table in the **Databases** tab tree
2. Open the table's **Object View** tab (right-click **Open in Tab**)
3. Select the **References** sub tab




By selecting one or more tables, links and columns are highlighted using colors to indicate usage.



By selecting (clicking on) a specific relation, the corresponding table columns are highlighted:




Check Viewing Entity Relationships section for information about the references graph showing all tables in a schema.

# Navigating Table Relationships

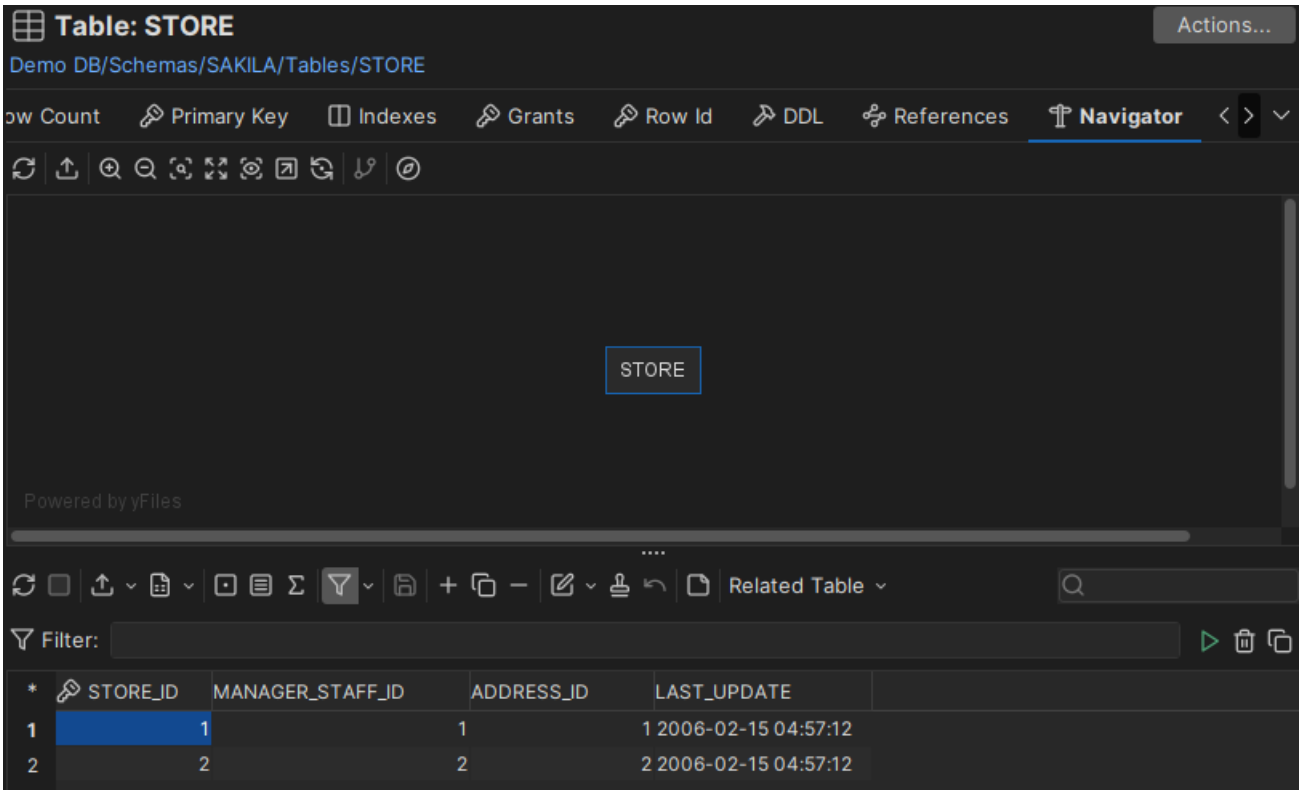
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

A powerful way to study database data is to navigate between the tables in a schema by following table relationships declared by Primary and Foreign Keys. DbVisualizer includes a **Navigator** feature for this purpose, visualizing the relationships graphically while making the data for each navigation case easily accessible in a data grid.

## Opening the Navigator

To launch the **Navigator:**

1. Locate the table you want to start the navigation from in the **Databases** tab tree,
2. Double-click the table node to open its **Object View** tab,
3. Select the **Navigator** sub tab.

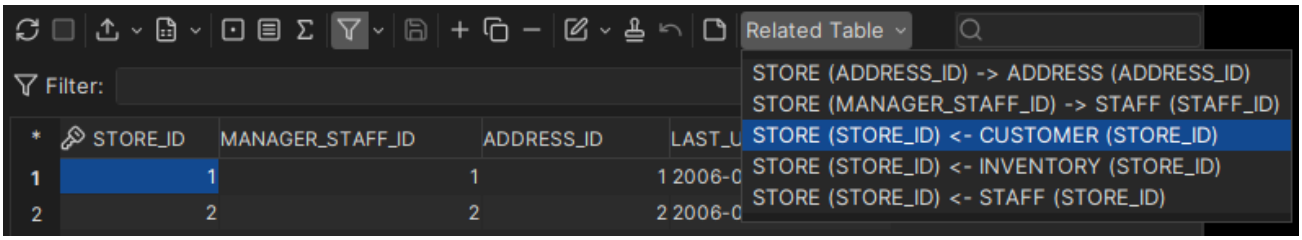


The **Navigator** tab has two parts: a graphical view and a data grid. Initially, the graphical view shows just the selected start table, and the data grid shows the data for the start table.

The data grid is of the same type as you encounter in other parts of DbVisualizer, such as in the Data tab, but extended with a **Related Table** list and a **Tag** button.

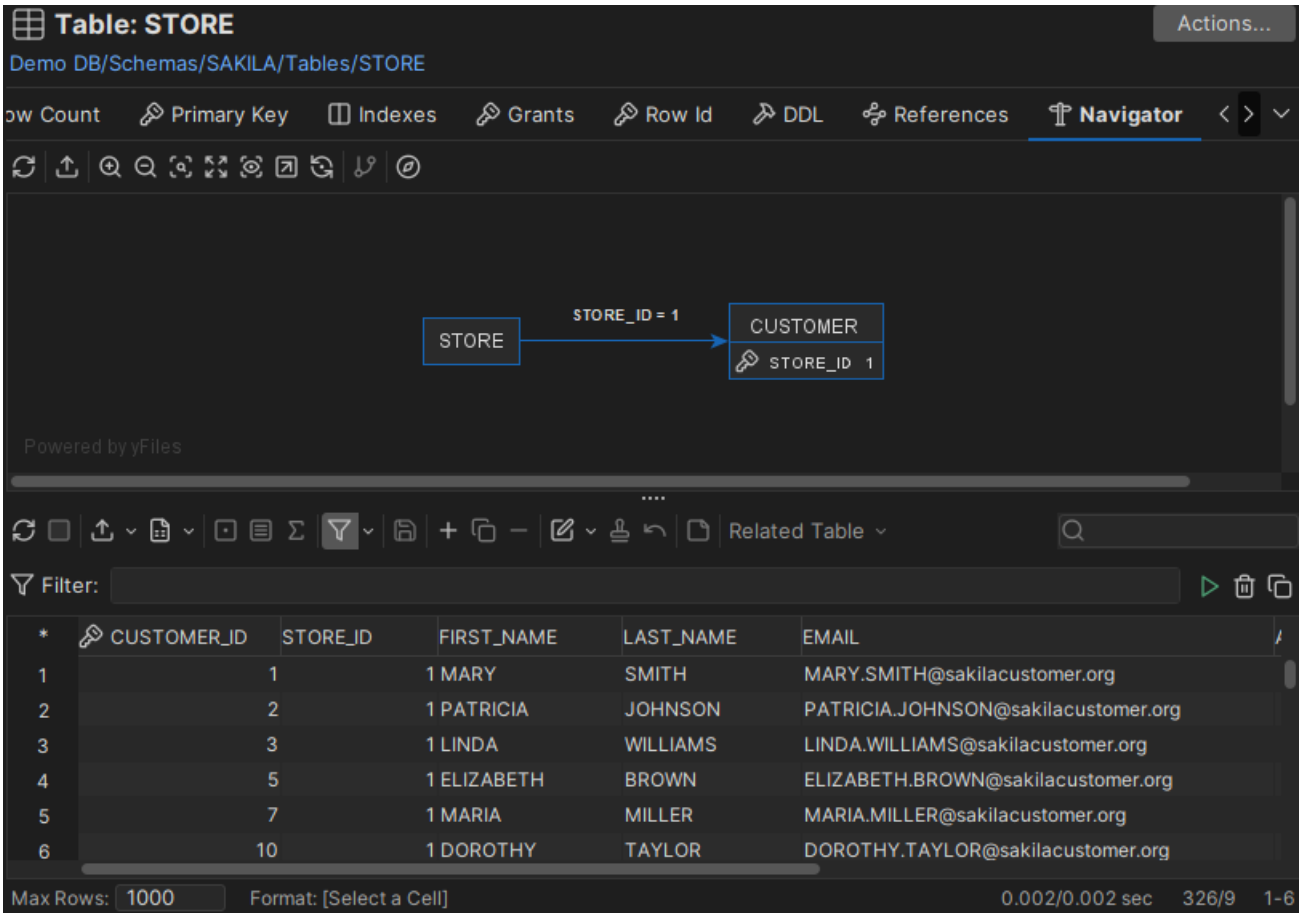
## Navigating Relationships

Data navigation in DbVisualizer means following table relationships declared by Primary and Foreign Keys, using a unique key value. In the example schema shown in the screen shots in this section, there is a table named STORE with a primary key named `STORE_ID`. Another table named `CUSTOMER` has a foreign key constraint, declaring that values in its `STORE_ID` column refer to primary key values in the column with the same name in the `STORE` table.




If you use STORE as your start table, you can easily navigate to the `CUSTOMER` table for different `STORE_ID` values. In the data grid, select one or more columns in the row that holds the `STORE_ID` you want to use for navigation. In the figure above, the store where `STORE_ID = 1` is selected.

Next, bring up the **Related Table** list. It lists all tables the `STORE` table is related to through Primary and Foreign Keys, with the key columns within parenthesis. A forward arrow (->) between the table names means that the STORE table has a foreign key relation to the named table. A backward arrow (<-) means that the named table has a foreign key relation to the `STORE` table.




When you select "STORE(STORE_ID) <- CUSTOMER(CUSTOMER_ID)" in the **Related Table** list, a node is added to the graph for the CUSTOMER table, with an arrow from the STORE table node to show the navigation direction. We call this a navigation case.

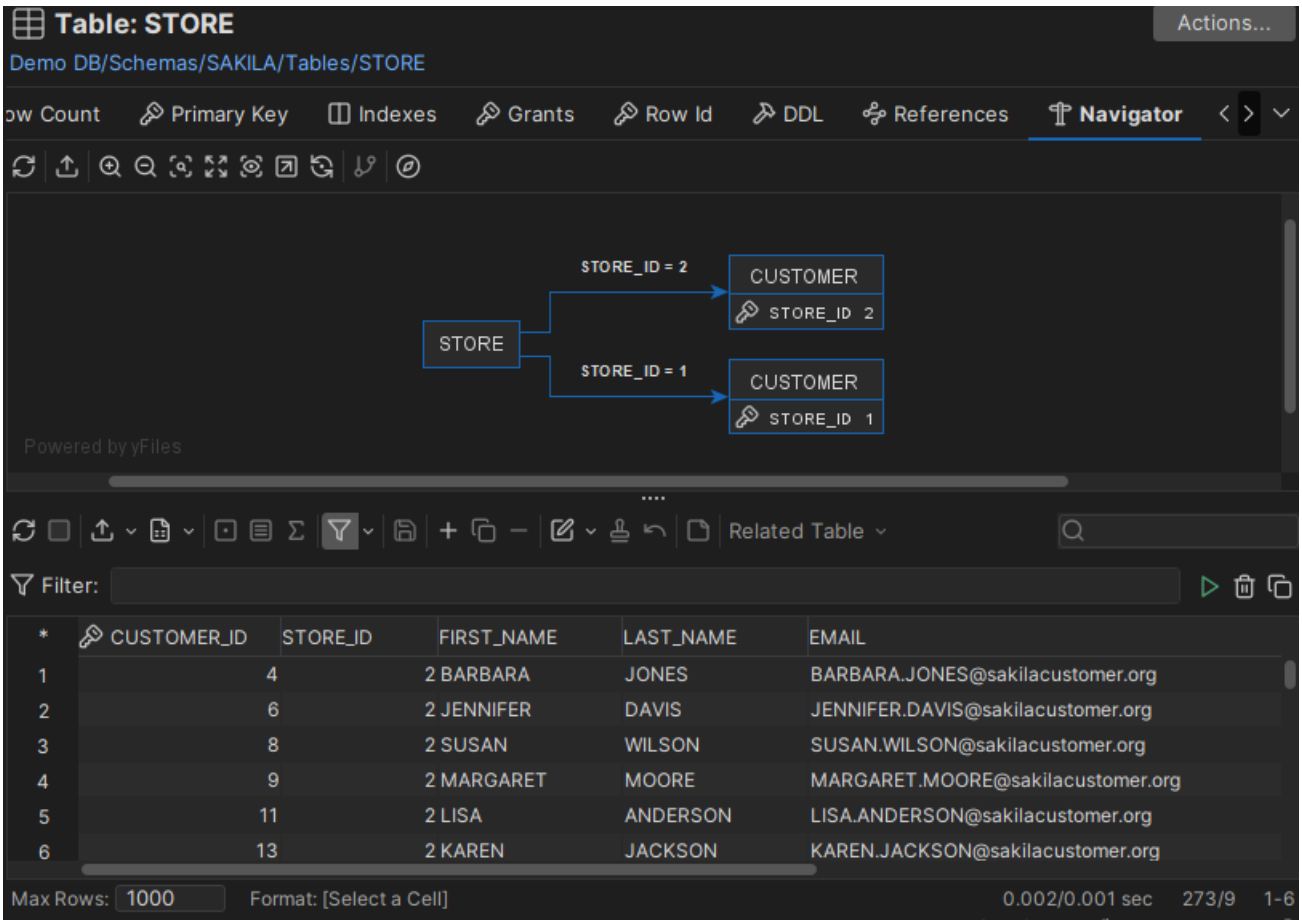
The CUSTOMER node contains the key columns (just one in this example) and their values.

The arrow between the nodes is labeled with the key column name. In addition, the arrow label also shows the name and value of the column that you selected in the `STORE` table when you created this navigation case, i.e., the STORE_ID column. If you select multiple columns when you create a navigation case, all non-key column names and values are included in the arrow label. This can make it easier to see at a glance what a navigation case represents.

The grid is also updated when you create a navigation case, to show all rows in the table you navigated to that has a key value corresponding to the selected key value in the table you navigated from. In this case, it shows all rows in the `CUSTOMER` table with `STORE_ID` equal to `1`.

> You can edit the grid values, but be aware that if you change the value of a key in the grid for a navigation case, the row will disappear from the grid since the grid only shows rows with keys matching the navigation case key value.

You can continue to create more navigation cases from any node in the graph. For instance, if the schema contains a table with job history information for employees, you can navigate to the rental history for an employee from the CUSTOMER node. Or, you can select the STORE node in the graph to navigate to the `CUSTOMER` table for a different store. Just click on the STORE node, select another row in the data grid and then the same **Related Table** list entry.




> If you want to create multiple navigation cases from one table to another using the same relationship, you can select columns in multiple rows in the first table. When you make a selection in the **Related Table** list, one navigation case per row is created.

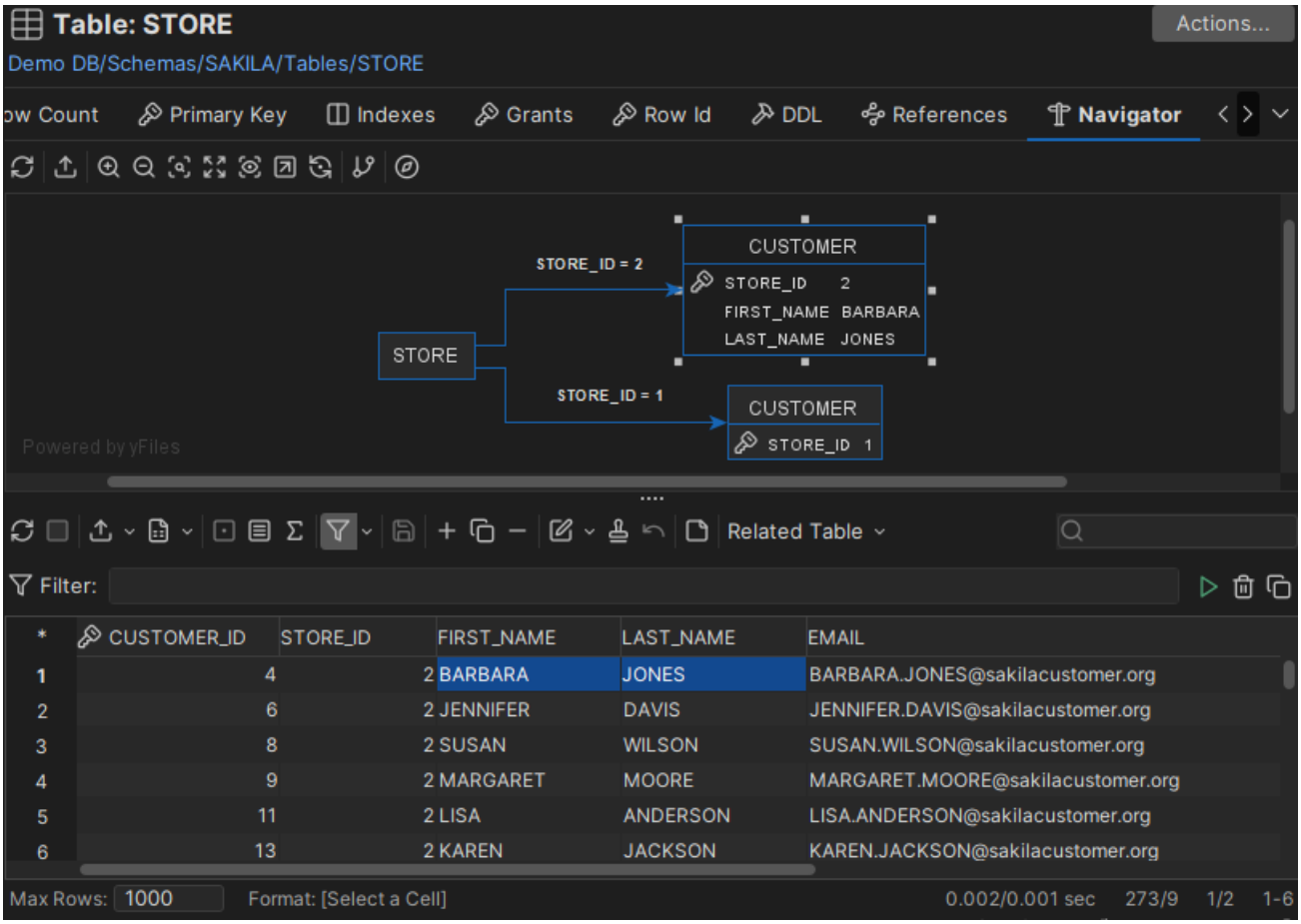
Every time you select a node in the graph, the data grid is updated to show the corresponding data. The grid settings for one node are independent of the settings for another node. For instance, if you define a filter for one node, the filter is only associated with the grid for that node.

## Adding Context Information to the Graph

The navigation node always shows the key columns and their values, but sometimes you may want to add other columns to the node to better describe what it represents. This is called tagging the node.

There are two ways to do so: drag and drop cells from the grid to any node, or use the **Tag** button in the grid toolbar to tag the currently selected node with the currently selected cells in the grid.

To drag and drop cells to a node, select one or more cells in the grid. With the left mouse button pressed and the mouse positioned over one of the selected cells, drag the cells over a node in the graph and release the mouse button. The cells are added to the node.




## Arranging the Graph

As you add navigation cases, you may find that you need to move nodes around, remove some nodes, zoom and move around in the graph, etc. You can rearrange the layout of the graph by selecting a node and, with the left mouse button pressed, drag it around. The arrow and its label move with the node. The toolbar for the graph offers a number of tools to help you with other tasks.

## Exporting and Printing the Graph

You can also export the graph to an image file or print it. Use the corresponding toolbar buttons to do this

When you print the graph, you are prompted for information about what to print (the Graph or the View, i.e., just the portion visible in the display area) and how many rows and columns to split the printing over (one page is used for each row/column).

## Opening the Navigator from the Data tab

Sometimes, you may realize that you want to analyze the relationships for a table when you are working with it in the **Data** tab. If you have configured the **Data** tab to show only filtered data, sorted in a specific way, etc. opening the **Navigator** tab and making all the same configurations there may be a bit of a hassle. A more convenient way is to just pick **Show in Navigator** in the right-click menu in the **Data** tab. It opens the table in the **Navigator** tab with all the same configurations as you made in the **Data** tab.

# Viewing the Table DDL

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

To see the DDL (CREATE statement) for a table:

1. Locate the table node in the **Databases** tab tree,
2. Double-click the table node to open its **Object View** tab,
3. Select the **DDL** sub tab.

The DDL shown is based on metadata retrieved from the database, and will not not include some database-specific information, such as storage clauses. For some databases, there is an additional sub tab named **Native DDL** (or similar) that shows the DDL as generated by the database itself, including all clauses.

# Filtering Tables in the Tree

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

If you have many tables in the tree, it may be hard to find the ones of most interest. You can then define a filter so that only a few tables are shown, as described in Filtering Database Objects.

# Showing Row Count in the Tree

You can use the **Database → Show/Hide Table Row Count** menu choice to see the number of rows within parenthesis next to the table name in the Database tab tree.

> Enabling this property results in a performance degradation.

# Using Permissions for Table Data Editing

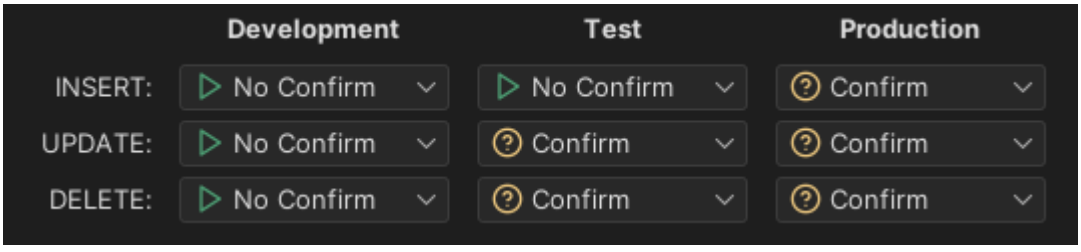
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The **Permission** functionality is a security mechanism, where you can specify that certain database operations must be confirmed. You configure permissions in the Tool Properties dialog, in the **Permissions** category of the General tab, per connection mode (Development, Test and Production). You specify which connection mode to use for a connection in the **Properties** tab of the Object View tab for the connection. By default, a connection mode is specified to be Development.

> The permission feature is part of DbVisualizer and does not replace the authorization system in the actual database.

For table grid edits, you can pick the permission type from a drop-down list for each operation:

| Permission Type | Description |
|---|---|
| **Confirm** | A confirmation window is displayed, and you can accept the operation or cancel it |
| **No Confirm** | The SQL operation is performed without any confirmation |



# Scripting a Table

To open the Script Table dialog, where you can insert generated text for a table in an SQL Commander editor:

1. Select one or more table nodes in the Databases tab tree,
2. Choose **Script Table** from the right-click menu.

You can also launch the dialog by dragging and dropping one or more nodes of the same type in an SQL Commander editor.

> If you just want to insert the object names in the editor, hold down the **Ctrl** key (or the **Alt** key on macOS) while dragging and dropping.

This behavior can be reversed in **Tool Properties**, in the **SQL Commander** category, so that dropping without pressing a key inserts the names and pressing the key launches the dialog."

The Script dialog provides a choice of which type of statement to generate, options for formatting, use of delimited identifiers, qualified names and statement delimiters. You can also pick an open SQL Commander or a new as the destination, and where in the SQL Commander editor to insert the text.

# Managing Table and Column Comments

> This feature is only available for some databases. Please execute the corresponding SQL in the SQL Commander if it is not available for your database.

Many databases support adding comments for various database objects, such as tables and columns. How this is done is database-dependent, but for some of the databases with specific support in DbVisualizer, you can enter and edit comments using an action in the right-click menu for the object node selected in the **Databases** tab. For instance, for an Oracle database, you can manage table comments like this:

1. Select the table node in the tree,
2. Choose **Comment Table** in the right-click menu,
3. Add or edit the comment.

For Oracle, you can manage comments for other database object types, such as columns and views, in the same way. An Oracle **Tables** node also has a **Table Comments** tab in its Object Views tab, listing then comments for all tables in a schema.

# Working with Views

## Working with Views

DbVisualizer provides many ways to work with views.


- Creating a View
- Altering a View
- Editing a View
- Exporting a View
- Viewing the View DDL
- Filtering Views in the Tree
- Scripting a View



- Creating a View
- Altering a View
- Editing a View
- Exporting a View
- Viewing the View DDL
- Filtering Views in the Tree
- Scripting a View

# Creating a View

There is no GUI dialog for creating a view, but you can:

1. Use the Query Builder to create the SELECT statement graphically,
2. Load the generated SELECT statement into the SQL Editor by clicking the corresponding button in the toolbar,
3. Add `CREATE VIEW name AS` before the SELECT statement,
4. Execute the CREATE VIEW statement.

# Altering a View

Views can typically not be altered; they must be dropped and recreated. You can:

1. Select the view in the **Databases** tree,
2. Double-click the view node to open its **Object View** tab,
3. Open the **DDL** sub tab,
4. Select **Copy to New Editor** from the **DDL** tab's right-click menu, which opens an **SQL Commander** tab with the DDL,
5. Remove the CREATE VIEW part in the **SQL Commander** editor so you are left with just the SELECT statement,
6. Load the SELECT statement into the **Query Builder** and alter it graphically,
7. Launch the **Drop View** assistant from the view node's right-click menu, and click **Execute** to drop it,
8. Create the new view from the altered SELECT statement.

# Editing a View

You can edit view data the same as you edit table data.

# Exporting a View

You can export a view the same way as you export a table.

# Viewing the View DDL

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

To see the DDL (CREATE statement) for a view:

1. Locate the view node in the **Databases** tab tree,
2. Double-click the view node to open its **Object View** tab,
3. Select the **DDL** sub tab.

# Filtering Views in the Tree

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

If you have many views in the tree, it may be hard to find the ones of most interest. You can then define a filter so that only a few views are shown, as described in Filtering Database Objects.

# Scripting a View

> This feature may only be available for some databases.

To open the Script View dialog, where you can insert generated text for a view in an SQL Commander editor:

1. Select one or more view nodes in the Databases tab tree,
2. Choose **Script View** from the right-click menu.

You can also launch the dialog by dragging and dropping one or more nodes of the same type in an SQL Commander editor.

> If you just want to insert the object names in the editor, hold down the **Ctrl** key (or the **Alt** key on macOS) while dragging and dropping.

This behavior can be reversed in **Tool Properties**, in the **SQL Commander** category, so that dropping without pressing a key inserts the names and pressing the key launches the dialog."

The Script dialog provides a choice of which type of statement to generate, options for formatting, use of delimited identifiers, qualified names and statement delimiters. You can also pick an open SQL Commander or a new as the destination, and where in the SQL Commander editor to insert the text.

# Working with Procedures, Functions and Other Code Objects

## Working with Procedures, Functions and Other Code Objects

Many databases offer the capability to store custom code in the database, primarily as functions and procedures, where a function has a return value but a procedure does not (a procedure may instead have output parameters). In addition, some databases offer a package concept, which means that a collection of functions and/or procedures are grouped together in one unit. A package is the interface describing the functions and procedures, while the package body contains the implementation. Many databases also support triggers: code that is executed when triggered by an event such as deleting a row in a table.

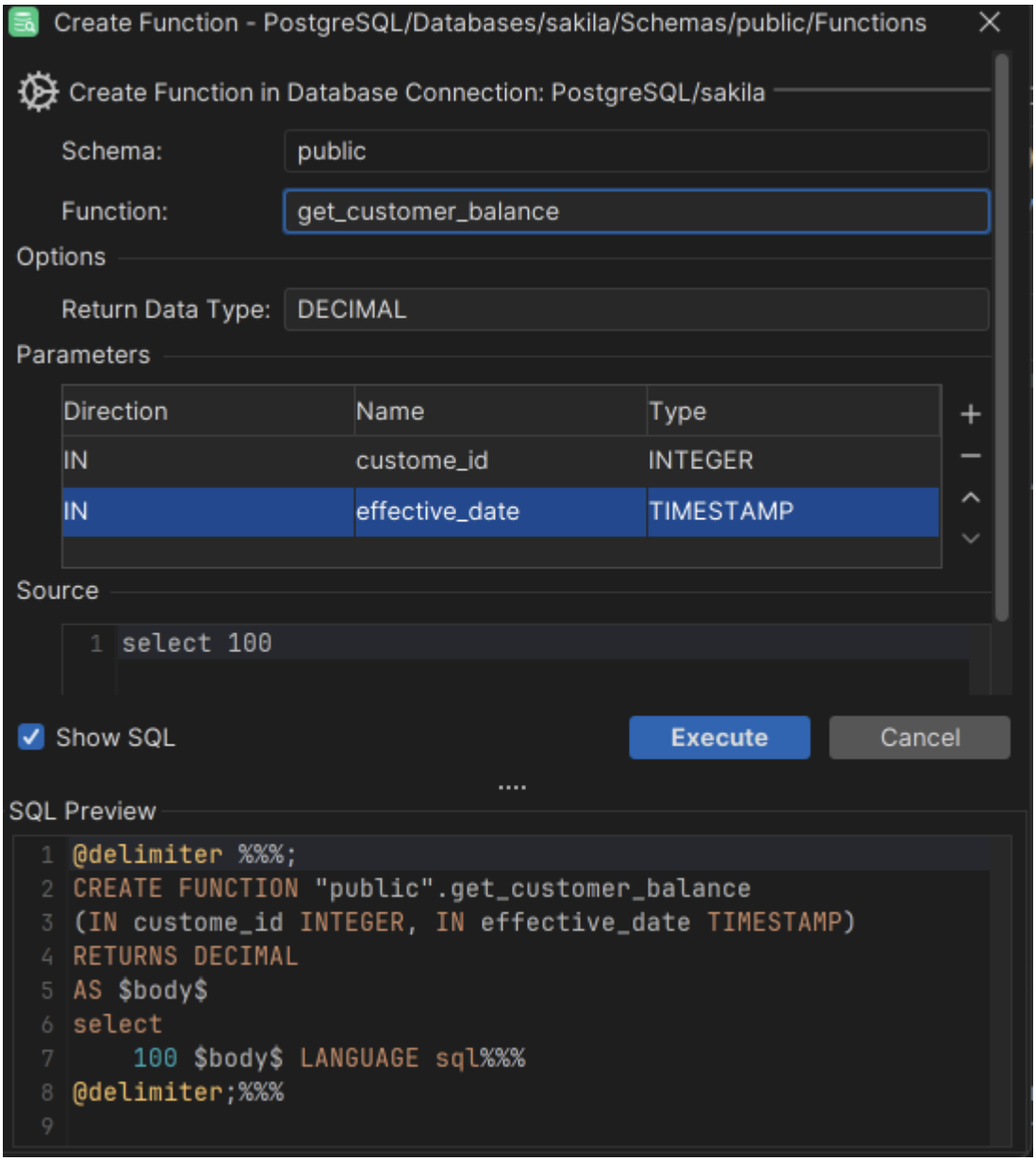
You can use DbVisualizer actions to create and drop procedural object of these types, and use the code editor to browse, edit and compile these object types. Procedures and functions can also be executed in the SQL Commander, with return values and parameters bound to DbVisualizer variables.

# Creating a Function

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

To create a new function:

1. Expand nodes in the tree under the connection node in the **Databases** tab tree until you reach the **Functions** node,
2. Select the **Functions** node and open the **Create Function** dialog from the right-click menu.




The details of the dialog depends on the database, but typically you need to:1. Enter an object name, 2. Click the **Add** button in the **Parameters** area to add parameters, 3. Enter a name and data type for each parameter. For some databases you can also enter a direction (typically IN, OUT, or INOUT) and a default value.

You can use the other buttons to the right of the parameter list to remove and move a parameter.

The dialog uses this information together with a simple sample body to compose a CREATE statement. For most databases, you can not enter the real code in the dialog. The real code is often complex and large, so DbVisualizer provides a more powerful editing environment than would fit in a dialog via the Code Editor. What you create with the dialog should be seen as a template that you then complete and work with in the Code Editor.

For some databases the sample code is editable because there is no way to write a generic sample that compiles. You must then modify the template to something that is syntactically correct, but we still recommend that you finish the real code in the Code Editor instead.

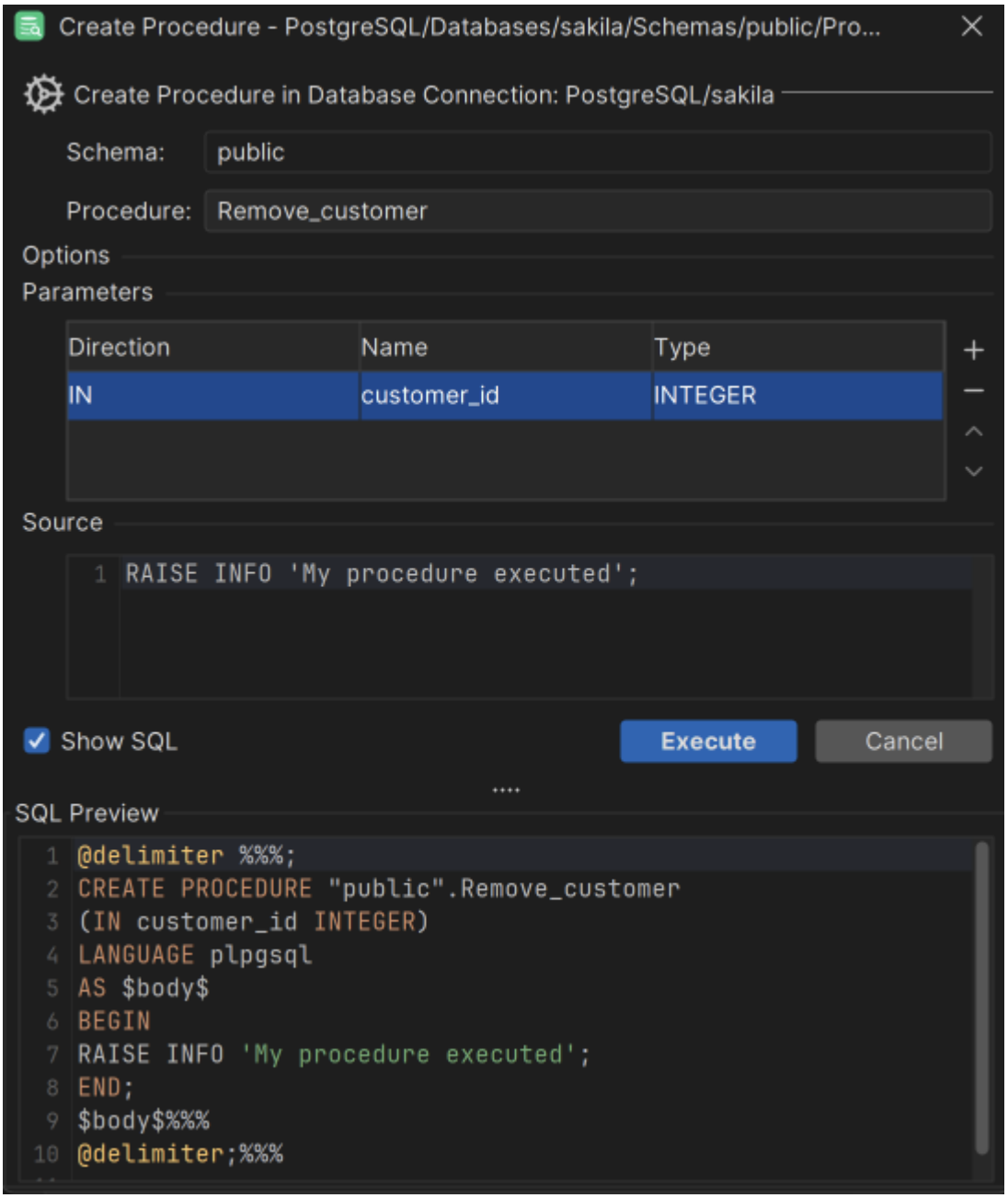
Click **Execute** in the dialog to create the new function.

# Creating a Procedure

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

To create a new procedure:

1. Expand nodes in the tree under the connection node in the **Databases** tab tree until you reach the **Procedures** node,
2. Select the **Procedures** node and open the **Create Procedure** dialog from the right-click menu.




The details of the dialog depends on the database, but typically you need to:

1. Enter an object name,
2. Click the **Add** button in the **Parameters** area to add parameters,
3. Enter a name and data type for each parameter. For some databases you can also enter a direction (typically IN, OUT, or INOUT) and a default value.

You can use the other buttons to the right of the parameter list to remove and move a parameter.

The dialog uses this information together with a simple sample body to compose a CREATE statement. For most databases, you can not enter the real code in the dialog. The real code is often complex and large, so DbVisualizer

provides a more powerful editing environment than would fit in a dialog via the Code Editor. What you create with the dialog should be seen as a template that you then complete and work with in the Code Editor.

For some databases the sample code is editable because there is no way to write a generic sample that compiles. You must then modify the template to something that is syntactically correct, but we still recommend that you finish the real code in the Code Editor instead.

Click **Execute** in the dialog to create the new procedure.

# Creating Other Code Objects

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

Some databases support other code object types in addition to function, stored procedure and trigger, e.g. Package in Oracle and Module in Mimer.

To create a new database-specific code object:

1. Expand nodes in the tree under the connection node in the **Databases** tab tree until you reach the group node for the code type, e.g. **Packages**,
2. Select the group node and open the **Create** dialog from the right-click menu.

The details of the dialog depends on the database, but typically you need to:

1. Enter an object name,
2. Click the **Add** button in the **Parameters** area to add parameters,
3. Enter a name and data type for each parameter. For some databases you can also enter a direction (typically IN, OUT, or INOUT) and a default value.

You can use the other buttons to the right of the parameter list to remove and move a parameter.

The dialog uses this information together with a simple sample body to compose a CREATE statement. For most databases, you can not enter the real code in the dialog. The real code is often complex and large, so DbVisualizer provides a more powerful editing environment than would fit in a dialog via the Code Editor. What you create with the dialog should be seen as a template that you then complete and work with in the Code Editor.

For some databases the sample code is editable because there is no way to write a generic sample that compiles. You must then modify the template to something that is syntactically correct, but we still recommend that you finish the real code in the Code Editor instead.

Click **Execute** in the dialog to create the new code object.

# Editing a Code Object

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

To edit the code for an object, such as a function, stored procedure or database-dependent code object:

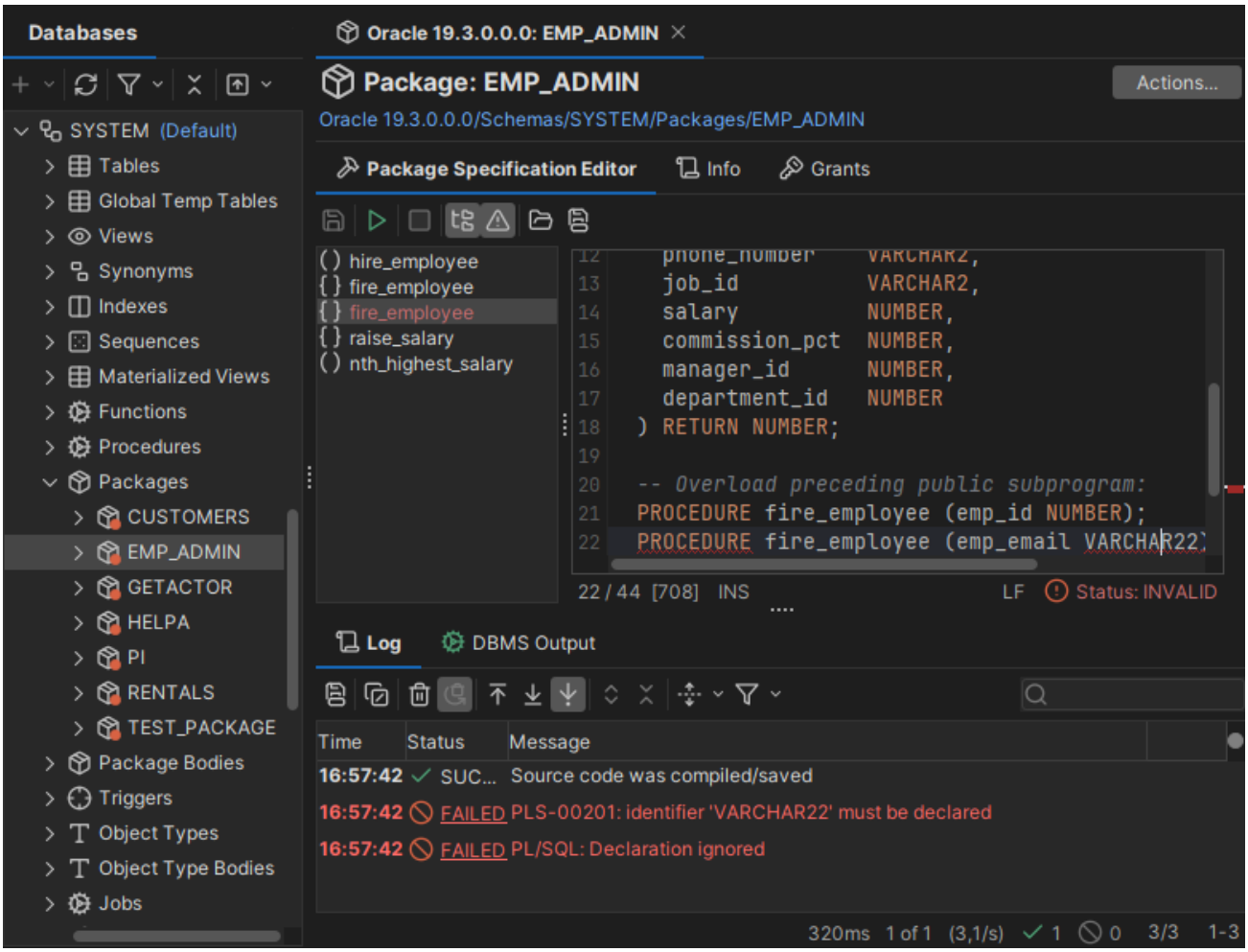
1. Expand nodes in the tree under the connection node in the **Databases** tab tree until you reach the node for the object you want to edit,
2. Double-click the node to open an **Object View** tab for it, and select the editor sub tab (e.g., the **Procedure Editor** tab).

The editor has a toolbar with various actions to save/compile the procedure, save and load the source to/from file and perform common editing operations.

If editing a package or package body, a package navigator (which can be toggled off) is also included to the left of the editor to ease navigating long packages.

The **Status** indicator shows whether the procedure is valid or invalid based on last compilation (not available for all databases).

Edit the source code and save/compile the procedure when you are happy with the code, using the **Save** toolbar button.




If errors occur, the corresponding text is underlined with a red wavy line. Hovering the mouse over the error indication shows the corresponding error message. The right margin contains markers for each error as well, and clicking on a marker scrolls the editor to the corresponding error. Alternatively, you can click the **FAILED** link in the Log tab grid to move the caret to the error location.

If you prefer to navigate between errors using the keyboard, you can use the defined key bindings for the **Insertion Point to Next Marker** and **Insertion Point to Previous Marker** actions in the Tool Properties dialog, in the **Key Bindings** category, in the Editor Commands group. Alternative is to use the **Goto Next Failed** and **Goto Previous Failed** in the right click menu of Log tab grid, and then click the link for the FAILED entries.

> Error location information is not available for some databases. In that case the complete statement is underlined in the editor.

## Disable Error Markers in the SQL Editor

It is possible to disable error markers in the **SQL Commander → SQL Commander Options** menus. With **Show Error Position Markers** turned on it shows markers only for databases that specifically report positions of errors in a

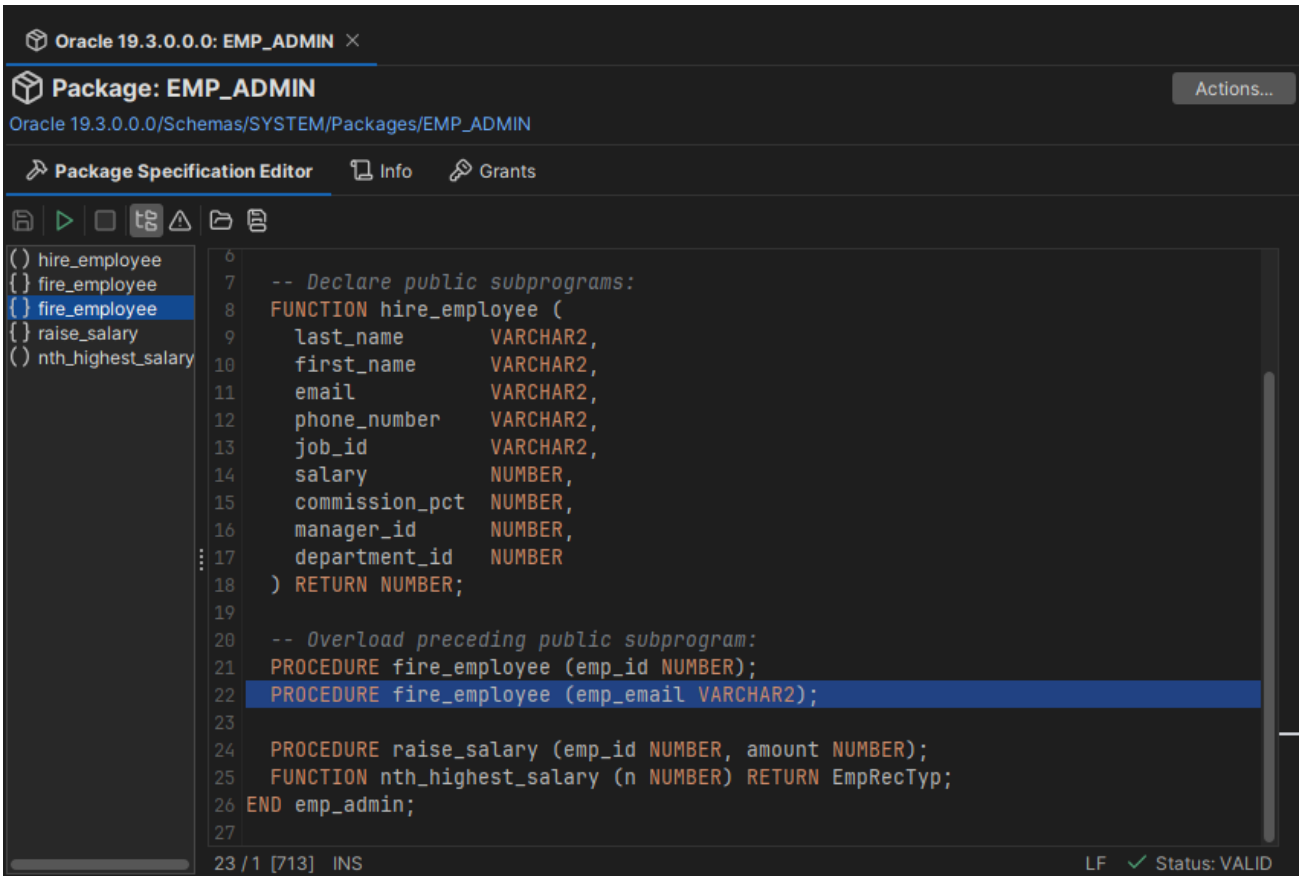
statement. With **Show Error Statement Markers** turned on the complete statement is highlighted if it fails during execution, but only if either the **Show Error Position Markers** is turned off or if the driver/database doesn't report positions of errors.

If you re looking to minimize the amount of error markers when executing scripts, turn off **Show Error Statement Markers** and keep the **Show Error Position Markers** checked (turned on). DbVisualizer will then report errors reported by the JDBC driver only.

To set default values for these settings visit **Tools → Tool Properties** and the **General / SQL Commander** category.

In addition to the **Status** indicator in the editor, the object icon in the tree shows a little red cross for invalid procedures, for databases that provide this information. You can see this for the **UPDATE_STATUS** procedure node in the screenshot above.

The figure below shows the result after correcting the errors and recompiling the procedure:




The status indicator now shows that the procedure is **VALID**.

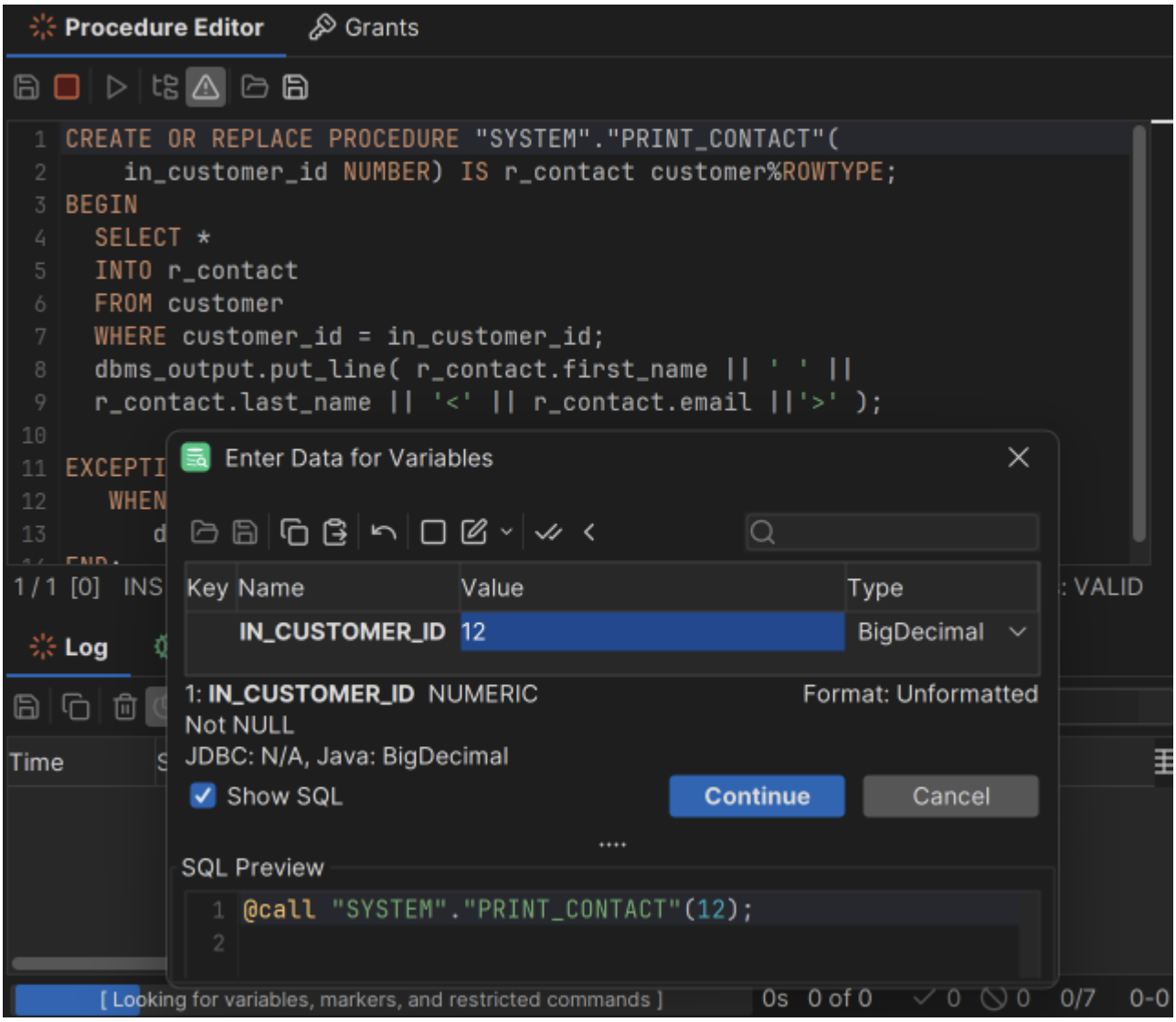
# Executing a Code Object

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

You can execute a code object, such as a function or stored procedure, either in the Code Editor or in the SQL Commander.

## Executing in the Code Editor

In the Code Editor, click the **Execute** button. DbVisualizer then generates a script for executing the code object, using variables for all parameters, and executes it.




Since the script contains variables, the **Variable Prompt** dialog pops up. Enter values for all parameters and click **Continue** to execute the procedure. The result is shown in the results area below the editor.

In the example shown in the figure, all parameters are input parameters but DbVisualizer also supports the execution of procedures with output parameters and functions returning a value. In this case, the generated script includes `@echo` statements to write the result in the **Log** tab. Please see below for more details.

## Executing in the SQL Commander

The scripts generated and executed by the Code Editor can also be included in a script and executed in an SQL Commander. Here's an example of a script calling a function and writing the result in the **SQL Commander Log** tab:

```
@call ${STATUS||(null)||String||noshow dir=out}$ = "HR"."GET_STATUS"(1002);
@echo STATUS: ${STATUS}$;
```

In this example, the result value from the `GET_STATUS` function is assigned to a variable named `STATUS`. Note that is has an option `dir=out`. This is a requirement for a variable that is assigned a value at runtime, whether it is used for a return value from a function call or for an output parameter in a procedure call. It also has the `noshow` option, to avoid getting prompted for a value for the variable. The value of the `STATUS` variable is then written to the log using the `@echo` command.

You can also use the output from one function or procedure as input to another, or even as a value in a SELECT or other SQL statement:

```
@call ${STATUS||(null)||String||noshow dir=out}$ = "HR"."GET_STATUS"(1002);
@call "HR"."UPDATE_STATUS"(1000, 2000, ${STATUS||||String||noshow dir=in}$);
```

Note that `dir=in` is specified for the `STATUS` variable when it is used in the `UPDATE_STATUS` procedure call. When you use a variable first for output and then as input with another `@call` command, you must change the direction option like this.

More formally, the `@call` command has this syntax when calling a function:

```
@call <OutVariable> = <FunctionName>(<ParamList>)
```

where the <FunctionName> may need to be fully qualified with a schema (and/or catalog/database) and the <ParamList> is a comma separated list of literal values or variables. Here's an example:

```
@call ${return_value||(null)||String||dir=out noshow}$ = get_some_value();
```

For a procedure, use this syntax:

```
@call <ProcedureName>(<ParamList)
```

where the <ProcedureName> may need to be fully qualified with a schema (and/or catalog/database) and the <ParamList> is a comma separated list of literal values or variables. Here's an example:

```
@call my_process('literal input',
                 ${var_in||(null)||String||dir=in}$,
                 ${var_out||(null)||String||dir=out noshow}$,
                 ${var_inout||'in_value'||String||dir=inout}$);
```

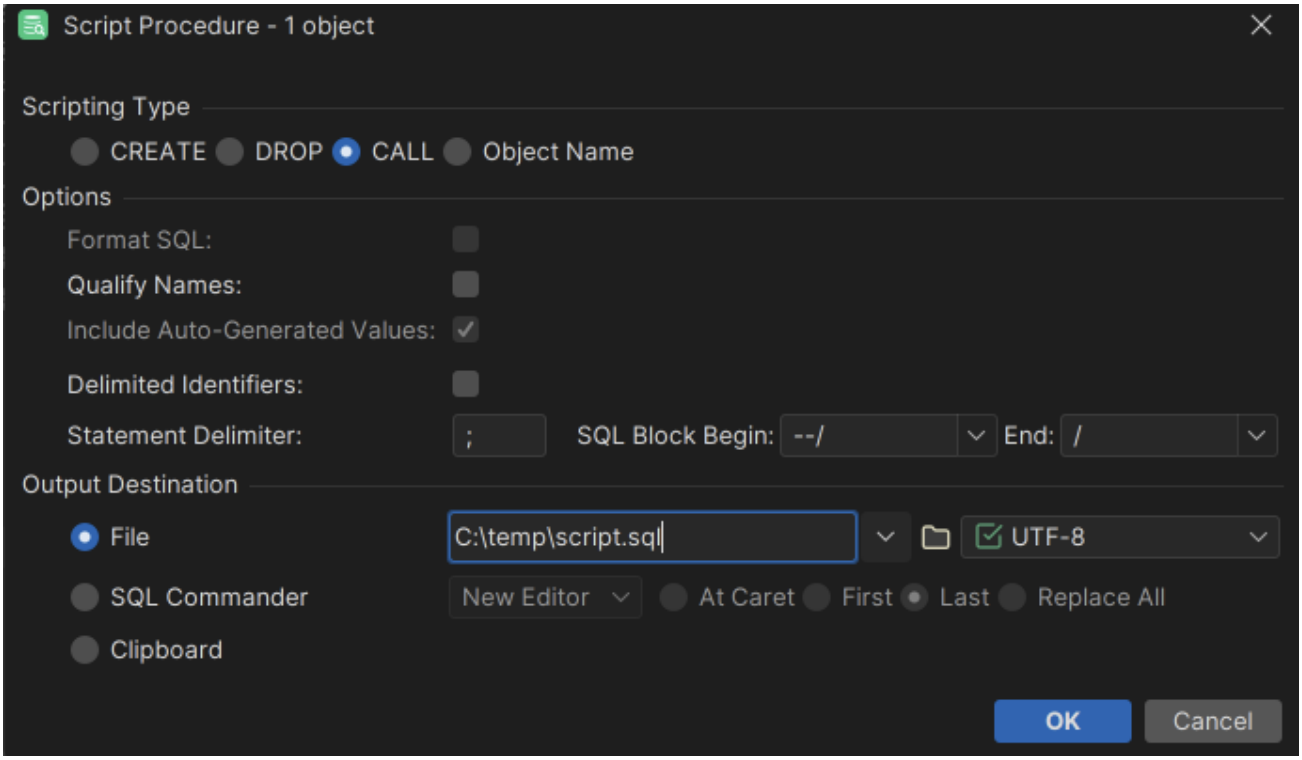
As shown in these examples, you must use the `dir` option to specify how the variable is to be used (in, out or inout) and you may use the `noshow` option to prevent being prompted for a value for an output variable.

You can use the @echo command to write the value assigned to an output variable to the log.

## Using the Script Object Dialog

Instead of writing a `@call` script by hand in an SQL Commander, you can use the Script Object (e.g. **Script Procedure** or **Script Function**) right-click menu choices for the object node in the tree.

This opens the Script Object dialog where you select that you want to generate a CALL script and can adjust settings for using delimiters and qualifiers, as well as the destination for the generated script. This is how the Script Dialog will look like when opened for a Procedure.



# Exporting a Code Object

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

To export a code object:

1. Select the object node in the **Databases** tab tree,
2. Open the Export Object dialog (e.g. **Export Procedure** or **Export Function**) from the right-click menu,
3. Select an **Output Format**, **Output Destination**, and **Options**,
4. Click **Export**.

For these object types, you can select either the **SQL** (CREATE statement) or **XML** Output Format and which delimiters to use in the Options area.

You can control whether to use delimited identifiers and/or qualified names in the DDL and INSERT statements generated for the SQL format.

# Scripting a Code Object

To open the Script Function/Procedure dialog, where you can insert generated text for a code object in an SQL Commander editor:

1. Select one or more code object nodes in the Databases tab tree,
2. Choose **Script Function/Procedure** from the right-click menu.

You can also launch the dialog by dragging and dropping one or more nodes of the same type in an SQL Commander editor.

> If you just want to insert the object names in the editor, hold down the **Ctrl** key (or the **Alt** key on macOS) while dragging and dropping.

This behavior can be reversed in **Tool Properties**, in the **SQL Commander** category, so that dropping without pressing a key inserts the names and pressing the key launches the dialog."

The Script dialog provides a choice of which type of statement to generate, options for formatting, use of delimited identifiers, qualified names and statement delimiters. You can also pick an open SQL Commander or a new as the destination, and where in the SQL Commander editor to insert the text.

# Working with Schemas

## Working with Schemas

DbVisualizer provides many ways to work with schemas.

- Creating a Schema
- Comparing Schemas
- Viewing Entity Relationships
- Exporting a Schema
- Filtering Schemas in the Tree

# Creating a Schema

To create a new schema:

1. Locate the **Schemas** node in the Databases tab tree,
2. Open the **Create Schema** dialog from the right-click menu,
3. Enter all required information (database dependent),
4. Click **Execute** to create the schema.

> This feature is only available for some databases. Please execute the corresponding SQL in the SQL Commander if it is not available for your database.

# Comparing Schemas

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

While DbVisualizer does not provide a way to fully show the differences between two schemas in terms of objects and data, you can compare different aspects of a schema to another schema one by one.

For instance, to compare the list of tables in one schema with the list of tables in another schema:

1. Expand the schema node for the first schema and select its **Tables** node and open it in a new Object View tab, select the **Tables** tab,
2. Do the same for the second schema,
3. Select **Compare** from the right-click menu in one of the Tables grids to compare their grid content, or select **Tools → Compare**.

You can do the same for all the other schema object types, such as views and stored procedures. You can also dig deeper and compare the individual objects, such as comparing individual tables.

# Viewing Entity Relationships

Use the References graph to see how tables are related to other tables through Foreign Keys.

You can view relations to/from one or more tables in different graph layouts: **Hierarchical**, **Organic**, **Orthogonal**, or **Circular**.Layout settings can be changed in the **Graph Control** area, which is shown or hidden with the settings toggle button in the toolbar. For instance, you can select how much information to include for each table in the graph: just the **Table Name**, the **Primary Key** column(s) or all **Columns**, which links to include, and whether or not to highlight links for selected tables. Settings vary slightly depending on whether you are viewing relations for a single table or multiple tables, and some settings (e.g. **Links to Columns**) are only available in **Hierarchical** layout.

If you choose the **Hierarchical** layout and select **Links to Columns** and **Highlight Links**, the graph will for all selected table nodes color code links and columns to indicate usage:

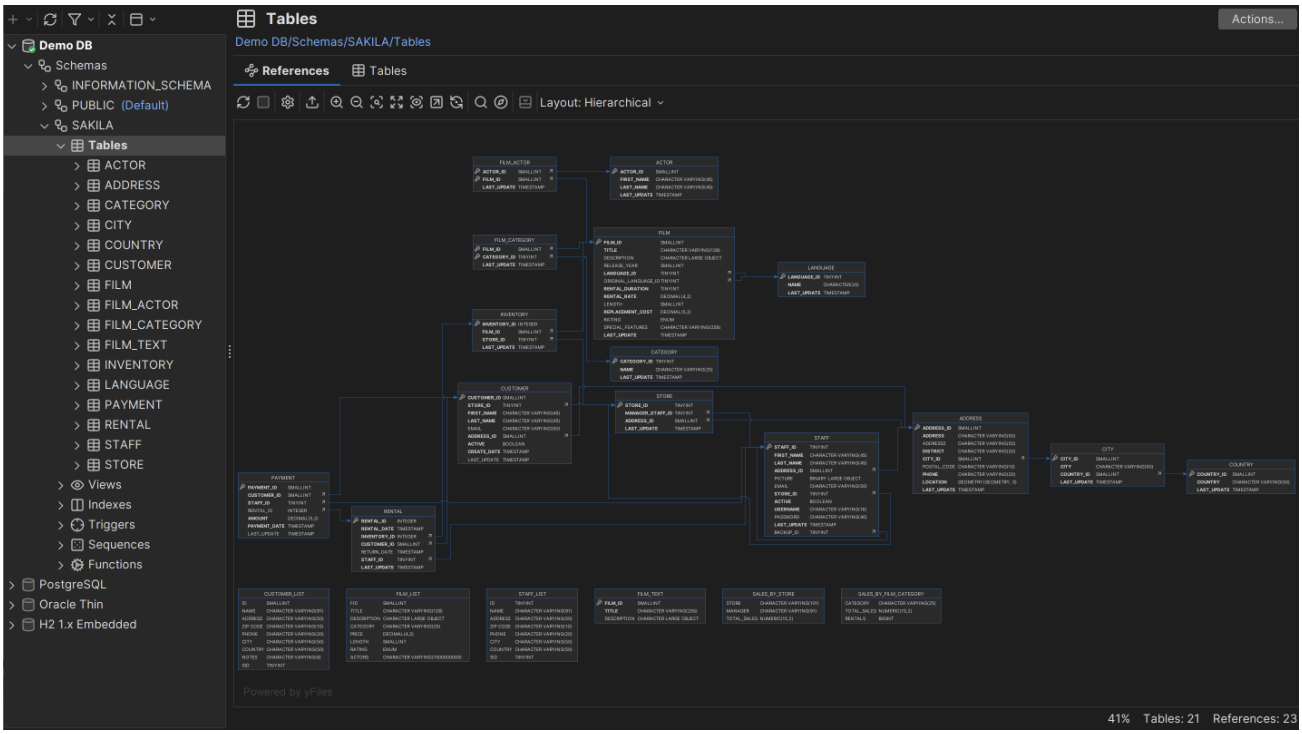
- foreign key columns and outbound links are green - primary key columns and inbound links are red - bidirectional columns (used for both inbound and outbound links) and bidirectional links (where both source and target tables are selected) are orange

The graph can be **Exported** to a file in **JPG**, **GIF**, **PNG**, **SVG**, **PDF** or **EMF**. It can also be saved s a **GML** (Graph Modeling Language) file that you can then open in the yEd tool from yWorks for further manipulation. The **GML** format is saved using **Save As** in the right-click menu. You can also control whether the table names should be qualified with the schema/catalog in the graph.

When opened on multiple tables, the graph shows all links between the tables; you cannot choose to show inbound or outbound relations since all links are both outbound (from one table) and inbound (to some other table). You can however exclude unreferenced tables and/or manually select which tables to include on the graph.

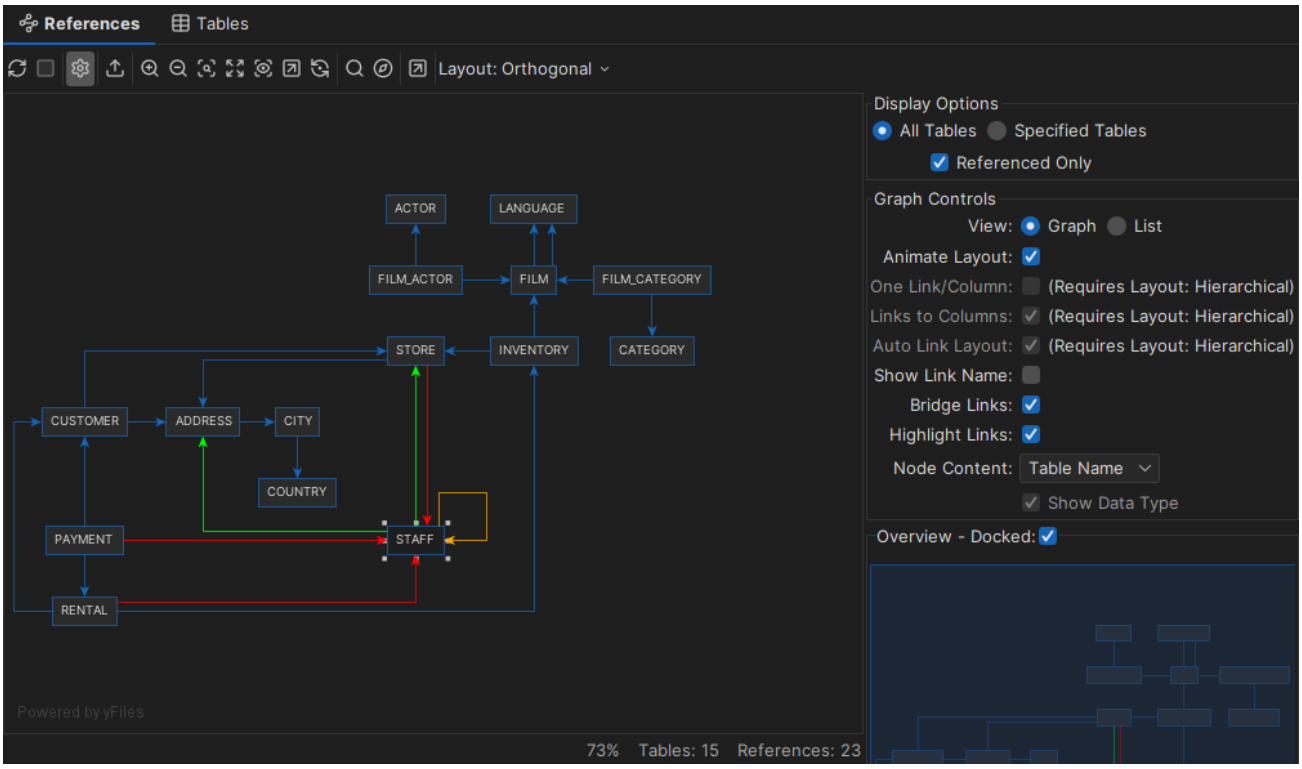
To zoom in on a specific table, you can also view table relationships for a specific table.

1. Locate the **Tables** node in the **Databases** tab tree
2. Open the **Object View** tab (right-click **Open in Tab**)
3. Select the **References** sub tab




Removing details and turning on highlighting to focus on certain aspects of the schema can be quite useful, especially if you choose to export the graph to discuss it with someone else.

Changing the layout, reducing the number of tables and suppressing some information makes it easier to focus on structure.




Check Viewing Table Relationships section for information about the references graph showing references for a specific table.

# Exporting a Schema

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

You can export all or selected objects in a schema using the Export Schema assistant.

To export a schema:

1. Select the schema node in the **Databases** tab tree,
2. Launch the **Export Schema** assistant from the right-click menu,
3. Select an **Output Format**, **Output Destination**, **Objects** to export and **Options**,
4. Click **Export**.

## Output Format

You can export objects in one of these formats: **CSV**, **HTML**, **SQL, TXT, XML**, **XLS** (Excel), or **JSON**.

The **CSV**, **HTML, TXT, XLS** and **JSON** formats are specifically for table data and are not supported for any other type of objects.

The **SQL** and **XML** formats can be used for all objects to export the DDL, and for tables you can also choose to include the table data in these formats.

You can control whether to use delimited identifiers and/or qualified names in the DDL and INSERT statements generated for the SQL format, using the controls in the Options area.

## Output Destination

The destination can be one of:

- a file,
- an open or new SQL Commander tab, with options for where in an open SQL Commander to insert the result,
- to the system clipboard.

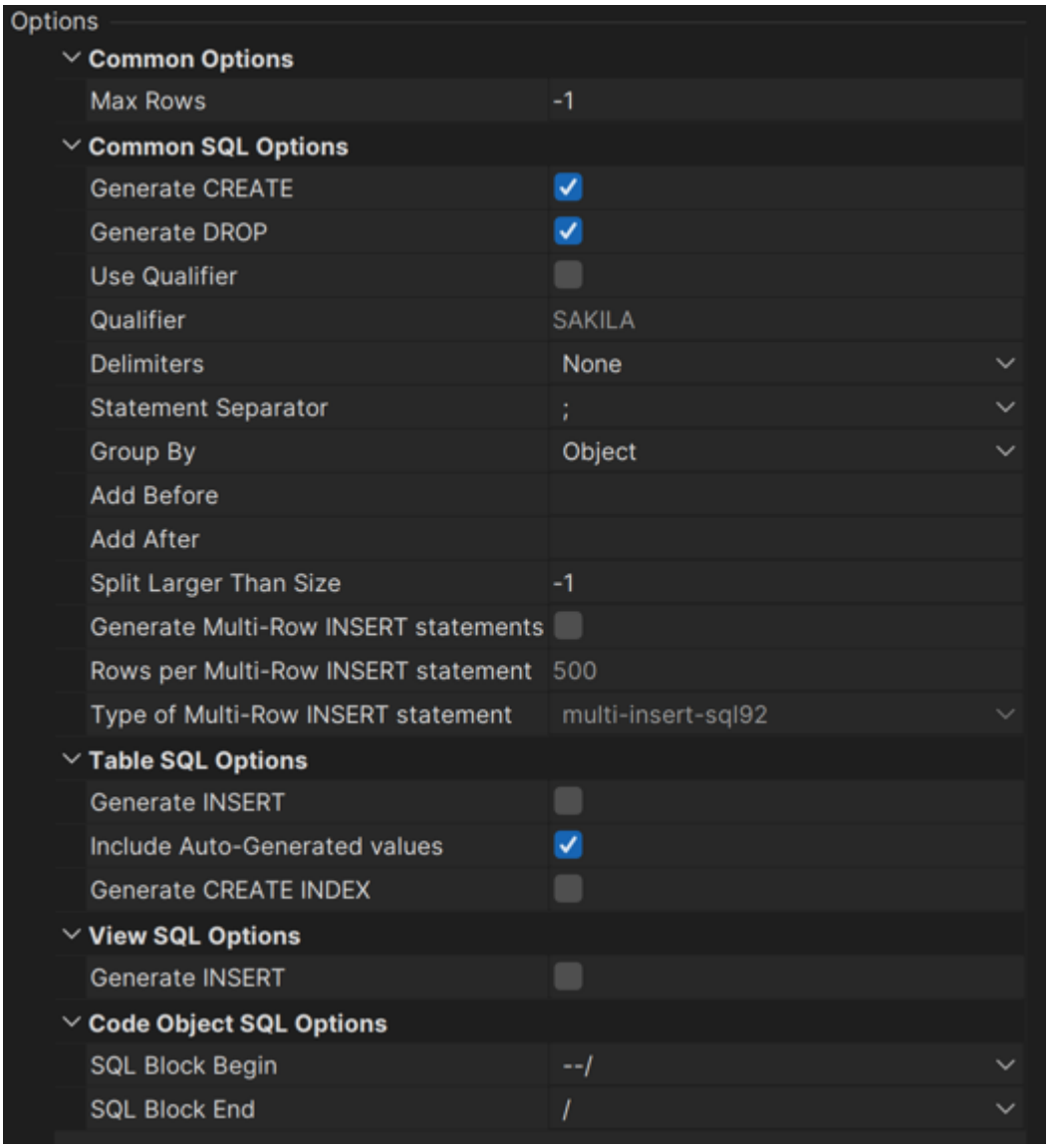
## Object Types

In the **Object Types** area you select what to export. You can check the checkbox for an object type to export all objects of that type, or expand a type node and select individual objects. To select all objects, just check the checkbox for the schema itself at the top of the tree.

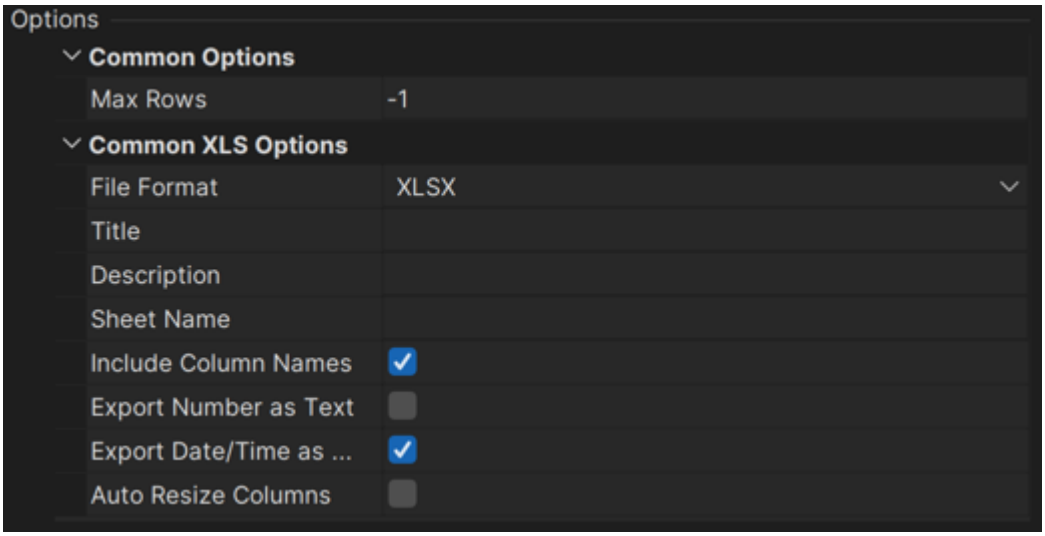
## Options

The **Options** section contains options common to all Output Formats at the top, followed by options for the selected format.

Example of the options for SQL:




Example of the options for Excel:




For the **SQL** and **XML** formats, you can choose to export the DDL, the DDL for indexes for a table and the table data: as INSERT statements for the SQL statement or in one of three XML formats.

For the **Excel** format, you can choose to export table data as either in the **XLSX** (default) or the legacy **XLS** format.

Most formats also let you specify other options, such as delimiters, title and descriptions. Just select an Output Format to see which options are available. All options are described in the context of the @export command, as the Export dialog is just a GUI for the command.

You can adjust the **Data Formats** specifically for the exported table data. By default, the formats defined in **Tool Properties** are used, but sometimes you need to export dates and numbers in a different format because you intend to import the data into a different type of database.

> If you are exporting table data in the **SQL** format from one database type (e.g. Oracle) to import it in a database of a different type (e.g. PostgreSQL) by executing the generated script, you need to be aware of differences in the literal formats for Date, Time and Timestamp data. If you connect to the other database using a JDBC client like DbVisualizer, you can select the **JDBC escape** format for these data format. This generates literals that the JDBC driver converts into a format the target database can interpret.

In the **Data Format Settings** dialog you can also specify how to quote text data and how to handle quotes within the text value.

## Using Variables in Fields

You can use some of the pre-defined DbVisualizer variables ( `${dbvis-date}$` , `${dbvis-time}$` , `${dbvis-timestamp}$` , `${dbvis-connection}$` , `${dbvis-database-type}$` and `${dbvis-object}$` ) in all fields that hold free text (e.g. title and description fields) and as part of the file name field.

Use the **${dbvis-object}$** variable as part of the filename if you want to export the DDL and/or data to a separate file for each object. The variable is replaced with the object type and object name, e.g. `${dbvis_object}$.sql` becomes `table_COUNTRIES.sql` for a table named COUNTRIES.

## Saving And Loading Settings

If you often use the same settings, you can save them as the default settings for this assistant. If you use a number of common settings, you can save them to individual files that you can load as needed. Use the **Settings** drop-down button menu to accomplish this:

- **Save as Default Settings** Saves all format settings as default. These are then loaded automatically when open an Export Schema dialog
- **Use Default Settings** Use this choice to initialize the settings with default values
- **Remove Default Settings** Removes the saved defaults and restores the regular defaults
- **Load…** Use this choice to open the file chooser dialog, in which you can select a settings file
- **Save As…** Use this choice to save the settings to a file
- **Copy Settings to Clipboard** Copies the settings to the system clipboard
- **Copy Settings to Clipboard** Use this choice to copy all settings to the system clipboard. These can then be pasted into the SQL Commander to define the settings for the @export editor commands.

# Filtering Schemas in the Tree

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

If you have many schema in the tree, it may be hard to find the ones of most interest. You can then define a filter so that only a few schemas are shown, as described in Filtering Database Objects.

# Working with SQL

## Working with SQL

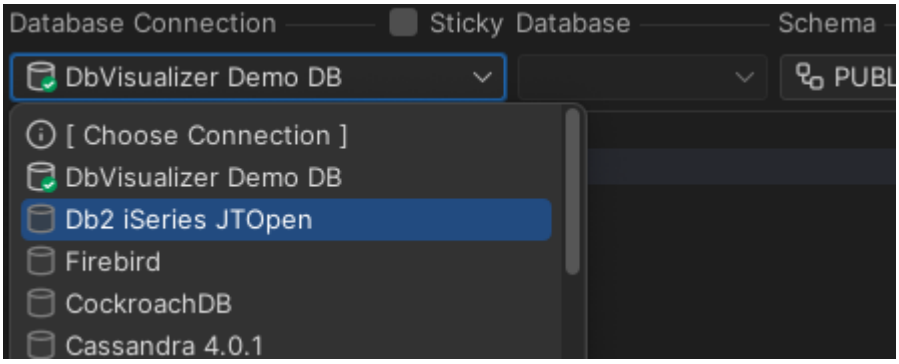
With DbVisualizer, you can use a powerful SQL editor or a graphical Query Builder to create and edit your scripts, save them as Bookmarks for easy access, and execute all or just a few of the statements, and lots more.

DbVisualizer depends on the JDBC Driver and the connected database to read meta data and execute SQL commands and will work with any type of SQL that the database supports. When you run a script, DbVisualizer parses the script to find statement delimiters, variables and client-side processing commands, everything else is passed as-is to the database for processing. The connected database defines what you can and cannot do and will report errors if it does not understand your SQL script.

DbVisualizer adapts to the database based on what Database Type you specify on the connection, this drives what objects you see, what keywords the editor recognizes, etc. The database type is usually automatically detected, but you can also specify it manually - see Create a New Database Connection for more info.

# Selecting Database Connection, Catalog and Schema

You use the **Database Connection** and **Database** (or Catalog) lists above the editor to specify which connection and database to use when executing the SQL in the SQL Commander. The list of connections shows all connections as they are ordered in the Databases tab tree, except that all currently active connections are listed first.




The screenshot shows the **Sakila** database being selected. When hovering over the name, its full path including any folders is listed in a tooltip.

If you check the **Sticky** box above the Database Connection, the current connection selection will not change automatically when passing SQL statements from other parts of DbVisualizer, for instance, when opening a Bookmark. Consider a Bookmark defined for database connection `Prod`. If the Sticky checkbox is not checked (i.e., disabled), the database connection is automatically changed to `Prod` when you open the Bookmark in the SQL Editor. However, if the Sticky checkbox is checked (i.e. enabled), the current database connection setting is unchanged. You can specify if you want to have Sticky enabled by default in the Tool Properties dialog, in the **SQL Commander** category under the General tab.

The **Database** list (or Catalog) defines which catalog in the connection is the target for the execution. Since not all databases use catalogs, this list may be disabled.

For most databases, the schema selected in the **Schema** list is used **only** to limit the tables the Auto-Completion feature shows in the completion pop-up; it does not define a default schema for tables referenced in the SQL since most databases do not allow the default schema to be changed during a session.

For databases that allow the default schema to be changed*, the selected schema is also used as the default schema, i.e., the schema used for unqualified table names in the SQL. If the Database Type is set to **Generic** for a connection, DbVisualizer tries to set the default schema (if **Use Schema** is chosen) but it depends on the JDBC driver if this works or not.

If you don't want the selected schema to be used as the default schema for a database, you can disable this behavior in the **Properties** tab for the connection, in the **SQL Commander** category.

*) Examples of databases that support setting a default schema include Db2 LUW, Db2 z/OS, Db2 iSeries, Greenplum, H2, JavaDB/Derby, Oracle, NuoDB, PostgreSQL, Vertica, and Yellowbrick.

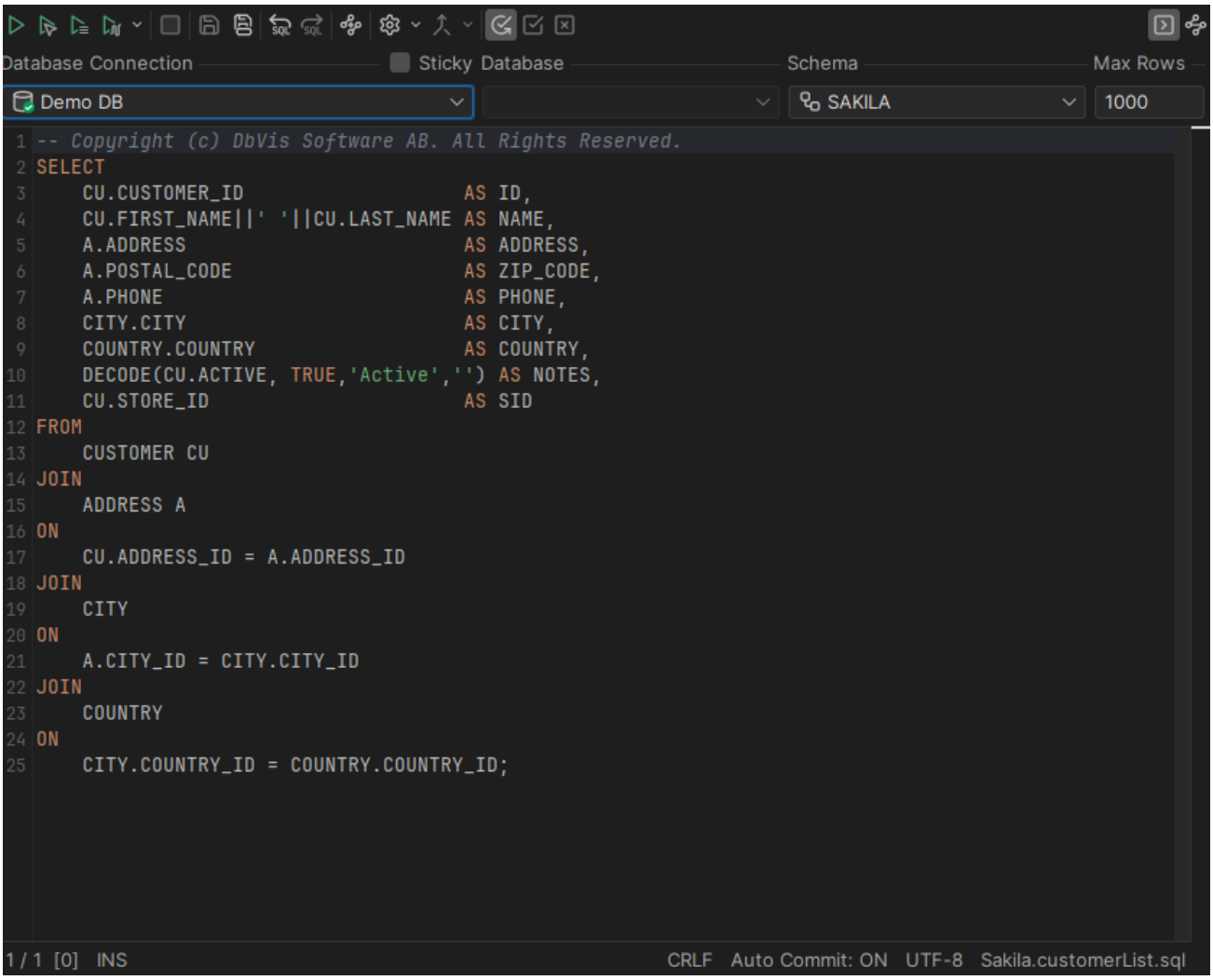
## Configuring the Initial Values

When a new Database Connection is selected, the initial values for the **Database** and **Schema** lists are by default based on the default values for the connection. You can change this behavior in the **Properties** tab for the connection, in the **SQL Commander** category. The initial value for each list that is applicable for the database type can be set to one of **The Connection Default**, **None** or **The Most Recently Used**.

# Editing SQL Scripts

The SQL Commander contains an SQL editor, used to edit SQL scripts.

The editor area looks like this:




Above the editor is a toolbar with buttons related both to execution of scripts and to editing. The editing related buttons are covered below.

The left margin shows the line numbers.

Below the editor, you see a Status Bar with the following fields, from left to right:

1. Position: the current caret position in the format: `<line>:<column> [<position from top>] </br>` The last figure, within square brackets, is the caret position from the top. This can be useful when you get an error message executing a script that contains this information rather than a line/column location.
2. Insert/Overwrite Mode: **INS** if characters you type will be inserted at the caret position or **OVR** if they will overwrite the current text at the caret position. You can toggle this mode using the **Toggle Typing Mode** keyboard shortcut, by default bound to the **Insert** key.
3. Macros: this field is only visible when working with macros, as described in the Recording and Playing Edit Macros section.
4. File Format: the format of the file as detected when the file was loaded (if any). Click it to select which format to use when saving the file; **Windows**, **Unix/Linux/macOS**, or **Old MacOS**.
5. Auto Commit Status: shows whether or not Auto Commit is enabled.
6. Character Set: the character encoding as detected when the file was loaded (if any). Click it to select which encoding to use when saving the file.
7. File: The name of the loaded file (if any). You can click on the filename to copy the file path or open the OS file chooser for the directory holding the file. If you just type into the editor without loading a file, the filename "Untitled" is shown instead. An asterisk (*****) after the filename indicates that there are unsaved edits.

The SQL Editor is like any editor you're used to when it comes to typing, scrolling etc. But it also offers additional features to help you specifically with editing SQL scripts. These are described in the following sections.

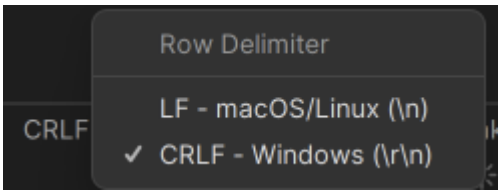
You can change how to display in the **Tools → Tool Properties** dialog, in the **General / Appearance** category. This is explained in more detailed below.

## Row Delimiter

This is how the SQL editor handles row delimiter:

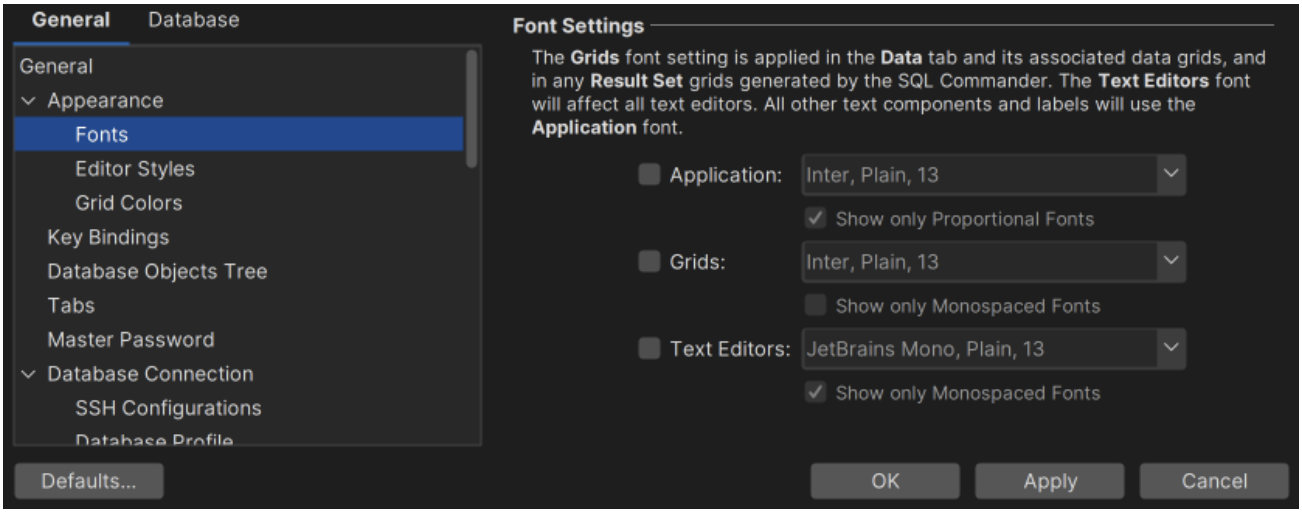
1. Linefeed (LF) is used as the line break between rows.
2. Loaded files/texts are scanned for carriage return (CR) or carriage return, line feed (CRLF). The first found line break that is not a LF is set as the line break style for the document.
3. Drop of a file to an existing document does not change the line break.
4. Paste text do not change the line break.
5. Copy text do not transfer the line break.
6. The setting in the SQL Editor determines the line break used at next save.
7. The setting in tool properties determines line break for new SQL Commanders (files).

In the editor status bar, the current line delimiter is shown. Click it to modify what line delimiter should be used at next save.




## Font Settings

In the **Appearance/Fonts** category, you can select the font for **Text Editors** to control the font in the SQL Editor (**Monospaced Fonts** are recommended). The **Grids** settings controls which font to use in all result set grids while the **Application** font controls the rest of the user interface.




### Zooming In the Editor

It is also possible to change the font size in individual editor tabs. To do so, press a modifier key as defined in **Tools → Tool Properties and in the** SQL Commander / Editor** category, and use the mouse wheel or the scroll gesture on your trackpad. This gives the effect of zooming in and out on the contents of the current tab and can be especially handy when sharing your screen in a meeting or discussing with a colleague in front of the display. The font size can be different in different tabs and is not preserved between sessions.

## Editor Styles

An SQL script consists of keywords, operators, object identifiers, quoted text, etc. It may also contain comments. To make it easier to see at a glance what is what, the SQL Editor displays words using different font styles depending on their classification. For instance, keywords are displayed with a bold blue font, while quoted text is displayed with a regular type red font.

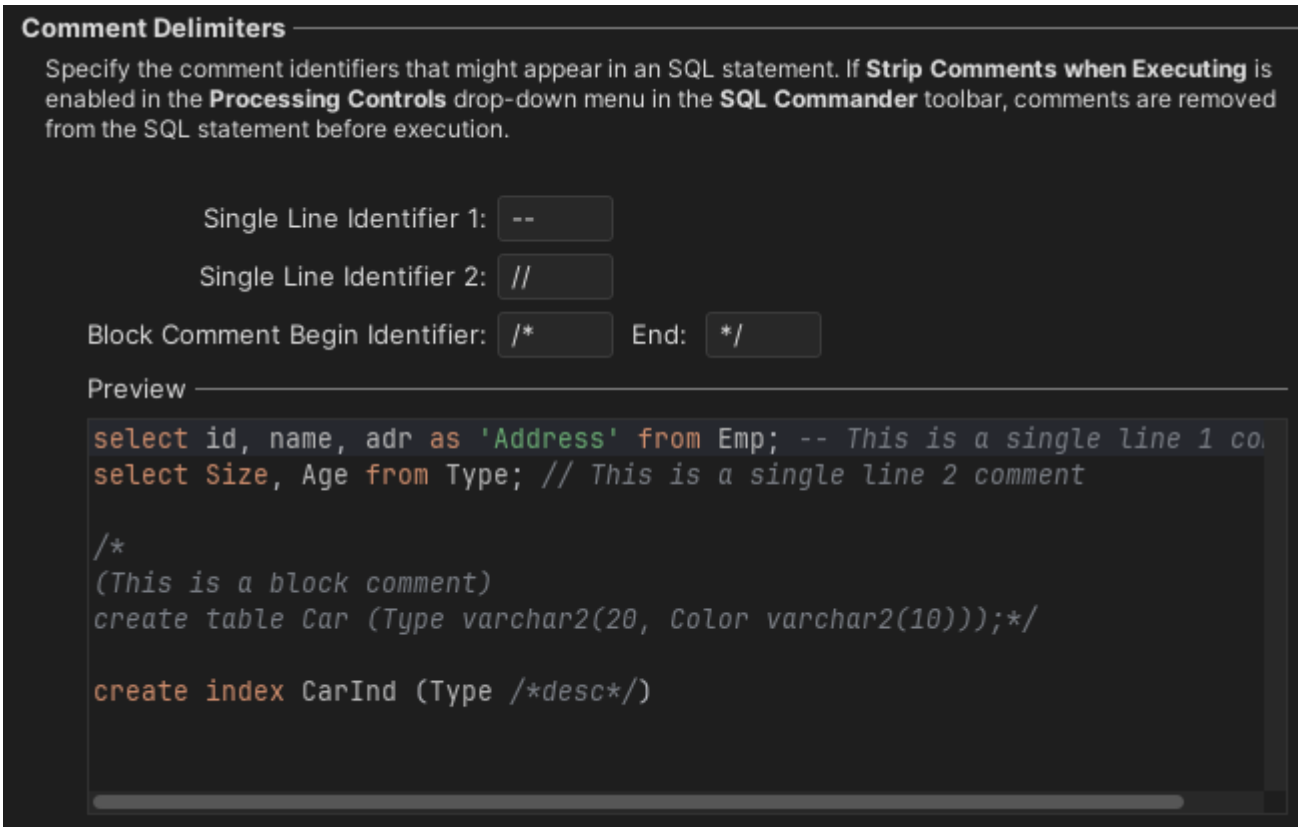
In the **Appearance/Editor Styles** category you can select colors for the different kinds of words, as well as the editor selection background color, the current line highlight color and the editor background color, and more.

The **SQL Editor Column Guide** is an optional visual guide that appears as a thin vertical line at specified column (this works best if you use monospaced fonts).



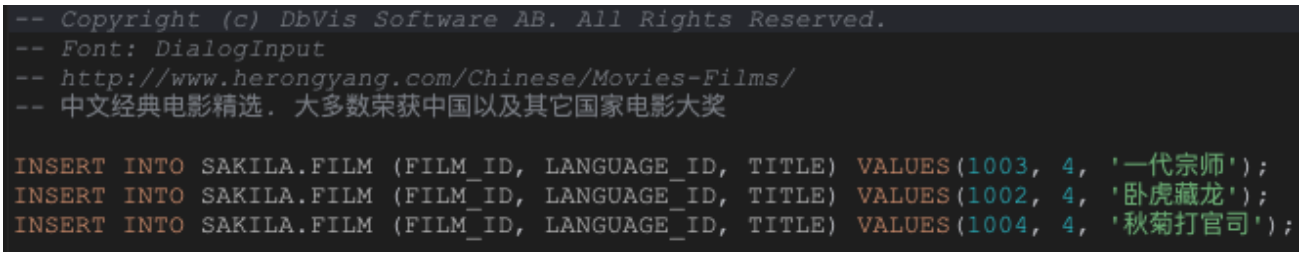
## Comments

The editor uses the Tool Properties settings from the **SQL Commander/Comments** category under the General tab to detect comments.




## Charsets and Fonts

You can also change the SQL Editor font family, which is useful and necessary in order to display characters for languages like Chinese, Japanese, etc., in **Tool Properties** in the **Appearance/Fonts** category to set the font for the SQL Editor (see Internationalization and Localization (i18N and L10N) for more information).



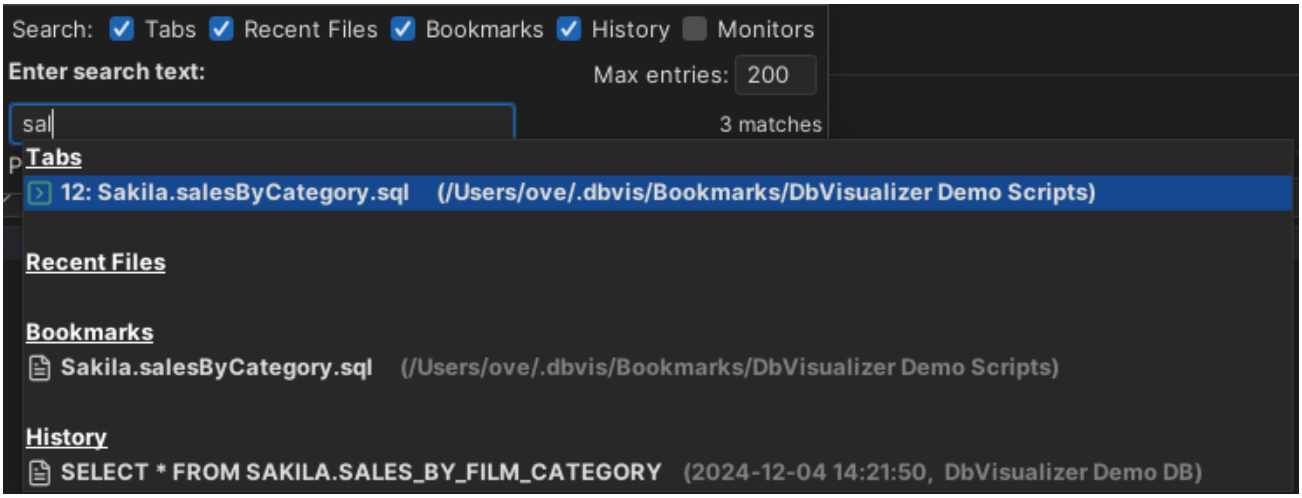
# Loading and Saving Scripts

The SQL editor supports loading statements from a file and saving the content of the editor to a file. Use the standard file operations, **Open**, **Save** and **Save As** in the **File** main menu or the main toolbar to accomplish this. Loading a file loads it into a new **SQL Commander** tab or activates the tab that already holds it.

The name of the loaded file is listed in the status bar of the editor, with the full file path shown in the window title. The editor tracks any modifications and indicates changes with an asterisk (*) after the filename. When you close the SQL Commander tab or exit DbVisualizer, you are asked what to do if there are any pending edits that need to be saved.

The **File → Open Recent** submenu lists the recently loaded files. How many recent files to keep track of can be specified in the Tool Properties dialog, in the **SQL Commander** category under then General tab.

An alternative to locating editor tabs, Recent Files, Bookmarks or Monitors from the Files tab and History entries from the History window is to use the **Quick Load** feature, by default bound to the **Ctrl+Alt+O** key combination. It is also available via a main toolbar button as well as in the **File → Quick File Open** menu.



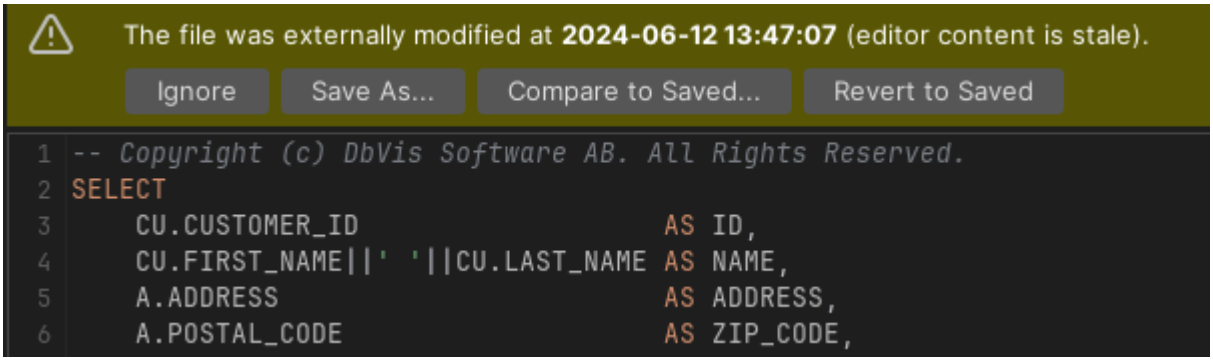

The Quick Load feature locates files with partly matching names from the categories you have selected, as you type. You can use an asterisk ("*") as a wildcard in the search string. Files already open in editor tabs is not displayed in Recent Files.

When you see the file you're looking for, just select it and press **Enter** to load it into an SQL Commander editor. If the file is already loaded in an editor, that tab is made visible instead.

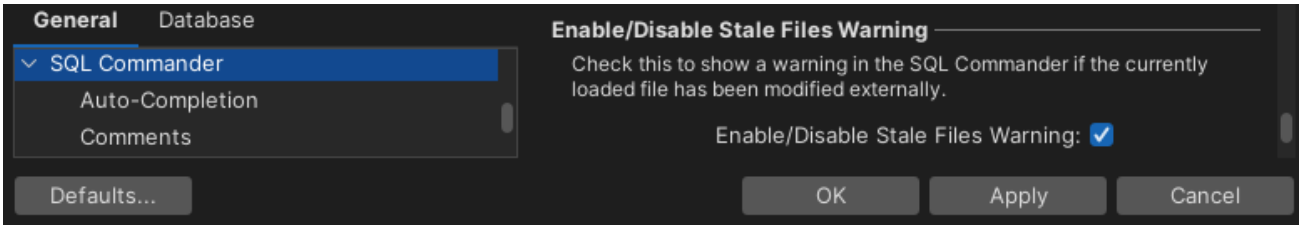
The number of matches is shown on the search panel as seen on the figure above. It the number of matches is big, and you want to change the size, you can press **Escape** and change the number in the **Max entries** field that is shown. Go then to the **Enter search text** field and press arrow down. The result list shown will be limited to your set number for each category.

## Stale Files Warning

**SQL Commander** monitors open files to detect external changes, for example if an open file is modified by an external editor, reloaded from a network connection or repository, etc. If a file is externally modified, you will see a warning ribbon at the top of the editor area, presenting options to handle the situation:




You can turn off this control in **Tool Properties/SQL Commander:**



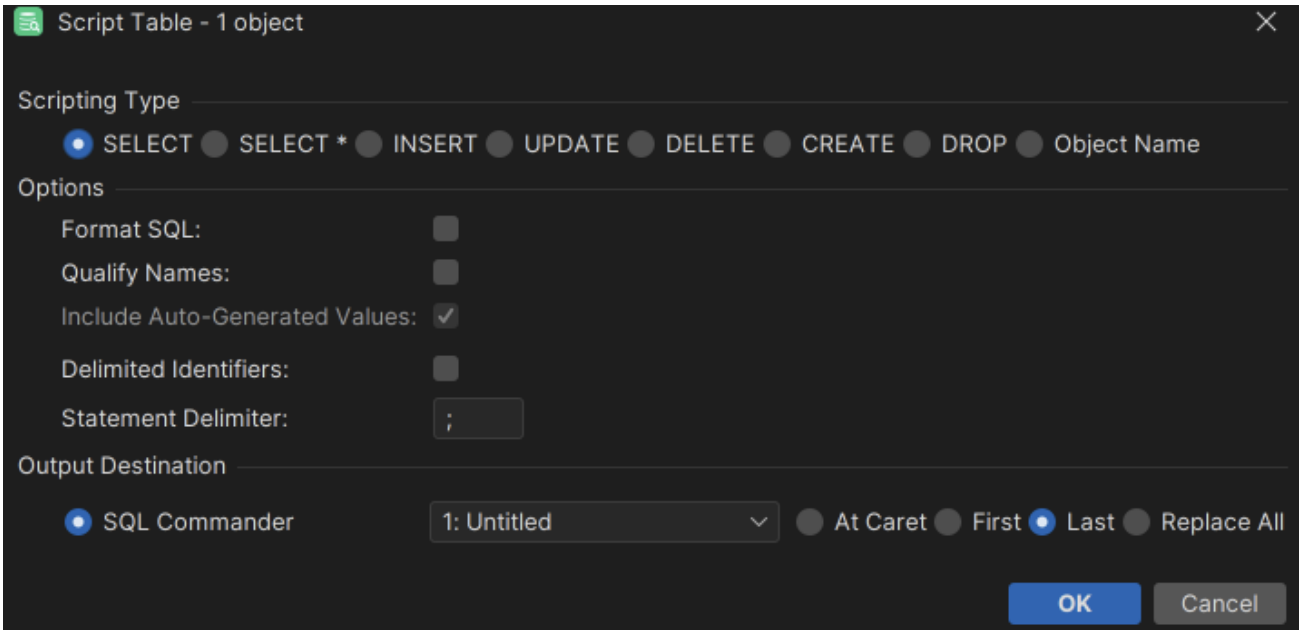

> This does not affect the check when you save the file; DbVisualizer will always check and warn if you're trying to save a file that was externally modified.

## Drag and Drop a File

You can also select a file in the platform's file browser and drop it somewhere in the DbVisualizer window. If you drop it in an editor, the file content is inserted at the caret position in the editor. If you instead drop it in the toolbar area, the file is opened in a new **SQL Commander** tab.

## Drag and Drop Database Objects

If you want to include an object shown in the database objects tree, you can select the node and drop it in the editor where you want it inserted. The **Script Object** dialog is shown where you can select exactly what you want to insert in the editor.




First of all, you can select to insert an SQL statement based on the dropped object, e.g. a SELECT statement or a CREATE statement. You can also choose to just insert the object name. The choices available depends on the type of object you drop.

In the **Options** area, you can opt to format the SQL before it is inserted and use qualifiers and quoted identifiers, and even change which statement delimiter to use.

The **Output Destination** is set to the SQL Commander tab you dropped the object on by default, but you can change your mind and pick another destination. If you stick with an **SQL Commander** as the destination, you can tell where in the editor to insert the text.

You can also open this dialog from the **Databases** tab, from the object's right-click menu.

> If you just want to insert the object names in the editor, hold down the **Ctrl** key (or the **Alt** key on macOS) while dragging and dropping.

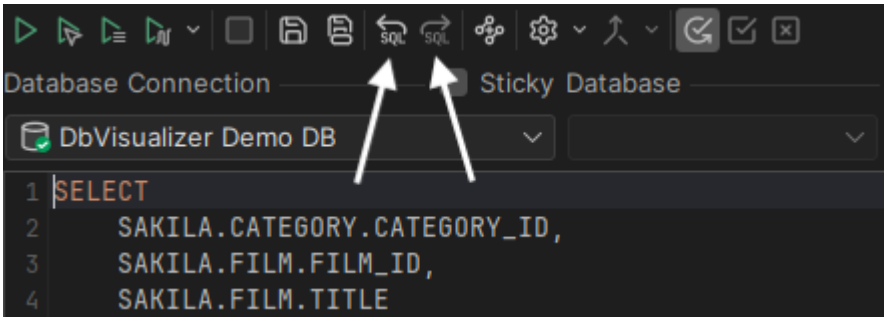
This behavior can be reversed in **Tool Properties**, in the **SQL Commander** category, so that dropping without pressing a key inserts the names and pressing the key launches the dialog."

## Loading and Saving Bookmarks and Monitors

Bookmarks and Monitors are also files, but with special meaning. See the Managing Frequently Used SQL for how to create and edit them in the SQL Editor.

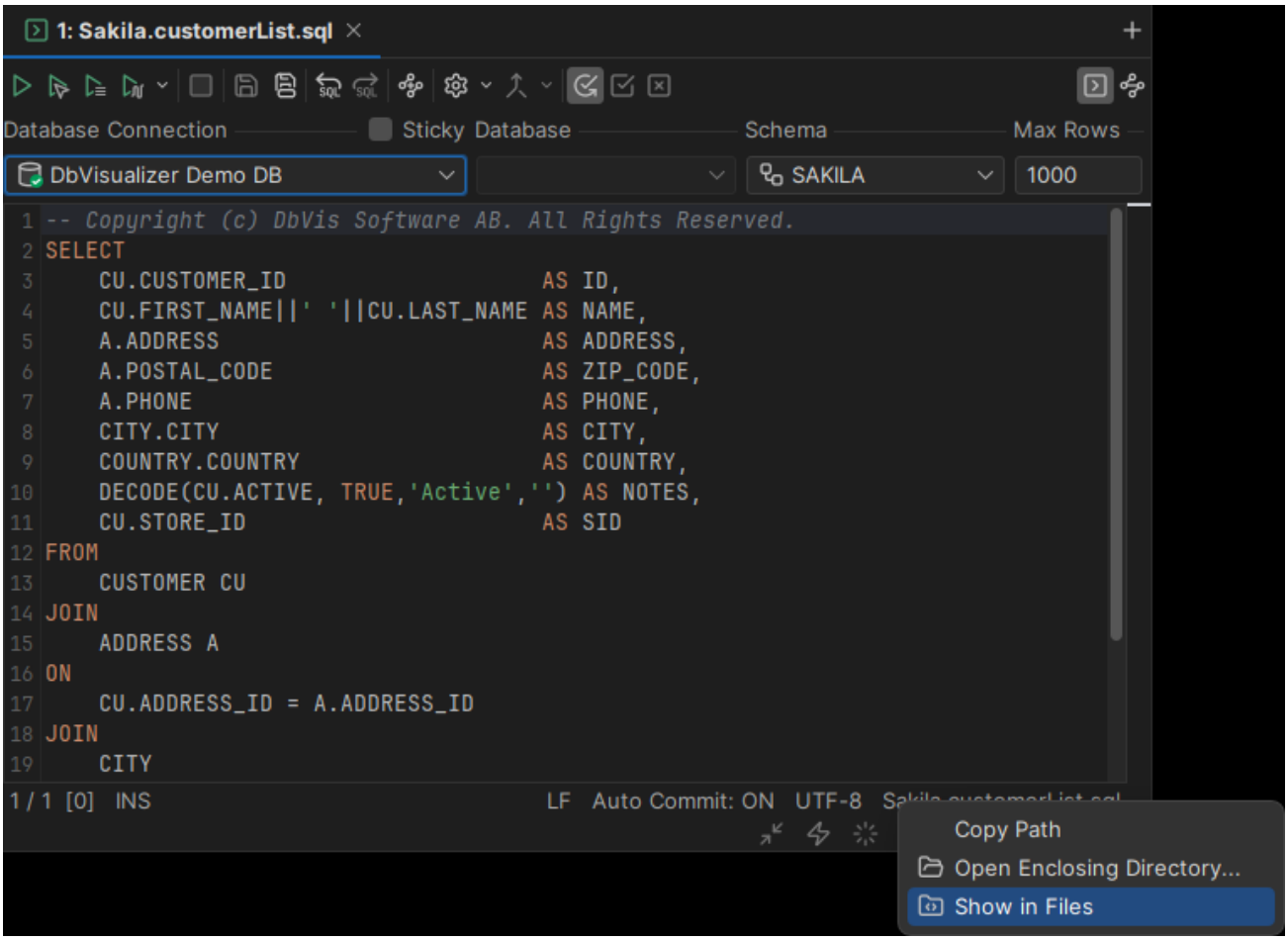
## Navigating Between History Entries

When you execute a script, DbVisualizer saves it as a history entry, see the Re-Executing SQL Statements section for details. You can use the **Previous** and **Next** buttons in the editor toolbar to navigate between (load) these entries.




## Navigating to Script Location

Clicking on the script filename in the right bottom corner of the editor brings up a menu that allows to copy the script's path (**Copy Path**), open the script's containing directory in the operating system's file browser (**Open Enclosing Directory**), or locate it in the Files tab (**Show in Files**).



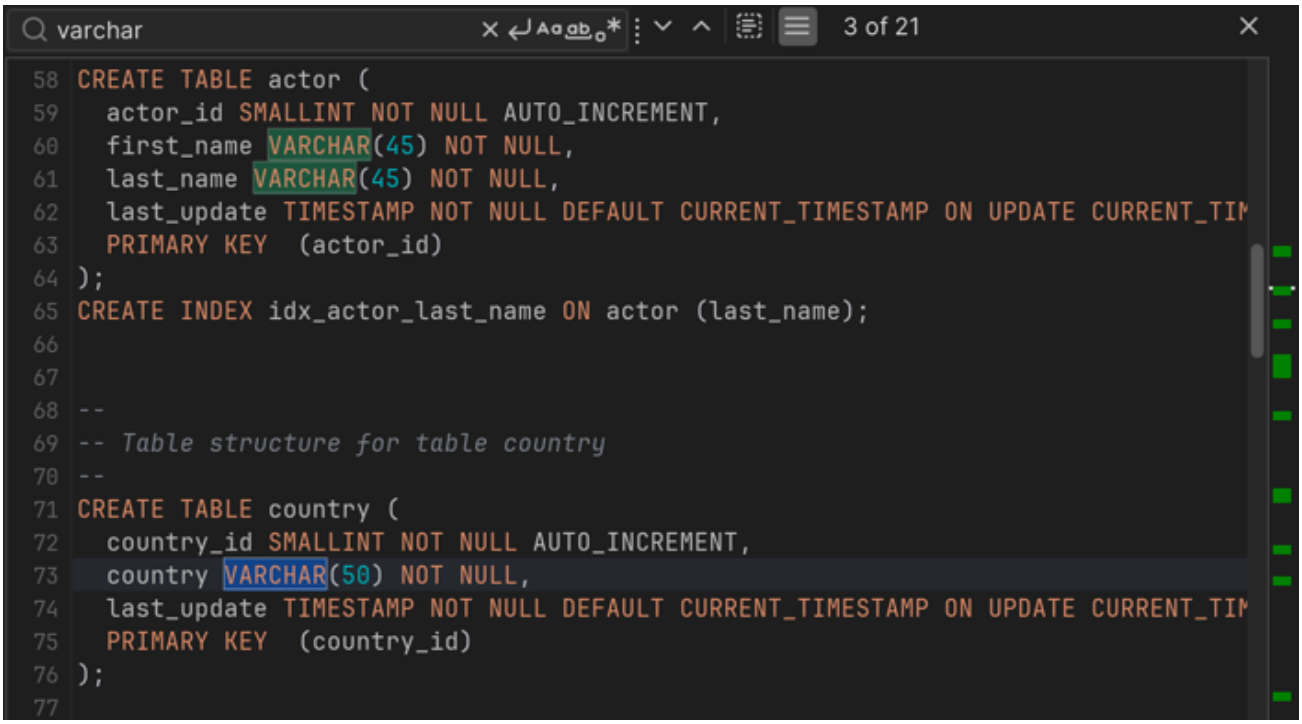

## Confirming Overwriting Unsaved Changes

By default, you have to confirm overwriting unsaved changes in an editor, e.g. when navigating between history entries, and when closing an SQL Commander tab with unsaved edits. You can disable these confirmation popups in the Tool Properties dialog, under the **SQL Commander** category under the General tab.

## Finding and Replace in the Editor

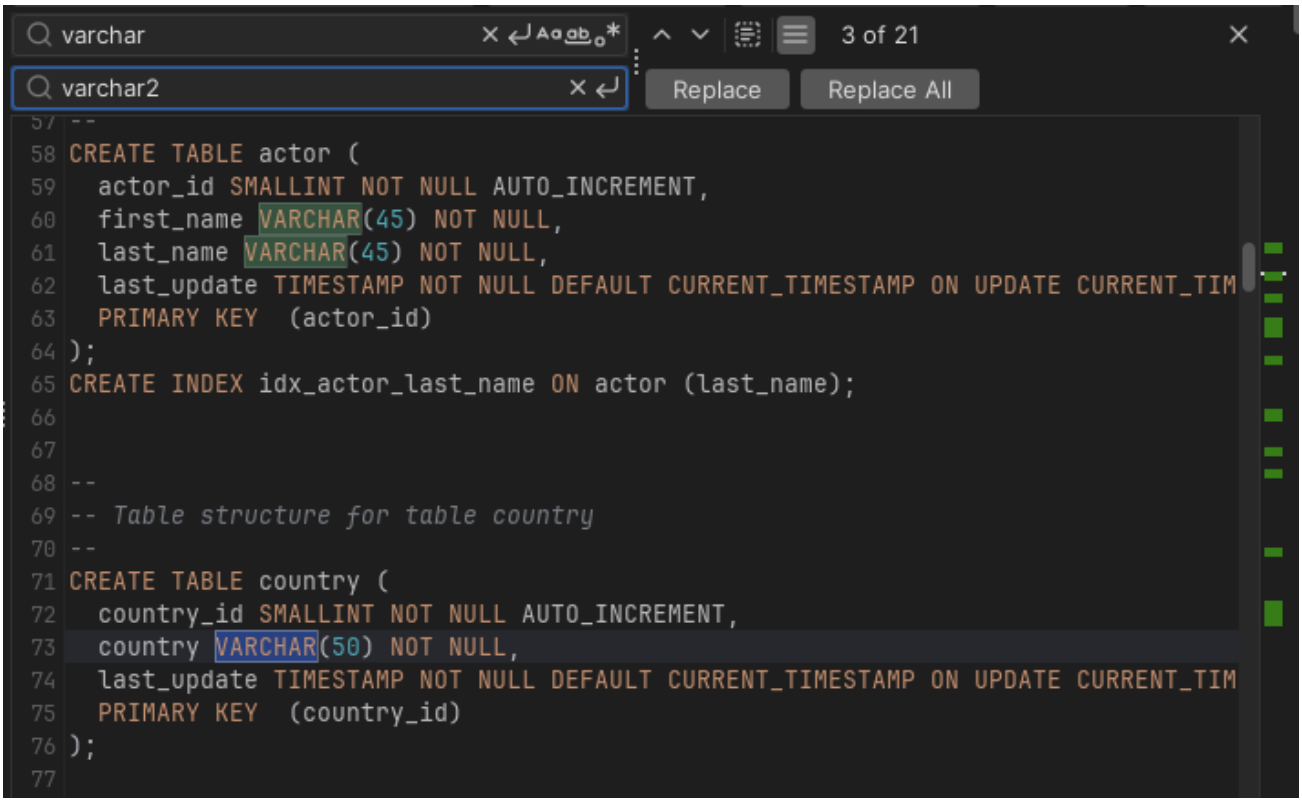
The **Edit** main menu and the editor right-click menu contains the **Find** (Ctrl+F) and **Replace** (Ctrl+R) actions.

**Find** shows a pane at top of the editor where you can enter what to look for, either as text, a regular expression or using wildcards. You can also limit the search to the current selection and use other options for a more precise search. Use **Find Next** and **Find Previous** to navigate to other matches, by default mapped to the **F3** and **Shift-F3** keys, or **Down** and **Up** keys.




Matches are highlighted while typing in the search field.

When in **Replace** mode, a second field is displayed used to enter the replacement text.



When you use a regular expression with group expressions in the **Find what** field, you can reference the captured text in the **Replace with** field using the **dollar sign** (**$**) plus the group number (e.g. `$1` for the first group).

### Regular Expression Example:

**Text in editor:**

```
SELECT
    firstname,
    lastname
FROM
    person;
```

**Find what:**

```
^(\s+)(\w+)(,?)$
```

**Replace with:**

```
$1person.$2$3
```

Deselect **Match whole word**, select **Entire scope** and **Use Regular Expression**.

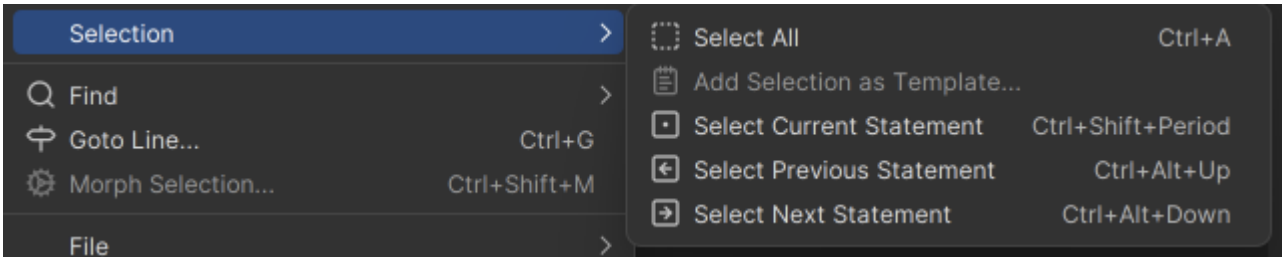
When you click **Find** and choose **All** in the following dialog, you get the following result:

**Resulting text:**

```
SELECT
    person.firstname,
    person.lastname
FROM
    person;
```

## Selecting SQL Statements

**Select Current Statement**, **Select Next Statement** and **Select Previous Statement** can be used to find individual statements in the editor. You find them in the submenu **Selection** of the right-click menu, or you can use the corresponding key bindings.



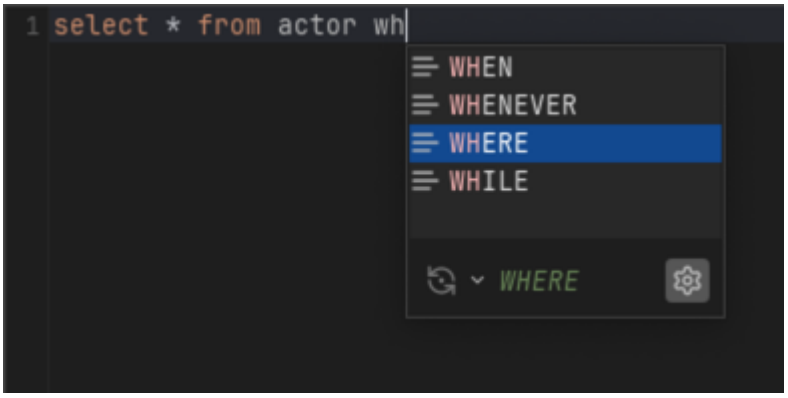
## SQL Formatting

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

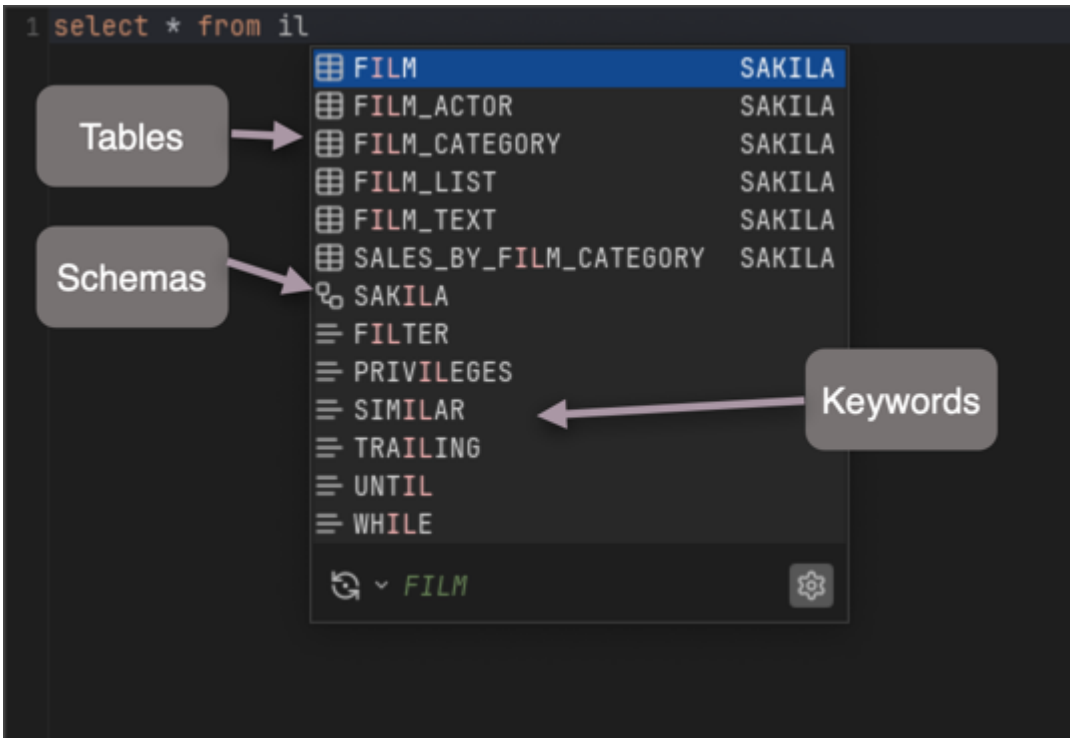
## Auto Completion

Auto completion is a convenient feature used to assist you when editing SQL statements and DbVisualizer commands. By default, you activate auto completion with the key binding **Ctrl-SPACE**, but you can also configure it to activate as you type (in the Tool Properties dialog, in the **SQL Editor/Auto Completion** category under the General tab).

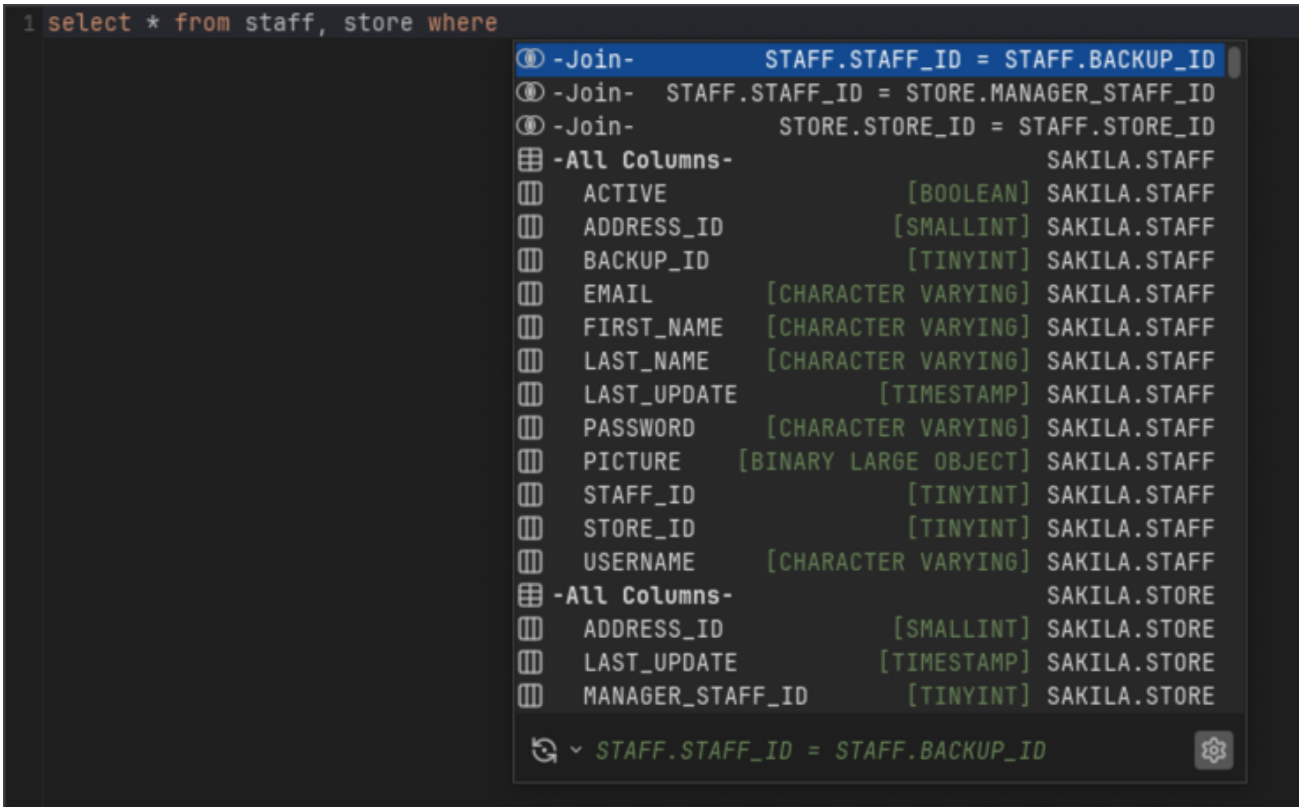
With the caret placed anywhere in a statement, activating auto completion displays a list of keywords that match the letters you have typed so far. As you continue to type, the list narrows. The list of keywords is database specific, selected based on the database type for the connection currently selected in the **Database Connection** list above the editor.




With the caret placed where a table or view name may be typed in a supported SQL statement type, the auto completion list also shows a list of tables and views from the currently selected database connection, assuming you are actually connected to the database. When relevant, the popup may also show catalogs or schemas. The following figure shows the completion pop up with schemas, tables and keywords that contain the letters **IL**.



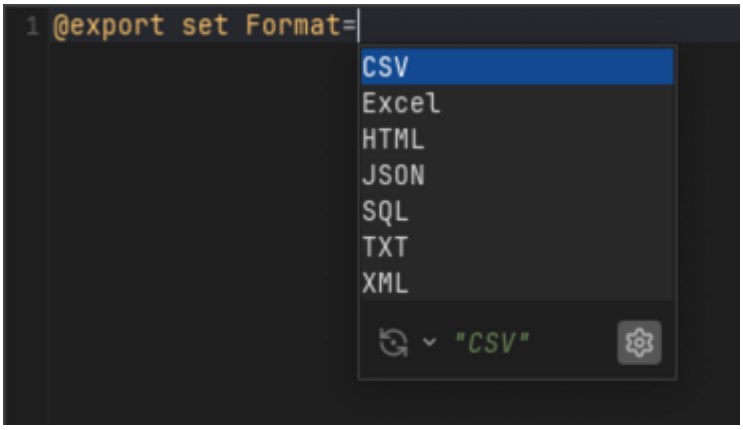

With the caret placed where a column name may be typed in a supported SQL statement type, the completion pop-up shows column names for the relevant tables. As a convenience when selecting from several tables, joinable columns are suggested as conditions.



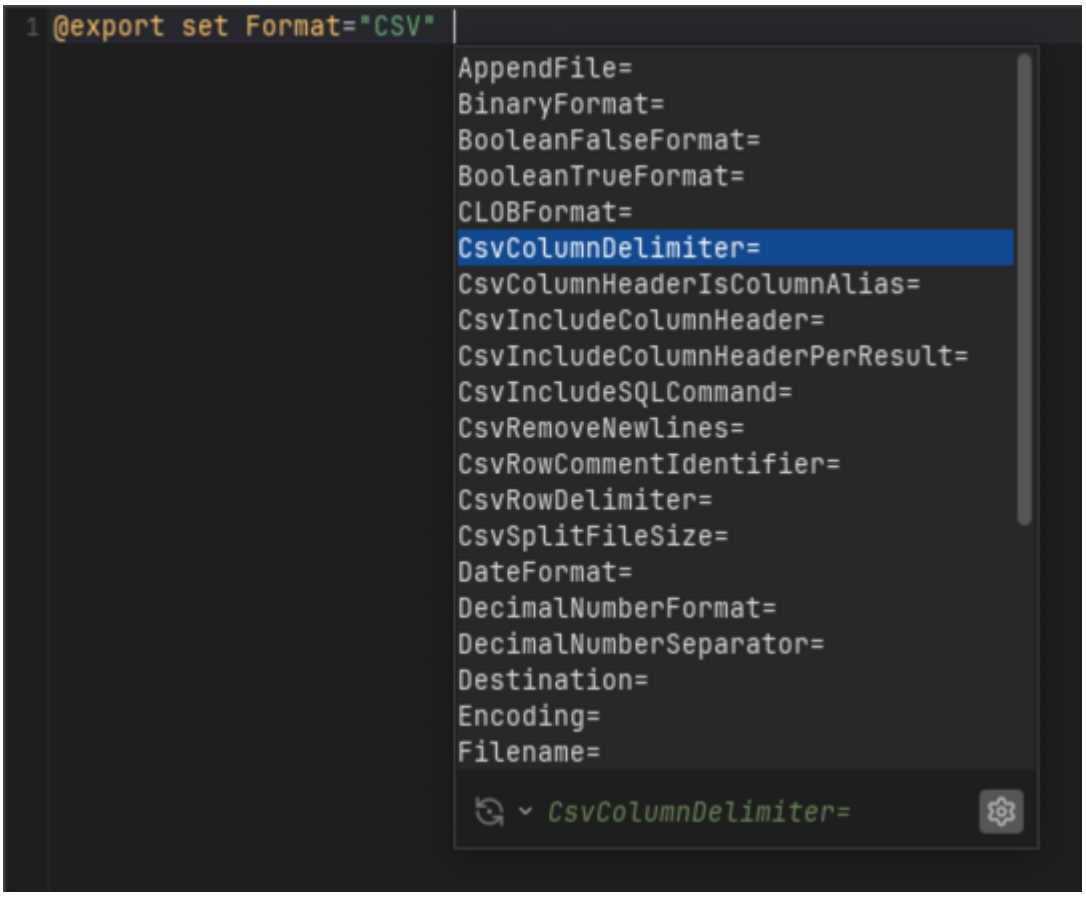
DbVisualizer provides auto completion for the following DML commands: - SELECT - INSERT - UPDATE - DELETE

And the following DDL commands: - ALTER - DROP

Auto completion for DbVisualizer commands is very similar. Activating it after a partial command name lists all matching commands. If you activate it after a complete command name, you get a list of all valid parameters for the command. After a parameter name, you can select from a list of valid values.




For the `@export set` command, the parameter list is adapted for the specified output format after you have entered the `Format` parameter setting, for instance only showing parameters that are valid for the CSV format.



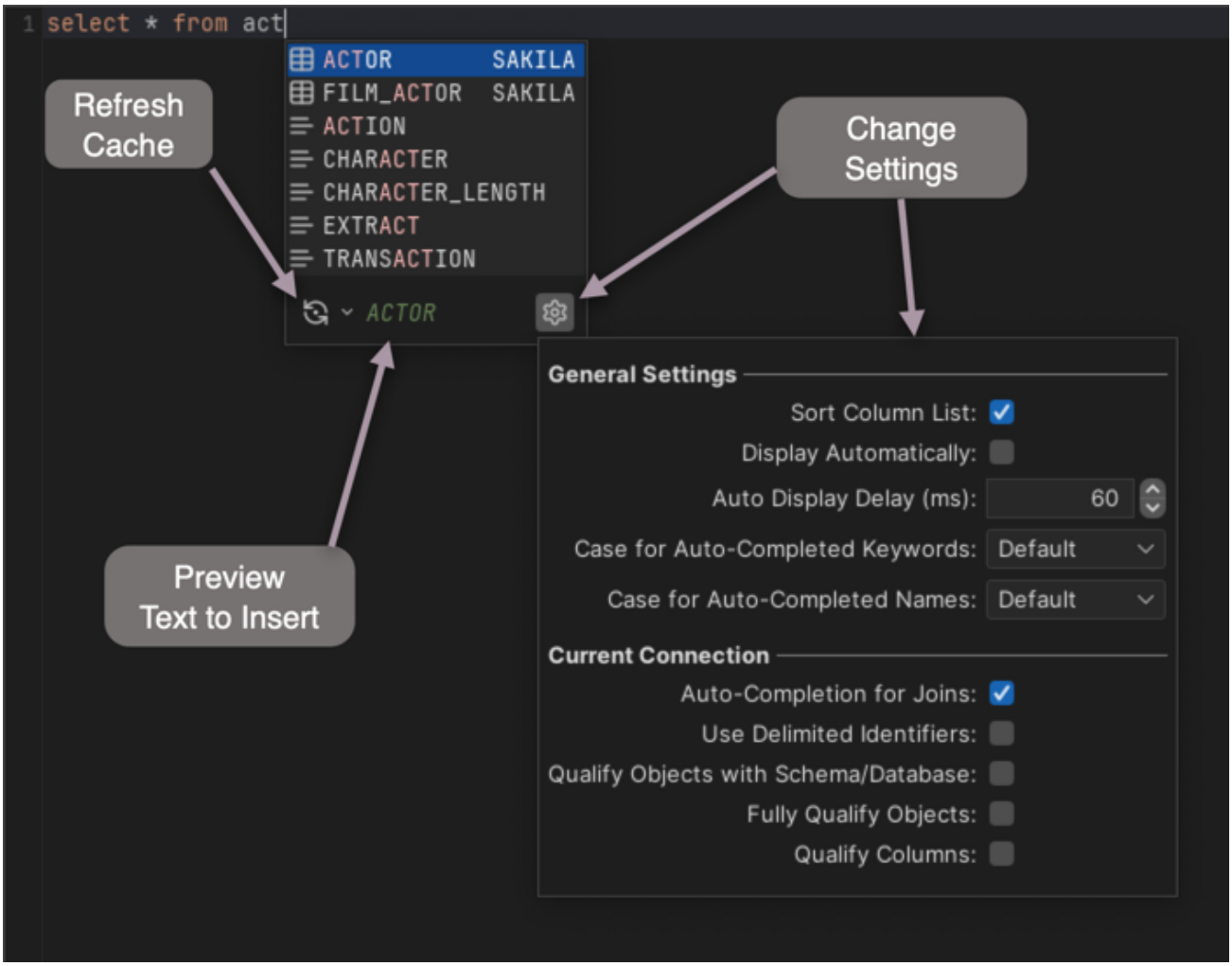

To display the completion pop-up, use the key binding **Ctrl-SPACE** (by default). You select an entry in the pop-up menu with a mouse double-click, the **ENTER** key, or the **TAB** key. To cancel the pop-up, press the **ESC** key.

> If there are several SQL statements in the editor, make sure to separate them using the statement delimiter character (the default is ";").
>
> In order for the column name completion pop-up to appear, you must first make sure there are table names in the statement. All names that have been listed in the completion popup are cached by DbVisualizer to make sure subsequent displays of the pop-up is performed quickly without asking the database. If you modify the schema (create or alter elements), the cache may need to be refreshed . The **Database** and **Schema** lists above the editor are used to limit the list of tables in the auto complete pop-up to those in the selected database and/or schema. It is possible to fine-tune how auto-completion works in the connection properties.

- Enable or disable the use of identifier qualifiers (i.e. qualifying table names with the schema name) in the **[Database Type]/Qualifiers** category. Ambiguous identifiers are automatically qualified.
- Enable or disable the use of delimited identifiers (e.g. quotes around a table name) in the **[Database Type]/Delimited Identifiers** category. Invalid or ambiguous identifiers are automatically delimited as necessary.

Sorting, when to show the popup, upper/lower case transformation, etc. can be configured in the **Tool Properties** dialog, in the **SQL Editor/Auto Completion** category under the **General** tab. Note that forcing case transformation may render invalid SQL; DbVisualizer will typically transform identifiers as expected by the connected database.

At the bottom of the auto-complete popup, you can see a preview of the text to be inserted, change the settings for auto-completion, and update the Metadata Cache to make the popup reflect recent changes.




## Recording and Playing Edit Macros

If you repeatedly need to run a sequence of edit operation, you can record them as a macro and play it as many times as needed during an editing session. The editor status bar indicates when a recording is in progress and when a macro is available to play.

As an example, suppose you have some plain text that you need to convert into INSERT statements:

```
12345 123456
89012 890123
45678 456789
```

Place the caret at the beginning of the first line and start the macro recording, using the right-click menu or the corresponding key binding, and then type text and use key bindings to perform the following operations:

1. Type `insert into mytable values('`
2. **Insertion Point to Next Word**
3. Type `',`
4. **Insertion Point to Next Word**
5. Type `'`
6. **Insertion Point to Next Word**
7. Type `');`
8. **Insertion Point Down**
9. **Insertion Point to Beginning of Line**

Then stop the recording. You now have a macro for converting a single line to an INSERT statement. To convert the remaining lines, just use Play Macro for each line. The result will look like this:

```
insert into mytable values('12345', '123456');
insert into mytable values('89012', '890123');
insert into mytable values('45678', '456789');
```

> The Find operation, by default mapped to the **Find** key and **Ctrl-F** key stroke, can not be recorded. You must instead use **Find Selection**, **Find with Dialog**, **Find Next** and **Find Previous**. Mouse gestures are also not recorded, only key strokes and menu selections.

## Folding

If you work with a large script containing many individual statements, it can sometimes be helpful to hide some. You can do so using the Folding feature. **Folding → Toggle Folding** in the right-click menu.

Here's an unfolded script with the statements visible:

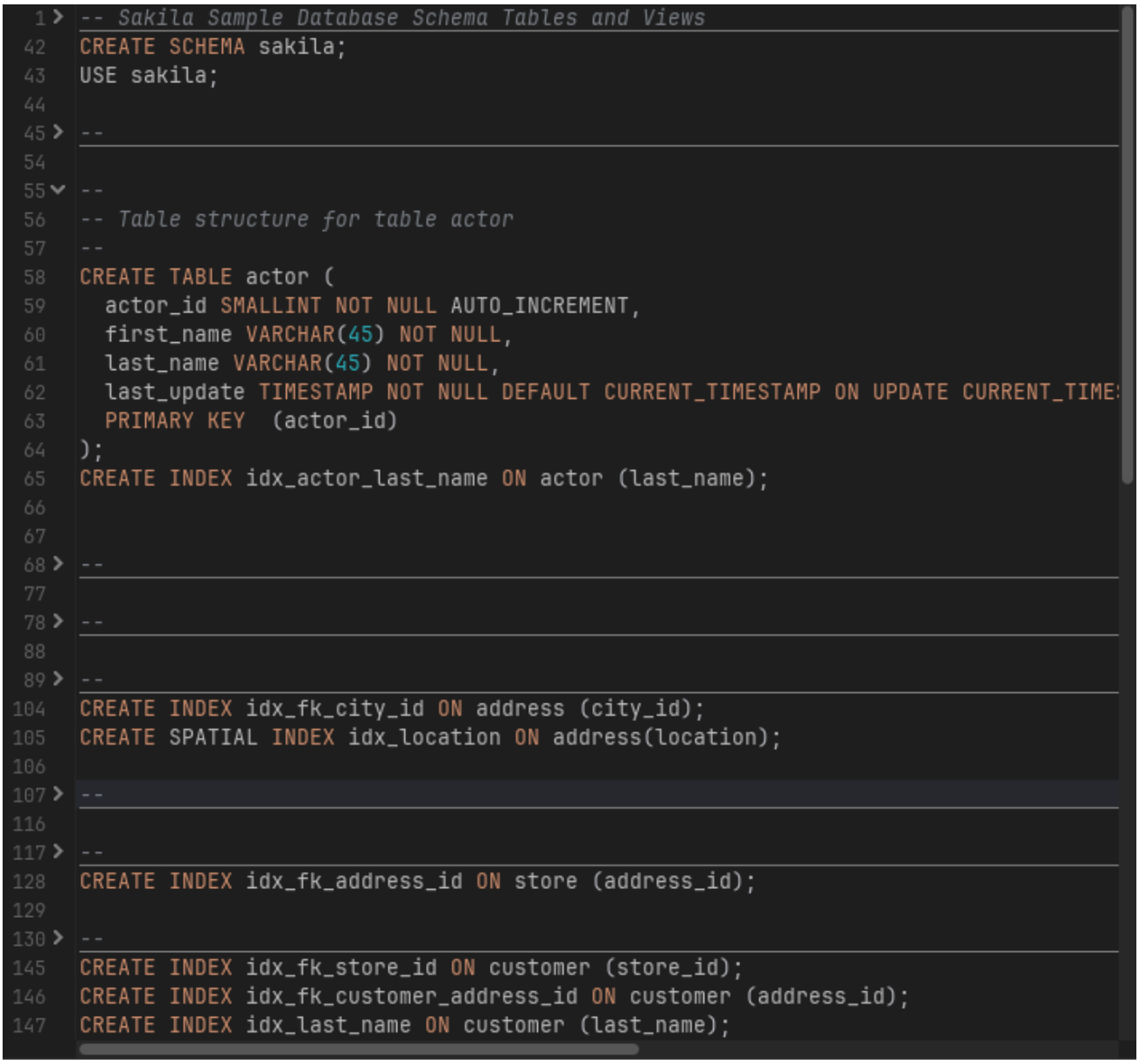
```
CREATE SCHEMA sakila;
USE sakila;

--
-- Table structure for table language
--
CREATE TABLE language (
  language_id TINYINT NOT NULL AUTO_INCREMENT,
  name CHAR(20) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIME
  PRIMARY KEY (language_id)
);

--
-- Table structure for table actor
--
CREATE TABLE actor (
  actor_id SMALLINT NOT NULL AUTO_INCREMENT,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIME
  PRIMARY KEY  (actor_id)
);
CREATE INDEX idx_actor_last_name ON actor (last_name);


--
-- Table structure for table country
--
CREATE TABLE country (
  country_id SMALLINT NOT NULL AUTO_INCREMENT,
  country VARCHAR(50) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIME
  PRIMARY KEY  (country_id)
);
```

Here is the same script with statements folded. Use the button in the row number area to expand/collapse a statement.



```
-- Sakila Sample Database Schema Tables and Views
CREATE SCHEMA sakila;
USE sakila;

--

--
-- Table structure for table actor
--
CREATE TABLE actor (
  actor_id SMALLINT NOT NULL AUTO_INCREMENT,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIME
  PRIMARY KEY  (actor_id)
);
CREATE INDEX idx_actor_last_name ON actor (last_name);


--

--

--
CREATE INDEX idx_fk_city_id ON address (city_id);
CREATE SPATIAL INDEX idx_location ON address(location);

--

--
CREATE INDEX idx_fk_address_id ON store (address_id);

--
CREATE INDEX idx_fk_store_id ON customer (store_id);
CREATE INDEX idx_fk_customer_address_id ON customer (address_id);
CREATE INDEX idx_last_name ON customer (last_name);
```

## Highlighting Matches

Instead of searching for occurrences of a text string and navigating to each occurrence, it is sometimes useful to get all occurrences highlighted. To do this, select a text string that is at least three characters long and contains at least one letter or digit. You can use the **Tool Properties** dialog to enable or disable this feature (in the **General / SQL Commander** category) or change the colors used (in the **General / Appearance / Editor Styles** category).



```
SELECT
    FILM_ID,
    TITLE,
    RELEASE_YEAR
FROM
    FILM
WHERE
    FILM_ID IN
    (   SELECT
            FILM_ID
        FROM
            FILM_CATEGORY
        WHERE
            CATEGORY_ID IN
            (   SELECT
                    CATEGORY_ID
                FROM
                    CATEGORY
                WHERE
                    NAME = 'Family' ) );
```

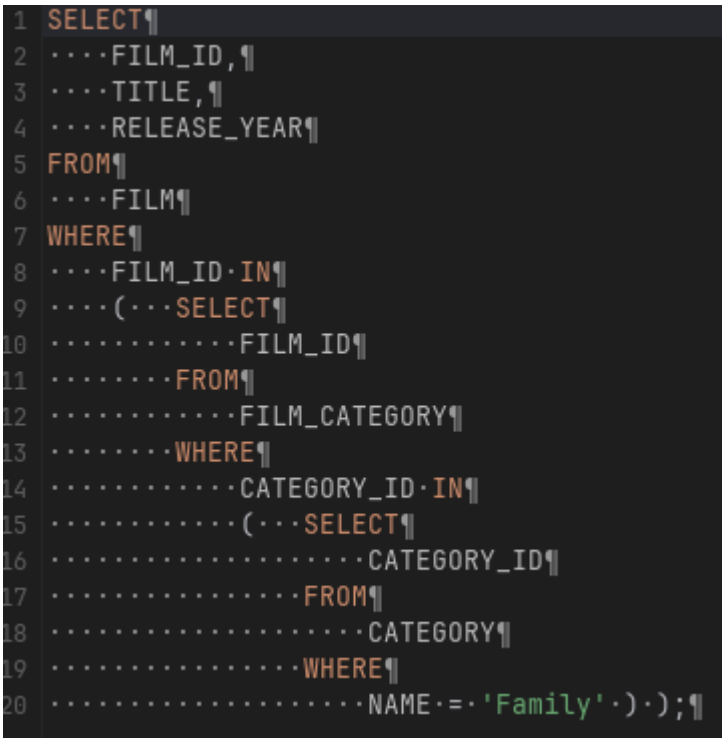
## Tab Key Treatment

Pressing the TAB key in the editor inserts eight (8) space characters by default. If you instead want a TAB character to be inserted, or want to insert another number of space characters, you can specify this in the **Tool Properties** dialog, in the **General / SQL Commander** category under the General tab.

## Show Special Characters

Sometimes it is useful to render characters that are not visually represented such as whitespace, tab, newline, etc. To do this, make sure an SQL editor is active and then open the **Edit** meny and toggle **Show Special Characters**.

With Show Special Characters enabled:




## Key Bindings

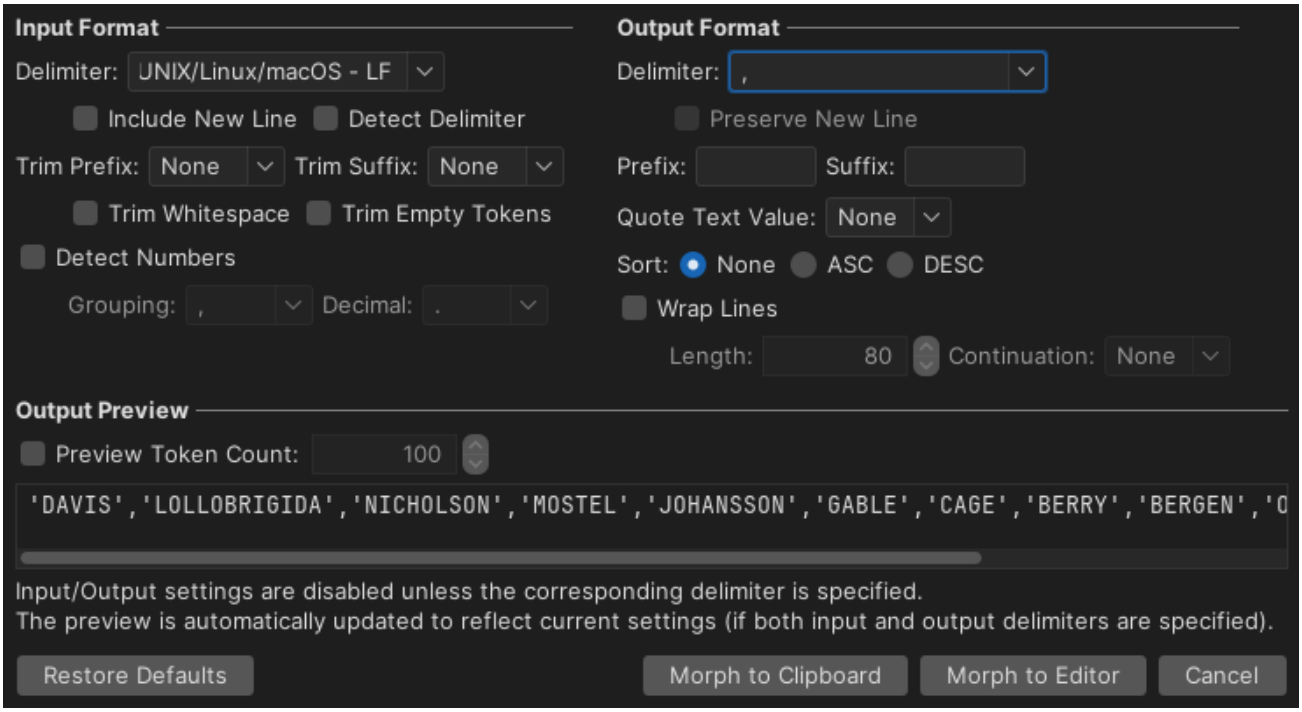
The editor shortcuts, or key bindings, can be redefined in the **Tool Properties** dialog, in the **Key Bindings** category under the **General** tab (see Changing Keyboard Shortcuts). Expand the **Editor Commands** node to manage all editor actions and the **Main Menu/Edit** node to manage the key bindings for the edit operations in the right-click editor menu and the main window **Edit** menu.

# Morph Selection

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

## Introduction

The **Morph Selection with Dialog** in the SQL editor allows you to convert a selection of text into a delimited list with the appropriate options. Simply select text in the editor and choose **Edit → Morph Selection with Dialog** (or from the editor right-click menu). This will prompt you for different options about the **Input Format** and the **Output Format**, with the **Output Preview** overwriting the input text with the result. This lets you paste text from a Grid, Excel or some other application into the editor and then convert it into a delimited list suitable for completing a query or other documentation.




## Basic Examples

Some basic examples of how various settings affect the transformations.

## Detect Input Delimiter

Detect the input delimiter by counting occurrences of the default delimiters (the predefined delimiters shown in dropdown). In this case, comma (,) is the most frequent of the predefined separators (single quote (') is not a predefined delimiter).

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | ON | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
'A','B':'C','D'
```

**Result**

```
'A';'B':'C';'D'
```

## Detect Numbers

Detect numbers and quote text tokens; recognized numbers (strings that match the specified number format) are not quoted.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | ON | Continuation | (N/A) |
| Grouping | , | preview setting | value |
| Decimal | . | Preview Token Count | OFF |

### Input

```
1,000.00:two thousand:3000,00:4000
```

### Result

```
1000.0;"two thousand";"3000,00";4000
```

## Input Prefix and Suffix

Trim prefix and suffix in input tokens. Text inside prefix/suffix are treated as one token.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | ( | Suffix | None |
| Trim Suffix | ) | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
A:(B:C)
```

**Result**

```
A;B:C
```

## Limit Preview

If the input is large, it may be useful to limit preview to speed up things while sorting out options. The limit is applied to number of input tokens as determined by the current settings.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | 2 |

**Input**

```
Token1:Token2:Token3
```

**Result**

```
Token1;Token2
```

## Missing Input Prefix

Ignore trimming of input prefix and suffix unless both are defined (here we only defined the prefix).

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | ' | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
A:'B:C
```

### Result

```
A;'B;C
```

## Missing Input Suffix

Ignore trimming of input prefix and suffix unless both are define (here we only defined the suffix).

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | ' | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
A:'B:C
```

**Result**

```
A;'B;C
```

## New Line as Input Delimiter

Transform delimiter and quote text spanning multiple lines. Since we recognize New Line as a delimiter, **Token2** and **Token3** are interpreted as separate tokens.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | ON | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
Token1:Token2
Token3:Token4
```

**Result**

```
"Token1";"Token2";"Token3";"Token4"
```

## No Delimiters

No transformation occurs unless both input and output delimiters are defined.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | None | Delimiter | None |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
1,000.00::' one thousand '
```

### Result

```
1,000.00::' one thousand '
```

## Output Prefix

We can add a single prefix or suffix to output tokens (we can define one without the other).

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | + |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
A:B:C
```

**Result**

```
+A;+B;+C
```

## Output Prefix and Suffix

We can add prefix and/or suffix to output tokens.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | [ |
| Trim Prefix | None | Suffix | ] |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
1,000.00::' one thousand '
```

**Result**

```
[1,000.00];[];[' one thousand ']
```

## Output Prefix and Suffix with Text Quoting

We can both quote text and add a prefix and/or suffix to output tokens. Prefix/suffix are added to all tokens outside the quoting symbols. Since we detect numbers, the recognized number token is unformatted and not quoted.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | [ |
| Trim Prefix | None | Suffix | ] |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | ON | Continuation | (N/A) |
| Grouping | , | preview setting | value |
| Decimal | . | Preview Token Count | OFF |

### Input

```
1,000.00:one thousand:1 000,00
```

### Result

```
[1000.0];["one thousand"];["1 000,00"]
```

## Preserve Output New Line

Transform delimiter and quote multiline text. Since we recognize and preserve New Line, we maintain a multiline output with quoted tokens.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | ON | Preserve New Line | ON |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
Token1:Token2
Token3:Token4
```

**Result**

```
"Token1";"Token2"
"Token3";"Token4"
```

## Quote Output Text Tokens

Quote text tokens. Since we do not detect numbers, or strip input prefix/suffix, all tokens are quoted and prefix/suffix preserved.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
1,000.00::' one thousand '
```

**Result**

```
"1,000.00";"";"' one thousand '"
```

## Repeated Delimiters

We use **Space** (" ") asa delimiter on an input with several repeated spaces and trim single quote (') suffix but do NOT trim whitespace. Spaces outside the prefix/suffix ar treated as a single delimiter, speces inside the prefix/suffix remain.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | Space | Delimiter | : |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | ' | Suffix | None |
| Trim Suffix | ' | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
1,000.00  ' one thousand '    1k
```

### Result

```
1,000.00: one thousand :1k
```

## Sort Ascending

Sort tokens in ascending order on a single line.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | ASC |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
A:C:B
```

**Result**

```
A;B;C
```

## Sort Descending

Sort tokens in descending order on a single line.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | DESC |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
A:C:B
```

**Result**

```
C;B;A
```

## Sort Multiline Output

This is an unsupported setup: sorting is done on tokens, not lines. Sorting tokens in a multiline text would be very confusing.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | ON | Preserve New Line | ON |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | DESC |
| Trim Empty Tokens | OFF | Wrap Lines | 8 |
| Detect Numbers | OFF | Continuation | \ |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
Token1:Token2
Token3:Token4
```

### Result

```
"Token1"\
;"Token2\
"
"Token3"\
;"Token4\
"
```

## Transform Delimiter

Basic transformation of delimiters using default settings. The defined input delimiters are replaced with the defined output delimiters.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
1,000.00::' one thousand '
```

### Result

```
1,000.00;;' one thousand '
```

## Transform Delimiters on Multi Line Tokens

Transform delimiters and quote text spanning multiple lines. Since we ignore New Line, the **Token2** and **Token3** are interpreted as one token with two embedded New Line. The resulting output is three tokens on three lines, where the middle token includes two consecutive New Line (producing an empty line).

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
Token1:Token2

Token3:Token4
```

**Result**

```
"Token1";"Token2

Token3";"Token4"
```

## Trim Input Empty Tokens

Since the input starts with a delimiter, we get an initial empty token that is trimmed. Since don't trim whitespace or consider New Line as a delimiter, the middle and trailing tokens are not trimmed.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | ON | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
: A : B :
: C : D :
```

**Result**

```
A ; B ;
; C ; D ;
```

## Trim Input Empty Tokens with Include Input New Line as Delimiter

Trim empty tokens while treating New Line as a delimiter. Sine the input starts with a delimiter, we get an initial empty token that is trimmed. Since New Line is considered a delimiter, we get an empty middle token that is trimmed. Since the input ends with a delimiter and a New Line, we get a trailing empty token that is trimmed.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | ON | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | ON | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
: A : B :
: C : D :
```

**Result**

```
A ; B ; C ; D
```

## Trim Input Whitespace

Trim whitespace outside any prefix/suffix (whitespace inside prefix/suffix is preserved). Since New Line is considered as whitespace, we get an empty middle token.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | ON | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
: A : B :
: C : D :
```

### Result

```
;A;B;;C;D;
```

## Trim Input Whitespace and Empty Tokens

Trim whitespace outside any prefix/suffix (whitespace inside prefix/suffix is preserved). Since the input starts with a delimiter, we get an initial empty token that is trimmed. Since New Line is considered as whitespace, we get an empty middle token that is trimmed. Since the input ends with a delimiter and a New Line, we get a trailing empty token that is trimmed.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | ON | Sort | None |
| Trim Empty Tokens | ON | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
: A : B :
: C : D :
```

### Result

```
A;B;C;D
```

## Trim Input Whitespace and Empty Tokens with Input New Line as Delimiter

Trim whitespace and Empty Tokens while recognizing New Line as an input delimiter. Since the input starts with a delimiter, we get an initial empty token that is trimmed. Since New Line is considered as whitespace, we get an empty middle token that is trimmed. Since the input ends with a delimiter and a New Line, we get a trailing empty token that is trimmed.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | ON | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | ON | Sort | None |
| Trim Empty Tokens | ON | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
: A : B

C : D :
```

### Result

```
A;B;C;D
```

## Trim Input Whitespace and Empty Tokens with Input New Line as Delimiter Preserved in Output

Trim whitespace and Empty Tokens while recognizing New Line as a delimiter and preserving it in the Output. Since the input starts with whitespace and a delimiter, we get an initial empty token that is trimmed. Since New Line is considered as whitespace, we get an empty middle token that is trimmed. Since the input ends with a delimiter and a New Line that is trimmed, we get a trailing empty token that is trimmed. Since we preserve New Line in Output, we get the first two tokens (A and B) on the first line and the last two tokens (C and D) on the second line.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | ON | Preserve New Line | ON |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | None |
| Trim Whitespace | ON | Sort | None |
| Trim Empty Tokens | ON | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
: A : B

C : D :
```

### Result

```
A;B
C;D
```

## Trim Input Whitespace with Prefix and Suffix

Whitespace is trimmed before trimming prefix/suffix - whitespace inside prefix/suffix is preserved.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | ( | Suffix | None |
| Trim Suffix | ) | Quote Text Value | None |
| Trim Whitespace | ON | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

### Input

```
1,000.00: ( 1k ) : one thousand
```

### Result

```
1,000.00; 1k ;one thousand
```

## Wrap Output Lines

Wrap lines at specified length, including any prefix, suffix or quoting symbols. Wrapping does not observe any tokens or words, it breaks here at specified length.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | 8 |
| Detect Numbers | OFF | Continuation | None |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
Token1:Token2:Token3
```

**Result**

```
"Token1"
;"Token2
";"Token
3"
```

## Wrap Output Lines with Continuation Symbol

Wrap a multiline output text with a symbol terminating each line that continues on the next line. Regular line breaks (not resulting from wrapping) are not terminated with the wrapping symbol.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | : | Delimiter | ; |
| Include New Line | ON | Preserve New Line | ON |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | OFF | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | 8 |
| Detect Numbers | OFF | Continuation | \ |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
Token1:Token2
Token3:Token4
```

**Result**

```
"Token1"\
;"Token2\
"
"Token3"\
;"Token4\
"
```

## Use Cases

A few "real world" use cases of how to use the Morph function to transform input data into the desired output format.

## Morph IN Clause Data to CSV Format

I have an IN clause with text values that I want to use as unquoted data in a CSV file. I need to trim whitespace and prefix/suffix before transforming the input delimiter to my desired output delimiter.

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | , | Delimiter | ; |
| Include New Line | OFF | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | ' | Suffix | None |
| Trim Suffix | ' | Quote Text Value | None |
| Trim Whitespace | ON | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
'BETTE', 'CHRISTIAN', 'GRACE', 'JENNIFER', 'JOE', 'JOHNNY', 'KARL', 'MATTHEW',
'UMA', 'ZERO'
```

**Result**

```
BETTE;CHRISTIAN;GRACE;JENNIFER;JOE;JOHNNY;KARL;MATTHEW;UMA;ZERO
```

## Morph Table Data into CSV Format

I copied a table from a web page into SQL Commander and want to save it as a file that can be used for import. I want the data in CSV format, lines sorted, text values quoted and any empty lines suppressed.

Input is tab separated, not quoted, in random order and a mix of strings and numbers. Output is not sorted, separated by semi-colon and all text values are quoted.

Note: it is not possible to sort the output lines. The morph operation is about tokens, not lines.

### Settings

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | Space | Delimiter | ; |
| Include New Line | ON | Preserve New Line | ON |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | " |
| Trim Whitespace | ON | Sort | None |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | ON | Continuation | (N/A) |
| Grouping | , | preview setting | value |
| Decimal | . | Preview Token Count | OFF |

### Input

```
1  PENELOPE GUINESS
6  BETTE NICHOLSON
3  ED CHASE
7  GRACE MOSTEL
2  NICK  WAHLBERG
4  JENNIFER DAVIS
5  JOHNNY   LOLLOBRIGIDA
8  MATTHEW  JOHANSSON
12 KARL  BERRY
9  JOE   SWANK
11 ZERO  CAGE
10 CHRISTIAN   GABLE
13 UMA   WOOD
```

### Result

```
1;"PENELOPE";"GUINESS"
6;"BETTE";"NICHOLSON"
3;"ED";"CHASE"
7;"GRACE";"MOSTEL"
2;"NICK";"WAHLBERG"
4;"JENNIFER";"DAVIS"
5;"JOHNNY";"LOLLOBRIGIDA"
8;"MATTHEW";"JOHANSSON"
12;"KARL";"BERRY"
9;"JOE";"SWANK"
11;"ZERO";"CAGE"
10;"CHRISTIAN";"GABLE"
13;"UMA";"WOOD"
```

# Morph Table Data to IN Clause Format

I copied a column from Excel into SQL Commander and want to convert it to a string that I can use as an IN clause in a SQL Query.

I want the data comma separated with a space (, ), all values quoted, whitespaces trimmed and any empty lines suppressed. Since there is no predefined delimiter that includes a space after the comma, I type this delimiter directly in the combo box 1.

Input is tab separated, not quoted, in random order and a mix of strings and numbers. Output is sorted, separated by semi-colon and all text values are quoted.

1 it is hard to see the trailing space in the settings below

**Settings**

| input setting | value | output setting | value |
|---|---|---|---|
| Delimiter | UNIX/Linux/macOS - LF | Delimiter | , |
| Include New Line | ON | Preserve New Line | OFF |
| Detect Delimiter | OFF | Prefix | None |
| Trim Prefix | None | Suffix | None |
| Trim Suffix | None | Quote Text Value | ' |
| Trim Whitespace | ON | Sort | ASC |
| Trim Empty Tokens | OFF | Wrap Lines | OFF |
| Detect Numbers | OFF | Continuation | (N/A) |
| Grouping | (N/A) | preview setting | value |
| Decimal | (N/A) | Preview Token Count | OFF |

**Input**

```
JENNIFER

  JOHNNY
BETTE
   GRACE
MATTHEW
      JOE
CHRISTIAN
ZERO
KARL
UMA
```

**Result**

```
'BETTE', 'CHRISTIAN', 'GRACE', 'JENNIFER', 'JOE', 'JOHNNY', 'KARL', 'MATTHEW',
'UMA', 'ZERO'
```

# Using Editor Templates

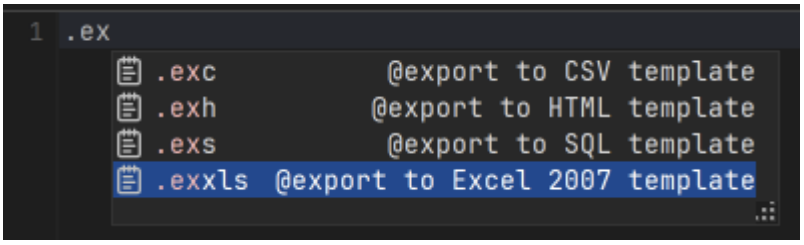
Editor Templates can be used to easily insert text that you often use, such as code snippets, current date and time, or anything you like.

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

## Using a Template

A template is an **Abbreviation** that can easily be replaced by its **Expanded Text**, and it may optionally have a **Description**.

To expand a template use the Tab or Ctrl+Shift-T key binding. This displays a list of matching templates that you can pick from, along with the description or a part of the corresponding expanded text for each.




Select a template by pressing the TAB or ENTER key and it will be expanded in the editor.




You can also enable **Instant Substitution** to immediately expand the text if you have typed enough of the abbreviation to match a single template when you press the TAB key:

1. Open **Tools → Tool Properties**,
2. Select the **General/SQL Commander/Editor Templates** category,
3. Click the **Instant Substitution** checkbox in the toolbar,
4. Click the **Apply** or **OK** button to save the updated setting.

You can use the **Show Editor Templates** entry in the main Edit menu or the editor right-click menu.

## Creating a new Template

DbVisualizer comes with some default templates, but you can create additional templates:

1. Open **Tools → Tool Properties**,
2. Select the **General/SQL Commander/Editor Templates** category,
3. Click the **Insert** button in the toolbar to add a new template, and enter an Abbreviation, Expanded Text, and optionally a Description,
4. Check off the checkbox in the **Format** column if you want the Expanded Text to be formatted by the SQL Formatter when it is inserted in the editor,
5. Click the **Apply** or **OK** button to save the new templates.

If you want to use an existing template as a starting point, you can select it and then click the **Duplicate** button instead of the Insert button. The Edit in Window button opens the selected cell in a separate window where it is easier to work with larger templates.

> Always use a special character, such as dot or hashmark, as the first character in the Abbreviation. The default templates use a dot but you can pick any special character you like. If you use a regular character as the first character, unexpected matches may be found if one abbreviation starts with the same characters as another abbreviation ends with.

## Editing or Deleting a Template

You can edit any piece of a template or delete templates you no longer need:

1. Open **Tools → Tool Properties**,
2. Select the **General/SQL Commander/Editor Templates** category,
3. To edit, double-click any cell you want to change and edit its value,
4. To delete a template, select any of its cells and click the Delete button,
5. Click the **Apply** or **OK** button to save the changes.

## Changing the Expand Keybinding

The keyboard shortcut used to expand a template, or bring up the list of matching templates if more than one, is the TAB key. You can add other shortcuts or change the default, as described in Changing Keyboard Shortcuts. The shortcut is named **Show Editor Templates** and you find it in the **Main Menu/Edit** category.

# Executing SQL Statements

In the SQL Commander, you can execute one or multiple statements. You can also control if the execution should stop or continue when the execution of a statement results in a warning or error.

## Execute a Script with Multiple Statements

Use the **SQL Commander → Execute** main menu operation to execute the SQL in the SQL Commander editor. The SQL Commander executes the statements one by one and indicates the progress in the log area. The currently selected Database Connection is used for all statements. The SQL Commander does not support executing SQLs for multiple database connections in one batch.

DbVisualizer uses the delimiters specified in the Tool Properties dialog, in the **General / SQL Commander / Statement Delimiters** category under the General tab, to separate one statement from the next. Usually semicolon ";" following the actual statement or "go" which should be the only command on a new line directly after the statement (then with no semicolon after) that should be executed. The "go" command supports setting the number of times the statement should be executed. For example "**go 5**" will then execute the statement 5 times.

The result of the execution is displayed in the results area based on the type of results result(s) that are returned. If there are several results and an error occurred in one of them, the Log tab is automatically displayed to indicate the error.

If you select a statement in the SQL editor and choose **SQL Commander → Execute** main menu option, only the selected statement is executed. This is a useful feature when you have several SQL statements in the SQL editor and you just want to execute one or a few of the statements.

> Comments in the SQL editor are sent to the database when you use **SQL Commander → Execute**, unless you have enabled **Strip Comments when Executing** in the **SQL Commander → SQL Commander Options** menu.

Note the option **Change Default Values** in the **SQL Commander → SQL Commander Options.** This offers a shortcut to navigate to the Connection properties section for the options. Setting the options in the Connection properties will result in that these values will be the default for SQL Commanders opened for this connection.

## Execute Only the Current Statement

The **SQL Commander → Execute Current** operation is useful when you have a script with several SQL statements. It lets you execute the statement at the cursor position without first having to select the SQL statement. The default key binding for execute current is **Ctrl-PERIOD** (Ctrl-.).

> **Execute Current** determines the actual statement by parsing the editor buffer using the standard statement delimiters.The current statement is the statement containing the caret or that ends on the line with the caret. This means that the caret may be after the statement delimiter as long as there is no other statement on the same line.

If you are unsure what the boundaries are for the current statement then use **Edit → Select Current Statement**. This will highlight the current statement without executing it.

## Execute Buffer

The **SQL Commander → Execute Buffer** sends all of the content in the SQL editor to the database in a single run. This is useful when executing complex SQL statements such as CREATE STORE PROCEDURE (or similar) where statement splitting on semicolon must not be done as with SQL Commander → Execute and SQL Commander → Execute Current.

## Control Execution after a Warning or an Error

You can control whether subsequent statements should be executed when a statement results in an error, a warning or returns or affects no rows.

Open **Tools → Tool Properties** and select the **SQL Commander** category under the **General** tab. There you find **Stop on Error, Stop on SQL Warning**, and **Stop on No Rows** check boxes for enabling these features in all **SQL Commander** tabs.

Alternatively, you can use DbVisualizer client side commands to enable or disable these features in a script.

```
@stop on error;
@stop on sqlwarning;
@stop on norows;

@continue on error;
@continue on sqlwarning;
@continue on norows;
```

## Stopping Execution

Apart from clicking the **Stop the current script execution button** the client side command " `@stop now"` may be used to stop the script execution.

## Handling Connection loss

If DbVisualizer detects that the database connection has been lost, during the execution of a script, DbVisualizer can automatically restore the lost connection. To enable this change the settings in the **Handling loss of Connection** section in the **Connection Properties** tab and the **SQL Commander** category

- **No Reconnect**: Do not automatically reconnect
- **Reconnect**: Try to automatically restore the connection. If the connection was successfully restored the script execution is continued. Note that automatically restoring a connection is only possible if the connection is in **auto-commit** mode.
- **Reconnect and re-execute**: As **Reconnect** and also try to re-execute the command that encountered the loss of connection.

> **re-executing** commands is not enabled for all types of commands. E.g for client side commands such as @commit it is not enabled.
>
> Also note that, when a SQL statement is re-executed, there will be two SQL log entries for the command. A first entry showing the failure encountered and a second entry showing the result of the re-execution. If the re-execution is successful the result of the total script execution is considered OK.

# Re-Executing SQL Statements

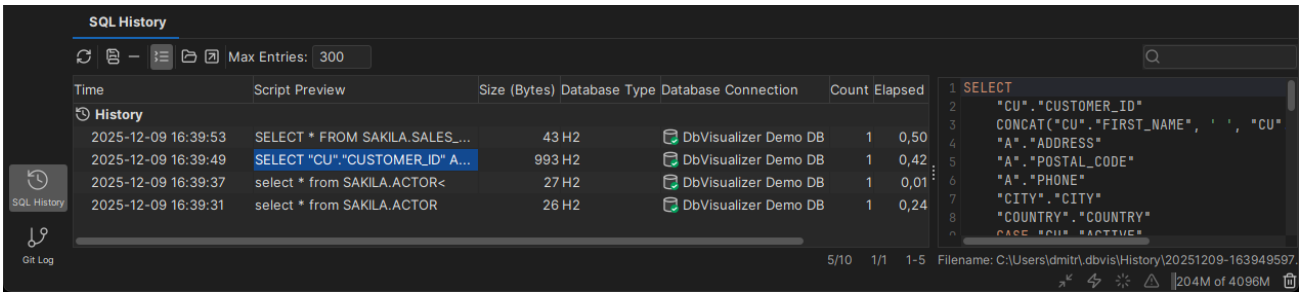
As you execute SQL statements in SQL Commander, DbVisualizer saves them as history entries along with information about the connection that was used and the execution result. This makes it easy to locate statements and scripts you have executed in the past.

## Using SQL Commander actions

If you just want to go back and forth between statements you have executed recently, you can use the **Load Previous History Entry** and **Load Next History Entry** actions in the toolbar on in the context menu in SQL Commander.

## Using SQL History

To look through all saved statements, you can display all history entries by clicking the **SQL History** button in the main window's sidebar.




The entries are ordered with the most recently executed entries at the top by default, but you can reorder them by clicking on the column headers. The complete content of the selected entry is shown below the list, unless you disable it by clicking the Show Details toolbar button.

The columns show when the entry was executed, a part of the script/statement, the size of the complete statement/script, and then the database type and connection it was executed for, and how long it took to execute. The **Success** and **Errors** columns show how many of the statements in a script were executed successfully or that caused an error. Finally, the **Rows** column show the number of rows retrieved or affected by the script.

You can use the filter field in the top right corner to search for entries matching a criteria. Clicking on the magnifying glass reveals a configuration menu where you can, among other things, specify which columns to search in and if you want to search through the complete script rather than just the part of the script shown in the Script/SQL Statement column.

In the Tool Properties dialog, in the **SQL History** category under the General tab, you can specify that sequential executions of the same SQL statement/script should be collected into a single history entry. When this feature is

enabled, the **Count** field number is increased for each execution. In the same Tool Properties category you can also specify rules for when not to create a history entry.

### Reusing a history entry

When you have found the entry you're looking for, you can open it in an SQL Commander by double-clicking it or clicking the corresponding toolbar button.

You can also add the content of an entry to the current content of an SQL Editor. Select the entry in the list, drag it with the mouse key depressed to the position in the editor where you want to add it, and drop it there by releasing the mouse button.

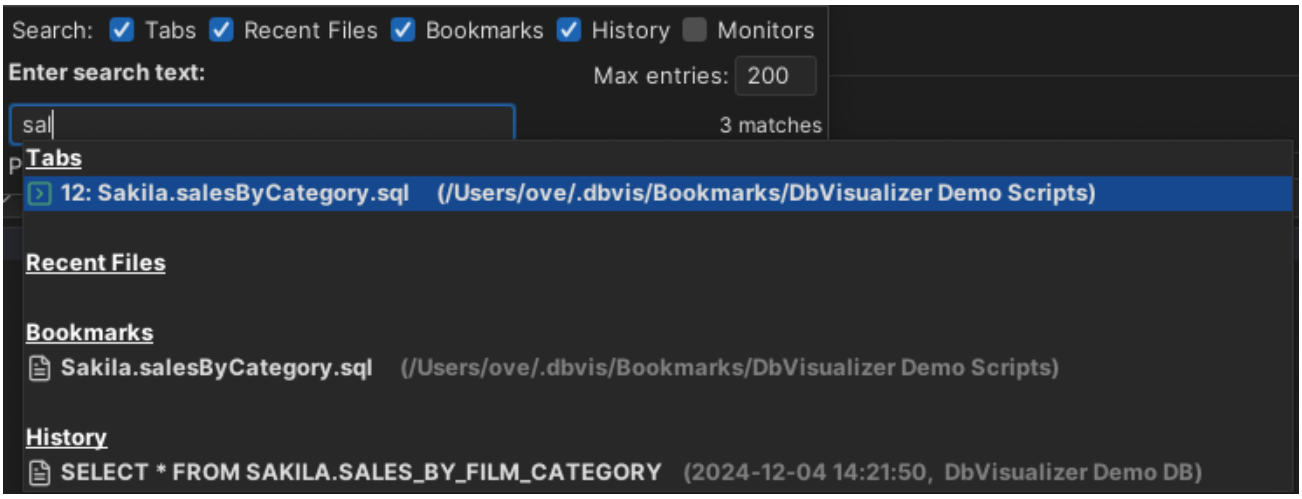
### Saving a history entry as a bookmark or file

If you realise that you need easy access to a History entry in the future, you can save it is as a Bookmark or other file. Just select the entry and use the **Save As** operation, or just drag it to the **Bookmarks** tree and drop it.

You can also locate the file holding the history entry in the file system using the **Open Enclosing Directory** right-click menu entry or toolbar button. This opens a file chooser for the directory holding the file.

## Using Quick Load

An alternative to locating editor tabs, Recent Files, Bookmarks or Monitors from the Files tab and History entries from the History window is to use the **Quick Load** feature, by default bound to the **Ctrl+Alt+O** key combination. It is also available via a main toolbar button as well as in the **File → Quick File Open** menu.




The Quick Load feature locates files with partly matching names from the categories you have selected, as you type. You can use an asterisk ("*") as a wildcard in the search string. Files already open in editor tabs is not displayed in Recent Files.

When you see the file you're looking for, just select it and press **Enter** to load it into an SQL Commander editor. If the file is already loaded in an editor, that tab is made visible instead.

The number of matches is shown on the search panel as seen on the figure above. It the number of matches is big, and you want to change the size, you can press **Escape** and change the number in the **Max entries** field that is shown. Go then to the **Enter search text** field and press arrow down. The result list shown will be limited to your set number for each category.

# Executing Complex Statements

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

If you need to execute a complex statement that itself contains other statements in the SQL Commander, such as a CREATE PROCEDURE statement, you may need to help DbVisualizer figure out where the complex statement starts and ends. The reason is that DbVisualizer needs to send statements to the database for execution one by one.

> To create or edit single procedures, functions, triggers, and similar types of complex objects, we recommend that you use the Create Procedure function and Procedure Editor to work with these.

The following explains the options that are available to run complex statements in the SQL Commander:

## Using Execute Buffer

The **SQL → Execute Buffer** operation sends the complete editor buffer for execution as one statement. No comments are removed and no parsing of individual statements based on any delimiters is made. You can use this feature if the complex statement is the only statement in the SQL Commander editor.

## Using an SQL Dialect

DbVisualizer will understand the syntax for complex statements for most of the officially supported databases. Open **Tools → Tool Properties** and in the **General / SQL Commander / Statement Delimiters** category and check **Allow SQL Dialects** to enable this feature.

Here is an example of a complex statement for My SQL:

```
CREATE TRIGGER upd_check BEFORE UPDATE ON account
       FOR EACH ROW
       BEGIN
           IF NEW.amount < 0 THEN
               SET NEW.amount = 0;
           ELSEIF NEW.amount > 100 THEN
               SET NEW.amount = 100;
           END IF;
       END;
```

**Note:** Depending on the complexity of the dialect, the overhead of parsing and analyzing all statements before executing them may be significant. If you notice degraded performance when executing very large scripts we recommend that you disable the dialect feature and use some of the other methods to handle complex statements.

## Using an SQL Block

To tell DbVisualizer that a part of a script should be handled as a single statement, you can insert an SQL block begin identifier just before the block and an end identifier after the block. The delimiter must be the only text on the line. The default value for the **Begin Identifier** is `--/` and for the **End Identifier** it is `/`.

Here is an example of an SQL block for Oracle:

```
--/
script to disable foreign keys

declare cursor tabs is select table_name, constraint_name
  from user_constraints where constraint_type = 'R' and owner = user;

begin
  for j in tabs loop
    execute immediate ('alter table '||j.table_name||' disable constraint'||
j.constraint_name);
end loop;
end;
/
```

## Using the @delimiter Command

With the **@delimiter** command you can temporarily change the statement delimiter DbVisualizer uses to separate the statements and send them one by one to the database. Use it before the complex statement, and after the statement if the script contains additional statements. Here's an example:

```
@delimiter ++;
CREATE OR REPLACE FUNCTION HELLO (p1 IN VARCHAR2) RETURN VARCHAR2
AS
BEGIN
 RETURN 'Hello ' || p1;
END;
++
@delimiter ;++
@call ${returnValue||(null)||String||noshow dir=out}$ = HELLO('World');
@echo returnValue = ${returnValue}$;
```

The first @delimiter command sets the delimiter to `++` so that the default `;` delimiter can be used within the function body in the CREATE statement. The `++` delimiter is then used to end the CREATE statement, and another **@delimiter** command sets the delimiter back to `;` for the remaining commands in the script.

## Calling a Function or Procedure

As a general rule, DbVisualizer takes no part in how you execute the query; the SQL code is simply sent to the server for interpretation and execution. You should be able to write and execute the SQL code just as you do in any other tool, subject to the server's ability to understand it.

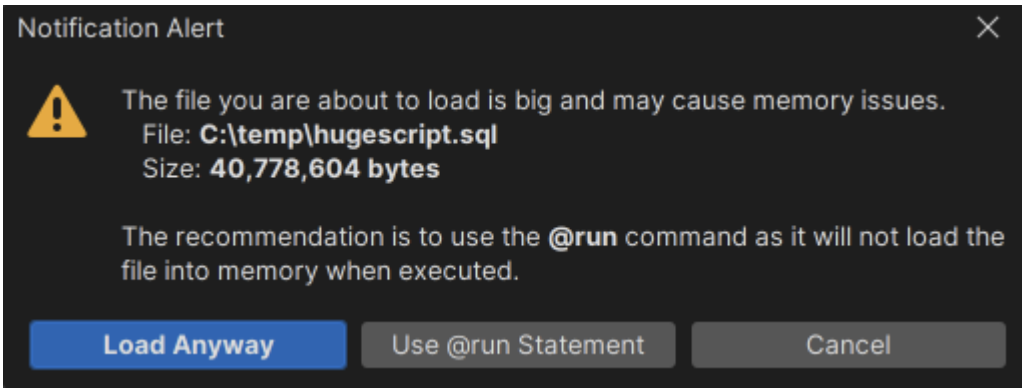
However, the command for executing a function or procedure varies among vendors. For instance, **EXECUTE** runs a procedure in Microsoft Transact-SQL and Oracle SQL*Plus, whereas **CALL** does the same thing in PostgreSQL and MySQL.

**@call** is the way to do it in DbVisualizer. This is also transparent to the underlying database; you use the same command irrespective of the database you are connected to (see Executing a Code Object and Using Client-Side Commands for more details).

# Executing an External Script

If you have a very large script to execute, you may not have enough memory available to be allocated for DbVisualizer to load it into an SQL Commander editor.

To save memory, you can use the **@run** command. If you try to load a very large file, DbVisualizer suggests using the @run command automatically:




The **@run** command executes a script file by only loading one statement at a time, minimizing the memory requirements. A related command is the **@cd** command for changing the current directory.

- **@run <file> [ <variables> ]** Request to execute the file specified as parameter, optionally with a list of variables
- **@cd <directory>** Change the working directory for the following **@run** command. The directory is reset after every execution, so the **@run** command must be run in the same execution.

Here's an example of a script using these commands:

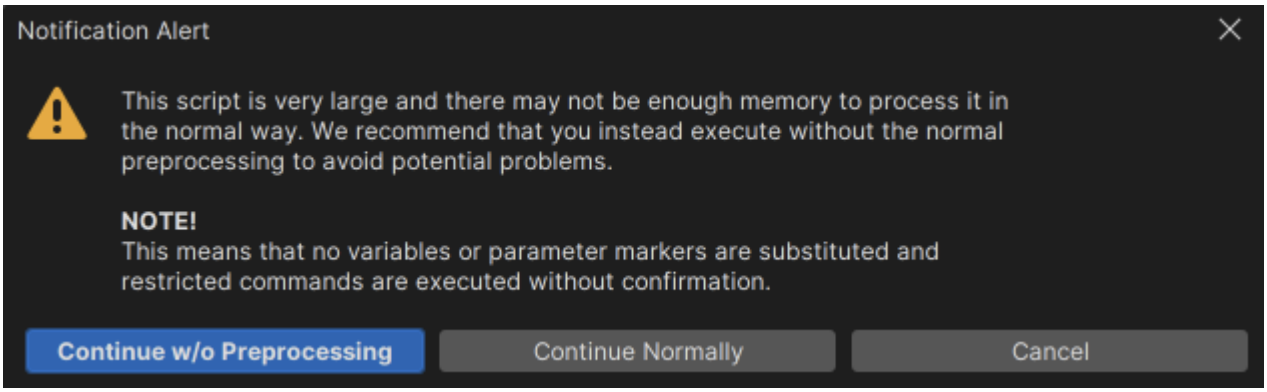
```
@run createDB.sql;      -- Execute the content in the
                        -- createDB.sql file without loading into the SQL editor.
                        -- The default location is the same as the working
directory
                        -- for DbVisualizer (when not using an absolute path).
                        -- Without context, it will be the program installation
directory.
                        -- If a file is opened in a commander tab, it'll be the
file's parent directory.
@cd /home/mupp;         -- Request to change directory to /home/mupp
@cd myscripts;          -- Request to change directory relative to current, i.e.
to /home/mupp/myscripts
@run loadBackup.sql;    -- Execute the content in the loadBackup.sql
                        -- file relative to current directory. This file will now
be read from the
                        -- /home/mupp/myscripts directory.

-- Note: this must run in the same execution, otherwise the directory will be
reset.
```

You can also include DbVisualizer variables as parameters to the **@run** command, with values to be used for the corresponding variables in the script:

```
@run monthlyReport ${month||2021-01-14||Date||noshow}$ ${dept||HR||String||noshow}
$
```

Even though the **@run** command reads one statement at a time from the file, there are other parts of the execution process that require the whole file to be read before the statements can be executed: parsing the script for variables, parameter markers, and restricted commands, as well as counting all statements in order to provide progress information. When you run a script that is large enough (more than 10 MB) for these things to potentially cause memory problems and slow down the processing, DbVisualizer gives you a chance to turn off this preprocessing and progress reporting so that the statements instead can be executed directly as the are read from the file, one at a time.



To ensure that you don't have any problems running scripts this large we strongly recommend that you click **Continue w/o Preprocessing**, thereby disabling all variable, parameter and restricted commands processing. Only click **Continue Normally** if you know for sure that you have enough memory available and have adjusted your installation so that DbVisualizer can use it. With the preprocessing disabled, you should be able to execute scripts of any size (we have tested with scripts as large as 4 GB).

Another alternative for execution of large scripts is to use the DbVisualizer command line interface instead of the GUI application. This option is the most efficient and fastest.

Running without preprocessing is always more efficient, so if your script does not use any variables or parameter markers and you do not use the Permissions feature, you can disable preprocessing even for scripts smaller than 10 MB by unchecking the **SQL Commander Options → Preprocess Script** checkbox in the SQL Commander menu.

# Locating SQL Errors

If errors occur, the corresponding text is underlined with a red wavy line. Hovering the mouse over the error indication shows the corresponding error message. The right margin contains markers for each error as well, and clicking on a marker scrolls the editor to the corresponding error. Alternatively, you can click the **FAILED** link in the Log tab grid to move the caret to the error location.

If you prefer to navigate between errors using the keyboard, you can use the defined key bindings for the **Insertion Point to Next Marker** and **Insertion Point to Previous Marker** actions in the Tool Properties dialog, in the **Key Bindings** category, in the Editor Commands group. Alternative is to use the **Goto Next Failed** and **Goto Previous Failed** in the right click menu of Log tab grid, and then click the link for the FAILED entries.

> Error location information is not available for some databases. In that case the complete statement is underlined in the editor.

## Disable Error Markers in the SQL Editor

It is possible to disable error markers in the **SQL Commander → SQL Commander Options** menus. With **Show Error Position Markers** turned on it shows markers only for databases that specifically report positions of errors in a statement. With **Show Error Statement Markers** turned on the complete statement is highlighted if it fails during execution, but only if either the **Show Error Position Markers** is turned off or if the driver/database doesn't report positions of errors.

If you re looking to minimize the amount of error markers when executing scripts, turn off **Show Error Statement Markers** and keep the **Show Error Position Markers** checked (turned on). DbVisualizer will then report errors reported by the JDBC driver only.

To set default values for these settings visit **Tools → Tool Properties** and the **General / SQL Commander** category.

# Analyzing (explain) Query Performance

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

You can analyze how a query is executed by the database, e.g. whether indexes are used or if the database has to do an expensive full scan. To analyze a query:

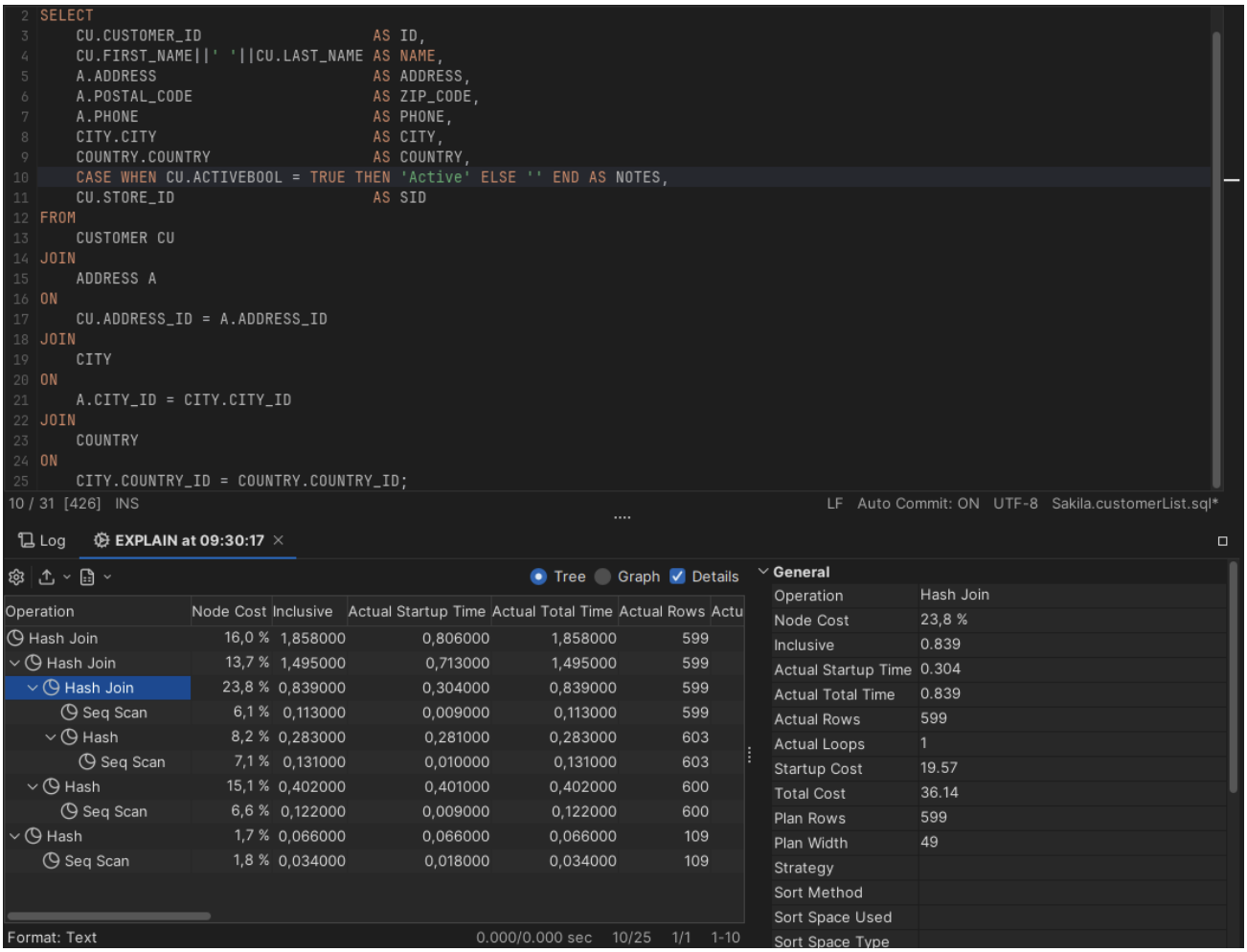
1. Enter the query in the **SQL Commander** editor,
2. Click **Execute Explain Plan** button in the toolbar,
3. Look at the result in the results area.

Please note that the available presentation options vary per supported database.

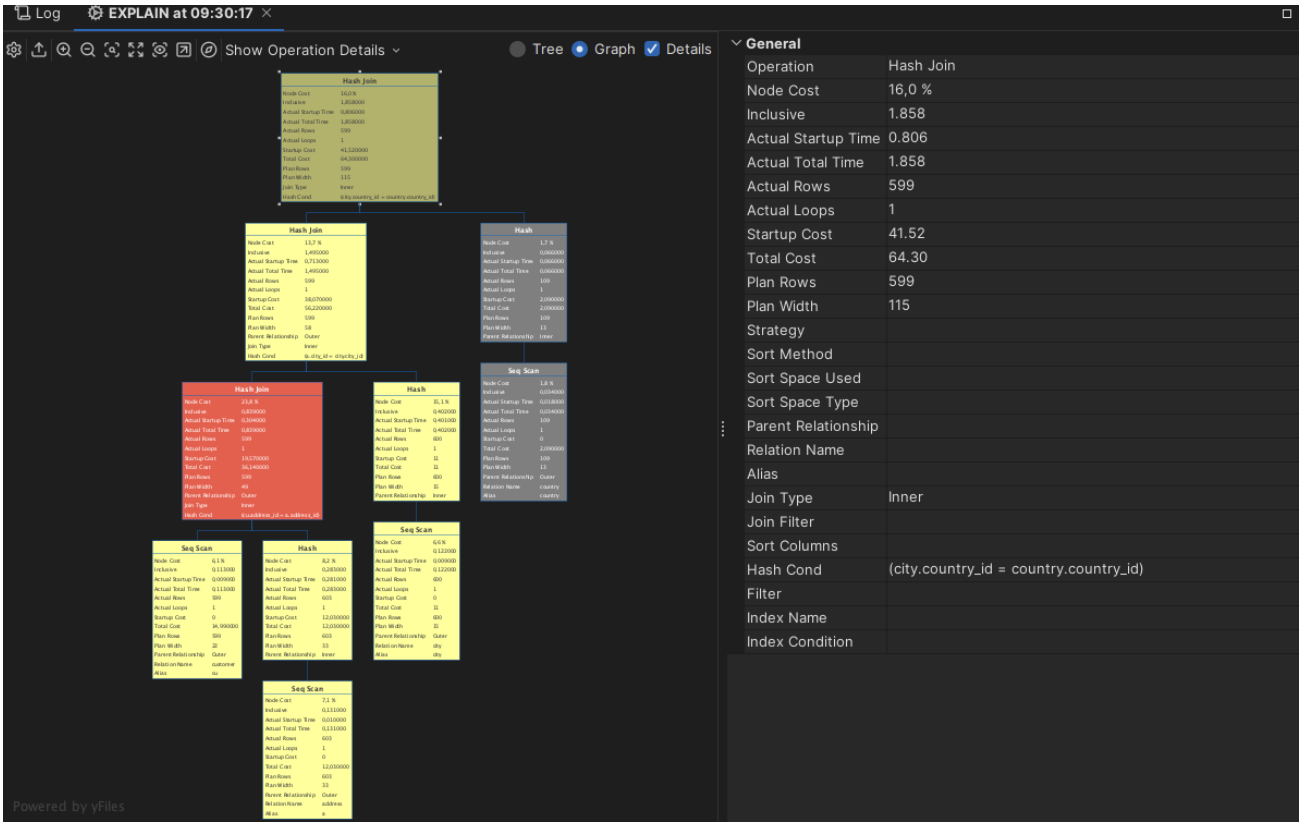
Explain Plan executes your query and records the plan that the database devises to execute it. By examining this plan, you can find out if the database is picking the right indexes and joining your tables in the most efficient manner. The explain plan feature works much the same as executing SQLs to present result sets; you may highlight statements, run a script or load from file. The explain plan results can easily be compared by pinning the tabs for different runs.

DbVisualizer presents the plan either in a tree style format or in a graph, or in a simple text format. The information depends on the database you use. In the tree view, put the mouse pointer on the column header for a tooltip description what that column represents.

The following screenshot shows the SQL in the editor at top and the resulting explain plan as a tree view. The relative cost is indicated using colored adornments on each node and you can select a node to see the details (if **Show Details** is checked).




The **Graph View** shows the plan as a graph. You can zoom in or out, choose detail levels, export it to an image file or print it using the toolbar buttons. The relative cost is indicated by node color, and you can select a node to see the details (if **Show Details** is checked).




The databases use different techniques to manage their explain plan support. You can make database-specific configurations in the **Properties** tab for a connection, in the **Explain Plan** category.

# Auto Commit, Commit and Rollback

With Auto Commit enabled, all changes you make to the database data is automatically committed after the successful execution of an SQL statement. Auto Commit is enabled for a connection by default. You can change the default in the **Options** area of the **Object View** tab for the connection. Note that this change can only be done when the connection is disconnected.

You can toggle the Auto Commit setting for an open SQL Commander tab using the **SQL Commander** main menu item of the corresponding button in the SQL Commander toolbar. The setting is also available both on the right-click menu under the **Transaction** menu item.

> Not all databases supports transactions and thus always commits executions

Alternatively, you can use this command in a script to set it:

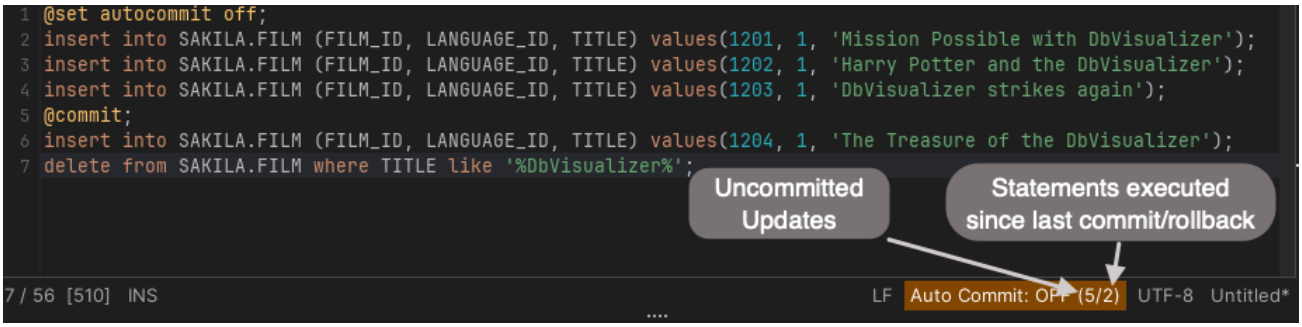
```
@set autocommit on/off;
```

If Auto Commit is disabled, it is very important to manually issue the commit or rollback operations when appropriate. Use the **Commit** and **Rollback** buttons in the SQL Commander toolbar or the corresponding operations in the **SQL Commander** main menu to commit and rollback transactions.

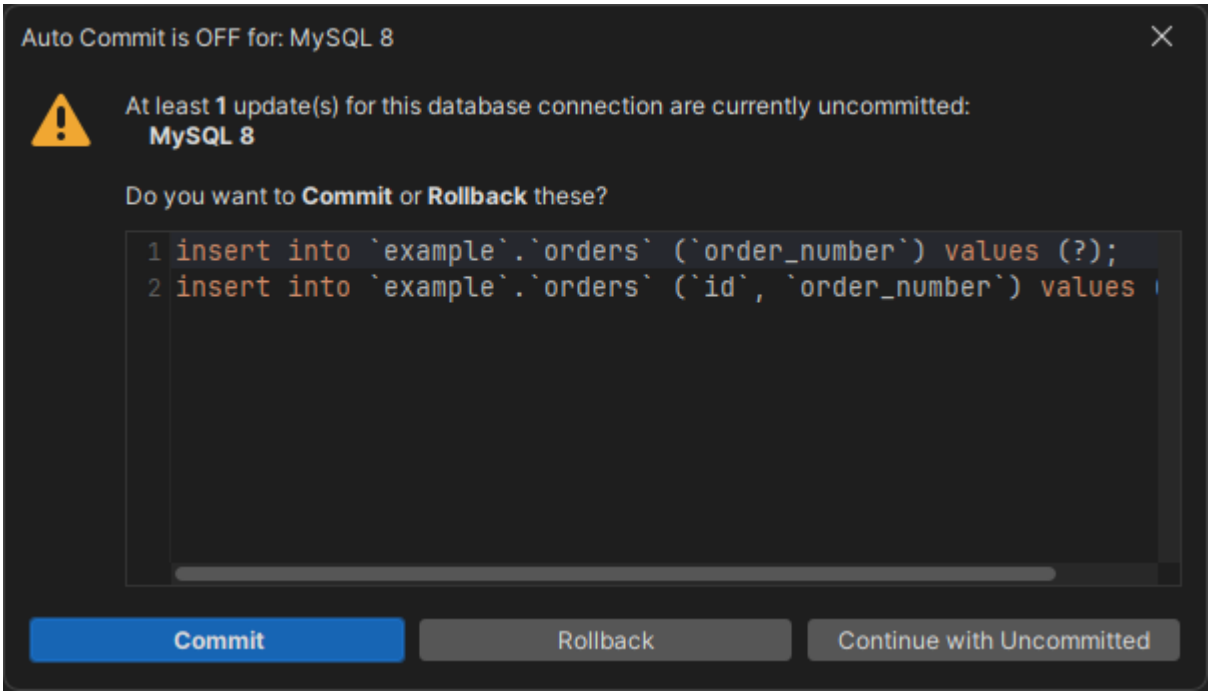
Alternatively, you can use the following commands in a script executed in the SQL Commander:

```
@commit;
@rollback;
```

There is an **Auto-Commit: ON/OFF** indicator in the editor status bar; the first number shows the number of records updated in the database since the last commit/rollback, the second shows the number of statements (except SELECTs) executed since last commit/rollback.



When performing certain actions like disconnecting, or closing a tab with a connection that has uncommitted records, a prompt will appear and ask what to do with those records. When disconnecting, the choice to **Continue with Uncommitted** is replaced with **Cancel**.




Having Auto Commit off for a connection should be handled with great care since transactions may lock parts of the database (this is database dependent). To minimize the risk of forgetting uncommitted transactions, there is an **Ask when Auto Commit is OFF** settings in the connection **Properties** tab, in the **Transactions** category, that can be set to warn you when there are changes that hasn't been committed. You can set it to **Always** or **When Uncommitted Updates**. When set to **When Uncommitted Updates**, you are warned when there is at least one updated record reported by the database. For database that do not accurately report updated records, you can set it to **Always** to be warned if at least one statement (other than SELECT) has been executed since the last commit or rollback.

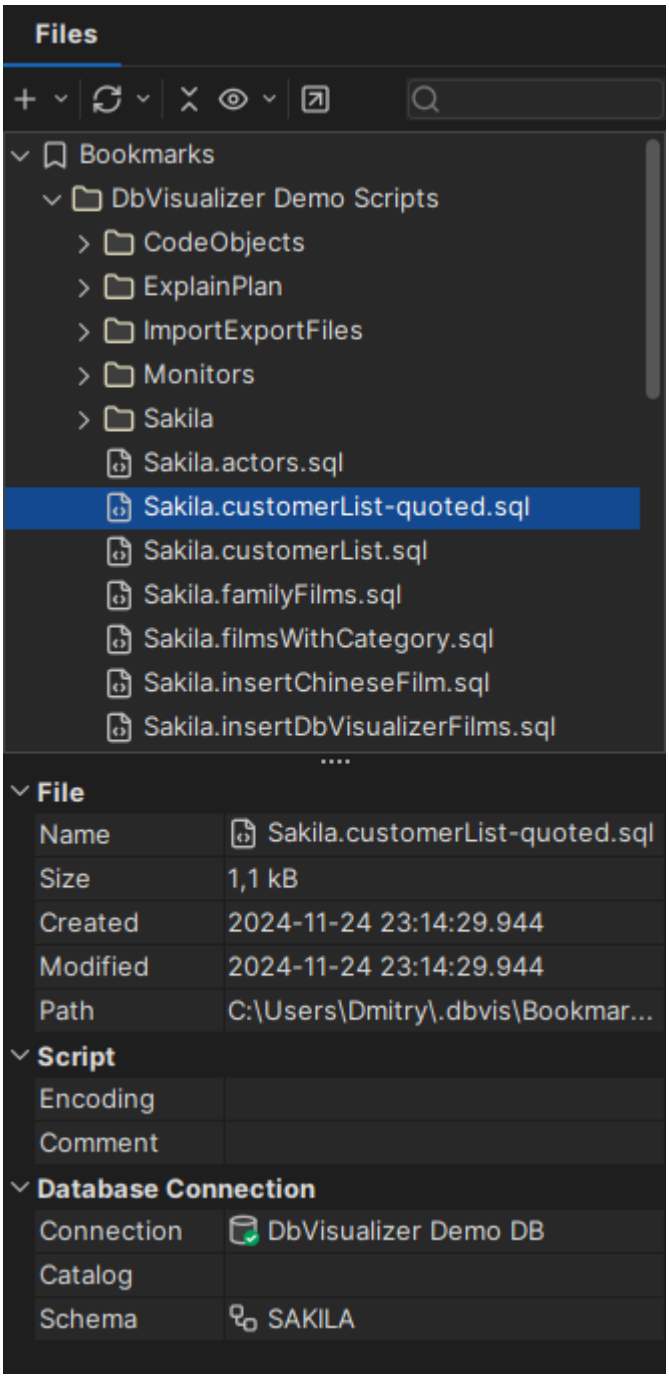
There is also a **Pending Transactions at Disconnect** setting in the Tool Properties dialog, in the Transaction category under the General tab. It specifies what DbVisualizer should do when you disconnect a connection that has pending changes, and you can set it to **Commit**, **Rollback** or **Ask**.

# Managing Frequently Used SQL

> This page describes the **Bookmarks** node in the Files tab. Starting with DbVisualizer 25.1, much of the functionality supported for Bookmarks applies to all mounted folders. For more details, see Using the Files Tab.

You may have a set of SQL statements that you use over and over to perform frequent tasks. You probably have them saved in script files that you can load into an SQL Commander, but DbVisualizer **Bookmarks** make it even easier to work with them. A Bookmark is a script visible in the Files tab in the navigation area.

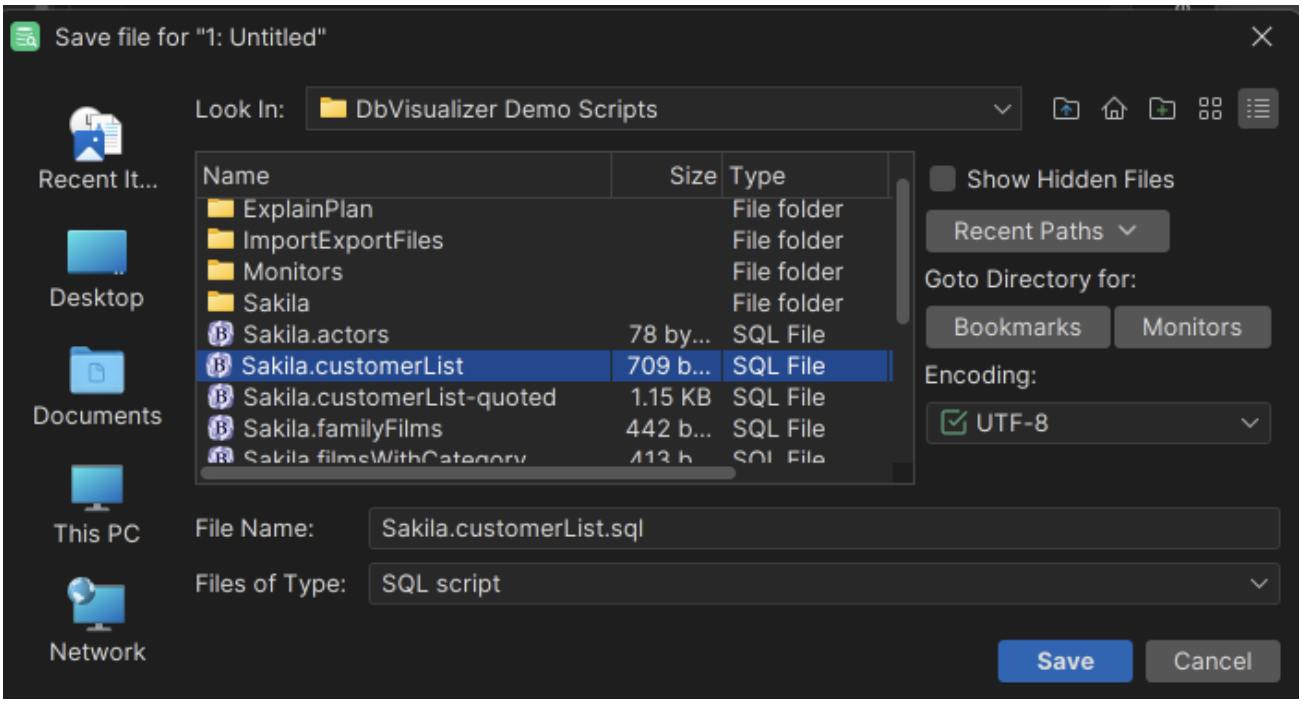
You find your Bookmarks under the Files tab in the navigation area to the left in the main DbVisualizer window. The content of a Bookmark is one or more SQL statements. It may also be associated with a Connection, a Catalog and a Schema, to be used when executing the statements. This information is displayed, and can be edited, in the lower part of the Files tab, along with information about the file that holds the Bookmark. If you don't want to see these details, you can disable it with the Show Details toggle control in the right-click menu for a node.




## Creating, Editing and Organizing Bookmarks

You can create a new Bookmark by selecting the **Bookmarks** node in the tree and clicking the **Create File** toolbar button. This adds a new node in the tree, with the default name selected so that you can replace it with the name you want to use. You can also rename the Bookmark later using the **Rename** right-click menu item.

A Bookmark can also be created from the current content in an SQL Editor. Click the **Save File As** toolbar button to open a file chooser dialog, and click the **Bookmarks** button in the file chooser dialog to go to the Bookmarks root directory. Enter a filename for the Bookmark and click **Save**.




To put some SQL statements in a new empty Bookmark or to edit the contents of an existing Bookmark, you need to open the Bookmark in an SQL Commander. Double-click the Bookmark node in the tree or click the corresponding toolbar button to open a new SQL Commander tab for the Bookmark, or activate the SQL Commander that already holds the Bookmark. When you are done with your edits, use the **Save** toolbar button in the SQL Editor to save them.

You can also add the content of a Bookmark to the current content of an SQL Commander editor. Select the Bookmark node, drag it with the mouse key depressed to the position in the editor where you want to add it, and drop it there by releasing the mouse button.

Folders can be used to organize your Bookmarks. Click the **Create Folder** toolbar button to create a new folder and give it the name you want. You can then drag an existing Bookmark node to the folder, and create new Bookmarks and subfolders in the folder by selecting it and clicking the **Create File** and **Create Folder** buttons.

> The folders and the Bookmarks within a folder are ordered alphabetically and cannot be changed manually.

## Executing Bookmarks

With a Bookmark opened in an SQL Commander tab, you can of course execute its statements by clicking the **Execute** toolbar buttons as usual, but you can also open and execute a Bookmark directly by selecting it in the tree and using the **Open in SQL Commander and Execute** operations in the right-click menu.

## Adding a Bookmark as a Favorite

If you are using a Bookmark very often, you may find it more convenient to add it to the Favorites area. You can drag and drop a Bookmark from the Files tab to the Favorites area, or via the **Add to Favorites** right-click menu operation.

## Sharing Bookmarks

It's easy to share your Bookmarks with someone else because they are stored as regular files. The files are located in a subfolder of the DbVisualizer user preferences folder named Bookmarks. The user preferences folder is typically a subfolder named .dbvis in your home folder.

The main Bookmark content is stored in a file with exactly the same name as the node in the Files tab. The additional data associated with the Bookmark is stored in a file with the same name plus the .met extension.

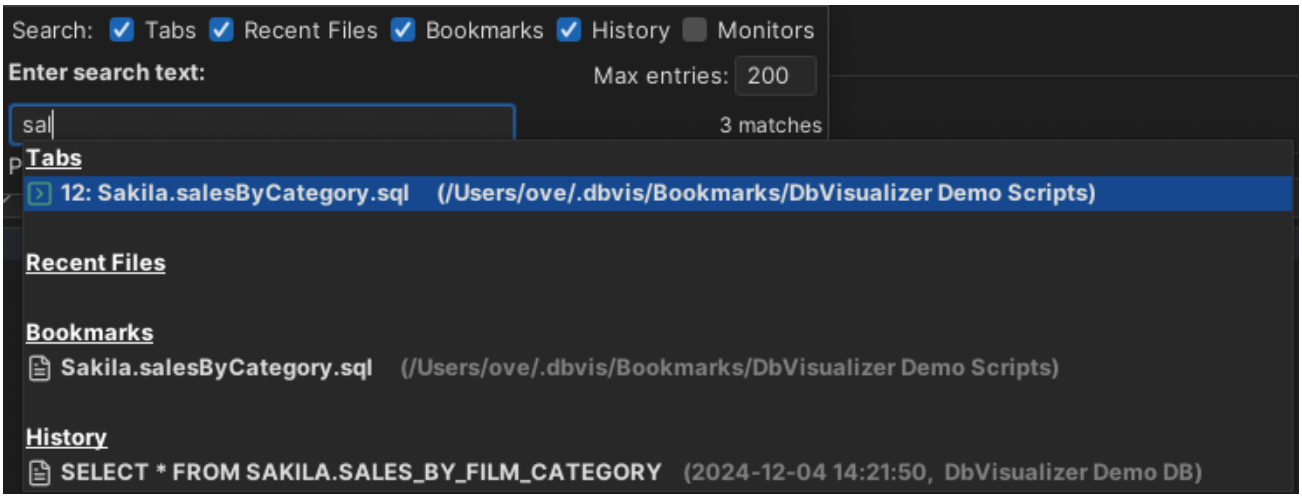
To share some of your Bookmarks with someone, we recommend that you use DbVisualizer to create a separate Bookmarks subfolder for the shared Bookmarks.

You can share your Bookmarks in two ways:

1. use an external tool to create a file archive (e.g. a ZIP file) of the subfolder and send it to your friend or colleague. He or she can then extract the files into their local Bookmarks folder.
2. create a **Linked Folder** that points at a network resource shared by your colleagues. **Notes:**
    a. DbVisualizer will try to detect if a file was externally modified, but file locking and access permissions are subject to the server capabilities and controlled by the system administrator of the server.
    b. In order to create linked folders on Windows, you may have to enable "Developer Mode" in the Control Panel

## Using Quick Load

An alternative to locating editor tabs, Recent Files, Bookmarks or Monitors from the Files tab and History entries from the History window is to use the **Quick Load** feature, by default bound to the **Ctrl+Alt+O** key combination. It is also available via a main toolbar button as well as in the **File → Quick File Open** menu.




The Quick Load feature locates files with partly matching names from the categories you have selected, as you type. You can use an asterisk ("*") as a wildcard in the search string. Files already open in editor tabs is not displayed in Recent Files.

When you see the file you're looking for, just select it and press **Enter** to load it into an SQL Commander editor. If the file is already loaded in an editor, that tab is made visible instead.

The number of matches is shown on the search panel as seen on the figure above. It the number of matches is big, and you want to change the size, you can press **Escape** and change the number in the **Max entries** field that is shown. Go then to the **Enter search text** field and press arrow down. The result list shown will be limited to your set number for each category.

# Creating Queries Graphically

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

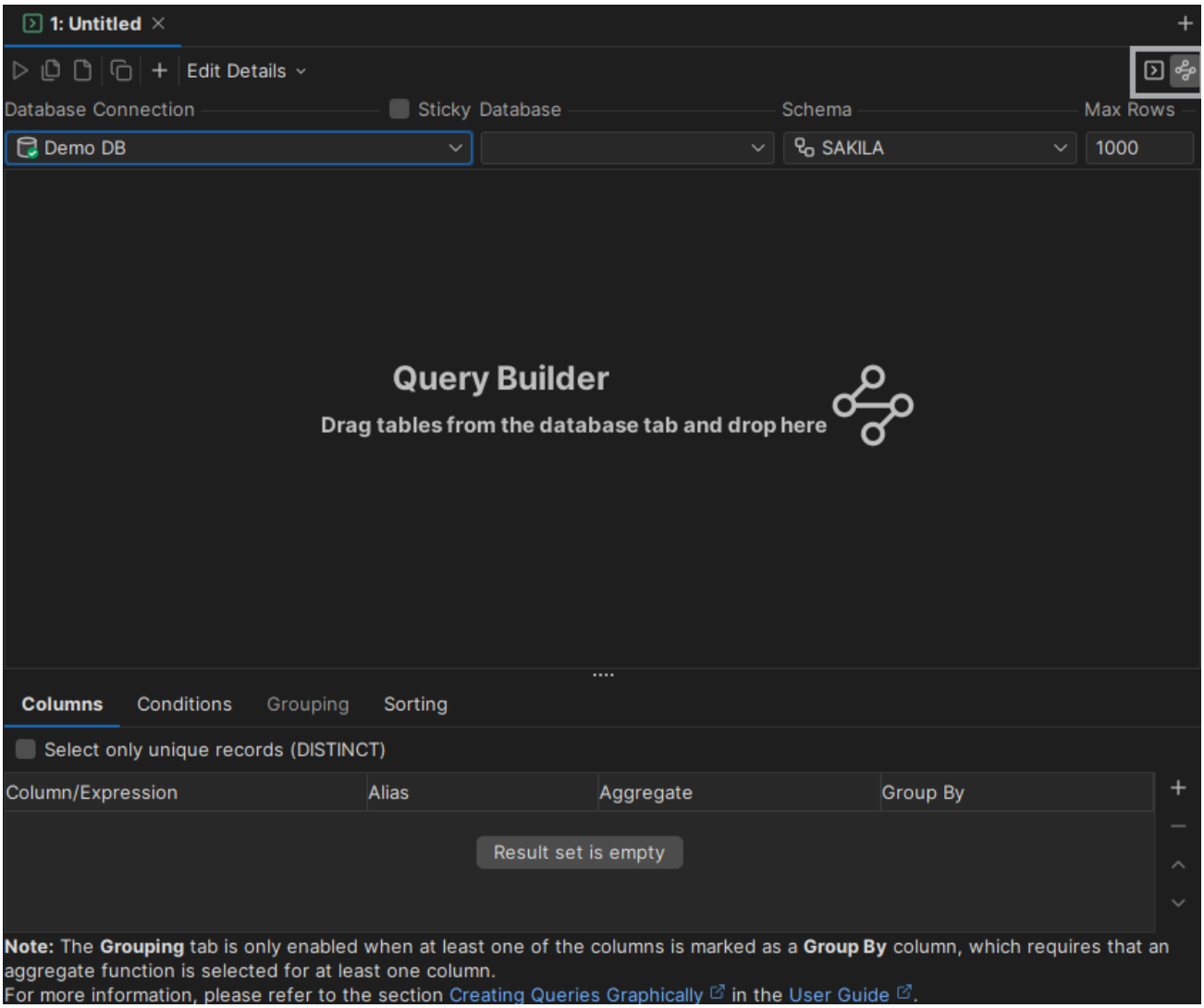
The Query Builder provides an easy way to develop database queries. The Query Builder provides a point and click interface and does not require in-depth knowledge about the SQL syntax.

The Query Builder is part of the SQL Commander, alongside the SQL Editor. When you are ready to test a query built with the Query Builder, you just load it to the SQL Editor for execution.

> This document talks only about Tables even though the Query Builder supports both table and view objects.

## Creating a Query

To create a query, open the query builder using the **SQL Commander → Show Query Builder** menu choice or click the vertical **Query Builder** button in the SQL Commander. Make sure that the controls in the top section of the Query Builder are set correctly.




The easiest way to jump between the Query Builder and the SQL Editor is by clicking the vertical control buttons to the right in the SQL Commander. Clicking these buttons changes the display, but does not copy the query from one display to the other. To copy the current query from the Query Builder to the SQL Editor, use the toolbar buttons at the top of the Query Builder:



From left to right, the button ...

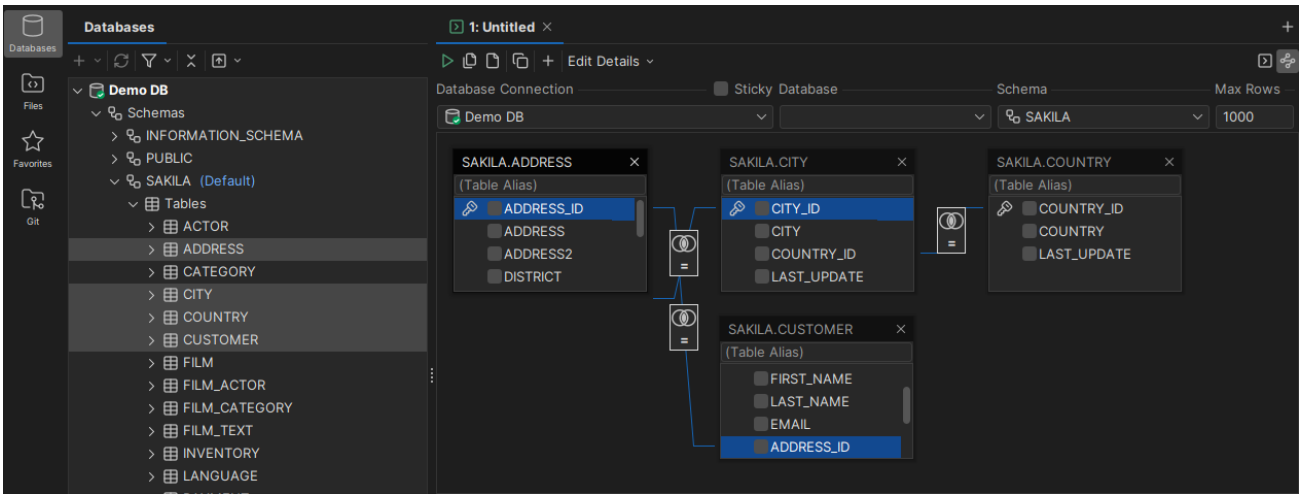
1. replaces the content of the SQL Editor with the query SQL and executes it.
2. replaces the content of the SQL Editor with the query SQL, without executing it.
3. adds the query last in the SQL Editor.
4. copies the query to the system clipboard.
5. opens a dialog that lets you add tables matching a search criteria.
6. is a drop-down menu for selecting what to show below the diagram area: query details or the SQL preview.

The first three buttons automatically change the display to the SQL Editor.

You can also load a query from the SQL Editor into the Query Builder, as described in detail below.

## Adding tables

To add tables using drag and drop, make sure the database objects tree and the table and/or view objects are visible and that the SQL Commander is showing the Query Builder view. Then select and drag one or more nodes from the tree into the diagram.




When a table is dropped in the diagram area, it is shown as a window with the table name as the window title.

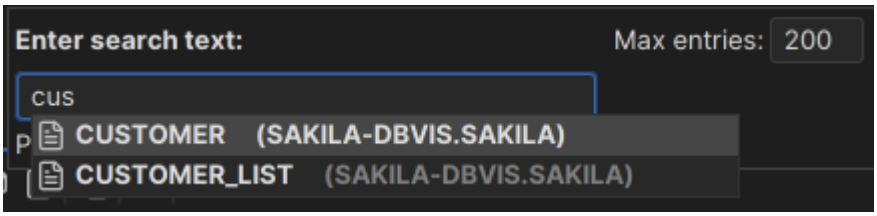
Below the table title is a text field where an optional table alias can be entered. If a table alias is specified, it is used in the Query Builder and the generated SQL statement to refer to this table.

Under the table alias field is a list of all table columns. Use the check box in front of each name to select whether the column should be included in the query result set. Columns selected for the query result set also appear in the **Columns** and **Sorting** details tabs.

Tables are added using a default layout; you can rearrange and resize them as you see fit.

QUICK TABLE ADD

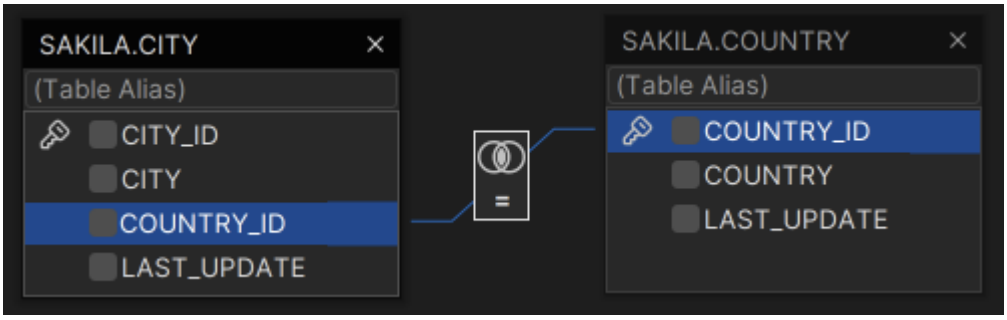
An alternative to dragging and dropping tables into the Query Builder is to use the Quick Table Add dialog.




It lists tables matching the search criteria as you type it in the search text field. An asterisk ("*") can be used as a wildcard for any characters.

## Joining Tables

To join two tables, select the column in the source table window with the mouse, drag it to the target table column, and drop it.

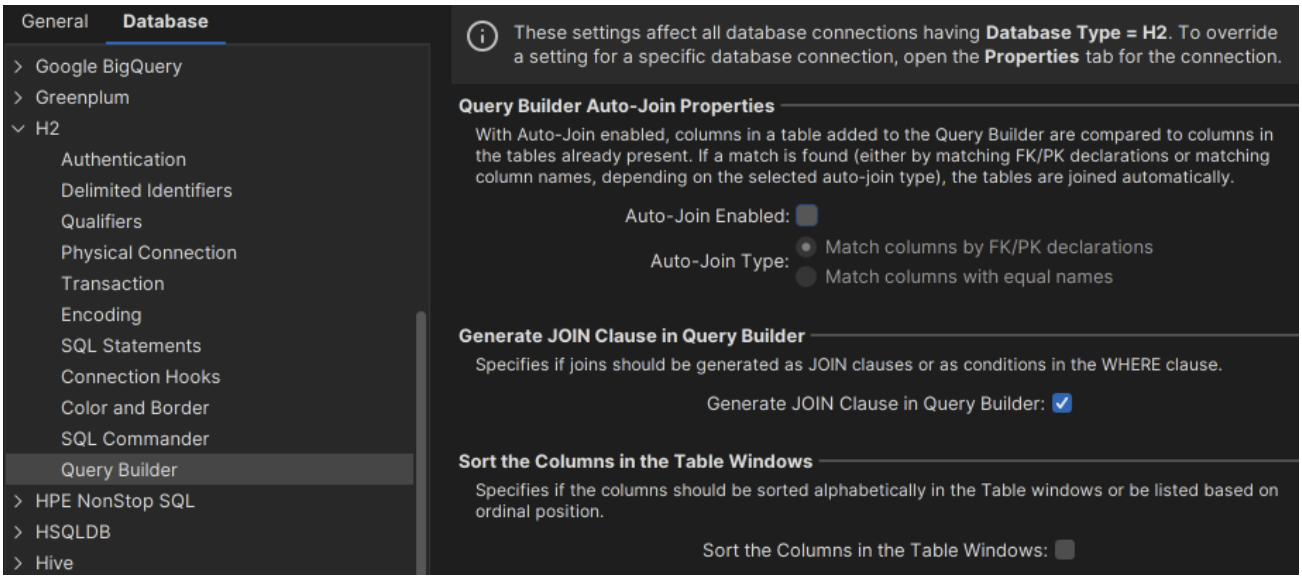



The two columns now represent a join condition, shown in the graph as a link between the columns. If more than one join condition is needed, link additional columns in the two tables by dragging and dropping the columns in the same way as for the first join condition. The default join type is an Inner join and the default condition is "equal to" (=), represented as an icon with overlapping circles with the shared area shaded and an equal sign below them.

## Autojoin

Some database schemas declare how tables are related using primary and foreign keys. Other schemas use column names to indicate these relationships. For instance, in the figure above, the CITY table has a column named COUNTRY_ID, which refers to the column with the same name in the COUNTRY table. The Query Builder can be configured to use both kinds of rules to automatically join the tables you add to the query builder.

The auto-join feature is disabled by default. You can enable it in the tool properties for the database type (**Tools → Tool Properties**, under the **Database** tab) or for a specific connection (the **Properties** tab in the **Object View** tab for the connection). In the Query Builder category, you can enable the auto-join feature and select whether to use key declarations (FK/PK) or column names to find out how the tables are related.




When you add a new table with auto-join enabled, the Query Builder automatically joins it to the tables already in the builder if table columns match the selected matching rule.

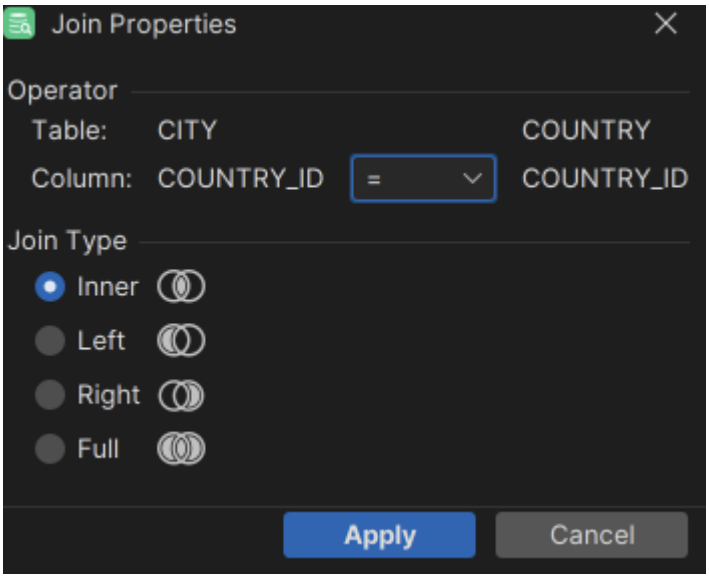
If columns in the table you add are related to other columns in the same table, the Query Builder creates two windows for the table and joins them based on the matching rule. In this case, a table alias is also added for one of the windows so that you can tell the two windows for the same table apart.

## Setting Join Properties

A Join Properties dialog can be opened by double-clicking the icon or selecting **Join Properties** from the right-click menu while the mouse pointer is over the join icon. The Join Properties dialog shows the source and target table columns and the conditional operator.

You can change the join type and the conditional operator in the Join Properties dialog. The join type defines how the records from the tables should be combined:

- **Inner** This is the most common join type as it finds the results in the intersection between the tables.
- **Left** This join type limits the results to those in the left table leaving 0 matching records in the right table as NULL.
- **Right** This is the same as left join but reversed
- **Full** A full join combines the results of both left and right joins.



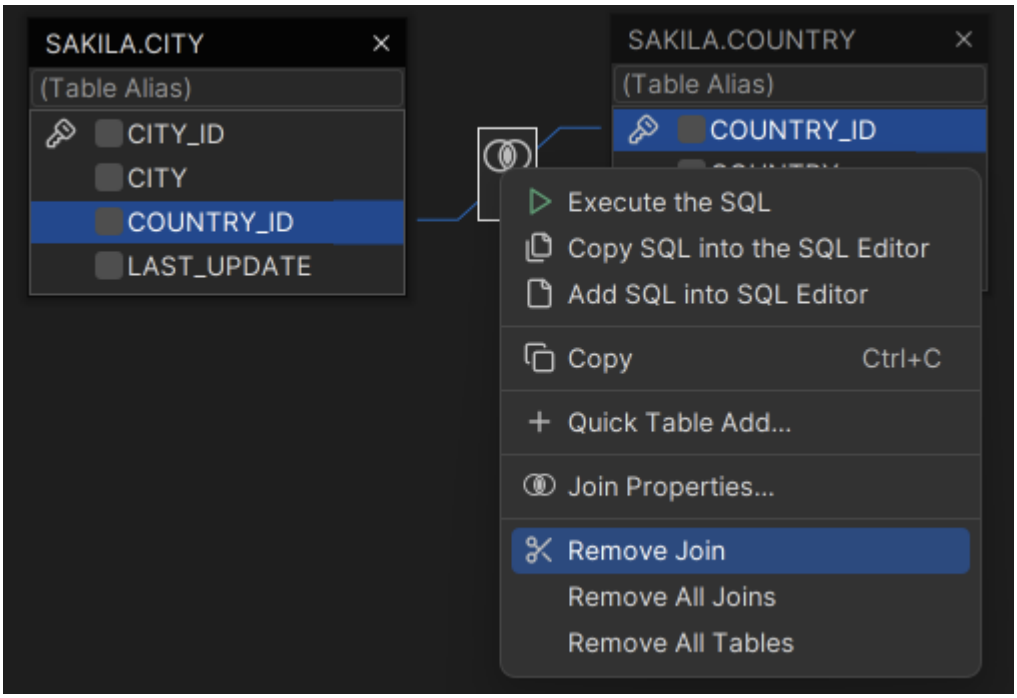

If you have multiple join conditions (linked columns) between two tables, you can specify different conditional operators for each join condition, but the join type is shared between all join conditions; if you change it for one join condition, it is changed for all the other join conditions linking the two tables. This is not a restriction in the Query Builder but rather how SQL is defined.

Here is the sample SQL generated from the previous join definition:

```
SELECT
    *
FROM
    SAKILA.CITY
INNER JOIN
    SAKILA.COUNTRY
ON
    (
        SAKILA.CITY.COUNTRY_ID = SAKILA.COUNTRY.COUNTRY_ID);
```

## Removing Tables and Joins

A table window is removed by clicking the close icon in the window header. A join is removed by selecting **Remove Join** in the right-click menu while the mouse pointer is over the join icon.




All tables and joins may be removed via **Remove All Joins** and **Remove All Tables.**

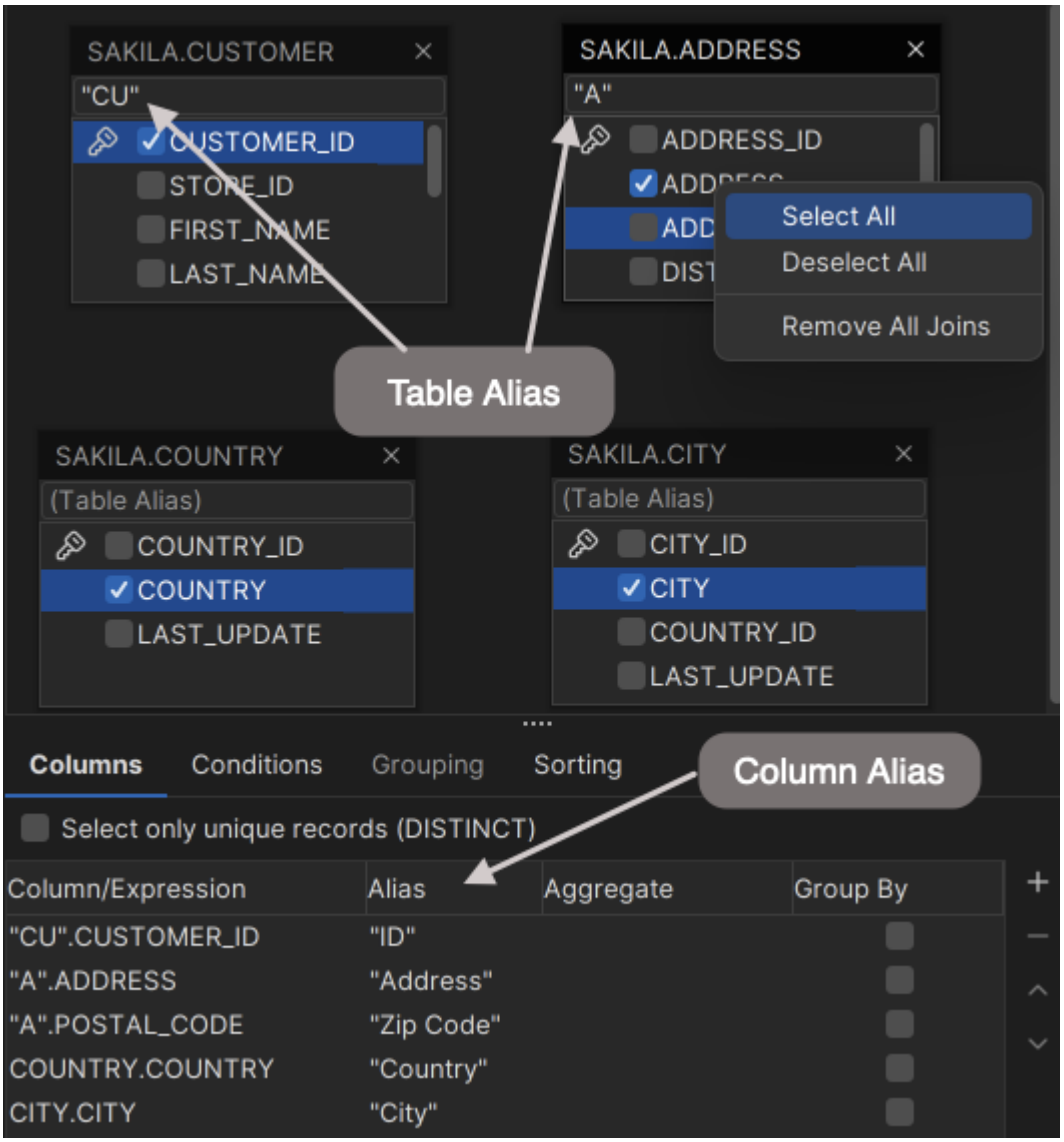
## Specifying Query Details

The Details tabs below the diagram area are used to define the various parts of the query. The tabs basically represent the following parts of the final SQL:

```
   SELECT <Columns>
     FROM <Tables>
    WHERE <Conditions>
 GROUP BY <Columns>
   HAVING <Grouping>
 ORDER BY <Sorting>
```

(The Tables clause is defined in the diagram, not by a tab).

Use the **Columns** tab to specify characteristics of the columns that are included in the query. The list is initially empty until a column is checked in a table window or a column expression is added manually (see below). Columns will appear in the list in the same order as they are checked but may be moved at any time with the up and down buttons. To include all columns from a table, right-click in the column list in the table window and choose **Select All**.




The previous screenshot shows a total of 5 checked columns in the four tables. These are presented in the columns list by their full column identifier, qualified by either the table name or the table alias. To remove a column from the list, uncheck the corresponding column in the table window.

The alias field is used to specify an optional alias identifier for the column. The alias is used as the identifier for the column in the final query and also appears as the column name in the result set produced by the query. Check the documentation for the actual database to see if the alias must be quoted since the Query Builder does not do this for you.

The **Aggregate** and **Group** by fields are used in combination:

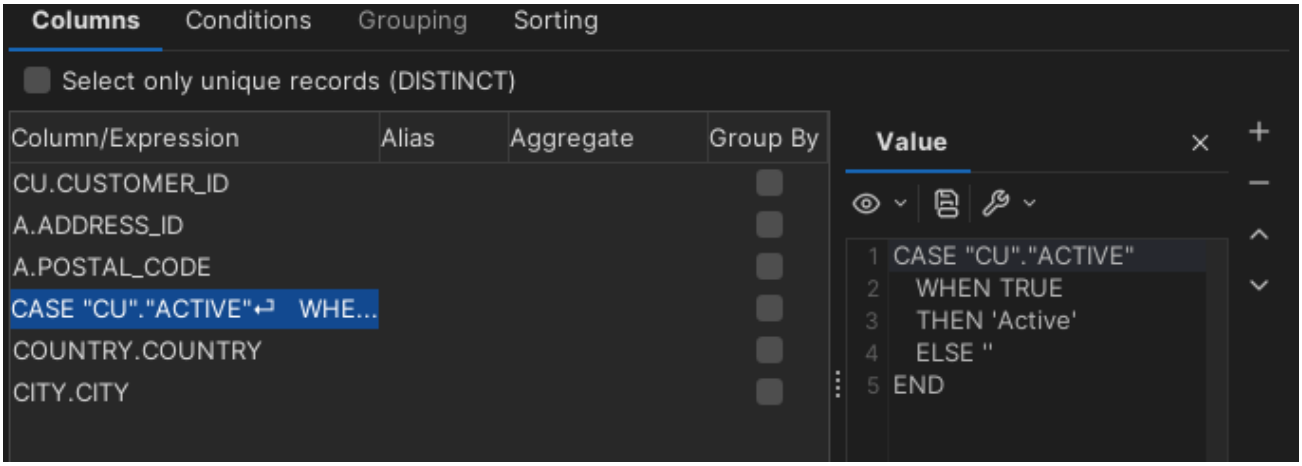
- The **Aggregate** field lists the available aggregation functions (AVG, COUNT, MAX, MIN, SUM) that may be used for columns
- The **Group By** field specifies whether the column should be included in the group for which aggregate columns are summarized

The **Group By** field is disabled unless an aggregate function is selected for at least one column, and once you select an aggregate function for one column, you must set **Group By** for at least one of the other columns to form a valid query. If you remove the aggregate function for all columns, **Group By** is automatically reset for all columns. **Group By** and **Aggregate** are also mutually exclusive options for one column, so when you select one of them, the field for the other is disabled for that column.

#### CUSTOM EXPRESSIONS

A custom expression may be added by entering data in the empty row last in the list, e.g., `col1 + col2` or `TO_CHAR(ts_col, 'DD-MON-YYYY HH24:MI:SSxFF')`. Once entered, press Add button to insert a new empty row. You can remove a custom expression by selecting it and clicking the Remove button.

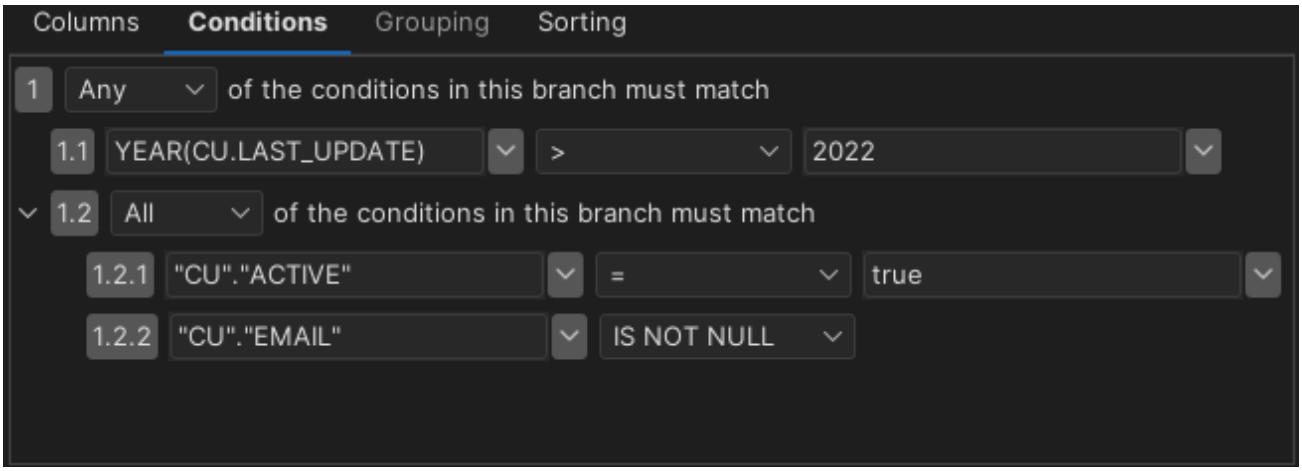
You can also launch a multi-line text editor for a custom expression, to make it easier to edit a large expression such as a CASE clause. Just double-click the expression cell, and then click on the editor launch button to the right.




#### CONDITIONS

The **Conditions** tab is used to manage the WHERE clause for the query. A WHERE clause may consist of several conditions connected by AND or OR. The evaluation order for each condition is defined by indentation in the condition list. Each level in the list will be enclosed by brackets in the final SQL.

Here is an example from the **Conditions** tab.

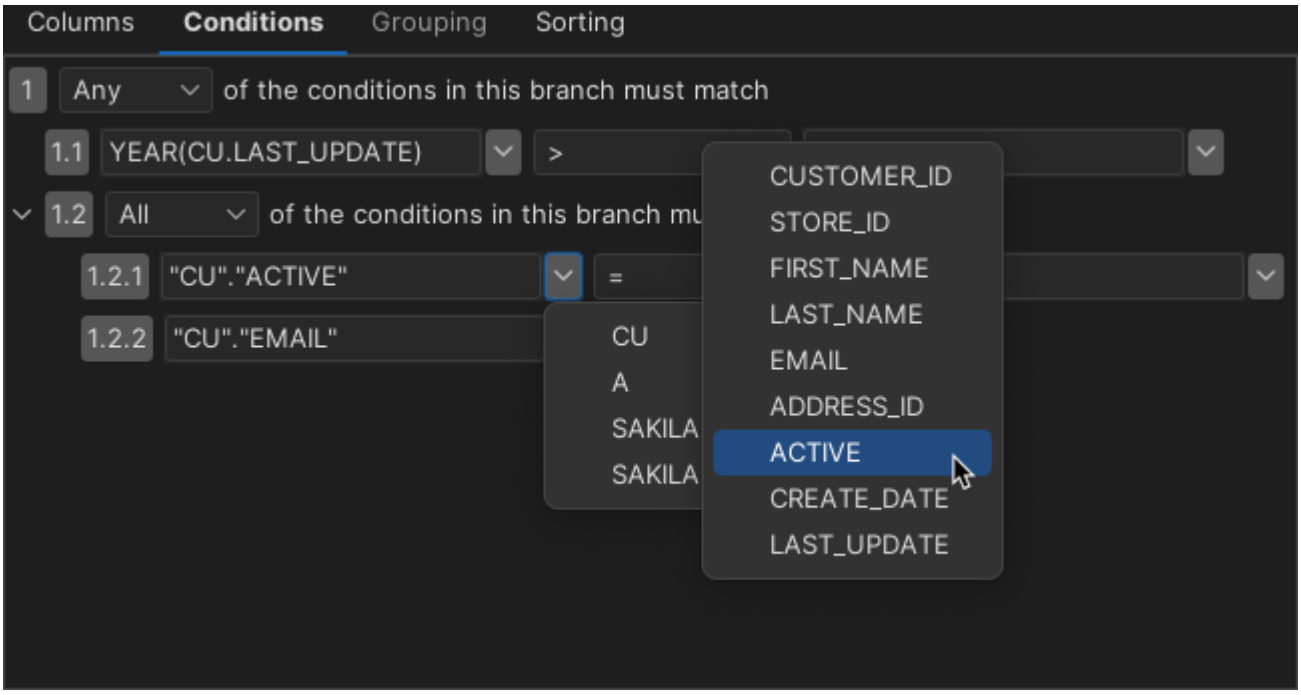



To create a new WHERE condition, press the indexed button in the list. In the menu that is displayed you may choose to create a new condition on the same level, a compound condition or delete the current condition.

For compound conditions you may choose whether **All** (AND), **Any** (OR), **None** (NOT OR) or **Not All** (NOT AND) conditions must be met for its sub conditions. The SQL for the **Conditions** tab in the figure is:

```
WHERE
    YEAR("CU"."LAST_UPDATE") > 2019
OR  "CU".ACTIVE = TRUE
AND "CU".EMAIL IS NOT NULL;
```

Next to the input field for each condition, there is a drop down button. When pressed it shows all columns that are available in the tables currently being in the Query Builder. You can pick columns from the list instead of typing these manually.




A condition field may also contain a custom expression, and just as for a custom expression in the columns list, you can launch a multi-line editor for the expression by selecting the field and click the editor launch button.

GROUPING

The **Grouping** tab is used to define the conditions for the HAVING clause that may follow a GROUP BY clause in an SQL query. This tab is only enabled when at least one of the columns in the Columns tab is marked as a Group By column.

The HAVING clause is similar to the WHERE clause, except that the HAVING clause limits what rows are included in the groups defined by the GROUP BY clause, after the WHERE clause has been used to limit the total number of rows to process.

You work with conditions in this tab in the same way as in the **Conditions** tab, with one exception regarding the drop-down button for the fields in a condition. In the **Grouping** tab, the drop-down shows all columns listed in the **Columns** tab, with an aggregate function expression for columns that have an aggregate function defined. This is because (according to the SQL specification) the conditions in a HAVING clause must only refer to columns that are being returned by the query.

Finally, the **Sorting** tab is used to specify how the final result set will be sorted. All columns for the tables in the graph, plus any custom expressions created for the selection list in the **Columns** tab, are listed in the **Sorting** tab.

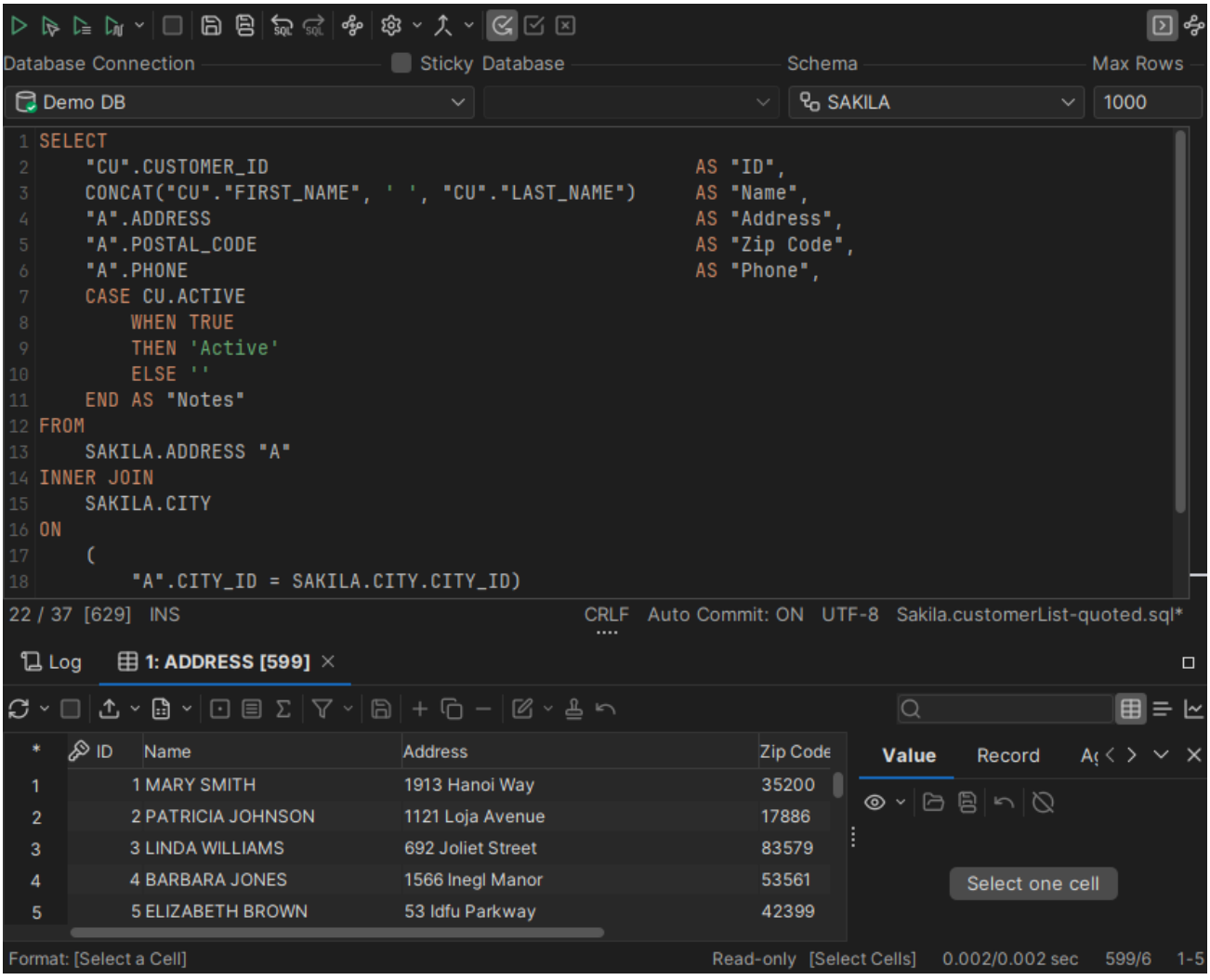
## SQL Preview

Select **SQL Preview** in the drop-down menu in the toolbar to show a preview of the final SQL. This is a read-only view and cannot be modified.




## Testing the Query

To test the query, simply press the appropriate toolbar button in the Query Builder to copy the SQL to the SQL Editor. Then execute the SQL as usual in the SQL Editor. Or click the **Execute** button in the Query Builder toolbar to copy and execute the SQL in one step.




To further refine the SQL press the Query Builder button and make the necessary changes.

## Loading a Query From the Editor

If you have an existing SQL query that you want to modify using the Query Builder, you can load it from the SQL Editor into the Query Builder by clicking the **Load in Query Builder** button in the SQL Editor toolbar.

It's important to be aware that the Query Builder does not support all features of the SQL SELECT statement, such as comments, UNION, and database-specific keywords. If you load a query into the Query Builder that contains unsupported constructs or keywords, they are ignored and a dialog pops up with a warning about this fact. You can then use the SQL Preview tab in the Query Builder to compare the SQL as it is represented in the Query Builder with the original SQL that you loaded to understand what was ignored.

## Properties for the Query Builder

In addition to the **Auto Join** setting discussed above, there are a few other properties that control how the Query Builder works and the SQL it generates. You can set these properties for the database type (**Tools → Tool Properties**, under the **Database** tab) or for a specific connection (the **Properties** tab in the **Object View** tab for the connection).

### Express joins as JOIN clause or WHERE condition

The **Generate JOIN Clause in SQL Builder** property is available in the **[Database Type] → Query Builder** category. Joins can be expressed either via the standardized SQL JOIN clause or a WHERE clause, using database-specific syntax for the Left and Right join types. The database-specific WHERE clause syntax is somewhat different between the supported databases and the Full outer join type is not supported. The default for this property is to use a JOIN clause.

A simple inner join expressed as a JOIN clause:

```
FROM SAKILA.CITY
INNER JOIN SAKILA.COUNTRY
ON ( SAKILA.CITY.COUNTRY_ID = SAKILA.COUNTRY.COUNTRY_ID );
```

Here is the same join expressed as a WHERE condition:

```
FROM SAKILA.CITY, SAKILA.COUNTRY
WHERE ( SAKILA.CITY.COUNTRY_ID = SAKILA.COUNTRY.COUNTRY_ID );
```

The syntax for expressing Inner and Outer joins in WHERE conditions is different between databases. Oracle, for example, uses the "(+)" sequence to the left or right of the conditional operator to express left or right joins. SQL Server and Sybase use "*=" or "=*" for the same purpose.

DbVisualizer automatically uses the correct join notation when generating joins as WHERE conditions for databases that support left and right joins using WHERE conditions. For databases that do not provide syntax for left and right joins, the join type is ignored and the WHERE condition that is generated produces an inner join result.

### Table and Column Name qualifiers

Whether to qualify table names with the schema or database name and whether to qualify column names with the table name are defined in the **[Database Type] / Qualifiers** category.

### Delimited Identifiers

Identifiers that contain mixed case characters or include special characters need to be delimited. Define this in the **[Database Type] / Delimited Identifiers** category.

## Current Limitations

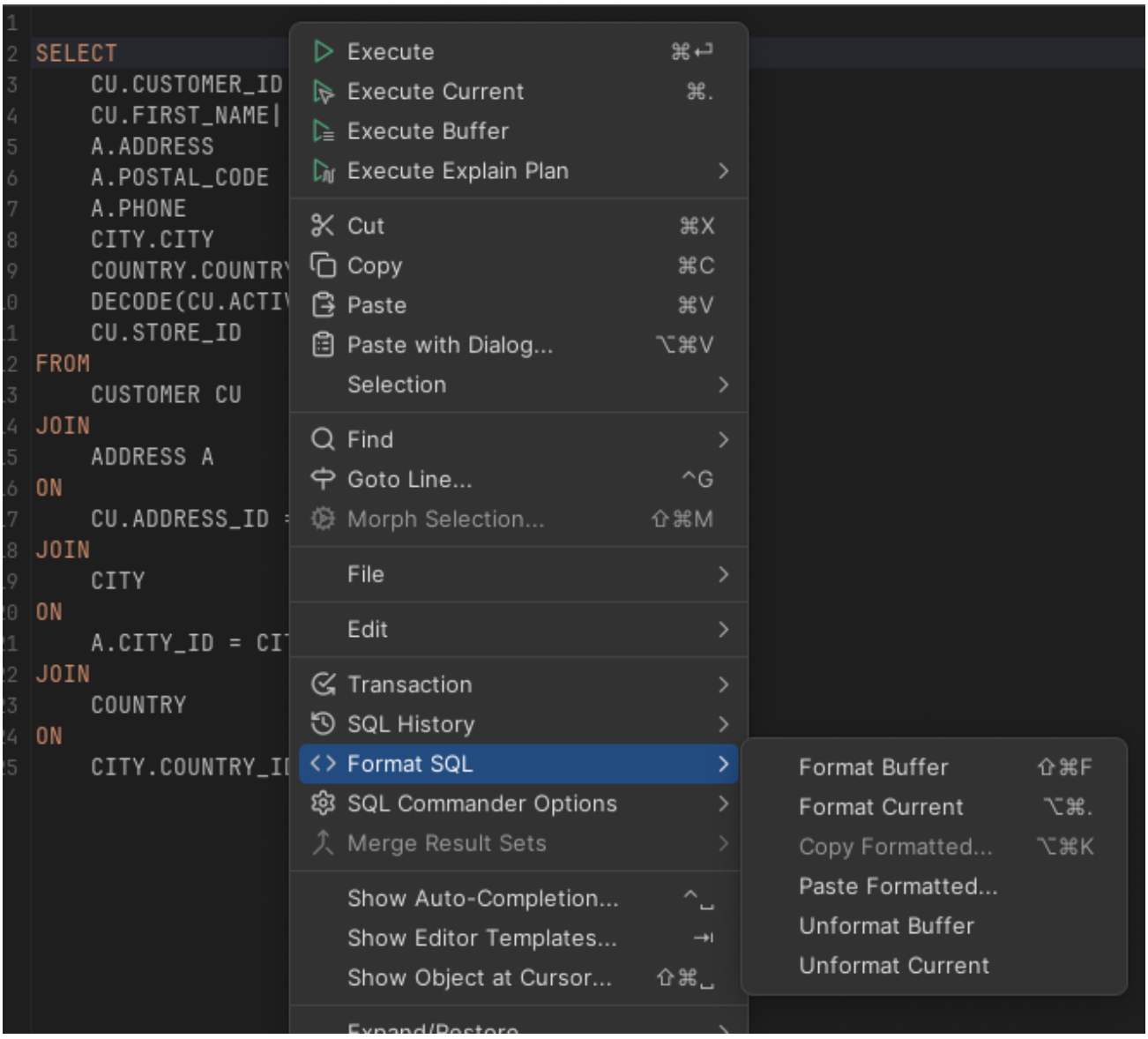
These are the current limitations in the Query Builder:

- Unions and sub queries are not supported.
- Not all join types are supported when joins are expressed as WHERE clause conditions. The **Inner** join type is supported for all databases, but the **Left** and **Right** types are only supported for databases with proprietary syntax to express these types, e.g., Oracle, SQL Server and Sybase. The **Full** type is not supported for any database. If a join type is not supported, the setting in the **Join Properties** dialog is silently ignored.
- When importing an SQL query from the SQL Editor, comments, unsupported keywords and statement clauses are ignored. A dialog tells you which parts of the query are being ignored when unsupported parts are found in the imported statement.
- There is only limited support for the CASE clause, in that everything between CASE and END is treated as uninterpreted text. This means that, as opposed to plain object references in the select list or conditions, column names and other identifiers within a CASE clause are not affected by changes to the Query Builder property settings, such as Delimited Identifiers and Qualifiers.

# Formatting SQL

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The **SQL Commander** main menu on the (or right-click in the editor) and its **Format SQL** sub menu contains operations for formatting SQL statements.




- **Format Buffer** and **Format Current** formats the complete editor content or the current SQL (at cursor position) respectively.
- **Copy Formatted** and **Paste Formatted** are powerful tools for copying SQL statements between programs written in languages like Java, C#, PHP, VB, etc. and the SQL Editor. Both operations display a dialog where you can adjust some of the formatting options, most importantly the **Target SQL** option and the **SQL is Between** option. **Target SQL** can be set to a number of common programming language formats.
- **Unformat Buffer** and **Unformat Current** produces compact statements by removing unnecessary whitespace.

**Example:**

To copy an SQL statement and paste it as Java code for adding it to a Java StringBuffer:

1. Select the statement. Example: `SELECT * FROM SAKILA.STAFF`)
2. Choose **SQL Commander → Format SQL → Copy Formatted**,
3. Set **Target SQL** to Java StringBuffer,
4. Click **Format** to place the formatted statement on the system clipboard,
5. Paste it into your Java code. Example: `StringBuffer sql = new StringBuffer();` `sql.append("SELECT ");` `sql.append(" * ");` `sql.append("FROM ");` `sql.append(" SAKILA.STAFF");`

To copy a statement wrapped in code from a program:

1. Select the code containing an SQL statement in your program,
2. Copy it to the system clipboard,
3. Choose **SQL Commander → Format SQL → Paste Formatted**,
4. Check **SQL is Between** and enter the character enclosing the SQL statement in the code,
5. Click **Format** to extract the SQL statement and paste the formatted SQL in the editor.

## Settings

All formatting is done according to the settings defined in the **Tool Properties** dialog, in the **SQL Commander/SQL Formatting** category under the General tab.

There are many things you can configure; use the default example or your own SQL to check the effect of the settings. After making some changes, press **Apply** and format again to see the result.

**Example:**

```
-- Basic SELECT example, with Sub-SELECT and JOIN
SELECT e.LAST_NAME AS "Last Name", e.FIRST_NAME AS "First Name",
d.DEPARTMENT_NAME AS "Department", e.SALARY AS "Salary", e.SALARY + e.SALARY *
e.COMMISSION_PCT, e.COMMISSION_PCT * 100 || '%', ROUND(e.SALARY / ( SELECT
MAX(SALARY) FROM HR.EMPLOYEES), 2) * 100 AS "Percentage of Max" FROM HR.EMPLOYEES
e INNER JOIN HR.DEPARTMENTS d ON ( e.DEPARTMENT_ID = d.DEPARTMENT_ID) WHERE
d.DEPARTMENT_ID IN (10, 20, 90, 210) AND e.SALARY > 3000;
-- CASE example
SELECT FIRST_NAME, LAST_NAME, SALARY, CASE WHEN SALARY > 10000 THEN 'High' WHEN
SALARY BETWEEN 5000 AND 999 THEN 'Midlevel' ELSE 'Low' END AS "Income Level",
CASE DEPARTMENT_ID WHEN 40 THEN 'Administration' WHEN 20 THEN 'Sales Related'
ELSE 'Other' END AS "Special Departments" FROM EMPLOYEES;
-- JOIN example, with GROUP BY, HAVING and ORDER BY
SELECT COUNT(d.DEPARTMENT_NAME) AS "Departments per Location", c.COUNTRY_NAME,
l.STATE_PROVINCE FROM DEPARTMENTS d INNER JOIN LOCATIONS l ON d.LOCATION_ID =
l.LOCATION_ID INNER JOIN COUNTRIES c USING (COUNTRY_ID) GROUP BY c.COUNTRY_NAME,
l.STATE_PROVINCE HAVING COUNT(d.DEPARTMENT_NAME) > 1 ORDER BY 2, 3, 1;
-- UPDATE example
UPDATE EMPLOYEES SET COMMISSION_PCT = 10 WHERE COMMISSION_PCT = 0 AND SALARY <
5000;
-- INSERT example
INSERT INTO EMPLOYEES ( FIRST_NAME, LAST_NAME ) VALUES ( 'Roger', 'Bjarevall' );
-- DELETE example
DELETE FROM EMPLOYEES WHERE HIRE_DATE < to_timestamp('1900-01-10', 'RR-MM-DD');
-- CREATE TABLE example
CREATE TABLE DEPARTMENTS ( DEPARTMENT_ID NUMBER(4) NOT NULL, DEPARTMENT_NAME
VARCHAR2(30) NOT NULL, MANAGER_ID NUMBER(6), LOCATION_ID NUMBER(4), CONSTRAINT
DEPT_ID_PK PRIMARY KEY (DEPARTMENT_ID), CONSTRAINT DEPT_LOC_FK FOREIGN KEY
(LOCATION_ID) REFERENCES "LOCATIONS" ("LOCATION_ID"), CONSTRAINT DEPT_MGR_FK
FOREIGN KEY (MANAGER_ID) REFERENCES "EMPLOYEES" ("EMPLOYEE_ID"), CONSTRAINT
DEPT_NAME_NN CHECK ("DEPARTMENT_NAME" IS NOT NULL) );
```

**Formatted with default settings:**

```
-- Basic SELECT example, with Sub-SELECT and JOIN
SELECT
    e.LAST_NAME           AS "Last Name",
    e.FIRST_NAME          AS "First Name",
    d.DEPARTMENT_NAME     AS "Department",
    e.SALARY              AS "Salary",
    e.SALARY + e.SALARY * e.COMMISSION_PCT,
    e.COMMISSION_PCT * 100 || '%',
    ROUND(e.SALARY /
    (   SELECT
            MAX(SALARY)
        FROM
            HR.EMPLOYEES), 2) * 100 AS "Percentage of Max"
FROM
    HR.EMPLOYEES e
INNER JOIN
    HR.DEPARTMENTS d
ON
    (
        e.DEPARTMENT_ID = d.DEPARTMENT_ID)
WHERE
    d.DEPARTMENT_ID IN (10,
                        20,
                        90,
                        210)
AND e.SALARY > 3000;

-- CASE example
SELECT
    FIRST_NAME,
    LAST_NAME,
    SALARY,
    CASE
        WHEN SALARY > 10000
        THEN 'High'
        WHEN SALARY BETWEEN 5000 AND 999
        THEN 'Midlevel'
        ELSE 'Low'
    END AS "Income Level",
    CASE DEPARTMENT_ID
        WHEN 40
        THEN 'Administration'
        WHEN 20
        THEN 'Sales Related'
        ELSE 'Other'
    END AS "Special Departments"
FROM
    EMPLOYEES;

-- JOIN example, with GROUP BY, HAVING and ORDER BY
SELECT
    COUNT(d.DEPARTMENT_NAME) AS "Departments per Location",
```

```
    c.COUNTRY_NAME,
    l.STATE_PROVINCE
FROM
    DEPARTMENTS d
INNER JOIN
    LOCATIONS l
ON
    d.LOCATION_ID = l.LOCATION_ID
INNER JOIN
    COUNTRIES c
USING
    (COUNTRY_ID)
GROUP BY
    c.COUNTRY_NAME,
    l.STATE_PROVINCE
HAVING
    COUNT(d.DEPARTMENT_NAME) > 1
ORDER BY
    2,
    3,
    1;

-- UPDATE example
UPDATE
    EMPLOYEES
SET
    COMMISSION_PCT = 10
WHERE
    COMMISSION_PCT = 0
AND SALARY < 5000;

-- INSERT example
INSERT INTO
    EMPLOYEES
    (
        FIRST_NAME,
        LAST_NAME
    )
    VALUES
    (
        'Roger',
        'Bjarevall'
    );

-- DELETE example
DELETE
FROM
    EMPLOYEES
WHERE
    HIRE_DATE < to_timestamp('1900-01-10', 'RR-MM-DD');

-- CREATE TABLE example
CREATE TABLE
```

```
    DEPARTMENTS
    (
        DEPARTMENT_ID   NUMBER(4) NOT NULL,
        DEPARTMENT_NAME VARCHAR2(30) NOT NULL,
        MANAGER_ID      NUMBER(6),
        LOCATION_ID     NUMBER(4),
        CONSTRAINT DEPT_ID_PK PRIMARY KEY (DEPARTMENT_ID),
        CONSTRAINT DEPT_LOC_FK FOREIGN KEY (LOCATION_ID) REFERENCES "LOCATIONS"
("LOCATION_ID"),
        CONSTRAINT DEPT_MGR_FK FOREIGN KEY (MANAGER_ID) REFERENCES "EMPLOYEES"
("EMPLOYEE_ID"),
        CONSTRAINT DEPT_NAME_NN CHECK ("DEPARTMENT_NAME" IS NOT NULL)
    );
```

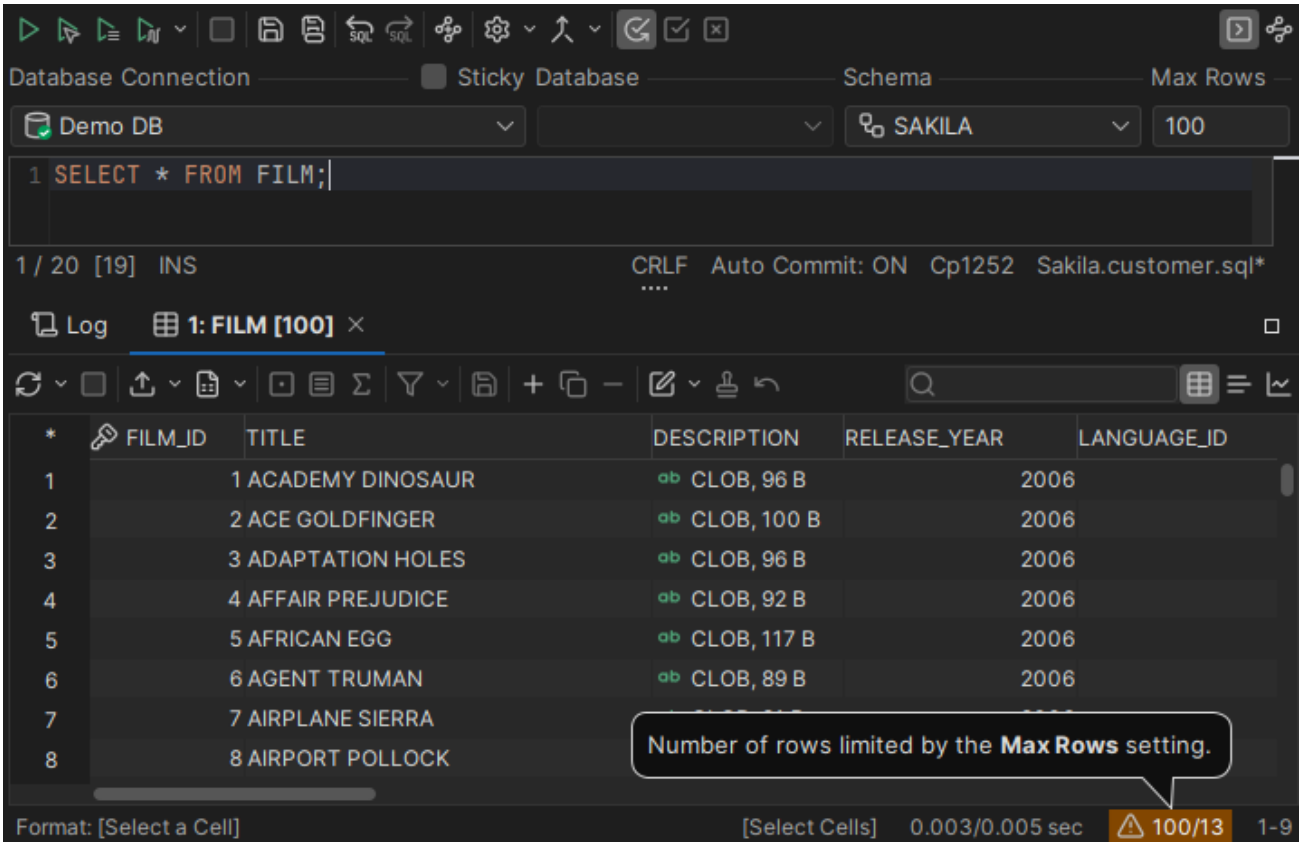
**Unformatted to compact form:**

```
/*-- Basic SELECT example, with Sub-SELECT and JOIN*/ SELECT e.LAST_NAME AS "Last
Name", e.FIRST_NAME AS "First Name", d.DEPARTMENT_NAME AS "Department", e.SALARY
AS "Salary", e.SALARY + e.SALARY * e.COMMISSION_PCT, e.COMMISSION_PCT * 100 ||
'%', ROUND(e.SALARY / ( SELECT MAX(SALARY) FROM HR.EMPLOYEES), 2) * 100 AS
"Percentage of Max" FROM HR.EMPLOYEES e INNER JOIN HR.DEPARTMENTS d ON (
e.DEPARTMENT_ID = d.DEPARTMENT_ID) WHERE d.DEPARTMENT_ID IN (10, 20, 90, 210) AND
e.SALARY > 3000;
/*-- CASE example*/ SELECT FIRST_NAME, LAST_NAME, SALARY, CASE WHEN SALARY >
10000 THEN 'High' WHEN SALARY BETWEEN 5000 AND 999 THEN 'Midlevel' ELSE 'Low' END
AS "Income Level", CASE DEPARTMENT_ID WHEN 40 THEN 'Administration' WHEN 20 THEN
'Sales Related' ELSE 'Other' END AS "Special Departments" FROM EMPLOYEES;
/*-- JOIN example, with GROUP BY, HAVING and ORDER BY*/ SELECT
COUNT(d.DEPARTMENT_NAME) AS "Departments per Location", c.COUNTRY_NAME,
l.STATE_PROVINCE FROM DEPARTMENTS d INNER JOIN LOCATIONS l ON d.LOCATION_ID =
l.LOCATION_ID INNER JOIN COUNTRIES c USING (COUNTRY_ID) GROUP BY c.COUNTRY_NAME,
l.STATE_PROVINCE HAVING COUNT(d.DEPARTMENT_NAME) > 1 ORDER BY 2, 3, 1;
/*-- UPDATE example*/ UPDATE EMPLOYEES SET COMMISSION_PCT = 10 WHERE
COMMISSION_PCT = 0 AND SALARY < 5000;
/*-- INSERT example*/ INSERT INTO EMPLOYEES ( FIRST_NAME, LAST_NAME ) VALUES (
'Roger', 'Bjarevall' );
/*-- DELETE example*/ DELETE FROM EMPLOYEES WHERE HIRE_DATE <
to_timestamp('1900-01-10', 'RR-MM-DD');
/*-- CREATE TABLE example*/ CREATE TABLE DEPARTMENTS ( DEPARTMENT_ID NUMBER(4)
NOT NULL, DEPARTMENT_NAME VARCHAR2(30) NOT NULL, MANAGER_ID NUMBER(6),
LOCATION_ID NUMBER(4), CONSTRAINT DEPT_ID_PK PRIMARY KEY (DEPARTMENT_ID),
CONSTRAINT DEPT_LOC_FK FOREIGN KEY (LOCATION_ID) REFERENCES "LOCATIONS"
("LOCATION_ID"), CONSTRAINT DEPT_MGR_FK FOREIGN KEY (MANAGER_ID) REFERENCES
"EMPLOYEES" ("EMPLOYEE_ID"), CONSTRAINT DEPT_NAME_NN CHECK ("DEPARTMENT_NAME" IS
NOT NULL) );
```

# Using Max Rows and Max Chars for Queries

## Max Rows setting

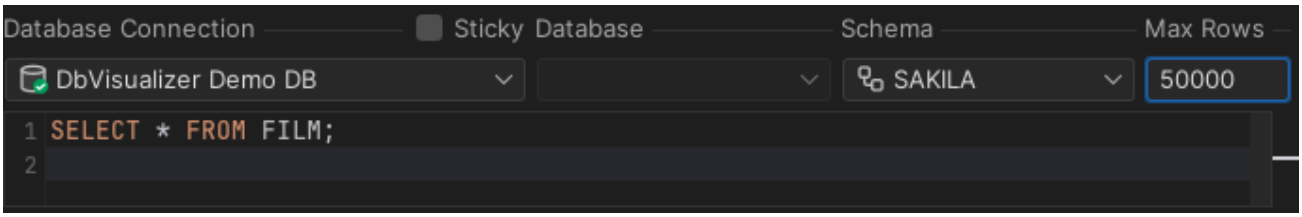
DbVisualizer by default limits the number of rows shown in a result set tab to 20,000 rows to conserve memory. The following is displayed when this limit is reached.




If the max rows limit prevents you from seeing the data of interest, you should first consider:

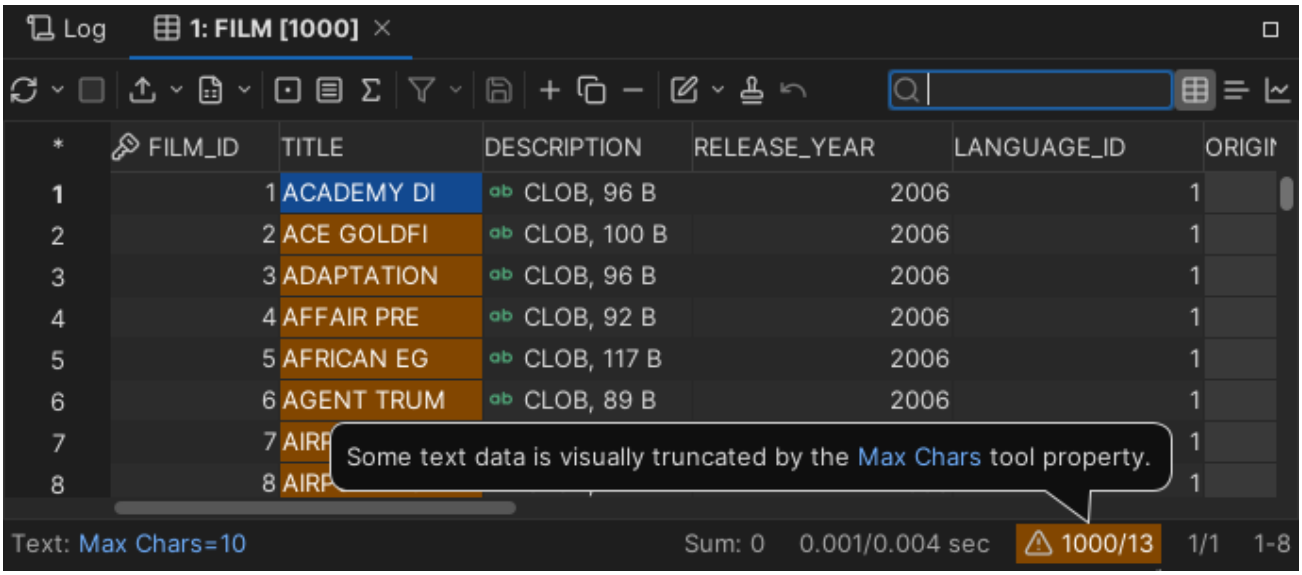
1. Using a `WHERE` clause in the query to only retrieve the rows of interest,
2. Exporting the table to a file.

If you really need to look at more than 20,000 rows, you can change the value in the **Max Rows** field in the SQL Commander toolbar. Use a value of 0 or -1 to get all rows.




## Max Chars setting

Text values presented in the result set grid may be very long, potentially consuming lots of memory and affecting performance. By default, DbVisualizer visually truncates text values at 1,000 characters, and when it happens, there is an indicator below the grid.




The **Max Chars** setting can be modified in the **Tool Properties** / **Data Formats** category or by clicking on any of the **Max Chars** links in the screenshot above.

> To view the full text of a truncated text value, double-click on it to open in the Value viewer

# Getting the DDL for an Object

You can use the `@dll` command in a script to get the DDL for a number of different database object types. The command supports this general syntax:

```
@ddl <objType>="<objId>" [ drop="true | false" ] [ constrCtrl="<constrCtrl>" ]
```

where `<objType>` is one of `table`, `indexesfortable`, `view`, `procedure`, `function`, `package` (Oracle only), `packagebody` (Oracle only), `objecttype` (Oracle only), `objecttypebody` (Oracle only), `sequence` (Oracle only), `synonym` (Oracle only), `module` (Mimer SQL only) or `trigger`, and `<objId>` is the qualified identifier for the object (case sensitive). Here's an example:

```
@ddl table="SAKILA.STAFF";
```

If `drop` is set to `true`, a DROP statement is included before the CREATE statement.

The `constrCtrl` parameter only applies to tables. It accepts two values: `noconstr` means that no constraints should be included in the statement that can potentially cause creating the table or inserting data into it to fail (FK and CHECK constraints), while `onlyconstr` means that an ALTER statement adding the remaining constraints should be generated instead of a CREATE statement.

# Using the Log Tab

The Log reports progress and results when executing in the **SQL Commander**, **Import**, **Export Schema/Database**, **Procedure Editor**, and when running **Database Object Actions**. The appearance of the Log may differ slightly depending on which of these functions is used. The Log was completely redesigned in 10.0 and now saves unlimited number of log entries to file. Due to memory constraints, the Log grid in the DbVisualizer tool keeps up to 10 000 entries until it starts truncating the oldest entries.

Note that the detail level in a log entry message depends on the driver and database that is being used. Some databases are very good at telling you what went wrong and why, while others provide less detail.

## Preprocessing Script

The preprocessing step is performed just before a script is executed. It locates any variables, parameters, and restricted commands (defined in the Permissions category in **Tools → Tool Properties**) in the script. Depending how large the script is, preprocessing may take some time to complete and DbVisualizer may even advise you to run without preprocessing if the script is too large.

Check Executing an External Script page for more information.

## Executing

### Disabled functionality during execution

When a script is executed some functions in the Log are disabled:

- Quick Search
- Sorting
- Filtering
- Expand row heights (double-click)
- Hyperlinks to highlight errors and statements
- Save Log to Text File

### Progress information

While executing there is progress information in the Log status bar showing overall metrics for the execution. It helps getting a quick overview of number of failed and successful statements, elapsed time, and number of statements executed per time unit.

The progress bar shows how much of the script has been executed. The progress bar switch to an indeterminate mode when **Preprocess Script** is switched off or when running a script using the **@run** client-side command. The reason is that the Log then doesn't count how many statements to execute.




### Auto scroll

When new log entries are added the Log grid is automatically scrolled to the bottom. This can be turned off in the toolbar or by right-click in the grid, and select **Auto Scroll**.

### Auto Clear log

Having auto clear checked, the log is cleared when execution is started. Disabling auto clear will keep all entries until number of entries exceeds 10 000 when truncation is triggered.

### Log truncation

When the number of entries in the grid exceeds 10 000 the log is truncated by removing the oldest entries. When the log is truncated an icon and "Truncated" text is displayed in the grid status bar. If having auto clear log disabled, you will still see the STARTED and FINISHED entries for a previous executions in the grid along with an entry telling how many log entries has been truncated for that execution.

## Auto resize row heights

The **Message** and **SQL/Command** columns may include multi-lined content. During execution only a single line of content is displayed in the grid. Once the execution has completed, and there are any FAILED statements these rows automatically expands to show all of its content. In the right-click menu under **Auto Resize Row Heights**, the auto resizing may be configured to also expand SUCCESS entries.

A cell having multi-lined content ends with three dots, "...". To manually expand/collapse a row with more content, double-click the row.

## Navigating to next/previous failed log entry

To navigate to next/previous failed statement, use the **Goto Next Failed** and **Goto Previous Failed** actions in the right-click menu. An alternative is to filter out all but FAILED log entries in the grid, see Filter and search section below for more information.

## Highlight statement or error in the SQL editor

If there is an editor associated with the log grid, such as the SQL Commander, the **Command** column for all log entries is represented by a hyperlink. Clicking the link will navigate to, and highlight the corresponding SQL/command in the SQL editor.

The same applies for any FAILED statements whereas the **Status** column is then a hyperlink, when clicked it will show the statement that failed. Depending on if the database reports error positions, clicking the link will set the caret at the position where the error started.

If a single statement produces multiple errors, clicking on the FAILED status link, will then show a menu with all errors. Selecting an error highlights it in the SQL editor.

For **@run** commands, clicking the @run link in the **Command** column shows a menu in which you can select to load the related file in a new SQL Commander tab.

The matching actions are in the right-click menu, **Highlight Statement** and **Highlight Failed Statement**.

## Saving all Log entries to text file

All log entries are written to file, even those that has been truncated in the Log grid. The **Save Log to Text File** right-click menu action saves this log to text. Note that the log file is cleared between executions if auto clear log is enabled and when manually clearing the log. It is also removed when closing down the SQL Commander tab.

The internal log file is stored in json format under the $HOME/.dbvis/sqllogs folder with the .dson extension. A log file may be loaded in the Log grid using **Load Log File** in the right-click menu.

## Copy Log Entries to clipboard

The copy log entries to clipboard actions transforms the selected rows to text format and puts these on the system clipboard. This is useful if you want to share all or some log entries with others. The difference with the standard **Copy Selection** is that the latter copies only the selected cells in standard grid copy format typically separated with tabs.

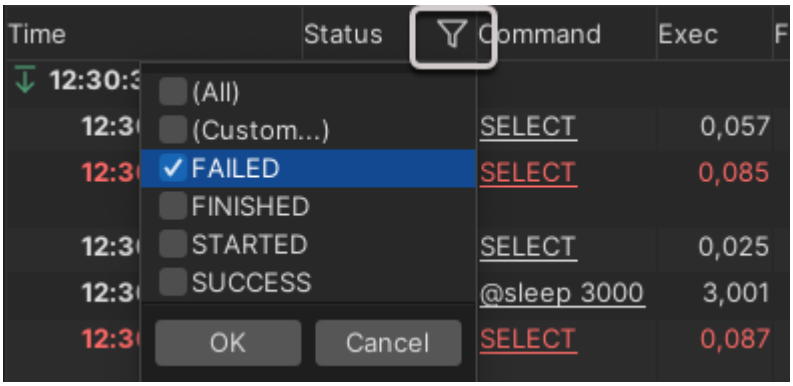
## Copy executed SQLs to the SQL Commander

The SQLs displayed in the log may be copied as a script to an existing or a new SQL Commander. Just select the rows you want to copy and then right-click and **Load Selected Commands into Editor**.

## Filter and search

The filtering capability is useful to quickly limit what entries are displayed in the grid. A good example is to click the filtering icon in the **Status** column header and then de-select all statuses than **FAILED** (or just double-click FAILED to de-select the others). The result is that only FAILED statements are displayed in the log grid. That type of filtering can be applied to any on the columns (and in fact in most grids in DbVisualizer). The quick search field can be used to further refine to do full text search on all entries in the grid.

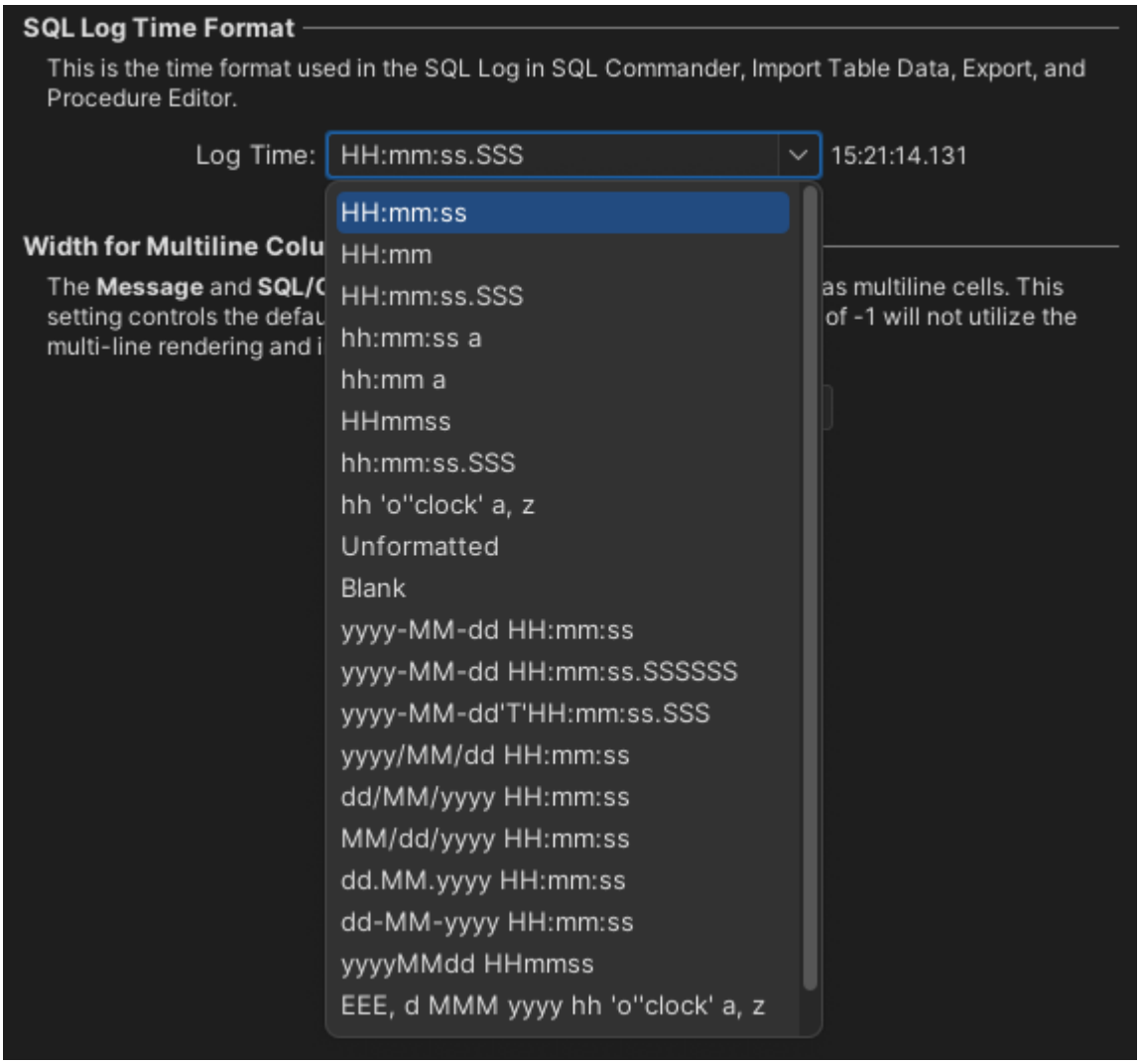
There is also a **(Custom...)** option that let you define your own condition to be used in the filter.




Applying a column filter will not have any effect on the **STARTED** or any **@run** log entries as these are grouping (indented) entries in the Log grid.

## Configure the Log

The **Time** column in the log shows by default **hour:minute:second** (HH:mm:ss) but can be configured to show date, milliseconds, and more. Change this in **Tools → Tool Properties** and the **General / SQL Commander / SQL Log** category.


SQL Log Time Format
This is the time format used in the SQL Log in SQL Commander, Import Table Data, Export, and Procedure Editor.
Log Time:
HH:mm:ss.SSS
15:21:14.131
HH:mm:ss
HH:mm
HH:mm:ss.SSS
hh:mm:ss a
hh:mm a
HHmmss
hh:mm:ss.SSS
hh 'o''clock' a, z
Unformatted
Blank
yyyy-MM-dd HH:mm:ss
yyyy-MM-dd HH:mm:ss.SSSSSS
yyyy-MM-dd'T'HH:mm:ss.SSS
yyyy/MM/dd HH:mm:ss
dd/MM/yyyy HH:mm:ss
MM/dd/yyyy HH:mm:ss
dd.MM.yyyy HH:mm:ss
dd-MM-yyyy HH:mm:ss
yyyyMMdd HHmmss
EEE, d MMM yyyy hh 'o''clock' a, z

# Writing to the Log Tab

You can use the @echo client side command to write information to the Log tab.

```
@echo <message>
```

The message may contain DbVisualizer variables, e.g. one of the predefined variables.

```
@echo Today is ${dbvis-date}$ and the time is ${dbvis-time}$
```

Variables can also be used to display values produced by executing a function or stored procedure:

```
@call ${STATUS||(null)||String||noshow dir=out}$ = "HR"."GET_STATUS"(1002);
@echo STATUS: ${STATUS}$;
```

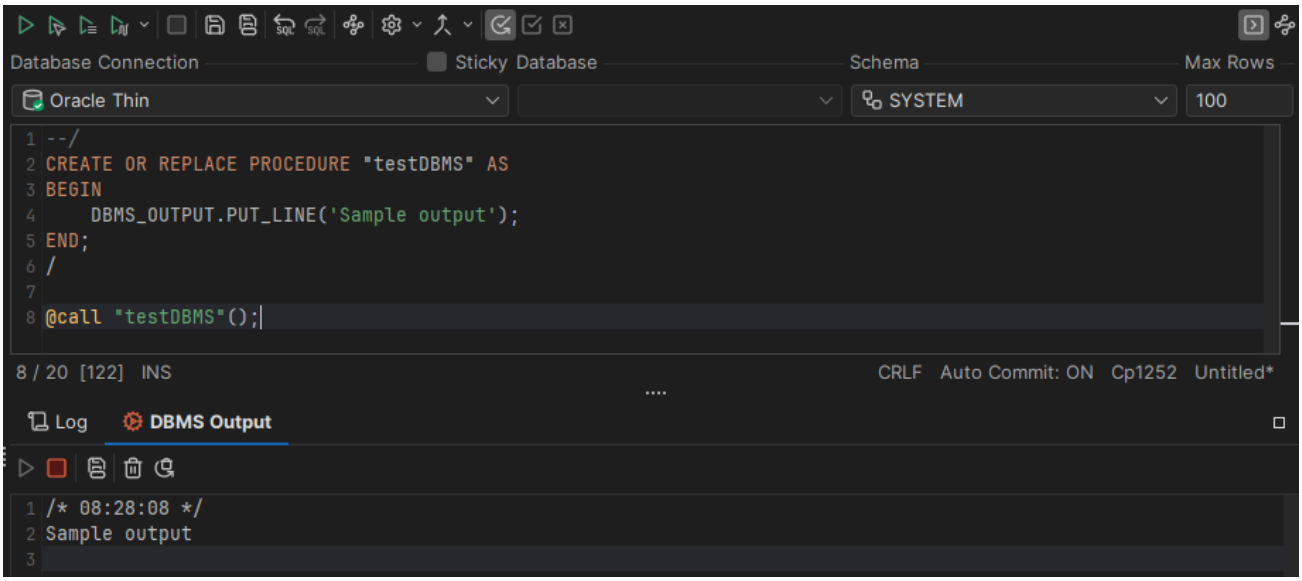
# Using the DBMS Output Tab

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The **DBMS Output** tab is only available for Oracle and Db2 LUW databases. It is used to capture messages produced by stored procedures, packages, and triggers using the DBMS_OUTPUT utility. These messages are typically inserted in the code for debugging purposes. For SQL*Plus users, the corresponding feature is enabled via the **set serveroutput on** command. To enable display of DBMS messages in DbVisualizer,

1. Select the **DBMS Output** tab
2. Press the **Enable** button.

Once DBMS output is enabled, the icon in the tab header is changed. Invoking a stored procedure that produces messages in the SQL Commander results in content similar to this in the output tab. Each block of output is separated with a timestamp.




> If you never use the DBMS Output tab and want to preserve screen real estate, you can select to not show it in the connection properties, in the SQL Commander category.

# Comparing SQL Scripts

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

You can compare a script in one SQL Commander editor to a scripts in another SQL Commander editor, or to the original file loaded into the editor.

- To compare the script to another script, select **Compare** from the right-click menu in one editor and pick the SQL Commander holding the other script in **Select Object** dialog to compare their text content.
- To compare the script to the original file, select **Compare to Saved** from the right-click menu.

# Using Permissions in the SQL Commander

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

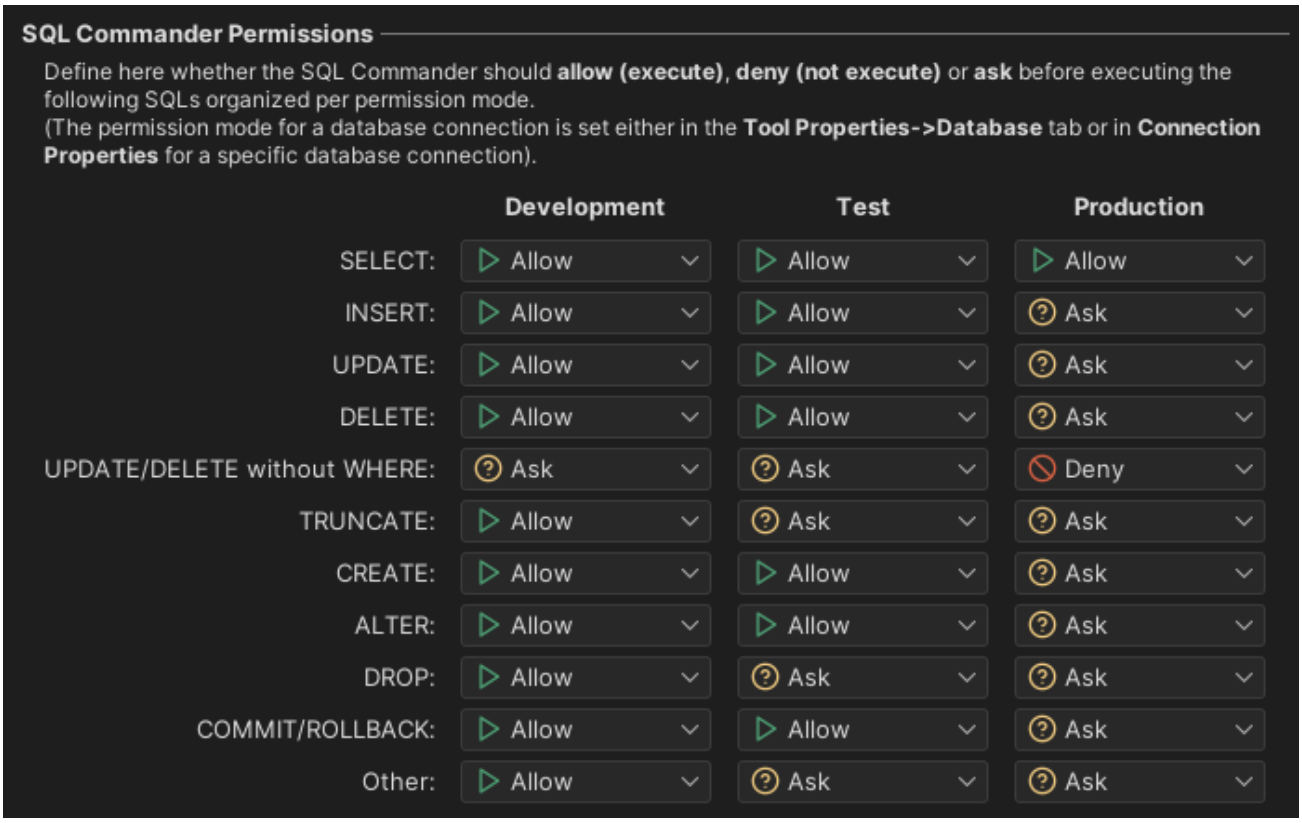
The **Permission** functionality is a security mechanism, where you can specify that certain database operations must be confirmed. You configure permissions in the Tool Properties dialog, in the **Permissions** category of the General tab, per connection mode (Development, Test and Production).

You specify which connection mode to use for a connection in the **Properties** tab of the Object View tab for the connection. By default a connection mode is specified to be Development.

> The permission feature is part of DbVisualizer and does not replace the authorization system in the actual database.

For the SQL Commander, you can pick the permission mode type from a drop-down list for each SQL command:

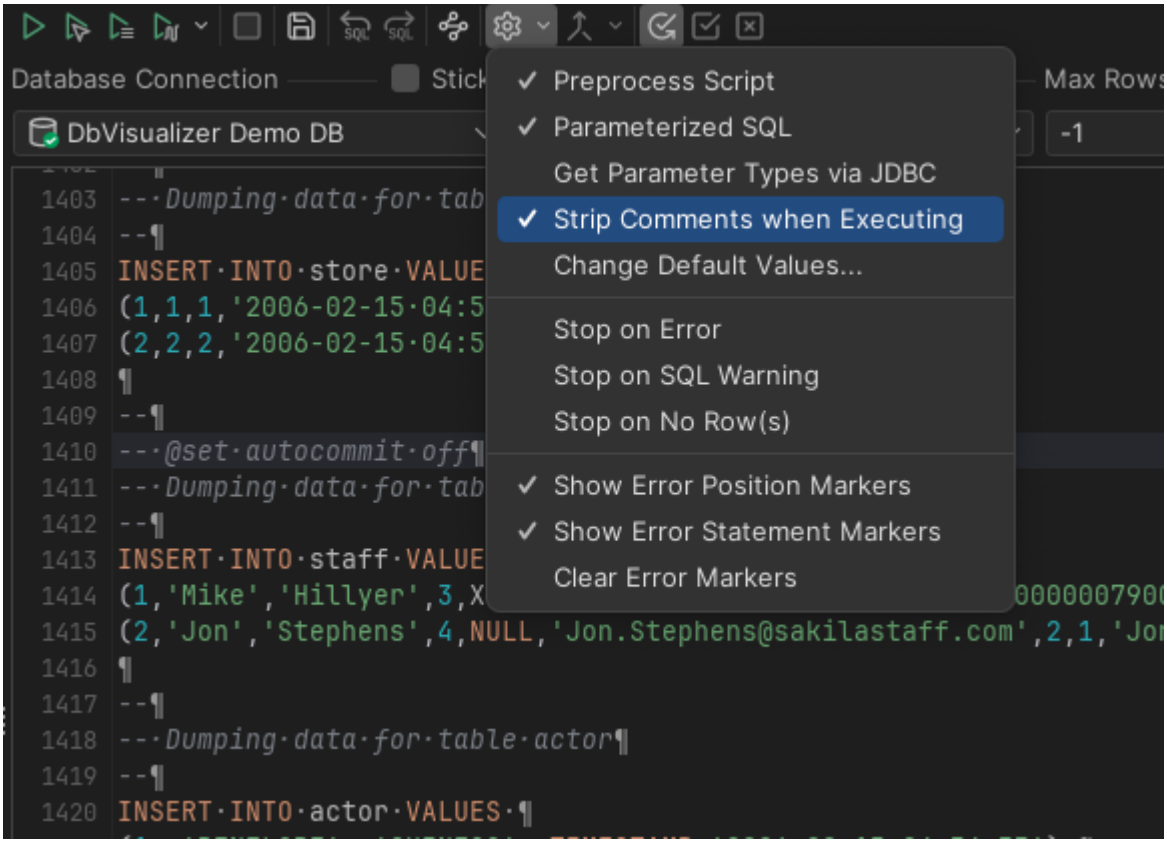
| Permission Type | Description |
|---|---|
| **Allow** | This permission type means that you can run the SQL statement without any confirmation |
| **Deny** | This permission type means that the SQL statement is not executed at all. |
| **Ask** | This permission type means that when executing an SQL statement, or a script of statements, the SQL Commander asks you whether the actual SQL command(s) should be executed. |




Note also the **Connection Permissions** in Tool Properties, in the **Permissions** category where you can specify if DbVisualizer should launch a confirm dialog when connection to the database when running a script.

# Comments in the a Script

Comments in an SQL script, identified by the delimiters defined in **Tools → Tool Properties** and in the **General / SQL Commander / Comments** category, are sent to the database by default. In some cases, you may not want to include the comments with the statement, for instance if your database does not handle them. You can then enable the stripping of comments using the **Strip Comments when Executing** toggle item in the **SQL Commander Options** menu button in the SQL Commander toolbar or in the **SQL Commander → SQL Commander Options** main menu.



# Using Client-Side Commands

## Using Client-Side Commands

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

### Introduction

A client-side command is a DbVisualizer specific command prefixed with the at-sign ( `@` ). These commands are used in SQL scripts executed either in the SQL Commander in the DbVisualizer UI or in the command-line interface, dbviscmd. Client-side commands are used to control the execution of scripts and run, for example, export and import functionality without needing to interact with the DbVisualizer UI. The `@mail` command sends emails with support for attachments, useful when using `@export` and then to mail the result of a query.

### Commands reference

You can use these DbVisualizer client-side commands in your SQL scripts. (These commands are processed by DbVisualizer and never sent to the database as-is).

| Command | Description |
|---|---|
| `@beep` | Use this command to emit a beep sound, e.g. to indicate a significant point in a script. |
| `@call` | Use this command to execute a function or procedure. |
| `@command ignore` | Use this command to ignore executing any following client-side commands. Example:<br>`@echo I will run;`<br>`@command ignore;`<br>`@echo I will not run;` |
| `@command resume` | Use this command to resume execution of following client-side commands. Example:<br>`@echo I will run;`<br>`@command ignore;`<br>`@echo I will not run;`<br>`@command resume;`<br>`@echo I will also run;` |
| `@commit` | Use this commands to run commit updates. |
| `@continue on error` | Continue execution on error. |
| `@continue on norows` | Continue execution on norows. |
| `@continue on sqlwarning` | Continue execution on sqlwarning. |
| `@echo <text>` | Use this command to write to the Log tab. |
| `@rollback` | Use this command to run rollback updates. |
| `@sleep <milliseconds>` | Use this command to halt processing the specified number of milliseconds. |
| `@stop now` | Stop the execution of the script. |
| `@stop on error` | Stop execution on error. |
| `@stop on norows` | Stop execution on norows. |
| `@stop on sqlwarning` | Stop execution on sqlwarning. |

SCRIPT CONTROLS

| Command | Description |
| --- | --- |
| `@delimiter <new_delimiter>` | Use this command to temporarily change the statement delimiter for a complex statement. |
| `@log <file_name> [spool \| close]` | Use this command to write log messages to a named file. Examples:<br>`@log /tmp/myLog.txt;`<br>`@log /tmp/myLog.txt close;`<br>`@log /tmp/allLog.txt spool;`<br>1. Logs subsequent commands to the file /tmp/myLog.txt. If we are already logging to another file as the result of a previous @log command that log is closed<br>2. Stops logging to the file<br>3. Writes all log messages up to this @log command to the named file |
| `@set autocommit off` | Use this command set Auto Commit off. |
| `@set autocommit on` | Use this command set Auto Commit on. |
| `@set dryrun [off]` | Execute a script or part of a script without actually executing the any commands. As an example below no actual sleep will be done nor will the **select** be executed.<br>`@set dryrun;`<br>`@sleep 2000;`<br>`select * from Actor;`<br>`@set dryrun off;`<br>The command is primarily used to debug scripts and check that script can run. An exception to this is when used in connection with client-side import commands. |
| `@set maxchars <number>` | Set Max Chars to limit the number of characters for text data columns. |
| `@set maxrows <number>` | Set Max Rows to limit the number of rows fetched for result sets. |
| `@set serveroutput off` | Use this command to disable output to the DBMS Output tab for databases supporting the DBMS package. |
| `@set serveroutput on` | Use this command to enable output to the DBMS Output tab for databases supporting the DBMS package. |

METADATA INFORMATION

| Command | Description |
| --- | --- |
| `@ddl <params>` | Use this command to get the DDL for a database object. |
| `@describe <table>` | Use this command to show column information for a table. The table name may be qualified with a schema and/or database name. |

RESULT SET PRESENTATION

| Command | Description |
| --- | --- |
| `@set resultset name <name>` | Use this command to name any following result set. When the SQL statement is executed, the tab showing the result set is named using the supplied <name> parameter. Example:<br>`@set resultset name MyActors;`<br>`select * from ACTOR;`<br>DbVisualizer will for the example above show the result of the select in a tab named **MyActors** |
| `@window iconify` | Use this command to lower (iconify) the DbVisualizer main window. |
| `@window restore` | Use this command to raise (restore) the DbVisualizer main window. |

RUNNING FILES AND EXPORTING DATA

| Command | Description |
| --- | --- |
| `@cd <directory>` | Use this command to change directory for any following file related command. execute an external script. |
| `@export` | Use this command to export the result of a query. |
| `@import` | Import table data from CSV, Excel, and TEXT (fixed width column) files |
| `@mail` | Use this command to send emails and attach files. |
| `@open <file> \| <url> [RunOnStop]` | Use this command to open the specified file in the associated tool. (Not supported in the command-line interface, dbviscmd). |
| `@run <file> [<variables>]` | Use this command to run the SQL script in the file execute an external script. |

# @export - Export query result

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

Instead of viewing and exporting query results from Result Set grids, you can export the result of one or more queries to a file from a script. For huge results, this may be the preferred choice due to memory constraints.

## Introduction

To do a simple export of the result from a query, create a script as:

1. an `@export on` command,
2. an `@export set` command,
3. one or more queries,
4. an `@export off` command.

Here is a basic example:

```
@export on;
@export set filename="c:\backups\orders.csv";
select * from Orders;
@export off;
```

The `@export set` command is used to define how the exported output should be generated. It reads parameters in the following syntax:

```
@export set parameter1="value" parameter2="value" ...;
```

The parameter name is case-insensitive, and the value is also case-insensitive for parameters requiring a defined value as per the parameter descriptions below. The supported parameters is based on the `format` parameter for the output which can be any of:

- `format="CSV"` (default)
- `format="HTML"`
- `format="XML"`
- `format="SQL"`
- `format="TXT"`
- `format="Excel"`
- `format="JSON"`

## Parameters for `@export set`

The following chapters lists all supported parameters.

These parameters are valid for all output formats.

| Parameter | Default | Valid Values |
| --- | --- | --- |
| AppendFile | false | • true<br>• false<br>• clear (start with a new file for the first result and then append following result sets to it)<br>Note that when `format="Excel"` and appending to an existing excel file any manually added formatting in the old sheets will be lost. |
| BinaryFileDir | | Path for data files when `BinaryFormat="File"`, see Exporting a Table for details. Note: Variables for column names must include the `scope` variable option when entered manually into a script, e.g. `C:\tmp\${dbvis-date}$\$` `{COUNTRIES||||||scope=post}$` |
| BinaryFormat | Don't Export | • **Don't Export**<br>• **Size**<br>• **Value**<br>• **Hex**<br>• **Base64**<br>• **File** |
| BooleanFalseFormat | false | • **false**<br>• **no**<br>• **0**<br>• **off**<br>(you can specify custom entries separated by comma) |
| BooleanTrueFormat | true | • **true**<br>• **yes**<br>• **1**<br>• **on**<br>(you can specify custom entries separated by comma) |
| CLOBFileDir | | Path for data files when `CLOBFormat="File"`, see Exporting a Table for details. Note: Variables for column names must include the `scope` variable option when entered manually into a script, e.g. `C:\tmp\${dbvis-date}$\$` `{COUNTRIES||||||scope=post}$.clob` |
| CLOBFormat | Don't Export | • **Don't Export**<br>• **Size**<br>• **Value**<br>• **File** |

| Parameter | Default | Valid Values |
| --- | --- | --- |
| DateFormat | yyyy-MM-dd | See valid formats in Changing the Data Display Format document |
| DecimalNumberFormat | Unformatted | See valid formats in Changing the Data Display Format document |
| DecimalNumberSeparator | . | The Decimal separator character to use. Note: this setting have no effect when `DecimalNumberFormat` or `NumberFormat` is `Unformatted`. |
| Destination | File | • **File**<br>• **Clipboard**<br>• **SQL Commander** |
| Encoding | UTF-8 | Check supported encodings for all encodings. (Use the encoding in the **Canonical Name for java.nio API** column). |
| Filename | | The output file name for exported file. This parameter is required if `Destination="file"`. If setting a relative filename the output path depends on the current working folder set by any `@cd` command. |
| Format | Based on file extension, or CSV if not specified. | • **CSV**<br>• **HTML**<br>• **XML**<br>• **SQL**<br>• **TXT**<br>• **Excel**<br>• **JSON**<br>If Format is not specified, the file extension is used to determine the format. If there is no recognized file extension, CSV is used as the default. |
| NumberFormat | Unformatted | See valid formats in Changing the Data Display Format document. |
| NumberGroupingSeparator | , | The number grouping separator to use. Example using `NumberFormat="#,###"` and `NumberGroupingSeparator="."` the formatting of 2000 will produce the result 2.000. If using `NumberGroupingSeparator="space"` the result will be 2 000. **Note:** this setting have no effect when `DecimalNumberFormat` or `NumberFormat` is `Unformatted`. |
| QuoteAllValues | false | • **true**<br>• **false** |
| QuoteDuplicateEmbedded | true | • **true** |

| Parameter | Default | Valid Values |
| --- | --- | --- |
| | | • **false**<br>(quote char is the same as QuoteTextData) |
| QuoteTextData | None | • **None**<br>• **Single**<br>• **Double** |
| Settings | | The path to an XML file containing all settings |
| ShowNullAs | (null) | The text representation for null values. |
| TextFunction | | • **Trim**<br>• **Right Trim**<br>• **Left Trim**<br>Applies the selected trim function to all text columns in the export. To trim specific columns only, use SQL functions like TRIM() directly in your SQL statement. |
| TimeFormat | HH:mm:ss | See valid formats in Changing the Data Display Format document |
| TimeStampFormat | yyyy-MM-dd HH:mm:ss.SSSSSS | See valid formats in Changing the Data Display Format document |

| Parameter | Default | Valid Values |
|---|---|---|
| CsvColumnDelimiter | \t (TAB) | The delimiter between columns. In addition to literals Unicode Code Points such as `\u2656` can be used. |
| CsvColumnHeaderIsColumnAlias | true | • **true**<br>• **false** |
| CsvIncludeColumnHeader | true | • **true**<br>• **false** |
| CsvIncludeColumnHeaderPerResult | false | • **true**<br>• **false**<br>(adds separate column headers for each query e.g. when you have several selects with different structure of the result sets) |
| CsvIncludeSQLCommand | Don't Include | • **Don't Include**<br>• **Top**<br>• **Bottom** |
| CsvRemoveNewlines | false | • **true**<br>• **false** |
| CsvRowCommentIdentifier | | |
| CsvRowDelimiter | \n | • `\n` (Linux/macOS)<br>• `\r\n` (Windows)<br>(you may set this to any literal) |
| CsvSplitFileSize | -1 | Split the result over multiple files if it is larger than the specified size, or -1 to never split.<br>The size must be specified as: `size` [ `K` \| `M` \| `G` ] |

HTML PARAMETERS

| Parameter | Default | Valid Values |
|---|---|---|
| HtmlColumnHeaderIsColumnAlias | true | • **true**<br>• **false** |
| HtmlConvertChars | true | • **true**<br>• **false**<br>Set to false if you have HTML code in the exported data, so that e.g. `<` and `>` characters are not converted to `<` and `>` |
| HtmlFooter | [ Generated by: DbVisualizer version ] | Any text to use in the document footer. Can be set to blank to remove the footer. |
| HtmlIncludeSQLCommand | false | • **true**<br>• **false** |
| HtmlIntroText | | Any description |
| HtmlPerTableHeader | | HTML code that describes the table. To fit into the rest of the HTML code, it must start with `<tr>` and end with `</tr>`. The pre-defined DbVisualizer variables can be used, e.g. `${dbvis-object}$` to include the table name. |
| HtmlPerTableHeaderGridExport | | When using the `append` parameter, use this paramater to set a header for each result set that is exported. |
| HtmlTitle | | The title of the HTML page. |

XML PARAMETERS

| Parameter | Default | Valid Values |
|---|---|---|
| XmlColumnHeaderIsColumnAlias | true | • **true**<br>• **false** |
| XmlIncludeSQLCommand | false | • **true**<br>• **false** |
| XmlIntroText | | Description of the XML. |
| XmlStyle | DbVisualizer | • **DbVisualizer**<br>• **XmlDataSet**<br>• **FlatXmlDataSet** |

| Parameter | Default | Valid Values |
| --- | --- | --- |
| SqlAfterExportStmts | | Any statements to include in the script after the SQL statements for the exported objects, e.g. `set foreign_key_checks = 1;` |
| SqlBeforeExportStmts | | Any statements to include in the script before the SQL statements for the exported objects, e.g. `set foreign_key_checks = 0;` |
| SqlBeginIdentifier | | Character to use to begin a quoted identifier. Note: To specify a double-quote, you must duplicate it since double-quote is also used to enclose the parameter value. |
| SqlBlockBeginDelim | | String to use to begin an SQL block when exporting complex DDL statements using the @ddl command. |
| SqlBlockEndDelim | | String to use to end an SQL block |
| SqlDelimitedIdentifiers | true | • **true**<br>• **false** |
| SqlEndIdentifier | | Character to use to end a quoted identifier. Note To specify a double-quote, you must duplicate it since double-quote is also used to enclose the parameter value. |
| SqlGenCreateWithStorage | false | • **true**<br>• **false**<br>Only applicable for Db2 and Cassandra. |
| SqlGenerateSqlMode | false | • **true**<br>• **false**<br>Only applicable for MariaDB and MySQL. |
| SqlGroupBy | Object | • **Object**<br>• **Statement**<br>Set to **Object** to generate `DROP`, `CREATE`, `INSERT`, and `ALTER` statements (where applicable) for each exported object in turn.<br>Set to **Statement** to group all statements of the same type together, e.g. first `DROP` statements for all exported objects, then `CREATE` statements for all exported objects, etc. |
| SqlIncludeAutoGeneratedValues | true | • **true** |

| Parameter | Default | Valid Values |
| --- | --- | --- |
| | | • **false**<br>Set to false to exclude columns declared as `AUTO_INCREMENT` or `IDENTITY` in the `INSERT` statements. |
| SqlIncludeCreateDDL | false | • **true**<br>• **false** |
| SqlIncludeSQLCommand | Don't Include | • **Don't Include**<br>• **Top**<br>• **Bottom** |
| SqlMergeStatement | false | • **true**<br>• **false** |
| SqlMergeStatementMatchColumns | | A comma-separated list of columns to use when matching rows |
| SqlMergeTemplate | | • **single-merge-sql92**<br>• **single-merge-oracle**<br><br>The following show the formats and the `MERGE` syntax that is generated.<br>• **single-merge-sql92**<br><br>`MERGE INTO T1 A USING (SELECT 1 AS C1, 'abc' AS C2, 'def' AS C3) B ON (A.C1 = B.C1) WHEN NOT MATCHED THEN INSERT (C1, C2, C3) VALUES (B.C1, B.C2, B.C3) WHEN MATCHED THEN UPDATE SET C2 = B.C2, C3 = B.C3;` |

| Parameter | Default | Valid Values |
| --- | --- | --- |
| | | • **single-merge-oracle**<br><br>`MERGE INTO T1 A USING (SELECT 1 AS C1, 'abc' AS C2, 'def' AS C3 from dual) B ON (A.C1 = B.C1) WHEN NOT MATCHED THEN INSERT (C1, C2, C3) VALUES (B.C1, B.C2, B.C3) WHEN MATCHED THEN UPDATE SET C2 = B.C2, C3 = B.C3;` |
| SqlMultiRowInsert | false | • **true**<br>• **false**<br><br>If **true** Multiple rows are included in each `INSERT` statement. The number of rows to include in each `INSERT` statement is limited by the `SqlMultiRowInsertLimit` parameter. |
| SqlMultiRowInsertLimit | 10 | The maximum number of rows to include in a multirow `INSERT`. |
| SqlMultiRowInsertTemplate | multi-insert-sql92 | The following show the formats and the `INSERT` syntax that is generated for one block having the `SqlMultiRowInsertLimit="2"` parameter set.<br><br>• **multi-insert-sql92**<br><br>`INSERT INTO TAB (ID, FI, LA) VALUES (1, 'abc', 'def'), (2, 'abc', 'def');`<br><br>• **multi-insert-cassandra** |

| Parameter | Default | Valid Values |
| --- | --- | --- |
| | | <pre>--/<br>BEGIN BATCH<br>INSERT INTO TAB (ID,<br>FI, LA) VALUES (1,<br>'abc', 'def');<br>INSERT INTO TAB (ID,<br>FI, LA) VALUES (2,<br>'abc', 'def');<br>APPLY BATCH;</pre><br>• **multi-insert-oracle**<br><pre>INSERT ALL<br>INTO TAB (ID, FI,<br>LA)<br>VALUES (1, 'abc',<br>'def')<br>INTO TAB (ID, FI,<br>LA) VALUES (2,<br>'abc', 'def')<br>SELECT 1 FROM dual;</pre><br>• **multi-insert-union**<br><pre>INSERT INTO TAB (ID,<br>FI, LA)<br>SELECT 1, 'abc',<br>'def' UNION ALL<br>SELECT 2, 'abc',<br>'def';</pre> |
| SqlQualifier | | Qualifier to use when qualifying table names. If not set, DbVisualizer tries to determine the schema and use it as the qualifier. |
| SqlQualifyColumnName | false | • **true**<br>• **false** |
| SqlQualifyObjectName | true | • **true**<br>• **false** |
| SqlRowCommentIdentifier | -- | |
| SqlSeparator | ; | Statement separator character. |
| SqlSplitFileSize | -1 | Split the result over multiple files if it is larger than the specified size, or -1 to never |

| Parameter | Default | Valid Values |
|---|---|---|
| | | split.<br>The size must be specified as: `size` [ `K` \| `M` \| `G` ] |
| TableName | | Can be set if DbVisualizer cannot determine the value for the `${dbvis-object}$` variable |

TXT PARAMETERS

| Parameter | Default | Valid Values |
|---|---|---|
| TxtAlignNumbers | left | • **left**<br>• **right** |
| TxtColumnDelimiter | \t (TAB) | The delimiter between columns. In addition to literals it can also be specified using Unicode Code Points as `\u2656`. |
| TxtColumnHeaderIsColumnAlias | true | • **true**<br>• **false** |
| TxtIncludeColumnHeader | true | • **true**<br>• **false** |
| TxtIncludeColumnHeaderPerResult | false | • **true**<br>• **false** |
| TxtIncludeSQLCommand | Don't Include | • **Don't Include**<br>• **Top**<br>• **Bottom** |
| TxtRemoveNewlines | false | • **true**<br>• **false**<br><br>Removes newlines when formatting text. This includes when displaying binary values as text. |
| TxtRowCommentIdentifier | | |
| TxtRowDelimiter | \n | • `\n` (Linux/macOS)<br>• `\r\n` (Windows)<br>(you may set this to any literal) |
| TxtSplitFileSize | -1 | Split the result over multiple files if it is larger than the specified size, or -1 to never split.<br>The size must be specified as: `size` [ `K` \| `M` \| `G` ] |
| TxtTruncateLength | -1 | Specifies the length at which truncation of a column will happen. |

EXCEL PARAMETERS

| Parameter | Default | Valid Values |
|---|---|---|
| ExcelAutoResizeColumns | false | • **true**<br>• **false** |
| ExcelColumnHeaderIsColumnAlias | true | • **true**<br>• **false** |
| ExcelFileFormat | | If `ExcelFileFormat` is not specified, the file extension is used to set this parameter to one of **xls** for binary (or legacy) Excel max 65 535 rows, or **xlsx** which is the current and recommended format. |
| ExcelIncludeColumnHeader | true | • **true**<br>• **false** |
| ExcelIncludeSQLCommand | false | • **true**<br>• **false** |
| ExcelIntroText | | A description of the Excel document. |
| ExcelSheetName | | Used when exporting to excel. Sets the name of exported excel sheet. |
| ExcelTextDateTime | true | • **true**<br>• **false**<br>Convert date, time and timestamp data to text in the Excel file if true. |
| ExcelTextOnly | false | • **true**<br>• **false**<br>Convert numeric values to text in the Excel file if true. |
| ExcelTitle | | Any title |

JSON PARAMETERS

| Parameter | Default | Valid Values |
|---|---|---|
| JsonColumnHeaderIsColumnAlias | true | • **true**<br>• **false** |
| JsonSplitFileSize | -1 | Split the result over multiple files if it is larger than the specified size, or -1 to never split.<br>The size must be specified as: `size` [ `K` \| `M` \| `G` ] |
| JsonStyle | Array | • **Array**<br>• **Rows** |

Here are a few examples using some of these settings.

### Automatic table name to file mapping

This example shows how to make the filename the same as the table name in the select statement. The example also shows several select statements. Each will be exported in the SQL format. Since the filename is defined to be automatically set, this means that there will be one file per result set and each file is named by the name of its table.

```
@export on;
@export set filename="C:\backups\${dbvis-object}$" format="sql";
select * from Orders;
select * from Products;
select * from Transactions;
@export off;
```

> There must be only one table name in a select statement to automatically set the filename with the `${dbvis-object}$` variable, i.e., if the select joins from several tables or pseudo tables are used, you must explicitly name the file.

The `${dbvis-object}$` variable is not substituted with a table name if using the `AppendFile="true/clear"` parameter.

### Multiple results to a single file

This example shows how all result sets can be exported to a single file. The `AppendFile` parameter supports the following values.

- **true**
  The following result sets will all be exported to a single file
- **false**
  Turn off the append processing
- **clear**
  Same as the true value but this will in addition clear the file before the first result set is exported

```
@export on;
@export set Format="sql" Filename="C:\Backups\alltables.sql" AppendFile="clear";
select * from Orders;
select * from Products;
select * from Transactions;
@export off;
```

## Using predefined settings

If you save settings when exporting a table or a schema, you can use the **Settings** parameter to reference the settings file.

```
@export on;
@export set Settings="C:\tmp\htmlsettings.xml" Filename="C:\backups\${dbvis-
object}$";
select * from Orders;
select * from Products;
select * from Transactions;
@export off;
```

## Limit the number of exported rows

You can use the `@set maxrows` command to override the Max Rows field value specified in the SQL Commander tab toolbar.

```
@set maxrows 10;
@export on;
@export set Format="sql" Filename="c:\Backups\alltables.sql";
select * from Orders;
select * from Products;
select * from Transactions;
@export off;
```

`@set maxrows="-1";` will export all rows.

## Other Ways to Export Table Data

- Export all or selected tables with the Export Schema or Export Table assistant
- Export query results by exporting the grid with the query results

# @mail - Send emails and attach files

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

## Mail with DbVisualizer

The `@mail` client side command is used to send emails from scripts executed in the SQL Commander and the command-line interface, `dbviscmd`. It offers full email setup including attaching multiple files and embedding data from files in the body of the email. There is also the capability to group settings for a mail server setup in a **Mail Server Account** enabling a simple reference for a mail server definition in the `@mail` command rather than a lot of parameters.

Here is a basic example with mail server parameters:

```
@mail To="<recipient>"
      Subject="<subject>"
      Body="<body>"
      MailServerHost="smtp.gmail.com"
      MailServerPort="587"
      MailServerUser="<user>"
      MailServerPassword="<password>";
```

| Parameter | Default | Values |
| --- | --- | --- |
| Subject | | The subject. |
| To | | Defining the recipients of the mail. Multiple email addresses may be specified separated with comma: E.g.: John <john.doe@nowhere.com>, jane.doe@nowhere.com. |
| Cc | | Defining Cc recipients of the mail. |
| Bcc | | Defining Bcc recipients of the mail. |
| From | Same as **MailServerUser** | This is by default the same address specified in MailServerUser. The mail server may reject sending from another email address depending how it is configured. |
| ReplyTo | Same as **From** (if specified) or **MailServerUser** | An address to which replies should be sent. |
| Body | | The mail message body. Read more in Body parameter. |
| FileSource | | File attachment parameter Should specify the path to the file being attached. The path can be absolute or relative to the working directory: E.g: `FileSource="/tmp/file.txt"` Read more in FileSource parameter. |
| FileSource_AttachType | Attach | Specifies how files are attached. Possible values:<br>• **Attach**<br>Add the file as an attachment<br>• **Inline**<br>Add the file as inline<br>• **Embed**<br><br>Note that it is not supported to specify inline attachments and message body in the same email. Since an inline attachment and body text can not be specified at the same time, DbVisualizer adds support to include text file content in the Body of the email from multiple files. By setting the `AttachType="Embed"` parameter DbVisualizer will not also attach the file. More about this in the section about specifying message content (Body). |
| FileSource_ContentType | | Optional, determined by the type of file attached. |
| MailHeaders | | For adding mail headers to the mail. E.g: `MailHeaders="X-Priority=""1"""` The above mail header will result in a mail marked as important. For information about this and other headers please use online resources as the interpretation of mail headers may differ between different mail clients. |

| Parameter | Default | Values |
| --- | --- | --- |
| MailTemplate | | Offers the possibility to use a file as a template for the mail. Read more in MailTemplate parameter. |

MAIL SERVER CONNECTION PARAMETERS

| Parameter | Default | Values |
| --- | --- | --- |
| MailServerHost | | The host name or IP address of the outgoing mail server. |
| MailServerPort | | The port number of the outgoing mail server. |
| MailServerUser | | The user account for the outgoing mail server. |
| MailServerPassword | | The user account password for the outgoing mail server. Note that the password is in plain text. The solution is to setup a mail server account. |
| MailServerSecurity | None | For connecting to the mail server securely. Possible values are:<br>• **None**<br>Secure connection not enabled<br>• **STARTTLS**<br>Use SSL |
| MailServerProperties | | Comma separated list of properties sent to the mail server. Eg.: `MailServerProperties="mail.smtp.startt ls.enable=""true"" mail.smtp.auth=""true"""` Note the double-quotes around the values. I.e., `=""true""` is correct. Using `="true"` will render an error. |
| MailServerAccount | | Refers to the name of a mail server account defined in **Tools → Tool Properties** and in the **General / Mail Server Account** category. By defining a mail server account, **MailServerAccount** is the only parameter you will need when using `@mail`. Read more in Defining Mail Server Accounts. |

## FileSource parameter - attaching files

The FileSource parameter should identify the path to an absolute or relative file that will be attached to the email. Files may be attached as a regular attachment, inline (for content and mail clients supporting it), and embedded as plain text (useful for text data).

Example of basic use resulting in the files being attached:

```
FileSource="C:\Data\file1.csv" FileSource_AttachType="Embed"
FileSource1="C:\Data\file2.xlsx"
FileSource2="C:\Data\file3.html" FileSource2_AttachType="Attach";
```

Restrictions with `AttachType:

- If using `AttachType="Inline"` there must only be a single file attached. If multiple files, all of them are sent as attachments.
- Having Body specified and `AttachType="Inline"` is not allowed. The file will then be attached.
- `AttachType="Inline, Embed"` is not valid. The file will then be embedded.
- A text file may both be attached and embedded in the body of the email, use `AttachType="Attach, Embed"`.

One more FileSource related parameter is the `FileSource_ContentType`. Use this to specify a MIME content type representing the file being attached. E.g.

- `FileSource_ContentType="image/png"`
- `FileSource_ContentType="text/html"`
- `FileSource_ContentType="application/vnd.ms-excel"`

**For embedded files, check the next chapter for information on how to specify where in the body content it should be inserted.**

## Body parameter

The Body parameter specifies the plain text content of the mail. With variables (placeholders) any files with `AttachType="Embed"` can be specified which when the email is sent is replaced with the file content. In addition, there is a collection of variables that can be included in the body related to the script execution. Read more about these in Expanding variables in the Body.

Example:

```
FileSource="C:\Temp\sales.csv"
FileSource_AttachType="Embed"
Body="Dear Mgmt,

Please find the requested product information:

{{FileSource}}

/Staff"
```

Sample output:

```
Dear Mgmt,

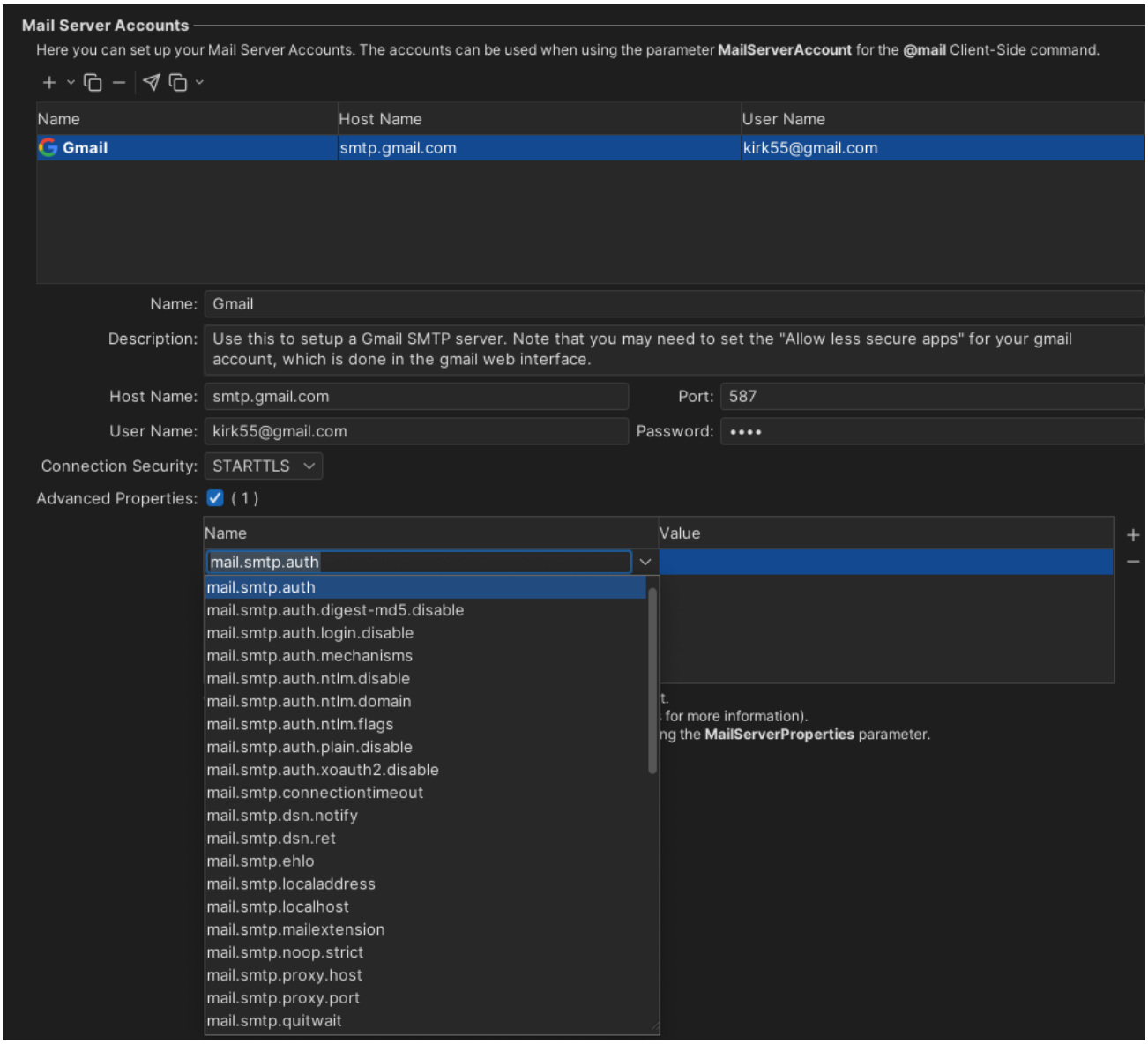
Please find the requested product information:

id  brand   length  width   hp  weight
1   Audi A4 48      76      28  44
2   Audi A6 96      6       51  86

/Staff
```

## Defining Mail Server Accounts

Sending an email in DbVisualizer requires mail server related information that includes userid, password, server name, port, etc. These parameters are prefixed `MailServerXXX` for the `@mail` command. Instead of specifying these as parameters, a **Mail Server Account** can be defined in **Tools → Tool Properties** and under the **Mail Server Accounts** category.




Use the left most button **Add new ...** to add a new account. Note choosing one of the top ones from the menu will create a new account from a named template. The template contains predefined values for well-known mail servers. E.g., the one named **Gmail** defines a temple for accessing an existing gmail account. Unless Google changes connection data for gmail, the only change that would be required is changing the **User Name** and **Password**.

The data entered for Mail Server Account are

| Field | Description | @mail parameter sample |
|---|---|---|
| **Name** | The name of the account. Example: "**My Gmail**". | `MailServerAccount="My Gmail"` |
| **Description** | Optional description of the mail server account | |
| **Host Name** | The name or IP address of the mail server. Example: "**smtp.gmail.com**". | `MailServerHost="smtp.gmail.com"` |
| **Port** | The port number of the mail server. Example: "**587**". | `MailServerPort="587"` |
| **User Name** | The mail account user name. Example: "**someuser@gmail.com**". | `MailServerUser="someuser@gmail.com"` |
| **Password** | The mail account password. Example: "**MySecret**". | `MailServerPassword="MySecret"` |
| **Connection Security** | For connecting to the mail server securely. Example: **STARTTLS** | `MailServerSecurity="STARTTLS"` |

#### Testing a Mail Server Account

By selecting an account in the list and pressing the **mail icon** in the toolbar, it is possible to send test mail.

#### Generating client side commands @mail

Select an account in the list and press the copy icon in the toolbar. Two menu alternatives exist:

- **Copy to clipboard as @mail command with all parameters**
  generates a `@mail` command with all mail server parameters as defined for the mail server account.
- **Copy to clipboard as @mail command with MailAccountServer parameter**
  generates a `@mail` command with only the **MailAccountServer** parameter.

### MailTemplate parameter

The `MailTemplate` parameter is used to specify a template file for the final email and some of the settings such as `To`, `Subject`, `Body`, etc.

Note:

The `Body` parameter must be defined as the last parameter in the template. Please see the section about Specifying Mail Content for a description of how to specify the `Body`.

Any parameters defined for the `@mail` command line will override the ones in the template. Example:

```
@mail MailTemplate="template.txt"
Subject="A better subject"
MailServerAccount="gmail";
```

And the **template.txt** file:

```
To: jane.doe@somewhere.com
Subject: Hey take care of this!
Body: The report
of the last ...
```

The mail will be sent to **jane.doe@somewhere.com** and the subject of the mail will be "Hey take care of this!"

## Usage Examples

SENDING MAIL WITH INLINE ATTACHMENT

The following script exports an HTML file, sends an email with that file attached as inline (most email clients will render the HTML in the body of the email). Note that Body parameter is not defined.

```
@export on;
@export set filename="g_actors.html" Format="HTML";
SELECT * FROM ACTOR WHERE last_name LIKE 'G%';
@export off;

@mail Subject="The actors" To="first.last@somewhere.com"
MailServerAccount="gmail"
FileSource="g_actors.html" FileSource_AttachType="Inline";
```

EXPORTING TO CSV INCLUDING CSV IN MAIL BODY

Please see Export and Mail example.

ALL PARAMETERS ON COMMAND LINE

```
@mail To="someone@somewhere.com"
Subject="The subject"
Body="The body"
MailServerHost="smtp.gmail.com"
MailServerPort="587"
MailServerUser="first.last@gmail.com"
MailServerPassword="password"
MailServerSecurity="STARTTLS";
```

## Expanding variables in the Body

The body of the mail can contain references to placeholders with values supplied by DbVisualizer during execution of the script in which `@mail` is executed. The values are inserted using the syntax: **{{place holder}}**

Example:

```
@mail Body="Hi, Number of records exported where {{dbvis-
current.rowCount}} ......"
```

The above example shows how we are referring the property `dbvis-current.rowCount`. `dbvis-current` refers to the current total result.

Following is a list of variables.

#### VARIABLES KEEPING A RESULT FOR THE LAST EXECUTED STATEMENT

These holds data for the last executed statement before the ´@mail` command.

| Variable/property | Description |
|---|---|
| {{dbvis-last.statusCode}} | The result status code. |
| {{dbvis-last.isError}} | "true" if the last command failed. |
| {{dbvis-last.executionMetrics.elapsed}} | The execution time. |
| {{dbvis-last.executionMetrics.begin}} | Execution start time. |

#### VARIABLES KEEPING SCRIPT EXECUTION RESULT

These holds data up to but not including the `@mail` command in which they are used.

| Variable/property | Description |
|---|---|
| {{dbvis-current.executionCount}} | Number of statements/commands executed. |
| {{dbvis-current.schemaChanged}} | "true" if the schema has changed. |
| {{dbvis-current.catalogChanged}} | "true" if the catalog has changed. |
| {{dbvis-current.updateCount}} | The number of updated records. |
| {{dbvis-current.numberOfResultSetsFetched}} | The number of result sets fetched. |
| {{dbvis-current.rowCount}} | The number of records returned. |
| {{dbvis-current.emptyResultSetsCount}} | The number of empty result sets fetched. |
| {{dbvis-current.successCount}} | The number of statements/commands successfully executed. |
| {{dbvis-current.errorCount}} | The number of statements/commands failed. |
| {{dbvis-current.warningCount}} | The number of statements/commands with result warning. |
| {{dbvis-current.stoppedOnErrors}} | "true" if the execution of the script was stopped on errors. |
| {{dbvis-current.stoppedOnSQLWarning}} | "true" if the execution of the script was stopped because of SQL Warnings. |
| {{dbvis-current.stoppedOnNoRows}} | "true" if the execution of the script was stopped because of result returned no rows. |
| {{dbvis-current.executorWasInterrupted}} | "true" if the execution of the script was stopped due to user interrupt. |
| File attachment related variables | |
| {{FileSource}} | Replaced in runtime with the content of the file FileSource is referring to. |

The use of variables is illustrated in the following example.

### Export and mail Example

```
@log "logfile.txt";

@export on;
@export set filename="g_actors.csv" Format="CSV";
SELECT * FROM ACTOR WHERE last_name LIKE 'G%';
@export off;

@export on;
@export set filename="f_contry.csv" Format="CSV" CsvIncludeColumnHeader="false";
SELECT country FROM COUNTRY WHERE country LIKE 'F%';
@export off;

@mail Subject="Last report" To="first.last@somewhere.com"
MailServerAccount="gmail"
Body="Hi,
The total export took: {{dbvis-current.commandElapsedExecTime}} sec and {{dbvis-
current.rowCount}} records where exported.

Actors with a last name starting with 'G':
{{FileSource}}

Countries starting with 'F':
{{FileSource1}}

The log file is attached.
"
FileSource="g_actors.csv" FileSource_AttachType="Embed"
FileSource1="f_contry.csv" FileSource1_AttachType="Embed"
FileSource2="logfile.txt";
```

# @import - Importing data

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

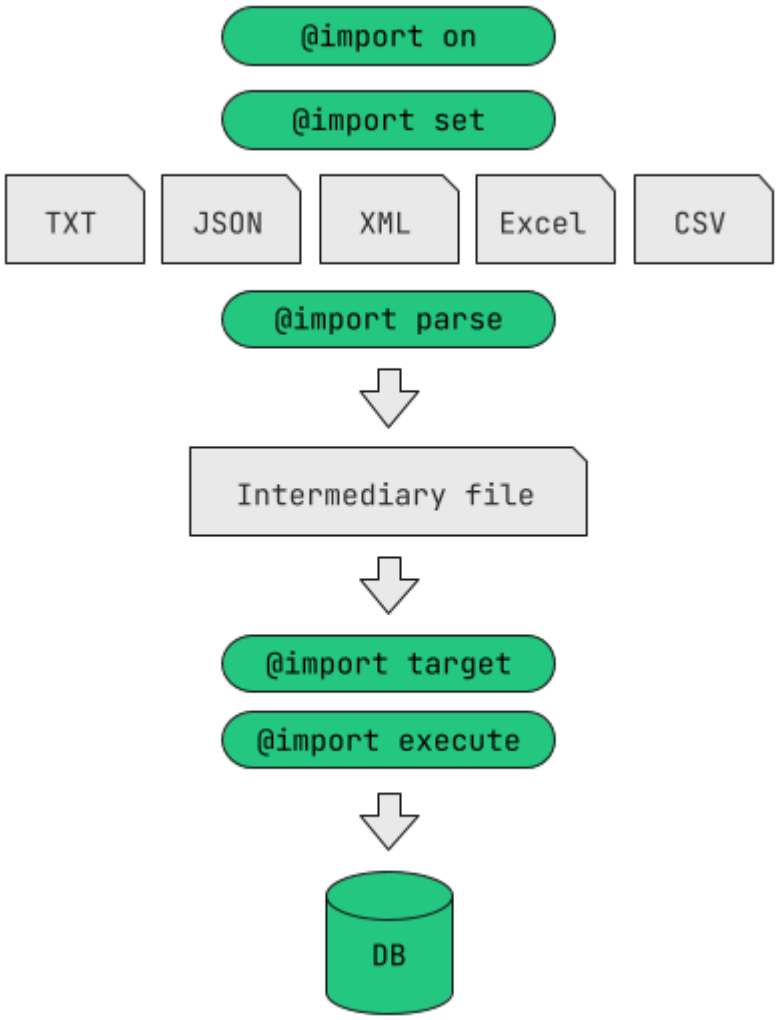
Instead of using the GUI to import data you can use client-side commands to import data, i.e **@import**. This enables you to use the DbVisualizer command-line interface to automate your imports and utilise other client-side commands such as @export, @mail, among others. Import data using the @import command supports the following formats:

- **Excel**
  xlsx or the legacy xls format
- **XML**
  The same XML formats that can be exported with DbVisualizer
- **JSON**
  The same JSON formats that can be exported with DbVisualizer
- **CSV**
  Importing CSV files supports a lot of configurations such as multi-symbol column separator, multi-line values, etc.
- **TXT**
  Importing fixed width text files

These are client-side commands for @import are:

| Command | Description |
|---|---|
| `@import on` | The command starts an import session |
| `@import set` | Set parameters for the import |
| `@import parse` | Parse and convert the source data to an intermediary format, and analyze the data. |
| `@import target` | Identify the target table and what target columns should be used |
| `@import execute` | Runs the export |

The import process is explained by the following figure.




The `@import parse` step (1) lays the foundation for the database import as it based on the source file format (csv, json, xlsx, etc), parses the data to an intermediary and internal file. This file is then analyzed to detect widths, data types and their sizes based on the data, row and cell references back to the source file, and a lot more.

The intermediary format stores the input data as **Records** associated with information from where the data **originates**.

Once the data has been parsed, some basic tests are performed to see if the properties of the data are compatible with the **target table** properties. E.g. if the size of the input data fits the targeted table columns. This is done when the `@import target` command is run (2).

The final step of an import is to **execute** (3-4) the actual operations towards the database to import the data. I.e. **execute** the INSERT statements. Any failures in the import will include specification of where in the source file the invalid data originates.

Following is a complete example where a simple CSV file is imported to a Target table Expenses.

## Importing data - Example

The example shows how data about fruits included in a CSV file **fruits.csv** is imported into a database table **fruits**.

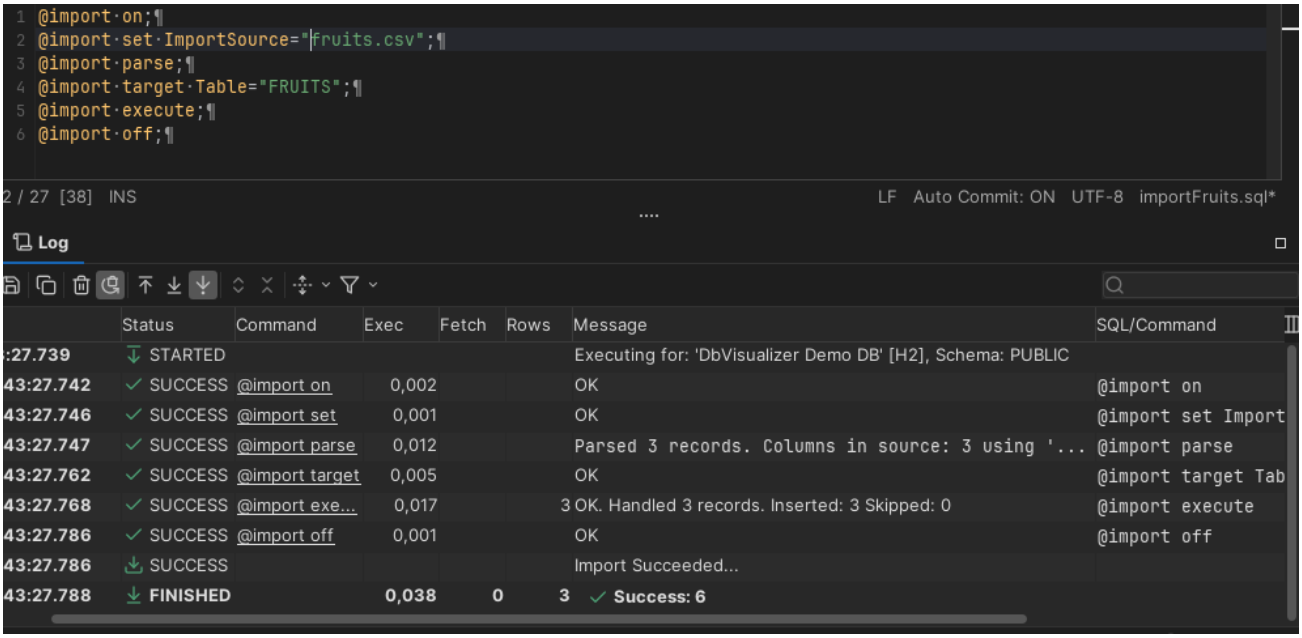
The example shows a minimal example where the columns of the CSV file are mapped directly to the table columns.

```
1,Banana,1.22
2,Apple,0.35
3,Orange,0.55
```

The SQL needed to import the file

```
@import on;
@import set ImportSource="fruits.csv";
@import parse;
@import target Table="FRUITS";
@import execute;
@import off;
```

Running the import script in DbVisualizer




As seen in the screenshot above, 3 records were inserted, and 1 was skipped (the header).

Read further on in the guide for how you can tailor your import using the different parameters of the import commands.

## The @import commands

@IMPORT ON

The `@import on` command initializes the import session. When a client-side command import session is started, an **output folder** is created where DbVisualizer generates data representing intermediate results of the import. The output folder is created under a root folder (importresults) in the DbVisualizer's current workspace. (By default, this is the .dbvis folder in the user's home folder). The name of the folder is generated to be unique.

These folders are automatically cleaned by DbVisualizer regularly.

**Note**: Normally there is no need to specify any parameters to the `@import on` command. Default values should be enough for most uses.

| Parameter | Default | Valid Values |
|---|---|---|
| ImportResultRoot | <WORKSPACE>/importresults | The root folder for parsing results and other temporary files produced by the import session. **Note**: Any results produced in this directory will automatically be removed on regular basis. |
| ImportResultDir | Automatically generated as a sub folder to `WORKSPACE` /importresults | The path to put the DbVisualizer import results. If the path is an absolute path the results are stored in this folder. If it's not an absolute path it is assumed that it's path relative to the `ImportResultRoot`. The folder is created when import is started. If the folder already exists, the import fails unless the **Clean** parameter is also used. If none of the `ImportResultRoot` and `ImportResultDir` is specified DbVisualizer will create a new uniquely named directory for every import session. |
| Clean | false | If true, the `ImportResultDir` is cleaned before import. If false import fails if directory exists. No need of specifying this parameter if `ImportResultDir` is not specified. |
| ContinueImportFile | | Specified if you are continuing an import. See chapter about Continuing an Import. |

**Example 1**

```
@import on ImportResultDir=/tmp/myimport Clean="true";
```

Import results are stored in the folder /tmp/myimport. Any existing folder /tmp/myimport will be deleted/cleaned.

**Example 2**

```
@import on ImportResultDir="newimport";
```

Import results are stored in <WORKSPACE/importresults/newimport. If this directory exists, the import will fail.

@IMPORT SET

The `@import set` command takes a parameter name followed by an equal sign and a value, e.g. `parameter="value"`. You can use the following parameters, where `ImportSource` is the only required parameter. All names are case-insensitive. **Note** that you may use multiple `@import set` commands but the first one must include the parameter `ImportSource` setting the input data to import.

These parameters are valid for all import formats.

| Parameter | Default | Valid Values |
|---|---|---|
| BooleanFalseFormats | false, no, 0, off | Comma separated string of values that should be interpreted as boolean false. |
| BooleanTrueFormats | true, yes, 1, on | Comma separated string of values that should be interpreted as boolean true |
| DateFormat | yyyy-MM-dd | See valid formats in Changing the Data Display Format document. |
| DecimalSeparator | . | The decimal separator to use when parsing decimal numbers |
| Encoding | As set in Tool Properties | The file encoding to use when parsing the file to import |
| ErrorIncludeStackTrace | false | • **true**<br>• **false**<br><br>If **true** the Java stack trace of any exception will be included in error messages. |
| FailOnConvertFailure | false | • **true**<br>• **false**<br><br>If **true**, the Import will fail if data conversion fails. |
| FailOnNoColumnsFoundFailure | false | • **true**<br>• **false**<br><br>If **true**, the Import will fail if we found no columns during continue import |
| FailOnParseFailure | false | • **true**<br>• **false**<br><br>If **true**, the Import will fail if we got a failure during parsing of the source data. |
| Format | | • **CSV**<br>• **XLS**<br>• **XLSX**<br>• **XLSM**<br>• **JSON**<br>• **TXT**<br>• **XML**<br><br>Specify the format of the file to import. Normally the format is auto detected which means that this parameter is optional. |

| Parameter | Default | Valid Values |
| --- | --- | --- |
| HeaderStartRow | 0 | The row index of the row where the header starts. If set to a number **n** the `StartRowOfData` parameter is automatically set to **n + 1.** The default value 0 indicates that the source data has no header information. |
| **ImportSource** | | A path to the file to import. Must be included in the first `@import set` command. The path is an absolute path or a relative path to the script location. If a `@cd` command has been run before the `@import set` command a relative path is relative to the `@cd` directory |
| MaxRows | -1 | The Maximum row to parse/import |
| ShowNullAs | | The value that should be considered as NULL. E.g. **(null)** |
| SkipEmptyRows | true | • **true**<br>• **false** |
| SkipHeader | true | • **true**<br>• **false**<br><br>If set to **false** the header is also imported. |
| SkipRowsStartingWith | | String |
| StartRowOfData | 1 | The row index of the row where data starts. See also `HeaderStartRow`. |
| ThousandSeparator | , | The thousand separator to use. |
| TimeFormat | HH:mm:ss | See valid formats in Changing the Data Display Format document. |
| TimeStampFormat | yyyy-MM-dd HH:mm:ss.SSSSS | See valid formats in Changing the Data Display Format document. |

| Parameter | Default | Valid Values |
|---|---|---|
| CsvColumnDelimiter | | Can be used to override the auto detection by specifying a column delimiter. `CsvColumnDelimiter=";"` The most common column delimiters are auto detected. |
| CsvColumnDelimiterType | Auto Detect | Valid values are: **String** or **Auto Detect**. If only `CsvColumnDelimiter` is specified in the command `CsvColumnDelimiterType` is set to String |
| CsvTextQuotedBetween | None | Having data quoted is needed if the data itself contains the delimiter use to separate columns, special characters such as tabs, multiline separators, etc. The `CsvTextQuotedBetween` parameter is used to specify the identifier which is used to enclose this type of data. Please note that there may not be multiple text quote identifiers in the same import source. Any character is valid to use. These alaises can be used to make the delimiters readable:<br><br>- **Single** same as specifying the value as `"'"`<br>- **Double** same as specifying the value as `""""`.<br>(Note that a `"` character need to be escaped with an additional `"`).<br>- **None** no quote character (default)<br><br>Example 1:<br>`CsvTextQuotedBetween="Single"`<br>Ex: `1 'Apple' 'Round fruit'`<br><br>Example 2:<br>`CsvTextQuotedBetween="Double"`<br>Ex: `2 "Lemon" "Yellow fruit"`<br><br>Example 3:<br>`CsvTextQuotedBetween="\|"`<br>Ex: `2 \|Lemon\| \|Yellow fruit\|`<br><br>If you have quoted data in your import source and do not specify `CsvTextQuotedBetween`, the data will be handled as text data. |

EXCEL PARAMETERS

| Parameter | Default | Valid Values |
|---|---|---|
| ExcelCellPolicyError | SKIP | Defines what to do for a formula cell where the cached value indicates an error. Possible values:<br>• **EMPTY**<br>Set the cell to blank.<br>• **SKIP_ROW**<br>Skip the complete row.<br>• **ERROR**<br>Produce an error for the cell/row.<br>• **TO_STRING**<br>Include the error as data. |
| ExcelSheetId | 0 | An id specifying the sheet id in a workbook. First sheet has **id = 0** Either this parameter or `ExcelSheetName` parameter can be used, not both. |
| ExcelSheetName | | A name specifying the sheet name in a workbook. |

TXT PARAMETERS

| Parameter | Default | Valid Values |
|---|---|---|
| TxtColumns | | Used when fixed columns text files are imported. Example **0, 4**. For detailed syntax of the `TxtTrim` parameter please see the example Importing fixed column width input data |
| TxtTrim | true | If **true**, the data will be trimmed. |

Example 1

```
@import set StartRowOfData="5" SkipRowsStartingWith="//";
```

We are starting to import data from row 5 of the source data file and skipping rows starting with "//".

Example 2

```
@import set HeaderStartRow="1";
```

The input data has header information at row 1. We are starting to import data from row 2. As `StartRowOfData` is not explicitly set is automatically set to 2.

@IMPORT PARSE

This command does all the parsing and analysing of input data.

Example

```
@import parse;
```

The source data file is parsed. The esult is stored in the folder defined by `ImportResultDir`.

Example generated by the `@input parse` command:

```
Parsed 5 records. Columns in source: 2 using ',' delimiter

INDEX NAME     TYPE       NULLABLE FROM ROW
----- -------- ---------- -------- --------
0     NAME     String(6)  No       2
1     BIRTHDAY String(16) No       4

Total bytes: 73 B
```

The information shows the number of parsed records along with the number of columns found. If the parsed file was a CSV file the used delimiter is printed.

For each column the following information is printed:

- **INDEX**
  The index of the column
- **NAME**
  If `@import set` parameter `HeaderStartRow` was specified and header information was extracted the extracted column name is printed.
- **TYPE**
  The type of data found. The size declaration (E.g. **16)** represents the longest string found.
- **NULLABLE**
  If the column is nullable or not.
- **FROM_ROW**
  From which row in the source file the data type (e.g. String) was determined. This number serves as a hint to investigate source data when an unexpected type is analyzed. E.g. The column name **BIRTHDAY** in the source data indicates that this data should be a date. By investigating the source data at row 4 you may find the reason why the column was analyzed as String column.

@IMPORT TARGET

This command is responsible of all preparation of the target table prior to import. This includes dropping, truncating, deleting from and creating the table.

When this is executed a check is done if the input data will actually fit the table. This is done by comparing of the analyze result with the specified target and column mapping**.** Depending on the parameters the check is performed at different occasions. Parameters for this command are:

| Parameter | Default | Valid Values |
| --- | --- | --- |
| Catalog | | The target table Catalog |
| CleanData | | Specifies if data should be cleared before import.<br>• **Drop**<br>The table is dropped before import. If this is used then one of `CreateTableSQL` or `CreateTableSQLFile` must be specified.<br>• **Clear**<br>The table is emptied before import. Either through the use of the SQL, **TRUNCATE** or **DELETE**. Default method is **TRUNCATE**. Override the default **Clear** method by specifying the parameter `ClearTableMethod`. Before the table is cleared the check of input data towards the target table is performed. |
| ClearTableMethod | | Specifies how the table data should be emptied prior to importing the ne data:<br>• **Truncate**<br>• **Delete** |
| ColumnMapping | | Specifies mapping of source columns to target columns. The default is to import the source columns to the target table column by index. First column in source is imported in the first column in the table. To specify another order or to ignore certain columns use the `ColumnMapping` parameter.Syntax: `ColumnMapping="<source column>=<target column>, <source column>=<target column>, ...`.<br>Source column can be identified by index starting with 0 or by its column name.<br>Target column can be identified by index starting with 1 or by its table column name.<br>Example: `ColumnMapping="0=2, 1=3"` or `ColumnMapping="id=no, color=col"`.<br>The mapping also supports overriding of the **data type** information of the input data column. I.e. overriding the type information deducted by DbVisualizer when parsing/ analysing the data. The syntax in this case, for a single column mapping is: `<sourde column>(<data type>)=<target column>` Where `<data type>` is one of: **String**, **Date**, **Time**, **Timestamp**, **Number,** Decimal Number**,** Boolean**,** BLOB |

| Parameter | Default | Valid Values |
| --- | --- | --- |
| | | **and,** CLOB** Example: `ColumnMapping="id(Number)=no, color=col"` |
| ColumnMappingFile | | A reference to a file containing the column mappings. Same syntax as for the `ColumnMapping` parameter with the addition that column entries may be specified in separate rows. Single line comments (using `--` or `//`) and block comments (using `/*...*/`) are also supported. |
| CreateTableSQL | | The SQL needed to create the table to import to. This parameter or the `CreateTableSQLFile` parameter is required if `CleanData="Drop"` is specified. |
| CreateTableSQLFile | | A file reference to a file containing the SQL to create the table. |
| DropTableSQL | | The SQL used to removing (drop) the table. |
| FailOnDropFailure | false | If true, the import will fail if the **DROP** table statement fails. |
| Schema | | The target table schema to import to. |
| SkipValidateColumns | | A comma separated list of target column names for which validation shall not be done. Example: `SkipValidateColumns="Column1,Column2"` |
| SkipValidateColumnNumbers | | A comma separated list of target column numbers for which validation shall not be done. Example: `SkipValidateColumnNumbers="1,5"` |
| SkipValidateJdbcTypes | | A comma separated list of JDBC type names for which validation shall not be done. Example: `SkipValidateJdbcTypes="INTEGER,TINYINT,VARCHAR"` |
| Table | | The target table to import to. |
| UseDelimitedIdentifiers | false | • **true**<br>• **false**<br>If **true** then table and column names will be delimited. |

Some examples

Example 1

```
@import target Table="MyTable" ColumnMapping="0=ID,2=NAME" CleanData="Drop"
               CreateTableSQL="CREATE TABLE MyTable (id SMALLINT, name
VARCHAR(45))";
```

The target table is the **MyTable** table. The table is dropped and recreated before import. The first column of the input data is mapped to the target column **ID** and the third column to the target column **NAME**".

As the table is dropped, we need to supply the DDL/SQL to create the table.

Example 2

```
@import target Table="MyTable" ColumnMapping="2=NAME" CleanData="Clear"
```

The table is cleared before the import. The default clear method is determined by DbVisualizer. For databases supporting this, **Truncate** is used.

@IMPORT EXECUTE

This command will run the import.

| Parameter | Default | Valid Values |
|---|---|---|
| BatchImport | true | • **true**<br>• **false**<br>Setting this to true significantly improve the import performance. Note that batch import may not be supported by all databases or JDBC drivers. In error situations it is also a good idea to switch off batch import. |
| BatchSize | 100 | For every 100 (or specified) number of rows being inserted, DbVisualizer will run a database **commit**. |
| FailOnInsertFailure | false | • **true**<br>• **false**<br>If **true**, the import will fail if an **INSERT** statement fails. |
| Example: | | |

Run the import in a non batch mode

```
@import execute BatchImport="false";
```

## Examples

SELECTING DATA TO IMPORT AND MAPPING COLUMNS

CSV File delimited by exclamation mark "!".

```
Volvo!XC90
BMW!F32 4 Series
Volvo!XC60
Mercedes!C197 SLS AMG
```

Note that the first column of the CSV file is the brand name ("Volvo") of the car. The table we are importing to have the columns in opposite order model, brand. I.e. there is a need to map the columns.

```
CREATE TABLE carmodel (model VARCHAR(50), brand VARCHAR(50));

@import on;
@import set ImportSource="cars.csv" CsvColumnDelimiter="!"
            SkipRowsStartingWith="BMW";
@import parse;
@import target Table="carmodel" ColumnMapping="0=brand,1=model";
@import execute;
```

**Parameters used**

- `CsvColumnDelimiter="!"` specifies that the data is delimited by an exclamation mark ("!").
- `SkipRowsStartingWith="BMW"` do not import entires starting with "BMW".
- `ColumnMapping="0=brand,1=model` column 0 of the CSV file is mapped to the **brand** column of the database table. Column **1** is mapped to the **model** column.

The table **carmodel** content after import:

```
model        brand
------------ --------
XC90         Volvo
XC60         Volvo
C197 SLS AMG Mercedes
```

OVERRIDING ANALYZED TYPE INFORMATION

When an input file is parsed, DbVisualizer analyzes the data to determine the data types of the input data. The algorithm for this is quite coarse, and it is not possible to override the analyzed data type.

**CSV data**

```
NAME,BIRTHDAY
August, "Sat, 21 Jul 1962"
Sven, "Fri, 21 Jan 1972"
Lotta, "NoData"
Bert, "Sat, 21 Jul 1962"
```

Note that the **birthday** of **Lotta** is **NoData** which is of course not a valid date. When the data is analyzed, it will come to the conclusion that the **BIRTHDAY** column is a String.

The result of `@import parse` will contain a table describing information about the data that was parsed.

```
Parsed 5 records. Columns in source: 2 using ',' delimiter

INDEX NAME     TYPE       NULLABLE FROM ROW
----- -------- ---------- -------- --------
0     NAME     String(6)  No       2
1     BIRTHDAY String(16) No       4

Total bytes: 73 B
```

As mentioned earlier you can see that column BIRTHDAY has been interpreted as a String. This was found examining row 4 (FROM ROW column is 4).

In connection with inserting this column in the database DbVisualizer would insert/set this as a string. This would result in total import failure and no rows would be inserted in the database.

This may be addressed by overriding the analysed type for BIRTHDAY (String) and set the type to date.

The script:

```
CREATE TABLE birthdays (name VARCHAR(40), birthday DATE);

@import on;
@import set ImportSource="birthdays.csv" CsvTextQuotedBetween="Double"
DateFormat="EEE, d MMM yyyy" HeaderStartRow="1";
@import parse;
@import target Table="birthdays" ColumnMapping="0=name,1(date)=birthday";
@import execute;
@import off;
```

Parameters

- `DateFormat="EEE, d MMM yyyy"`
  Defining the format to be able to interpret the dates in the CSV file.
- `ColumnMapping="0=name,1(date)=birthday"`
  Note the **1(date)=birthday** where we are mapping the column with index 1 to the target column birthday. The **(date) part specifies that column 1 should be interpreted as a date.**

The result of the import using the script above is that 3 rows are imported (August, Sven and Bert). The row representing Lotta is reported as a failure as Indicated below.

```
1 Record affected, Record: 0 originating at row: 4
DataRecordException: Convert error, Column: BIRTHDAY at index: 1
DataTypeConversionException: Value is 'NoData'. Not a valid date format. Valid
format: 'EEE, d MMM yyyy'
```

Text File

```
001   APPLE
002   SLEMON
003   ORANGE
```

The SQL Script

```
@import on;
@import set ImportSource="fruitlist.txt" TxtColumns="0,6";
@import parse;
@import target Table="fruitslist" Catalog="test";
@import execute;
```

The parameter **TxtColumns** parameter specifies the column character positions. In this case first column starts at character position 0 and the second column starts at character position 6.

The resulting imported table is

```
id name
-- ------
1  APPLE
2  LEMON
3  ORANGE
```

Note how "LEMON" is imported without proceeding blanks. This is because column values are trimmed (TxtTrim parameter default is true).

The TxtColumns parameter

The TxtColumns parameter supports a number of syntaxes as explained in the examples below.

An example when parsing a row "AAA BBB CCC"

| TxtColums parameter | Yields extracted columns |
|---|---|
| 0-2, 4-5 | "AAA" "BB" |
| 1-3, 8-9 | "AA" "C" |

Omitting the end index (as in the SQL script above)

| TxtColums parameter | Yields extracted columns |
|---|---|
| 0, 4, 8 ( same as 0-3,4-7, 8-end of line) | "AAA" "BBB" "CCC" |
| 1, 8-9 (same as 1-7,8-9) | "AA BBB" "C" |

Using the "+" sign

| TxtColums parameter | Yields extracted columns |
|---|---|
| 0+3, 4+3, 8 (Same as 0-3,4-7, 8-end of line) | "AAA" "BBB" "CCC" |
| 0+7, 8+1 (Same as 0-7, 8-9) | "AAA BBB" "CC" |

IMPORTING EXCEL DATA

Excel file

```
A   B
--  --------
1   A
2   #DIV/0!
3   C
```

Note how row 2 column B has the value **#DIV/0!.** This value represents a case where a cell is a calculated using formula where the calculation is producing an error.

SQL Script to import

```
CREATE TABLE exceldata (id INT, value VARCHAR(50));

@import on;
@import set ImportSource="excelData.xlsx" ExcelSheetName="mydata"
ExcelCellPolicyError="SKIP_ROW";
@import parse;
@import target Table="exceldata";
@import execute;
@import off;
```

**Parameters used**

- `ExcelSheetName="mydata"`
  Specifying that the sheet named mydata is is the sheet to import.
- `ExcelCellPolicyError="SKIP\_ROW"` Specifies that if we find a formula that produced an error we should skip the complete row.

**Resulting Table**

```
id value
-- -----
1  A
3  C
```

Note that row 2 is not imported as we instructed by `ExcelCellPolicyError="SKIP\_ROW"`.

If any of the input data cannot be imported DbVisualizer will keep track of this. This is done by storing the failed data in a specific errorRecords.drec file in the directory where the import process stores its intermediate results (See ImportResultRoot parameter).

Following is an example were the export fails. It also shows how to import the failed data again. Specifically, the first import fails as Clementine is a name that is too long to fit in the target table column.

```
ID,FRUIT,PRICE
1,Banana,1.22
2,Clementine,0.35
3,Orange,0.55
```

First Import SQL Script

```
CREATE TABLE fruits (id SMALLINT, name VARCHAR(6), price DECIMAL(10,2));

@import on  Clean="true" ImportResultDir="/tmp/importContinueFirstImport";
@import set ImportSource="fruits.csv" HeaderStartRow="1";
@import parse;
@import target Table="fruits" SkipValidateJdbcTypes="VARCHAR";
@import execute;
@import off;
```

**Note:** the table definition for the fruit table defines the column name to be **VARCHAR(6).** The input data **"Clementine" will not fit there.**

**Parameters used**

- `ImportResultDir="/tmp/importContinueFirstImport"`
  We get our results in this directory. Makes it easy to refer in the second import script.
- `SkipValidateJdbcTypes="VARCHAR"`
  Tells DbVisualizer not to validate VARCHAR columns. This is done for the purpose of this example. If not specified, the import would stop before any data has been imported.

When running this import the Database used in the example (MySQL) will fail when the import tries to insert the data row 3 as Clementine will not fit the column. The Failure printed in the DbVisualizer Log for this looks something like:

```
1 Record affected, Record: id = 0 originating at row: 3
DataRecordException: Error when importing data.
MysqlDataTruncation: Data truncation: Data too long for column 'name' at row 1
```

Note that the source data is pinpointed as **originating at row: 3**.

**The Table fruits content after import.**

```
id name   price
-- ------ -----
1  Banana 1.22
3  Orange 0.55
```

Second import SQL Script

```
ALTER TABLE fruits MODIFY COLUMN name VARCHAR(20);

@import on ImportResultDir="/tmp/importContinueSecond" UseImportFile="/tmp/
importContinueFirstImport/errorRecords.drec" Clean="true";
@import execute;
@import off;
```

The ALTER statement is dealing with the root cause of why the import failed. The name column was too small.

Note that when continuing an import, the commands **@import set** and @import target is not specified. The settings and target from the old import is used.

**Parameters used**

- `ImportResultDir="/tmp/importContinueSecond"`
  Specifying a separate directory for the results
- `UseImportFile="/tmp/importContinueFirstImport/Results/errorRecords.drec"`
  Pinpointing the file containing the data that contains the data that could not be imported.

**The Table fruits content after second import.**

```
id name       price
-- ---------- -----
1  Banana     1.22
3  Orange     0.55
2  Clementine 0.35
```

TESTING AN IMPORT - DRY RUN

The client-side import offers a way to run the import script to perform all client side data validation without changing anything in the database. This is done using the `@set dryrun` command.

Note that when running the import, without dry run, the import may fail nevertheless due to checks on the database side. E.g. primary- or unique key constraint checks.

```
@import set ImportSource="fruits.csv";
@import parse;
@set dryrun;
@import target Table="fruits" CleanData="Clear";
@import execute;
@set dryrun off;
@import off;
```

When running the script, no actual clearing of the table will be done as the parameter `CleanData` indicates. Nor does the `@import execute` lead to any rows being inserted in the database.

Since there is a `@set dryrun` command prior to the commands no changes to the database table will be performed.

# Parameterized SQL - Variables and Parameter Markers

## Parameterized SQL - Variables and Parameter Markers

A useful feature in the SQL Commander is to use variables or parameter markers in SQL scripts. Variables are used to express that certain parts of the SQL should be replaced with values when the SQL is executed. If you use a script to perform repetitive tasks, such as creating a user and granting permissions, just insert variables for the user name and permissions to grant in the script and DbVisualizer will prompt for the values at execution.

In addition to DbVisualizer's own variable syntax, two parameter marker syntaxes supported natively by some databases/drivers can also be used. This makes it easier to use SQL statements from other tools or code as-is in DbVisualizer, but you need to be aware of the limitations in how they are used compared to DbVisualizer variables.

The following gives an overview of the different formats and how they can be used.

> Even if DbVisualizer supports several variable formats it doesn't mean you can always copy/paste the SQLs including the parameter markers to another application and successfully execute it. You need to check the compatibility for the actual connector/driver/framework and even that the database itself supports the used syntax.

The following variable syntaxes are supported by DbVisualizer:

### DbVisualizer Variables

This is the syntax for a DbVisualizer variable:

```
${variable||value||type||options}$
```

This is the most flexible syntax as it supports setting a name, default value, data type, and other options. Check the DbVisualizer Variables section for details. A DbVisualizer variable can be used anywhere in the SQL as the specified value replaces the variable definition as a literal (unless a data type is specified; with a data type, its behavior is exactly the same as for Named Parameter Markers.

EXAMPLE

```
select *
from   EMPLOYEE
where  FIRST_NAME like '${First Name||Phil}$'
and    AGE > ${Age||20}$
```

The variable identifiers, **${...}$** can be modified in **Tools → Tool Properties** and in the **General / Variables** category.

## Named Parameter Markers

> Note that the support for named parameter markers is database-type-specific.

These are the parameter marker syntaxes supported by DbVisualizer.

`&name`
`:name`
`:{name}`
`:'name'`

This format allows only a name for the parameter and no other settings, such as type or default value. The parameter name is the name DbVisualizer shows in the prompt window.

Named parameter values are bound at runtime with the markers in the SQL statement. Some JDBC drivers/ databases require that the proper data type is set while some are more relaxed. For named (and unnamed) parameter markers, you may choose data type in the prompt window. The data type **Literal** means that the specified value will replace the variable as a literal in the final SQL statement. I.e., the value is not bound at runtime.

Named parameter markers should only be used in contexts supported by the actual database, usually for column values. For example, as opposed to a DbVisualizer variable, a parameter marker cannot be used for a table or column name. The only difference between `&name`, `:name`, `:{name}`, and `:'name'` is that the latter two, `:{name}` and `:'name'`, allows white spaces in the name.

EXAMPLE

```
insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, &FirstName, &LastName, &Address, &Age);

insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, :FirstName, :LastName, :Address, :Age);

insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, :{FirstName}, :{LastName}, :{Address}, :{Age});

insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, :'FirstName', :'LastName', :'Address', :'Age');
```

Read more about named parameter markers.

## Unnamed Parameter Markers

The question marker symbol is probably the most supported parameter marker among the supported databases.

`?`

It is also the most unintuitive marker since the user has to count the number of question marks to set the corresponding value.

Since there is no name associated with it, DbVisualizer shows these as the following:

- **Parameter 1**
- **Parameter 2**
- **Parameter 3**
- ...

For unnamed parameters, you can set a data type for the parameter when prompted for the value.

There is no technical difference between how unnamed and named parameter markers are handled internally in DbVisualizer or when processed by the database. All are bound with a prepared SQL statement.

EXAMPLE

```
insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, ?, ?, ?, ?);
```

In DbVisualizer it is recommended to use named parameter in favor of unnamed parameter markers for easier usability during execution. If you intend to run the SQL in application code, check the support for named parameters in the target database.

Read more about unnamed parameter markers.

> ⚠ It is not possible to mix DbVisualizer variables and parameter markers, or named and unnamed parameter markers, in the same script. If you do, you will only be prompted for values for one type and the execution will fail.

For more information about the different syntaxes check Using DbVisualizer Variables and Using Parameter markers.

# Using DbVisualizer Variables

DbVisualizer variables are used to build parameterized SQL statements and let DbVisualizer prompt you for the values when the SQL is executed. This is handy if you are executing the same SQL repetitively, just wanting to pass new data in the same SQL statement.

> A DbVisualizer variable that **doesn't specify a data type** will always be replaced with the **value as a literal**. This allows use of variables anywhere in an SQL statement. If a **data type is specified,** the prompted value will be bound with the SQL and variables can in this context **only be used where supported** by the target database.

## DbVisualizer Variables

This is the syntax for a DbVisualizer variable:

```
${variable||value||type||options}$
```

This is the most flexible syntax as it supports setting a name, default value, data type, and other options. Check the DbVisualizer Variables section for details. A DbVisualizer variable can be used anywhere in the SQL as the specified value replaces the variable definition as a literal (unless a data type is specified; with a data type, its behavior is exactly the same as for Named Parameter Markers.

EXAMPLE

```
select *
from   EMPLOYEE
where  FIRST_NAME like '${First Name||Phil}$'
and    AGE > ${Age||20}$
```

The variable identifiers, **${...}$** can be modified in **Tools → Tool Properties** and in the **General / Variables** category.

## Variable Syntax

The variable format supports setting a default value, data type and a few options as in the following example:

```
${FullName||Andersson||String||where pk}$
```

This is the complete syntax for a DbVisualizer variable:

```
${name || value || type || options}$
```

| Part | Default | Description |
| --- | --- | --- |
| name | <**name**> | Required. This is the name that appear in the prompt window. If multiple variables in a script have the same name, the substitution dialog shows only one and the entered value will be applied to all variables with that name |
| value | null | The default value for the variable |
| type | (literal) | The type of variable:<br>• String<br>• Boolean<br>• Integer<br>• Float<br>• Long<br>• Double<br>• BigDecimal<br>• Date<br>• Time<br>• Timestamp<br>• BinaryData<br>• TextData<br>The defines how the data should be passed between DbVisualizer and the database server. If no type is specified, it is treated as a **literal**. |
| options | none | |

The options part is used to express certain conditions. Separate these with a whitespace

- **pk**
  Indicates that the variable is part of the primary key in the final SQL. Represented with a symbol in the prompt window
- **where**
  Defines that the variable is part of the WHERE clause. A symbol indicate this condition in the prompt window
- **noshow**
  This option define that the variable should not appear in the prompt window. A value must be set when using this option, unless it is an output variable (see **dir** below)
- **nobind**
  Used in combination with when a **type** is set and defines that the variable should be replaced as a literal in the SQL rather than being bound as a parameter marker
- **dir=in | out | inout**
  The direction for a variable used with the **@call** command (it is ignored for other uses). A variable assigned the return value for a function must be declared as **dir=out**, and a variable used for a procedure parameter must use a dir type matching the procedure parameter direction declaration. **in** is the default
- **scope=post**
  The scope must be specified as **post** when using variables representing columns when exporting BLOB/CLOB values to separate files named based on column values.
- **choices=[...]**
  A comma-separated list of allowed options, which will be represented as a dropdown for the user. Example: **choices=['A','B','C']**.

## Pre-defined Variables

A few pre-defined DbVisualizer variables can be used anywhere in the SQL. These are replaced with actual values just before the SQL is sent to the DB server.

> Note that none of the pre-defined variables below will show in the prompt window.

`${dbvis-date}$`
`${dbvis-time}$`
`${dbvis-timestamp}$`
`${dbvis-connection}$`
`${dbvis-database-type}$`

By default, date/time variable values are formatted as defined in **Tool Properties → Data Formats**, but you can also specify a custom format for a single use of the variable, e.g.

`${dbvis-date||||||format=[yyyyMMdd]}$`

The following variables can be used only when monitoring a SQL statement that produce a result set and the **Allowed Row Count** for the monitor is > 0. The output format is seconds and milliseconds. Ex: 2.018

`${dbvis-exec-time}$`
`${dbvis-fetch-time}$`

The following variable holds the absolute path to the current directory, e.g. set by the `@cd` command:

`${dbvis-pwd}$`

In an sql script, the name on the result set produced by the next SELECT statement can be set with the `@set resultset name` command (see Using Client-Side Commands). This result set name is accessible through the variable

`${dbvis-resultset-name}$`

## Variables for Java System Properties and OS Environment Variables

You can use DbVisualizer variables to access the value of a Java system property by prefixing the property name with **java.** ( `java.<property>` ). Please note that some java properties include "java" in the property name! Examples:

```
@echo ${java.user.home}$  @echo ${java.java.io.tmpdir}$
```

To access the value of an operating system environment variable, prefix the variable name with **env.** ( `env.<variable>` ). Examples:

```
@echo ${env.USER}$  @echo ${env.TMP}$
```

This may be used in many cases, for instance when importing or exporting files. The export example below shows how to specify the DbVisualizer Bookmarks folder without explicitly giving the hardcoded folder path.

```
@export set filename="${java.dbvis.prefsdir}$/Bookmarks/myscript.sql" Format="SQL"
AppendFile="false";
```

A list of Java properties can be found in the **Java Properties** tab accessible from the **Help→About...** menu

## Using "now" in values for Time, Date, and Timestamp

For variables with the type set to `Time` , `Date` , and `Timestamp` , the value may be set to the literal `now` . The value is then converted to the specified type with the format defined in **Tool Properties** and **Data Formats** category.

```
${myDate||now||Date}$ yyyy-MM-dd | 2017-07-17 |
${myDate||now||Time}$ HH&#58;mm&#58;ss | 09&#58;02&#58;50 |
${myDate||now||Timestamp}$ | yyyy-MM-dd HH&#58;mm&#58;ss | 2017-07-17
09&#58;03&#58;11 |
```

For these types it is also possible to specify the value `now` in the variable prompting window.

## Variable Substitution in SQL statements

> For variable processing to work in the SQL Commander, make sure the **SQL Commander Options → Parameterized SQL** is checked in the **SQL Commander** main menu.

A simple variable may look like this:

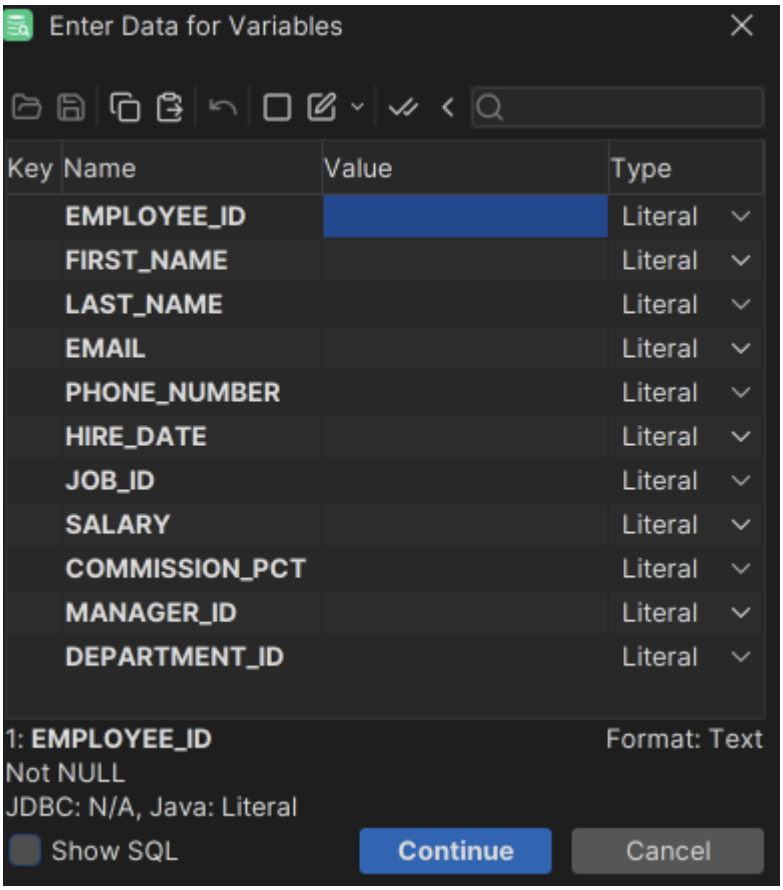
`${FullName}$`

A variable is identified by the start and end sequences, `${...}$` . (These can be re-defined in **Tool Properties**). During execution, the SQL Commander searches for variables and displays the prompt window with the name of each variable and an input (value) field. Enter the value for each variable and then press **Execute**. This will then replace the variable with the value as a literal and finally let the database execute the statement.

Consider the following SQL statement with variables. It is the simplest use of variables since it only contains the variable names. In this case it is also necessary to enclose text values with quotes since the prompt window cannot determine the actual data type for the variables.

```
INSERT
INTO
    EMPLOYEES
    (   EMPLOYEE_ID,
        FIRST_NAME,
        LAST_NAME,
        EMAIL,
        PHONE_NUMBER,
        HIRE_DATE,
        JOB_ID,
        SALARY,
        COMMISSION_PCT,
        MANAGER_ID,
        DEPARTMENT_ID
    )
    VALUES
    (
        ${EMPLOYEE_ID}$,
        ${FIRST_NAME}$,
        ${LAST_NAME}$,
        ${EMAIL}$,
        ${PHONE_NUMBER}$,
        ${HIRE_DATE}$,
        ${JOB_ID}$,
        ${SALARY}$,
        ${COMMISSION_PCT}$,
        ${MANAGER_ID}$,
        ${DEPARTMENT_ID}$
    )
```

Executing the above SQL will result in the following prompt window:




Using variables with no data type defined shows these as **Literal**. This means that the specified value will replace the variable as-is in the SQL statement.

The prompt window has the same look and functionality as the Form Data Editor, i.e. you can sort, filter, insert pre-defined data, copy, paste and edit cells in the multi line editor, plus a lot of other things. In addition the prompt window adds two new commands (leftmost in the toolbar and in the form right-click menu).

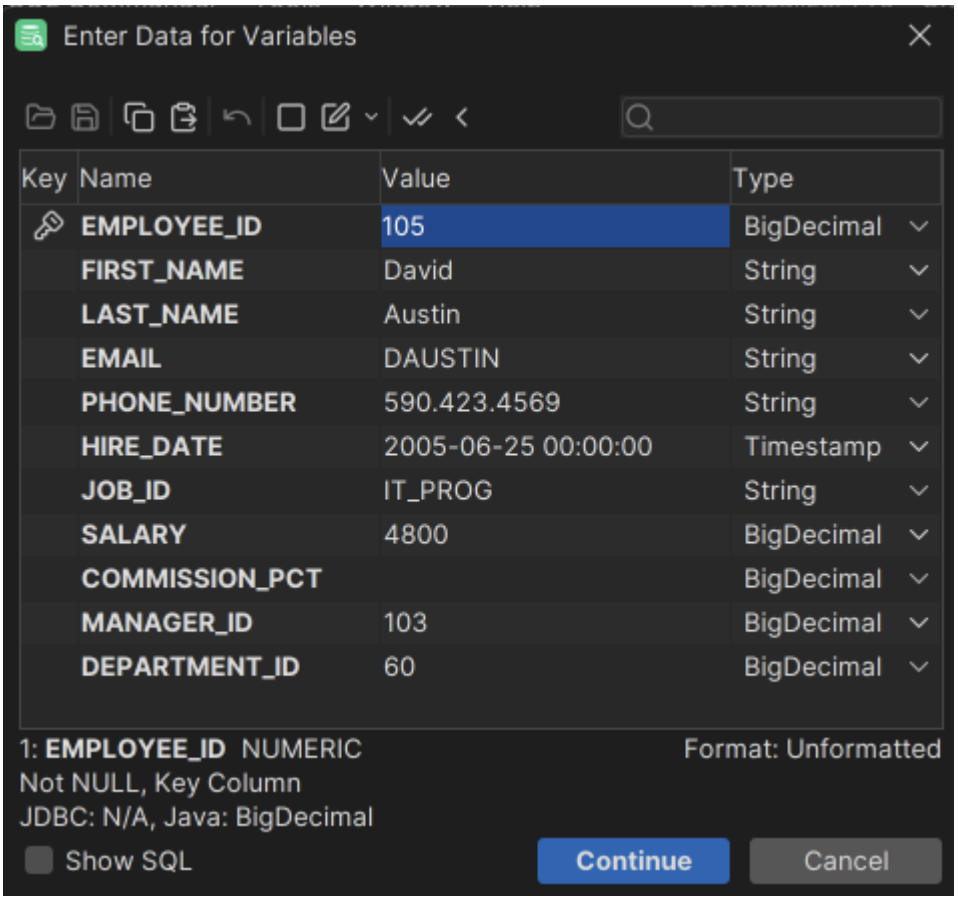
- Set Default Values
  This will set each value to the default value for the variable. If a default value was not specified in the variable, **(null)** will be shown
- Set Previously Used Values
  Set the value for each variable to the values (matched by name) that was used in the previous run (if there are no values from a previous run, this button is disabled)

The **SQL Preview** area shows the statement with all variables replaced with the values.

Here is an example of a more complex use of variables utilizing default value, data type and options:

```
INSERT
INTO
    EMPLOYEES
    (
        EMPLOYEE_ID,
        FIRST_NAME,
        LAST_NAME,
        EMAIL,
        PHONE_NUMBER,
        HIRE_DATE,
        JOB_ID,
        SALARY,
        COMMISSION_PCT,
        MANAGER_ID,
        DEPARTMENT_ID
    )
    VALUES
    (
        ${EMPLOYEE_ID||105||BigDecimal||pk ds=7 dt=NUMERIC}$,
        ${FIRST_NAME||David||String||nullable ds=20 dt=VARCHAR}$,
        ${LAST_NAME||Austin||String||ds=25 dt=VARCHAR}$,
        ${EMAIL||DAUSTIN||String||ds=25 dt=VARCHAR}$,
        ${PHONE_NUMBER||590.423.4569||String||nullable ds=20 dt=VARCHAR}$,
        ${HIRE_DATE||2005-06-25 00:00:00||Timestamp||ds=7 dt=TIMESTAMP}$,
        ${JOB_ID||IT_PROG||String||ds=10 dt=VARCHAR}$,
        ${SALARY||4800||BigDecimal||nullable ds=10 dt=NUMERIC}$,
        ${COMMISSION_PCT||(null)||BigDecimal||nullable ds=4 dt=NUMERIC}$,
        ${MANAGER_ID||103||BigDecimal||nullable ds=7 dt=NUMERIC}$,
        ${DEPARTMENT_ID||60||BigDecimal||nullable ds=5 dt=NUMERIC}$
    )
```

This example use the full capabilities of variables. This example is generated by the **Script to SQL Commander → INSERT COPY INTO TABLE** right-click menu choice in the **Data** tab grid. By default, it generates variables representing the actual values and the characteristics of the columns.




To highlight that a variable is part of the **WHERE** clause in the final SQL, it is represented with a green symbol in front of the name.

When executing an SQL statement that consists of variables, DbVisualizer replaces each variable with either the value as a literal or as a parameter marker. Using parameter markers to pass data with a statement is more reliable than literals. DbVisualizer will automatically generate a parameter marker if the variable has the data type set and if there is no **nobind** option specified.

The following will be replaced with a parameter marker:

```
${Name||rolle||String}$
```

These will be replaced with the value as a literal in the final SQL:

```
${Name||rolle}$
${Name||rolle||String||nobind}$
```

Variables in DbVisualizer may be used anywhere in a statement as long as there is no data type specified.

> If a data type was specified in the variable definition, it can be reverted to **Literal** by clicking the **Type** field for the variable.

## Changing the Delimiter Characters

You can change which identifiers should be used as the prefix, suffix and part delimiter in a variable expression in **Tools → Tool Properties**, in the **General / Variables** category.

# Using Parameter Markers

Parameter markers (also referred as **bind/host variables** or **place holders**) are commonly used in database applications where the SQL is composed of the static text combined with values represented as markers instead of actual values. These markers are processed during the preparation of the SQL statement and values are then bound with the markers. Each database has its recommendations, for how and when to use parameter markers, so this is not further discussed here.

DbVisualizer supports the most common syntaxes for parameter markers to comply with the supported databases. Parameter markers are categorized as either **named** or **unnamed** markers. The following sections explain their respective syntaxes.

> ⚠ It is not possible to mix DbVisualizer variables and parameter markers, or named and unnamed parameter markers, in the same script. If you do, you will only be prompted for values for one type and the execution will fail.

## Named Parameter Markers

> ✎ Note that the support for named parameter markers is database-type-specific.

These are the parameter marker syntaxes supported by DbVisualizer.

`&name`
`:name`
`:{name}`
`:'name'`

This format allows only a name for the parameter and no other settings, such as type or default value. The parameter name is the name DbVisualizer shows in the prompt window.

Named parameter values are bound at runtime with the markers in the SQL statement. Some JDBC drivers/databases require that the proper data type is set while some are more relaxed. For named (and unnamed) parameter markers, you may choose data type in the prompt window. The data type **Literal** means that the specified value will replace the variable as a literal in the final SQL statement. I.e., the value is not bound at runtime.

Named parameter markers should only be used in contexts supported by the actual database, usually for column values. For example, as opposed to a DbVisualizer variable, a parameter marker cannot be used for a table or column name. The only difference between `&name`, `:name`, `:{name}`, and `:'name'` is that the latter two, `:{name}` and `:'name'`, allows white spaces in the name.

EXAMPLE

```
insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, &FirstName, &LastName, &Address, &Age);

insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, :FirstName, :LastName, :Address, :Age);

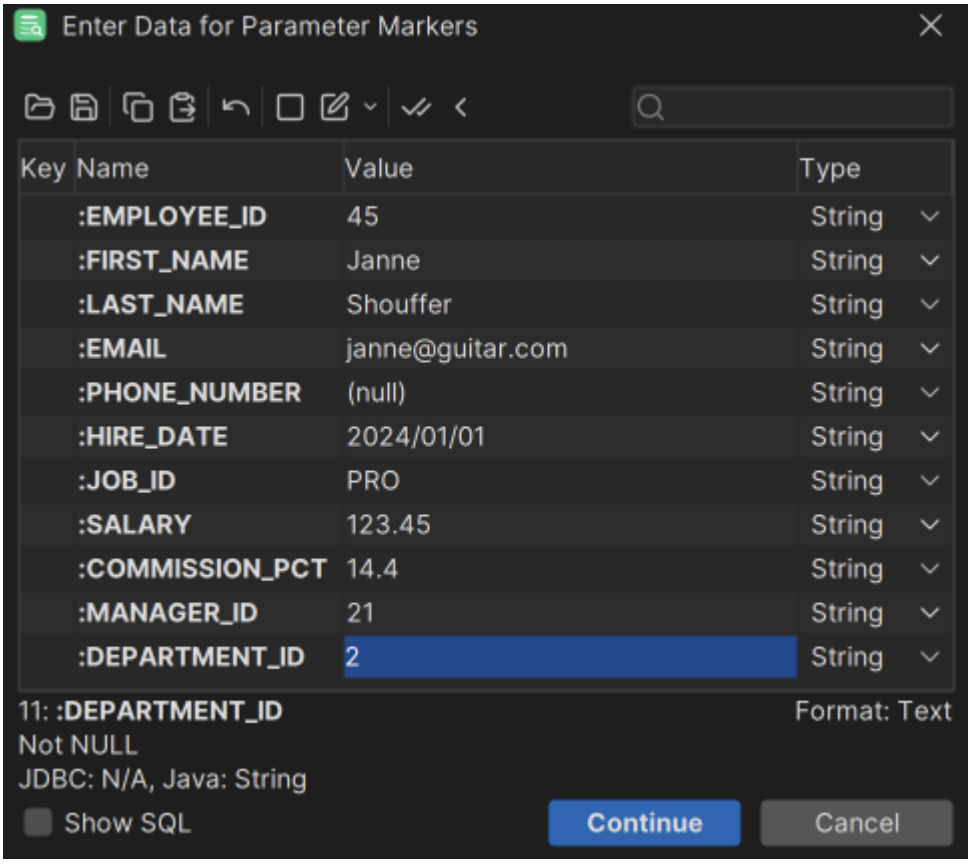
insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, :{FirstName}, :{LastName}, :{Address}, :{Age});

insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, :'FirstName', :'LastName', :'Address', :'Age');
```

The following is a sample SQL executed in the SQL Commander:

```
INSERT INTO EMPLOYEES
  (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE,
   JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, DEPARTMENT_ID)
VALUES
  (:EMPLOYEE_ID, :FIRST_NAME, :LAST_NAME, :EMAIL, :PHONE_NUMBER,
   :HIRE_DATE, :JOB_ID, :SALARY, :COMMISSION_PCT, :MANAGER_ID, :DEPARTMENT_ID);
```

The prompt window will show the markers with their respective names:



For parameter marker processing to work in the SQL Commander, make sure the **SQL Commander → Parameterized SQL** main menu option is checked.

> To apply the values, close the window and continue with the execution, use key binding **Ctrl+Enter** (**Command+Enter** on macOS).

## Unnamed Parameter Markers

The question marker symbol is probably the most supported parameter marker among the supported databases.

`?`

It is also the most unintuitive marker since the user has to count the number of question marks to set the corresponding value.

Since there is no name associated with it, DbVisualizer shows these as the following:

- **Parameter 1**
- **Parameter 2**
- **Parameter 3**
- ...

For unnamed parameters, you can set a data type for the parameter when prompted for the value.

There is no technical difference between how unnamed and named parameter markers are handled internally in DbVisualizer or when processed by the database. All are bound with a prepared SQL statement.

EXAMPLE

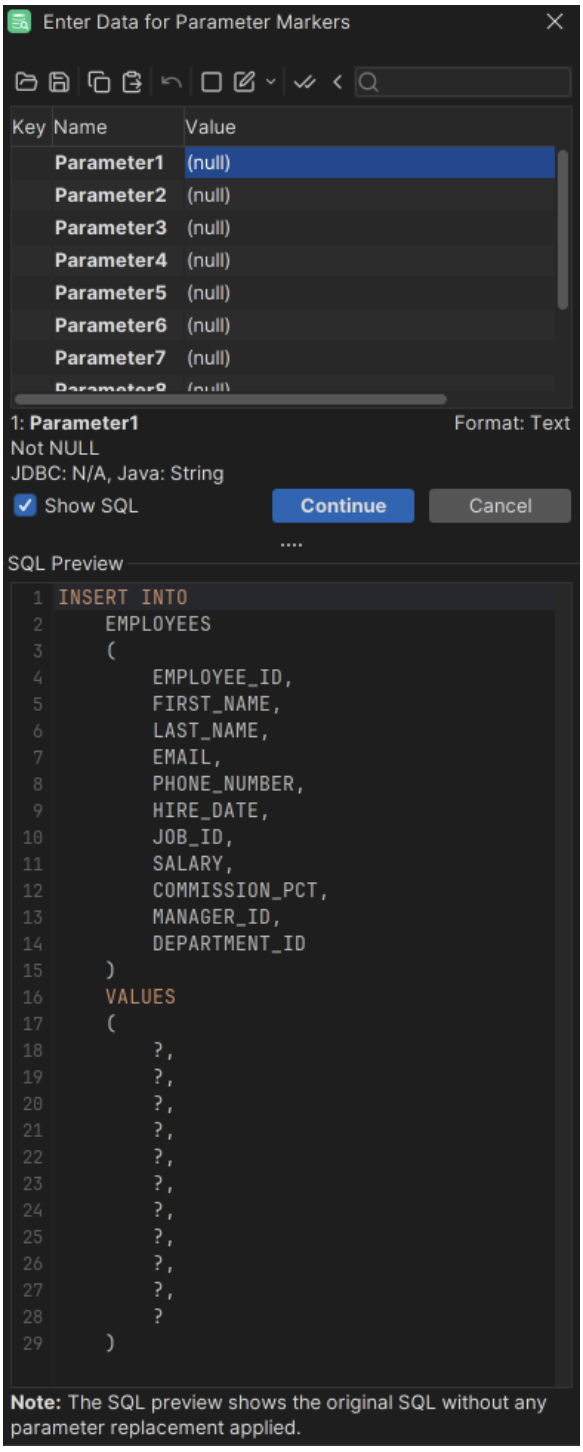
```
insert into EMPLOYEE (ID, FIRST_NAME, LAST_NAME, ADDRESS, AGE)
   values (null, ?, ?, ?, ?);
```

In DbVisualizer it is recommended to use named parameter in favor of unnamed parameter markers for easier usability during execution. If you intend to run the SQL in application code, check the support for named parameters in the target database.

This is the same SQL as used in the Named Parameter Marker section but here question marks are used as markers:

```
INSERT INTO EMPLOYEES
  (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE,
   JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, DEPARTMENT_ID)
VALUES
  (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
```

Since we're using the unnamed marker, **?**, the name of each parameter is displayed as **Parameter 1**, **Parameter 2** and so on:




Due to the use of unnamed markers it is not very intuitive what parameter correspond to which part in the statement. The **SQL Preview** may be handy to get an idea. The **Type** field is automatically adjusted based on what data is entered for a value. The data type may be manually set by left-click on the type and choose another type from the drop-down.

To apply the values, close the window and continue with the execution, use key binding **Ctrl+Enter** (**Command+Enter** on macOS).

## Get Parameter Types via JDBC

The processing of named and unnamed parameter markers is managed by DbVisualizer. By default there is no data type detection of the target columns identified by the markers and DbVisualizer will initially present these as **String** in the prompt window. When changing the value for a parameter in the prompt window, a data analyzer is triggered which will automatically detect the type and update the **Type** field accordingly.

Some drivers (far from all) have the capability to detect the real data type for the referenced columns in the SQL statement. To enable this processing, select the **Get Parameter Types via JDBC** action in the **SQL Commander** menu. DbVisualizer will then show the correct types in the prompt window.

Having **Get Parameter Types via JDBC** enabled while executing may decrease performance substantially as each SQL statement in the script is then pre-processed with the database before the prompt window is displayed.

## Database Support/Driver Support

The support for parameter markers may differ between databases. Please consult the documentation for the database to see what syntax it supports.

In some situations the database and DbVisualizer support for named parameters might be incompatible. An example is when using the same parameter name in multiple places in the SQL. When preparing a statement towards such a database, the database may report that the parameter is only used once. In these cases, DbVisualizer will trust the driver and revert to generating the names visible in the form as Parameter 1 , Parameter 2 and so on.

# Working with Result Sets

## Working with Result Sets

You can view result sets in different ways, edit simple result sets, and export or compare them.

- Viewing a Result Set
- Editing a Result Set
- Exporting a Result Set
- Comparing Result Sets
- Pinning Result Sets
- Show Result Sets in a Separate Window

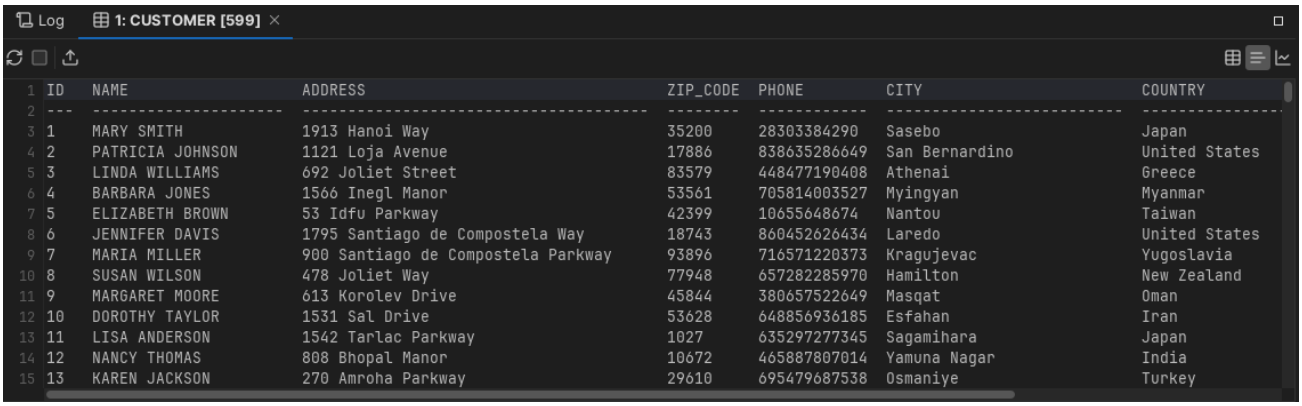
# Viewing a Result Set

You can view a result set as a grid, as text or as a chart. Which format to use by default can be specified in the **Tool Properties** dialog, under the **Default Display Mode** section shown in the **SQL Commander > Result Sets** category under the General tab.

Here you can also specify other things like if empty result sets should be shown at all, or which tab should be activated after a successful execution in the **SQL Commander**.

To change the view format for the current result set, use the buttons to the upper right in the grid toolbar.

Below is an example of a Result Set shown in text format.




```
ID  NAME                ADDRESS                               ZIP_CODE  PHONE          CITY                      COUNTRY
--- ------------------- ------------------------------------- --------- -------------- ------------------------- ---------------
1   MARY SMITH          1913 Hanoi Way                        35200     28303384290    Sasebo                    Japan
2   PATRICIA JOHNSON    1121 Loja Avenue                      17886     838635286649   San Bernardino            United States
3   LINDA WILLIAMS      692 Joliet Street                     83579     448477190408   Athenai                   Greece
4   BARBARA JONES       1566 Inegl Manor                      53561     705814003527   Myingyan                  Myanmar
5   ELIZABETH BROWN     53 Idfu Parkway                       42399     10655648674    Nantou                    Taiwan
6   JENNIFER DAVIS      1795 Santiago de Compostela Way       18743     860452626434   Laredo                    United States
7   MARIA MILLER        900 Santiago de Compostela Parkway    93896     716571220373   Kragujevac                Yugoslavia
8   SUSAN WILSON        478 Joliet Way                        77948     657282285970   Hamilton                  New Zealand
9   MARGARET MOORE      613 Korolev Drive                     45844     380657522649   Masqat                    Oman
10  DOROTHY TAYLOR      1531 Sal Drive                        53628     648856936185   Esfahan                   Iran
11  LISA ANDERSON       1542 Tarlac Parkway                   1027      635297277345   Sagamihara                Japan
12  NANCY THOMAS        808 Bhopal Manor                      10672     465887807014   Yamuna Nagar              India
13  KAREN JACKSON       270 Amroha Parkway                    29610     695479687538   Osmaniye                  Turkey
```

## Viewing as a Grid

When you view the result set as a grid, you have access to the same features as when viewing table data.

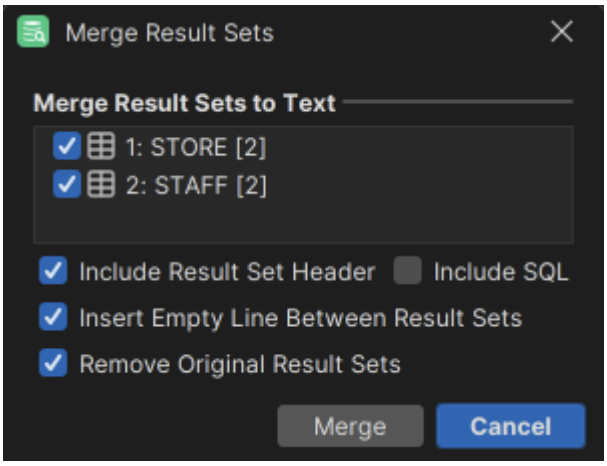
## Viewing as Text

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The text format for a result set presents the data in a tabular style. The column widths are calculated based on the length of each value and the length of the column label.

## Merge Result Sets

If you want to combine the text view of a number of result sets into one, select **Merge Result Sets** from the result set tab right-click menu. A dialog lets you select the result sets to merge and also do some configuration:




### Auto merging

Using the **Merge Result Sets** drop-down menu in the SQL Commander toolbar, you can enable **Auto Merge after Execution**. You can also select **Merge Result Sets > Configure Merging** from the drop-down menu to adjust various merge options.

## Viewing as a Graph

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

To view the result set as a chart, use the rightmost button in the grid toolbar. Please see the Working with Charts page for how you can arrange the chart.

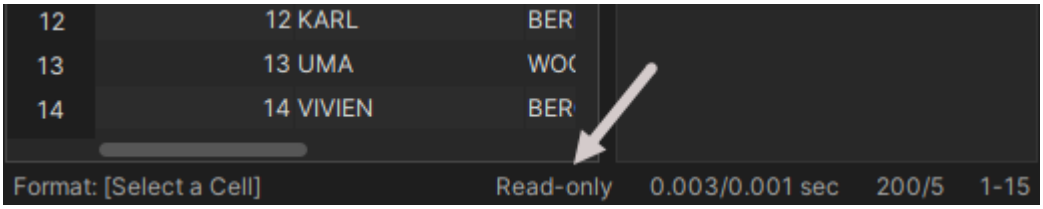
# Editing a Result Set

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

A result set from a query that fulfills these requirements is editable:

1. The SQL is a query command,
2. Only one table is referenced in the FROM clause,
3. All current columns exist by name (case-sensitive) in the identified table.

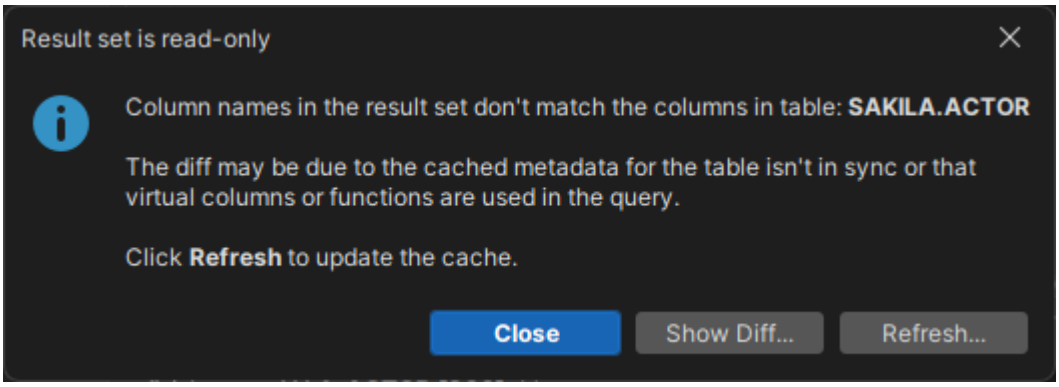
If the result set is not editable, a **Read-only** message will be displayed in the grid status bar. To show why, hover the mouse over the message or click it to see the details.



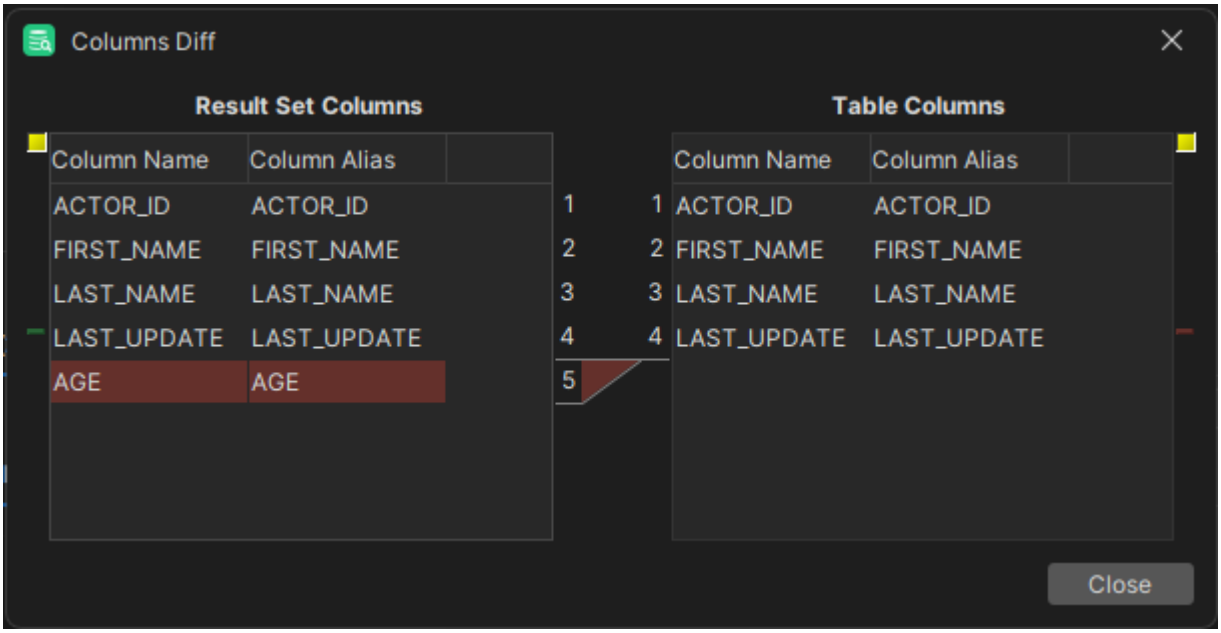

For performance reasons, metadata such as column information is saved in a local cache. If there is a mismatch between the local cache and the column metadata in the result set, the grid is read-only with an icon in front of the **Read-only** message.




Clicking the icon will display a dialog with the details of the mismatch and options to compare what columns differ and to refresh the local cache.




The diff window shows what columns don't match:




If you want results to always be read-only, you can enable the **Make Result Sets Read-Only** setting in the Tool Properties dialog, in the **SQL Commander/Result Sets** category under the General tab.

A result set like this can be edited in the same way as you can edit table data.

# Exporting a Result Set

You can export a result set as described in Exporting a Grid or using the `@export` client side command.

# Comparing Result Sets

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

You can compare a result set grid to tables and/or other result set grids.

To compare the grid data to the data of a table or a another result set:

1. Open the **Data** tab for another table or execute an SQL query to open a result set tab,
2. Select **Compare** from the right-click menu in one of the tabs to compare their grid content.

# Pinning Result Sets

Existing Result Set tabs are removed when you execute a script again. If you want to save a Result Set tab between executions, you can "pin" it using the **Pin Tab** right-click menu choice for the tab header, or by simply clicking on the tab icon. There is also a **Pin All** choice if you have multiple tabs you want to pin, and **Unpin All** to make them all be replaced at the next execution.

Whether Result Set tabs should be pinned by default can be controlled in the Tool Properties dialog, in the **SQL Commander/Result Sets** category under the General tab.

# Show Result Sets in a Separate Window

Results Sets and the Log tabs are located just under editor in a SQL Commander tab. Sometimes you may need to get the full screen height available for the editor and detach the result area tabs in its own window. To accomplish this click the symbol at the right-most position in the tab row.




Clicking the symbol while result area tabs are detached will bring them back into the original location.

Note that the symbol to detach and re-attach back can differ slightly depending on the Operating System and the current Theme.

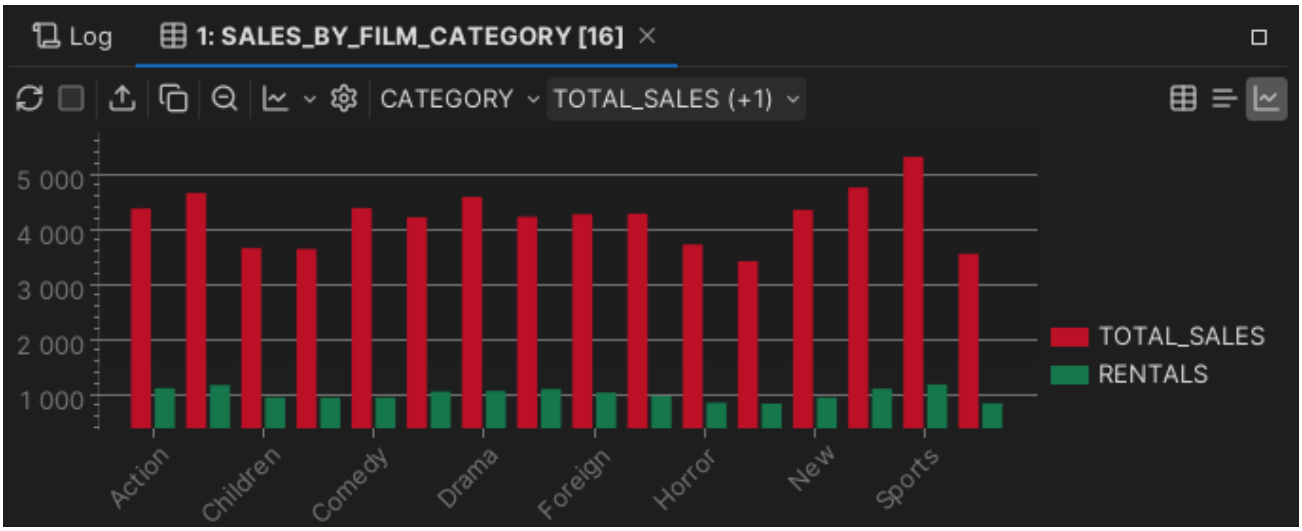
# Working with Charts

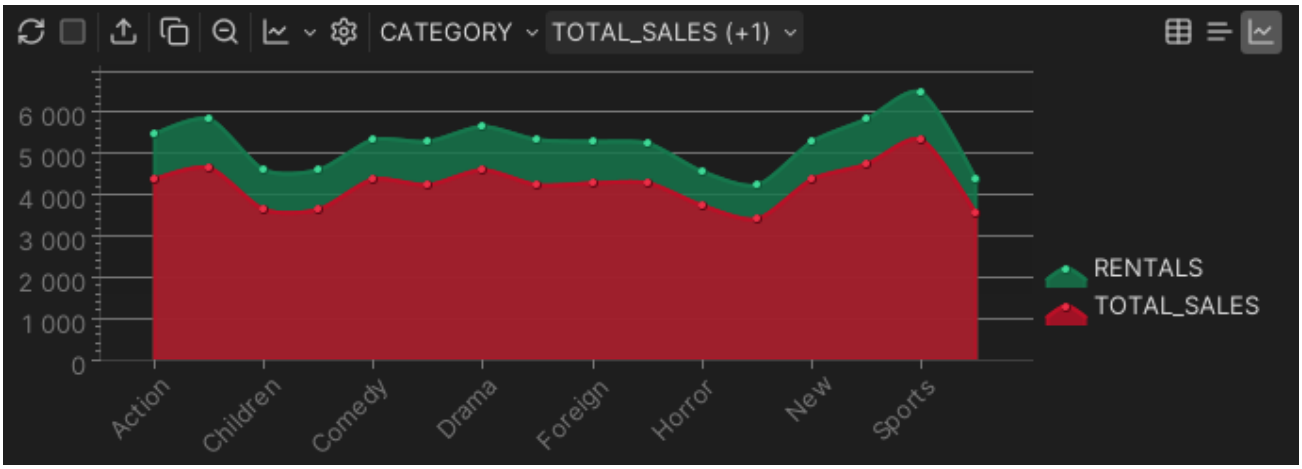
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

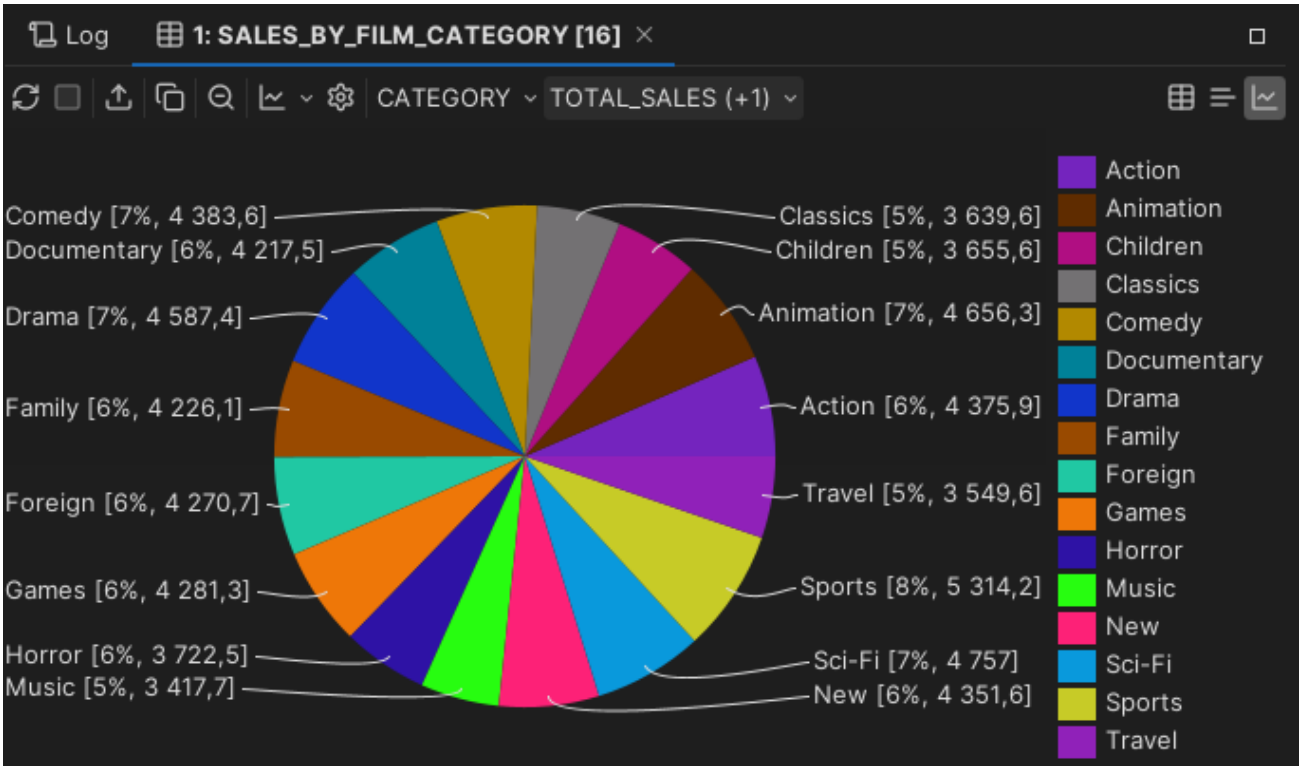
Result sets in the SQL Commander and in the Monitor tools can be viewed as charts.

The chart support in DbVisualizer presents data from any result set in a configurable chart displayed in a line, bar, area or pie style. It offers much of the charting support you find in MS Excel and other specialized charting tools. Charts may be exported as an image to file, printed and copied to system clipboard for easy sharing with other tools. Charts are configured and viewed in the SQL Commander and in the Monitor tool which is really powerful, delivering real time charts of many result sets simultaneously. Charts can be opened by clicking the icon at the the rightmost button in the result tab toolbar (see below).

Here are some sample charts:



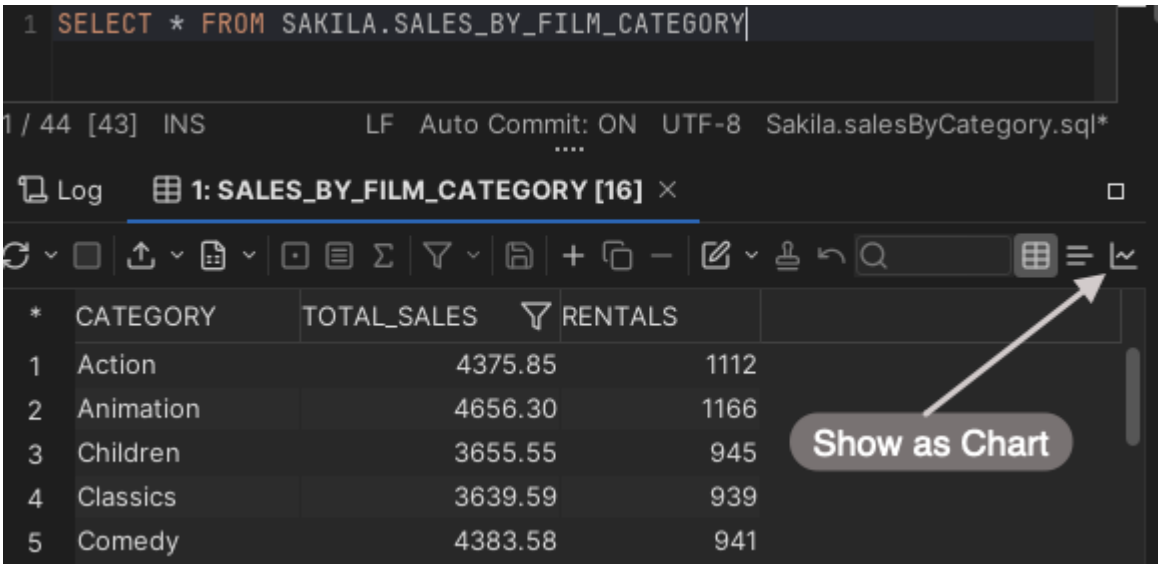


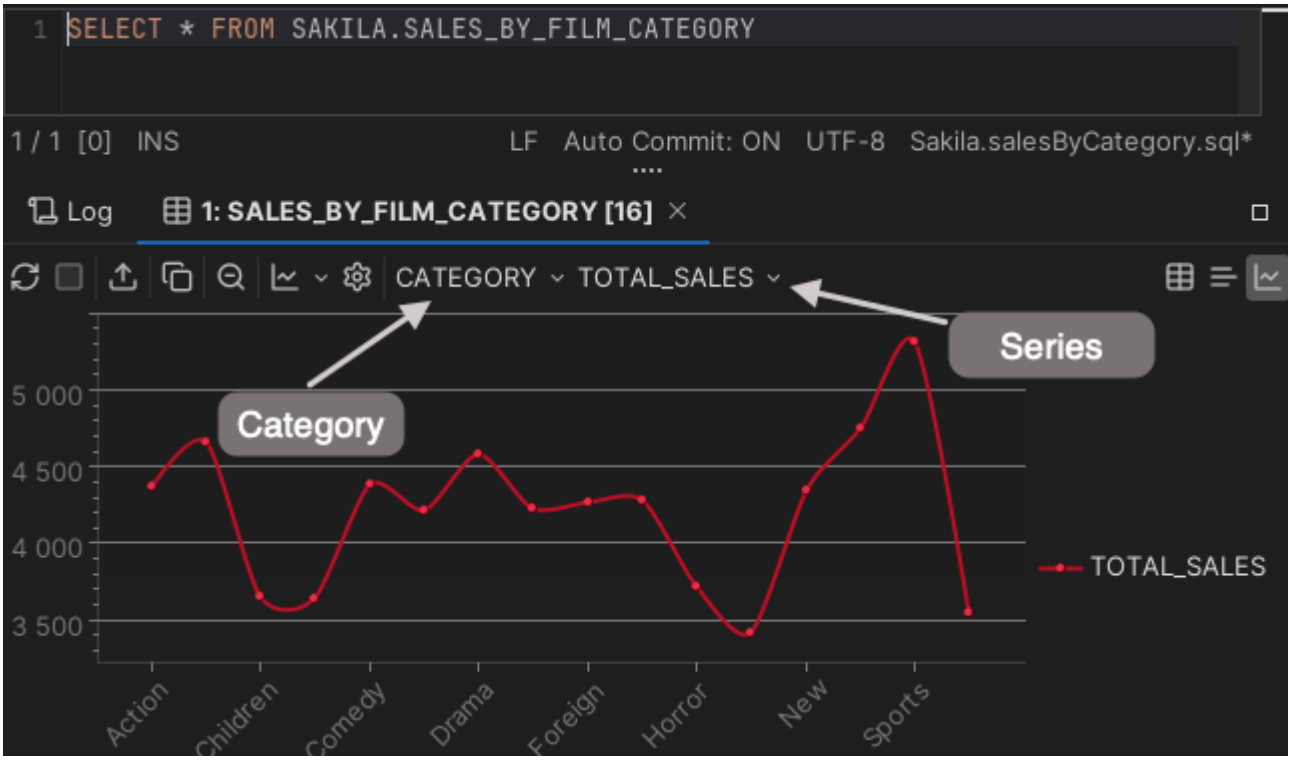

# Charting a Result Set

The basic setup of a chart is really easy. Just select one or more columns that should appear as series in the chart and what column to use as the category (X-axis). Further refinement of the chart can be made in the chart preferences window.

The normal output view of a result set in the SQL Commander or Monitor window is in a grid style as shown in the following screenshot. To activate the chart view click the rightmost button in the result tab toolbar:

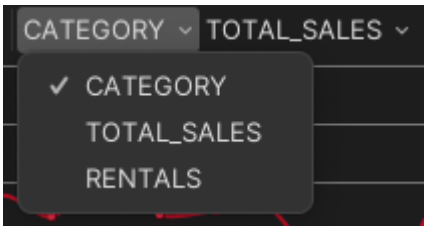



When switching to the chart view DbVisualizer automatically picks the first date or text column as the category (x-axis) for the chart and the first numeric column as the series (y-axis). In the following example it is the **CATEGORY** and **TOTAL_SALES** columns for this specific result set.



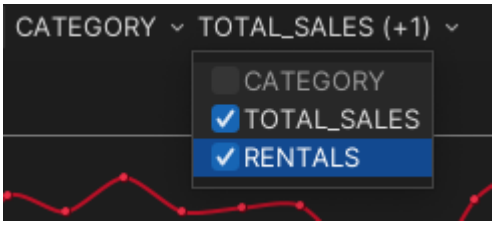

## Selecting the Category

Click the category button to pick the column to use for the x-axis. Let the mouse pointer stay on a column name for a second and a tip will show what data type it is.



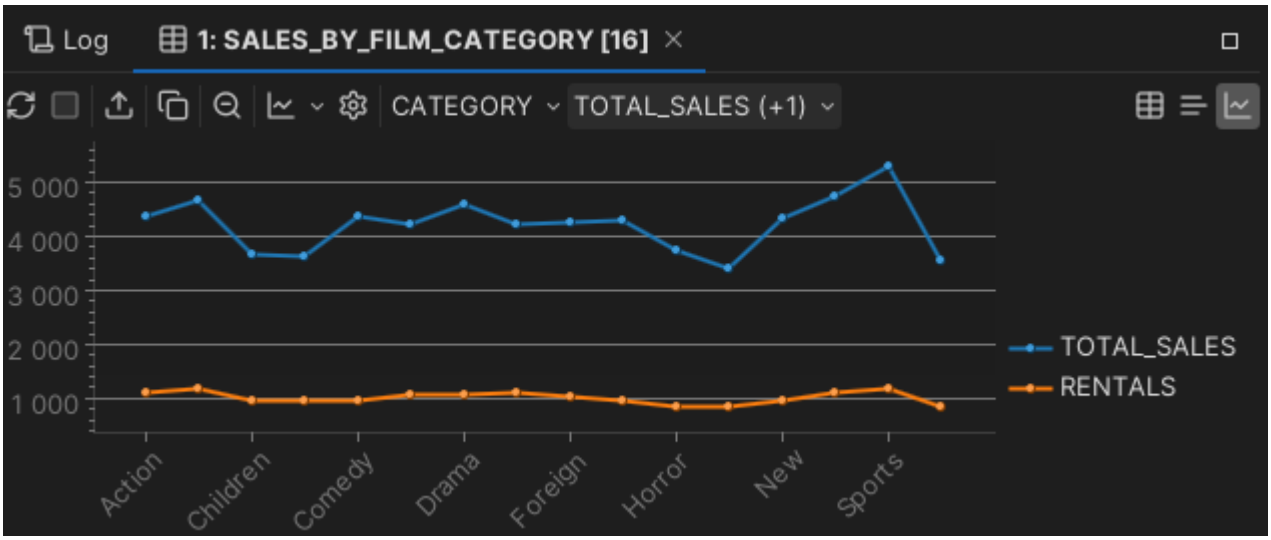

## Selecting the Series

Click the series button to change what series to display in the chart. This drop-down stays on screen while selecting one or more series (the changes are directly propagated in the chart). To close the list either press the **ESC** button or click outside the list. If only one series is selected then its name is listed in the button label. If additional series are selected then the number of selected series are listed in parentheses. Only columns that are of number data types can be selected as series.



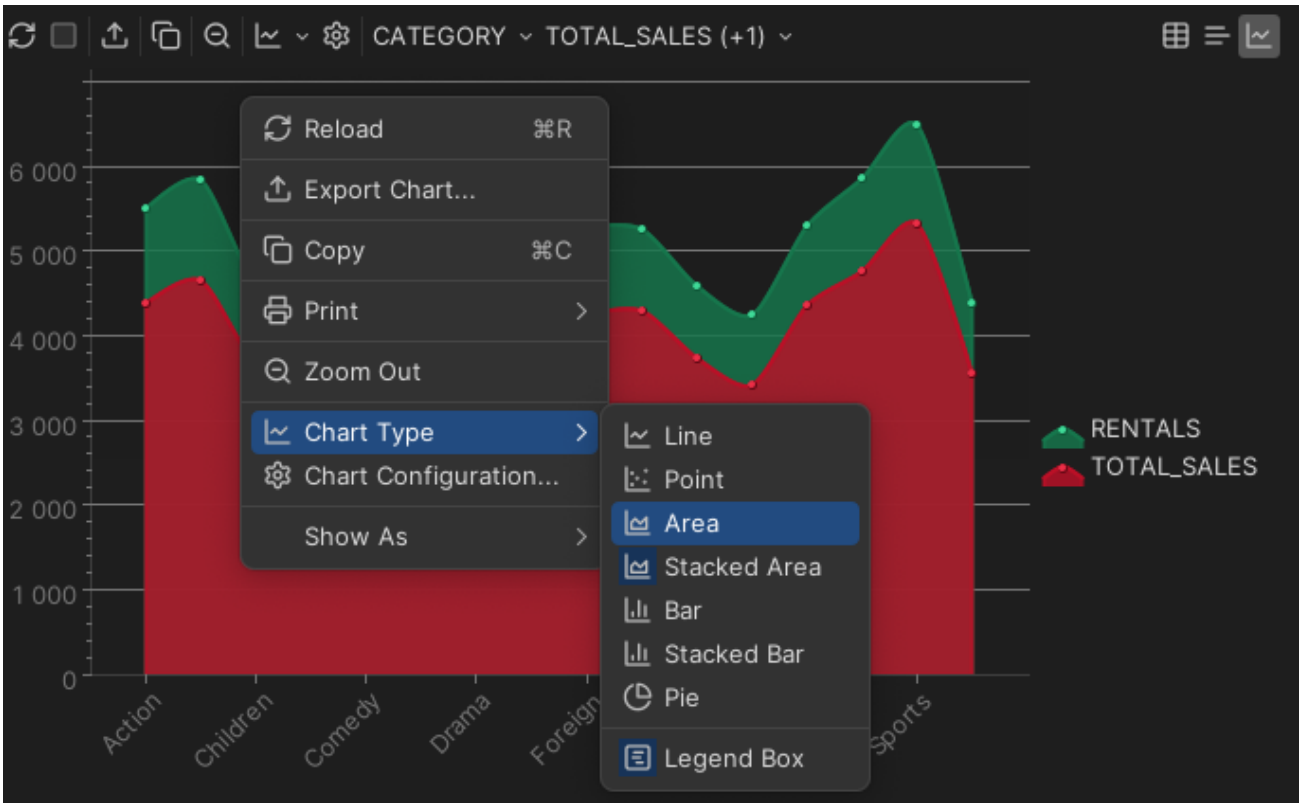

> Tip: Press the ALT key while selecting a series and all currently selected series will be de-selected.

This is the chart after applying category and 2 series:




## Chart Type

A chart can be displayed as one of several types. Select what type to use in the toolbar or on the right-click menu (which also offers some other controls):

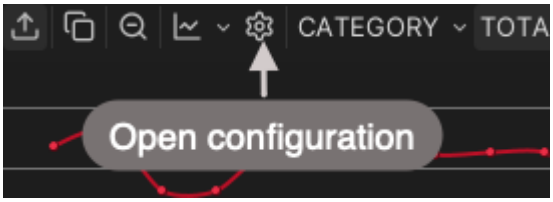


Note: For the Pie chart, only one series can be selected.

# Chart Configuration

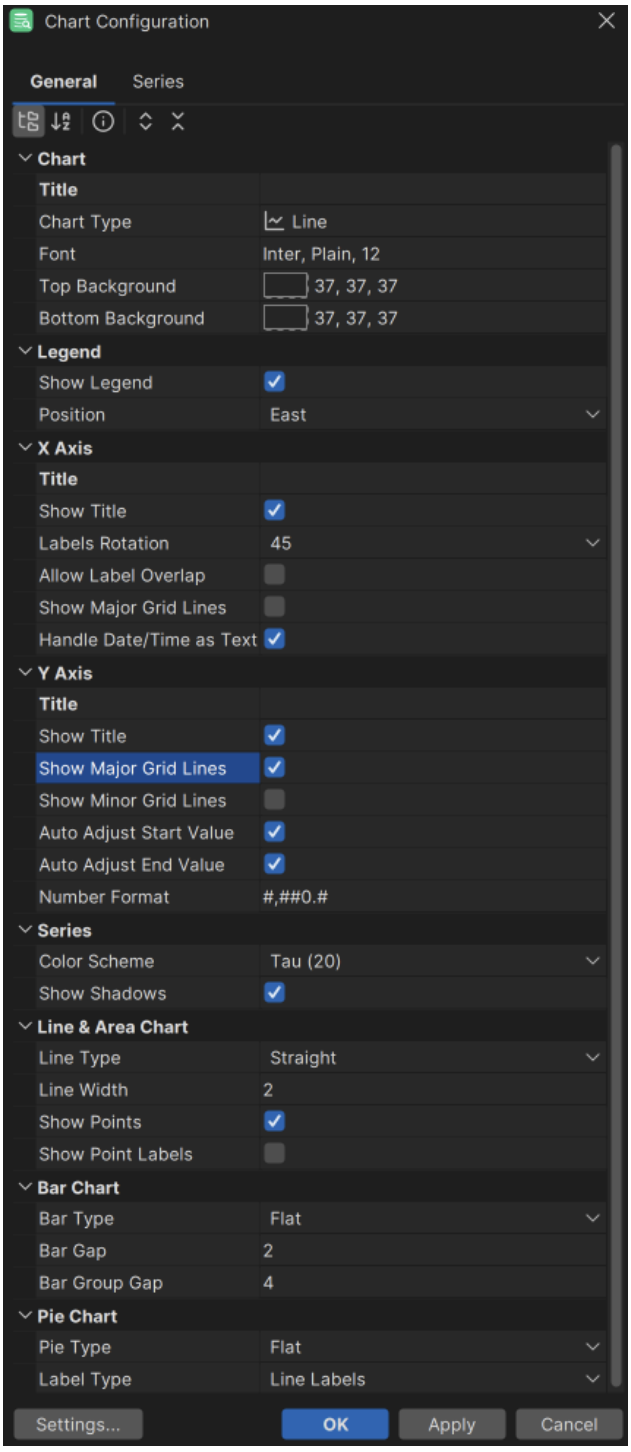
The chart can be configured to your preferences for titles, colors, legend position, etc. You can also set alternative names for the series in the chart. All appearance settings are automatically re-used when running subsequent queries in the SQL Commander during the same DbVisualizer session. If you save a query as a bookmark script then all appearance settings are saved with the chart. Loading the script at a later time will also load the chart settings.

Open the chart configuration dialog from the toolbar or the right-click menu:




## Appearance Preferences

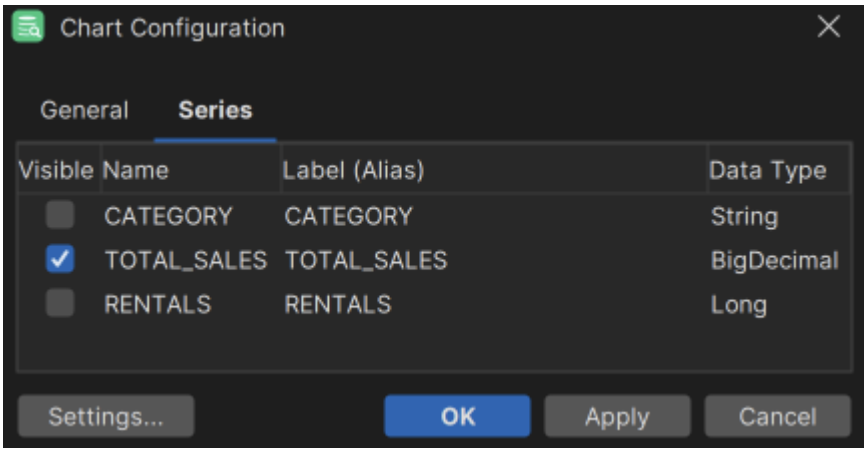
In the Chart Configuration panel, use the controls in the **General** tab to customize the layout and style of the chart (click the information button to show/hide information about the selected setting at the bottom of the window).




## Series Preferences

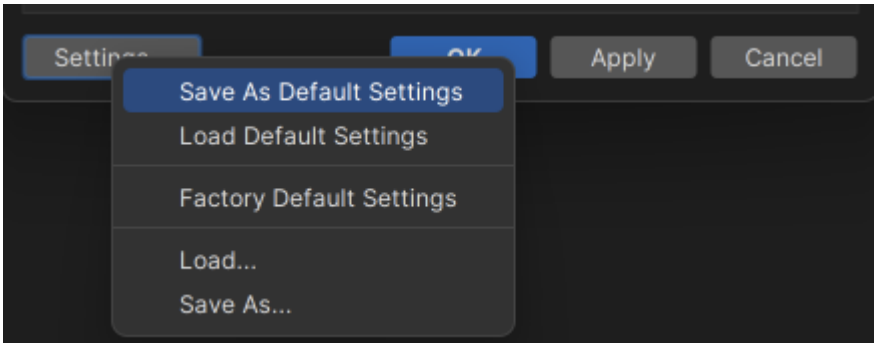
The default name of the series is the column name in the result set. In the **Series** tab you can set an alternative label name and also control what series should be visible.

Changes in the Series tab are propagated directly in the chart.




## Saving/Loading Preferences

You can also save or load default preferences by clicking the **Settings** button in the lower left corner:



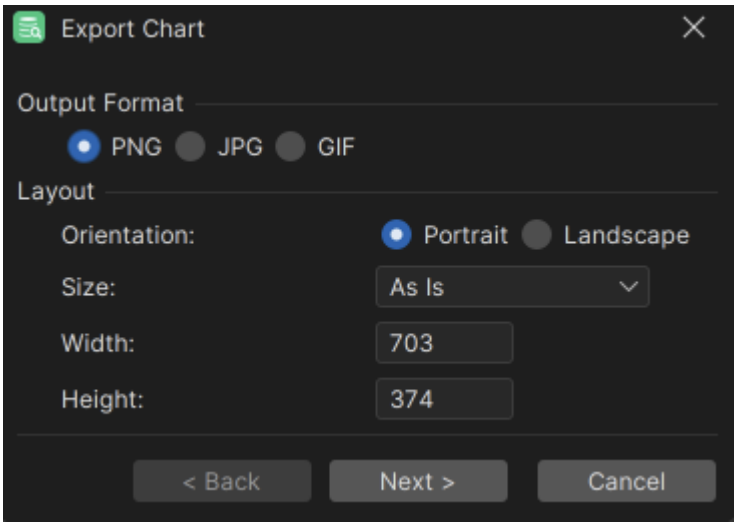

# Zooming

Charts support zooming by selecting a rectangle in the chart area. Selecting another rectangle in that zoomed area will zoom the chart even further, and so on. To unzoom one level, click the Zoom Out button.

> Note: You cannot zoom pie charts.

# Export

Charts can be exported in PNG, GIF or JPG formats.




The default size of the exported image is the same as it appears on the screen. To change the size, either select a pre-defined paper size in the Size list or enter a size in pixels.

# Working with Workspaces

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

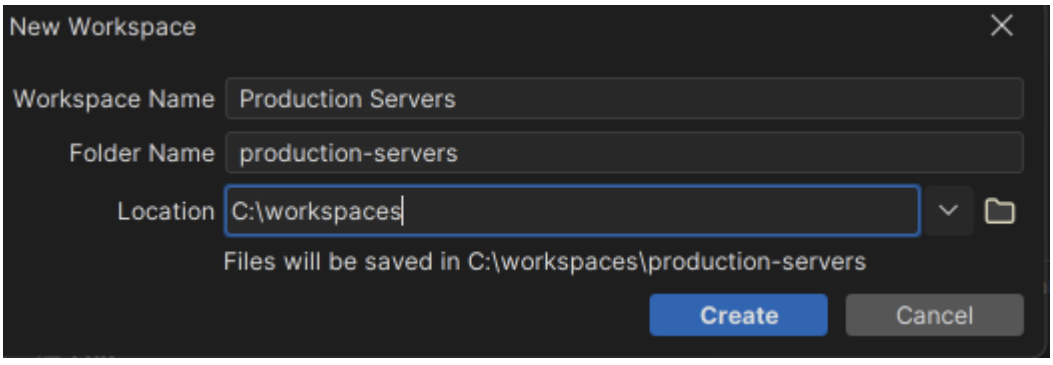
DbVisualizer allows you to create multiple workspaces. Each workspace contains its own independent set of DbVisualizer settings, allowing you to have the application configured in multiple different ways simultaneously.

As an example, let's say that you work with multiple companies, each with its own large collection of databases. Normally, that would result in a huge list of connections to keep track of, perhaps sorted into different folders to create some kind of structure. If working for the different companies required different settings (proxies, time zones, date formats, etc.), you might also need to constantly change the settings back and forth as you switch between working for different companies.

Workspaces give you an alternative way to structure your work by allowing you to separate unrelated sets of configuration from each other. Each workspace is independent from all the other ones, meaning that they have their own separate connections, settings, favorites, recent files, drivers, certificates, and so on. You can easily open another workspace, and even have multiple ones running simultaneously and separately from each other.

## Creating a Workspace

Go to **File → Workspaces → New Workspace** to open the New Workspace dialog:




Each workspace is stored as its own folder on your computer. The Folder Name and Location fields control the name and location of this folder, respectively.

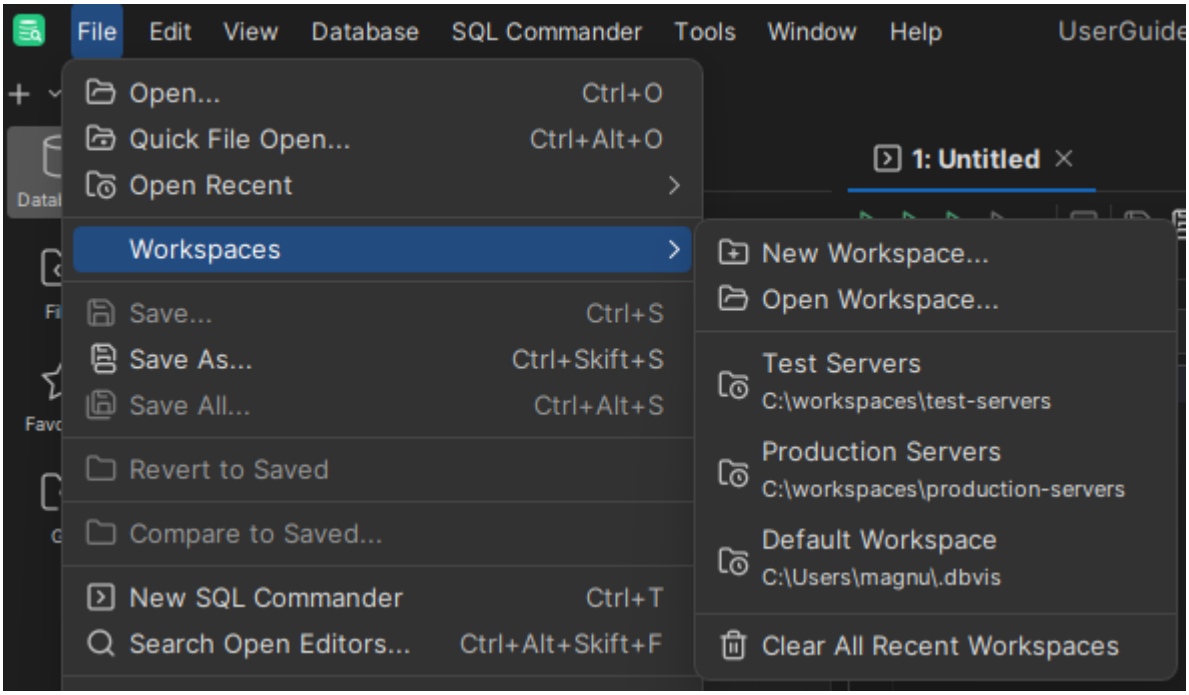
For instance, if the workspace's Folder Name is "tables" and the Location is C:\Users\bobby, the workspace and its files will be stored in the folder C:\Users\bobby\tables

By clicking **Create**, the workspace will be created and automatically opened in a new window. Changes made in this new workspace will not affect your previous workspace, and vice versa.

# Opening a Workspace

The **File → Workspaces** menu provides two different ways to open a workspace. The first is the **Open Workspace** option that allows you to open any existing workspace by selecting the folder where it's stored.

Second, if you've already opened at least one workspace in the past, the menu will also contain a list of the most recently opened workspaces. These can be opened by simply clicking on them.




When opening a workspace, it'll load into a new window. This allows you to use multiple workspaces simultaneously.

If there are recently opened workspaces in the menu, you can also use the **Clear All Recent Workspaces** to clear its list of workspaces. Note that this only clears the list in the menu; the workspaces themselves remain and can still be opened using Open Workspace.

# Renaming a Workspace

The purpose of a workspace may shift over time, and its name may need to change as a result. This can be done by opening the workspace you wish to rename and clicking on **Tools → Tool Properties** to open the Tool Properties window. There, under the **General/General** panel, you can use the **Workspace Name** field to rename the current workspace. Enter a new name and click **OK** or **Apply** to save the changes.

# The Default Workspace

When launching DbVisualizer for the first time, a workspace is automatically created for you. This is known as the Default Workspace, and it's the one you implicitly use if you don't create or open your own workspaces. It is also the one that automatically opens by default whenever you launch DbVisualizer.

Note that the Default Workspace has a slightly different status compared to user-created workspaces. To name a few differences, DbVisualizer is solely responsible for deciding where it's saved, it cannot be renamed, and it will always be considered one of your recently used workspaces.

# Command Line Support

Unless configured otherwise, DbVisualizer will always load the Default Workspace when launched. In case you want to launch DbVisualizer with some other workspace, this can be done using the `-workspace <folder>` command line argument.

For instance, if your workspace is located in the folder C:\business\my-workspace, you can launch DbVisualizer with the `-workspace C:\business\my-workspace` argument to launch directly into that workspace instead of the default one.

# -prefsdir Compatibility

Older versions of DbVisualizer didn't have the option to create and open workspaces from within the application itself, but still allowed one to manually create alternative "user preference directories" and use the `-prefsdir <folder>` argument to launch DbVisualizer with these preferences instead.

Workspaces are backwards compatible with this previous system, meaning that the preference directories that were manually created for older versions of DbVisualizer are still supported. They can be loaded and converted into workspaces by using **File → Workspaces → Open Workspace** and selecting the existing preference directory.

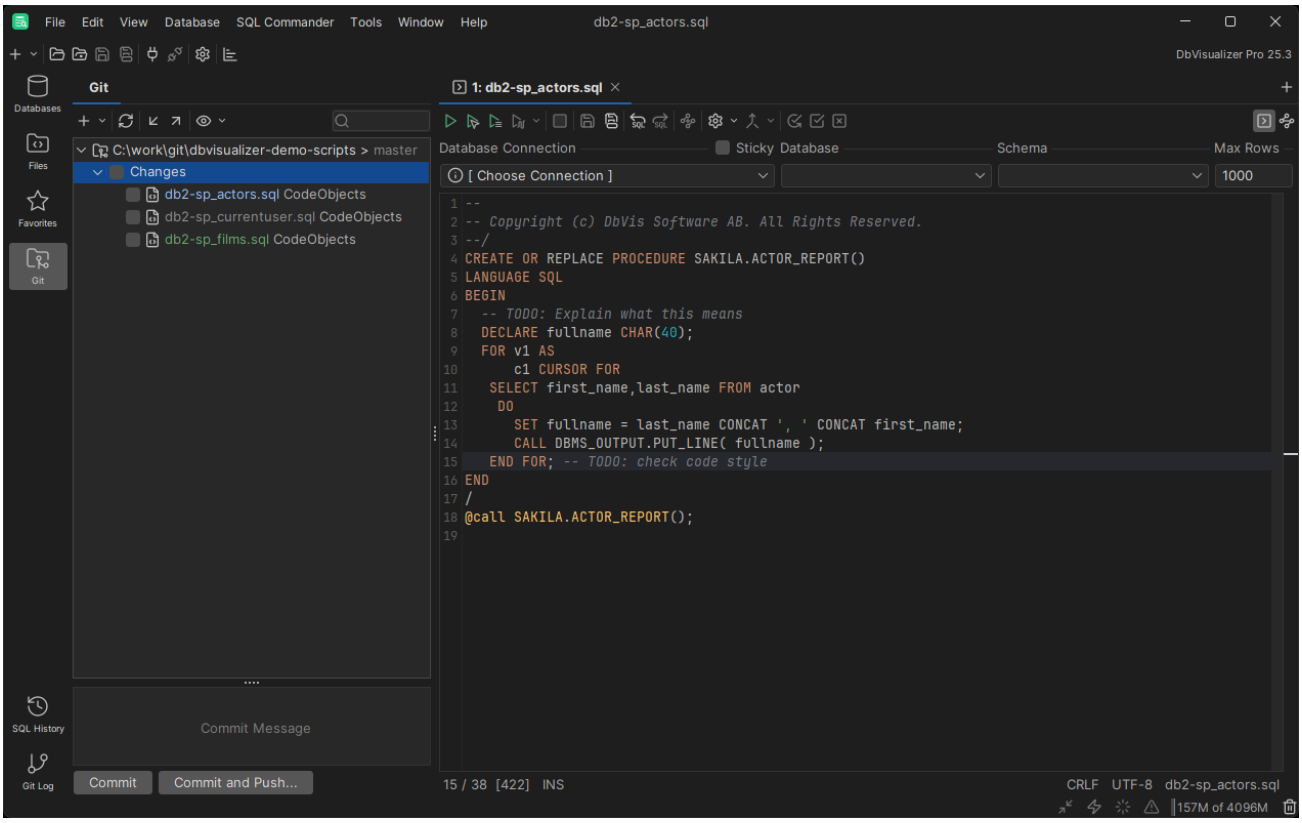
Unlike workspaces, the preference directories didn't have proper names. When converted to a workspace, the name will be automatically chosen based on the name of the folder it's stored in. This name can then be changed if needed; see "Renaming a Workspace" above.

The old `-prefsdir` argument can still be used as an alias for the new `-workspace` argument. However, the use of `-prefsdir` is not recommended and only exists for backwards compatibility with existing command line scripts. It may be removed in a future version.

# Working with Git

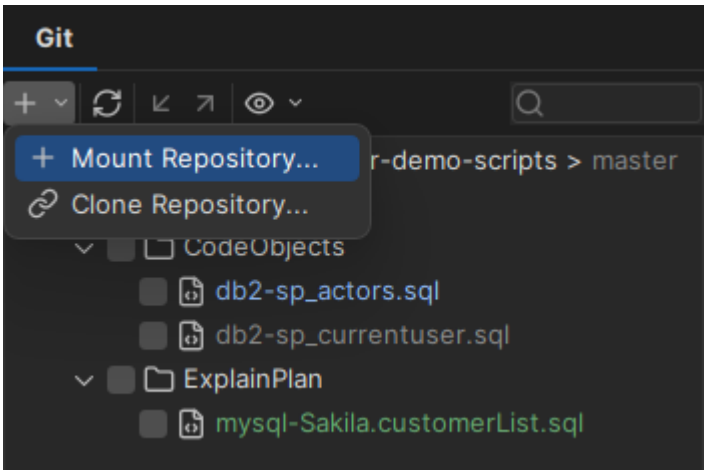
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

Using Git is a great option if you are looking to version control your scripts, but also to share your work with your team is a systematic and safe way. Version-controlled files are organised in repositories, and you can work with multiple repositories at the same time in DbVisualizer.

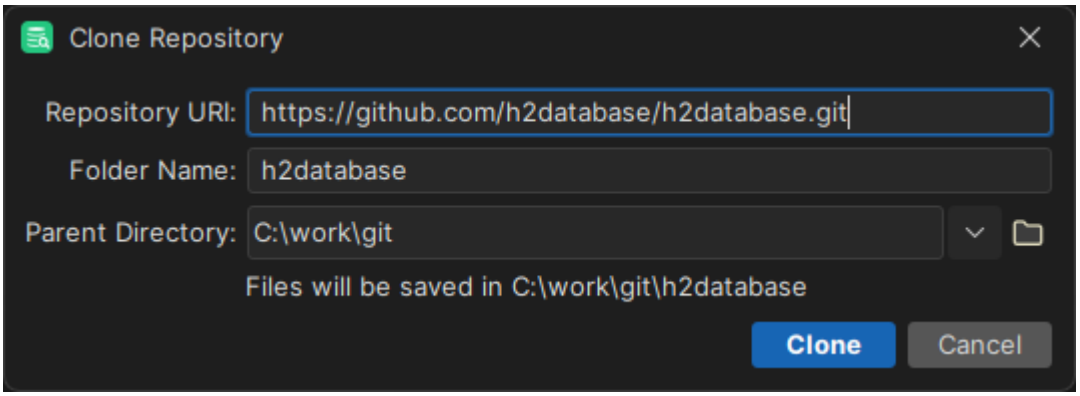



# Adding a repository

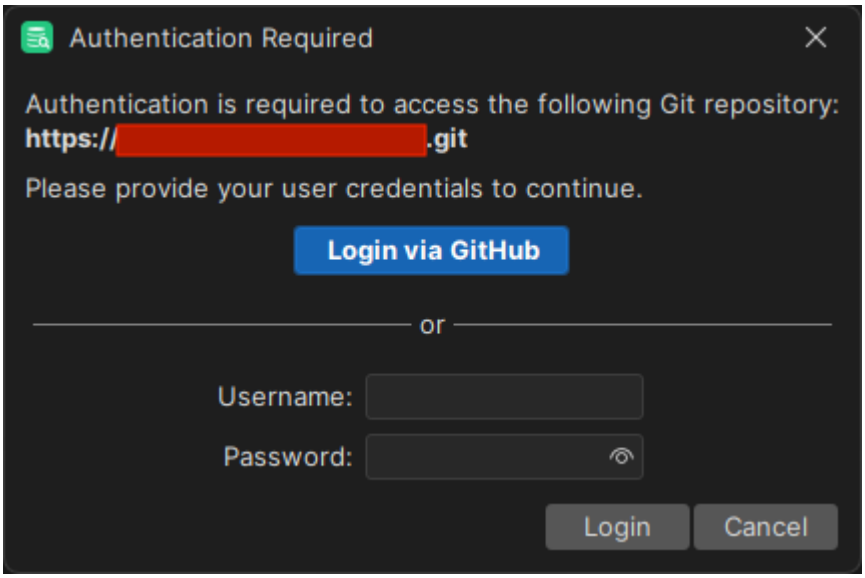
To make your version-controlled repositories available in DbVisualizer, open the Git tab. If the repository already exists in a directory in your environment, you can **mount** it using the corresponding action in the toolbar.




If the repository hasn't been copied to your environment yet, you can **clone** it using a dedicated action in the toolbar. When cloning, you will need to provide the URI of the repository and choose a target directory.

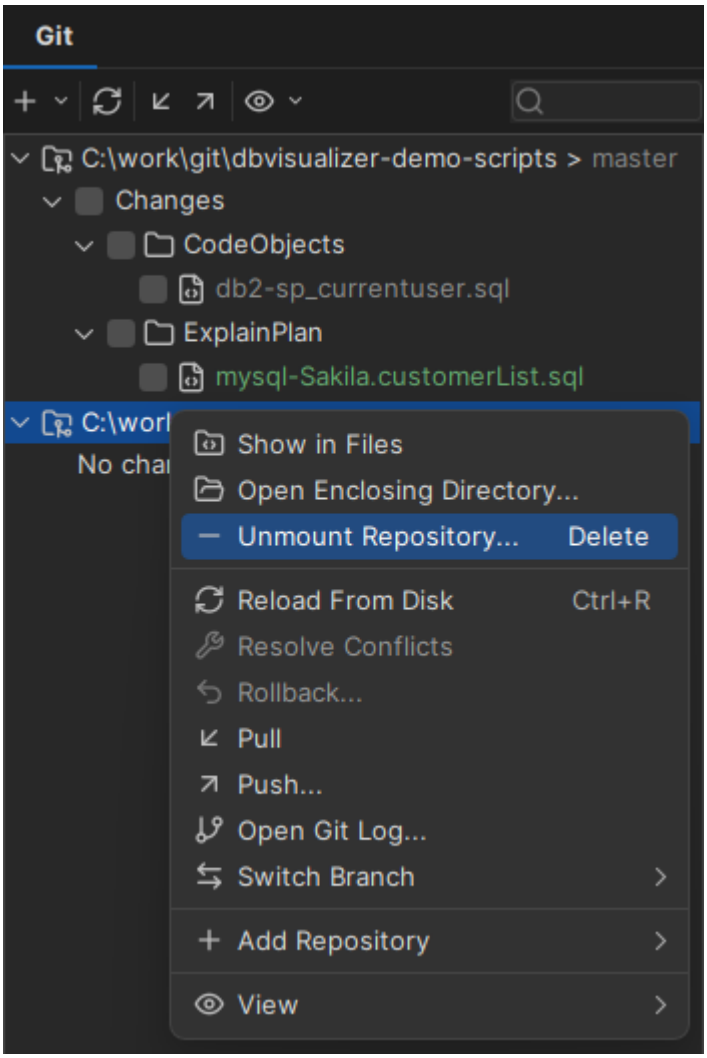



Several authentication methods are supported including integrated **GitHub authentication**, **username and password** (that can also be used for **personal access tokens**), as well as **SSH authentication**.




Mounted repositories can be **reordered** using drag and drop.

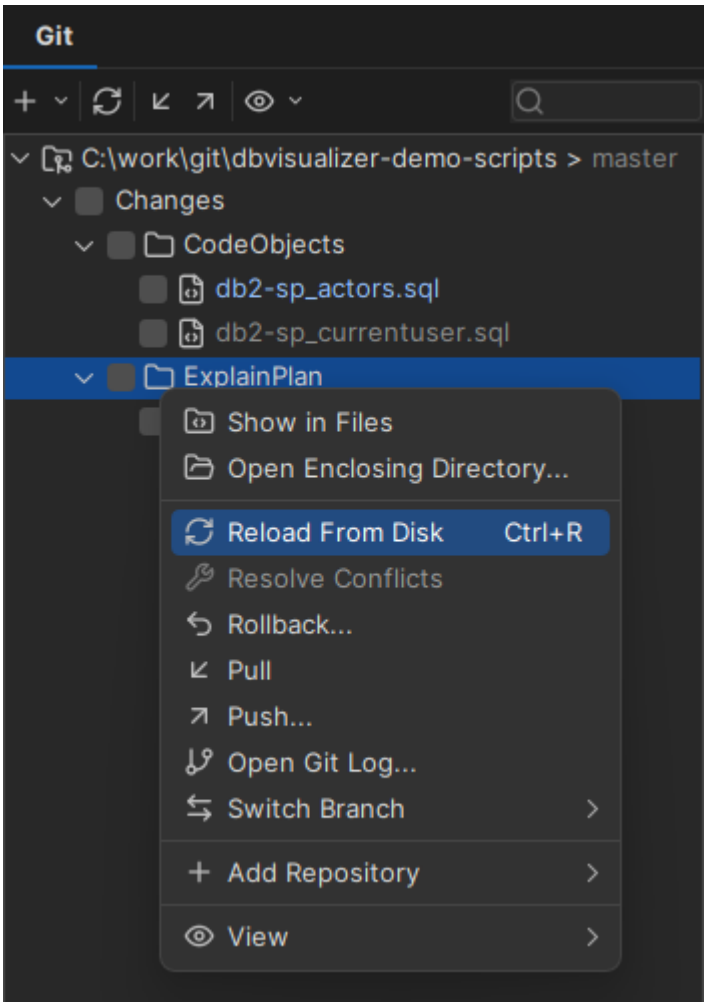
To **unmount** a repository, right-click and choose the corresponding action. Please note that while this will remove the repository from DbVisualizer, the actual files will remain intact.



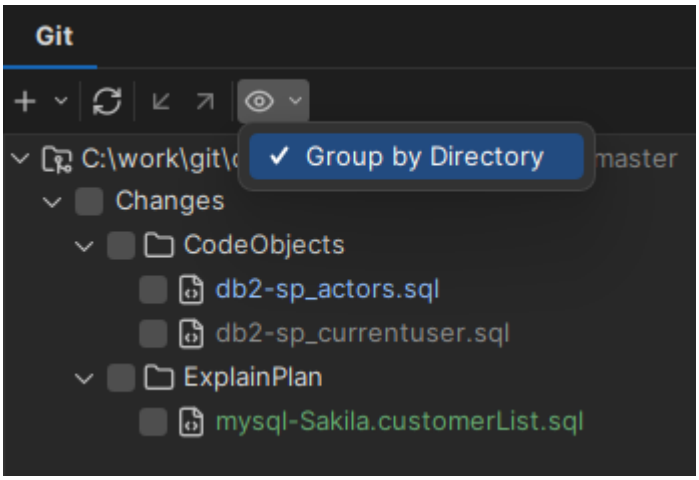

# Making changes

When you add, remove or modify files in a repository, the list of the changed files will appear under the corresponding repository node in the Git tab.

To refresh the list of changes, right-click on the repository and choose **Reload From Disk**. To reload all repositories at once, you can use the reload action in the toolbar.

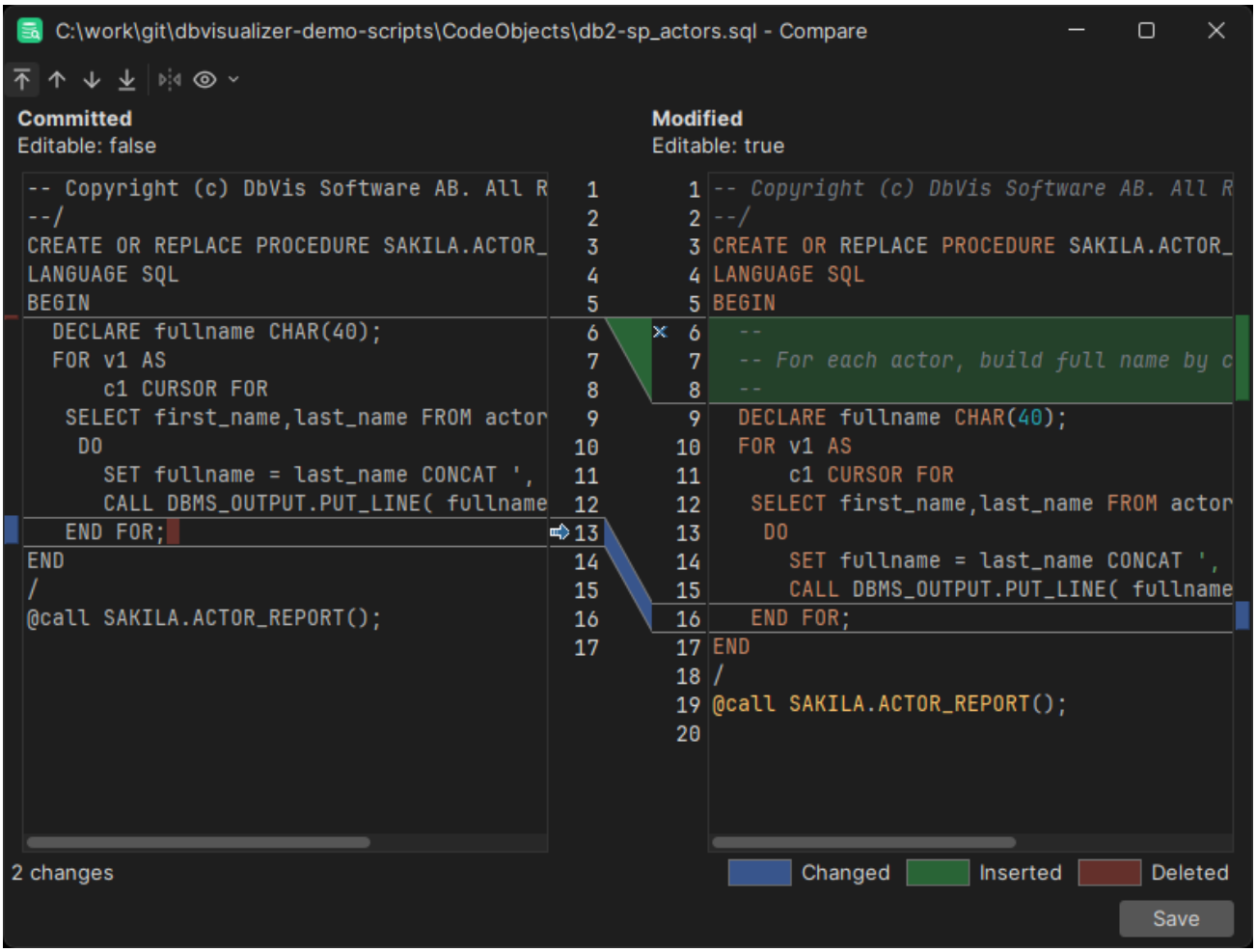



By default, the changed files are presented as a list. If you want to include the directory structure in the list of changes, you can change this using the View menu in the toolbar.

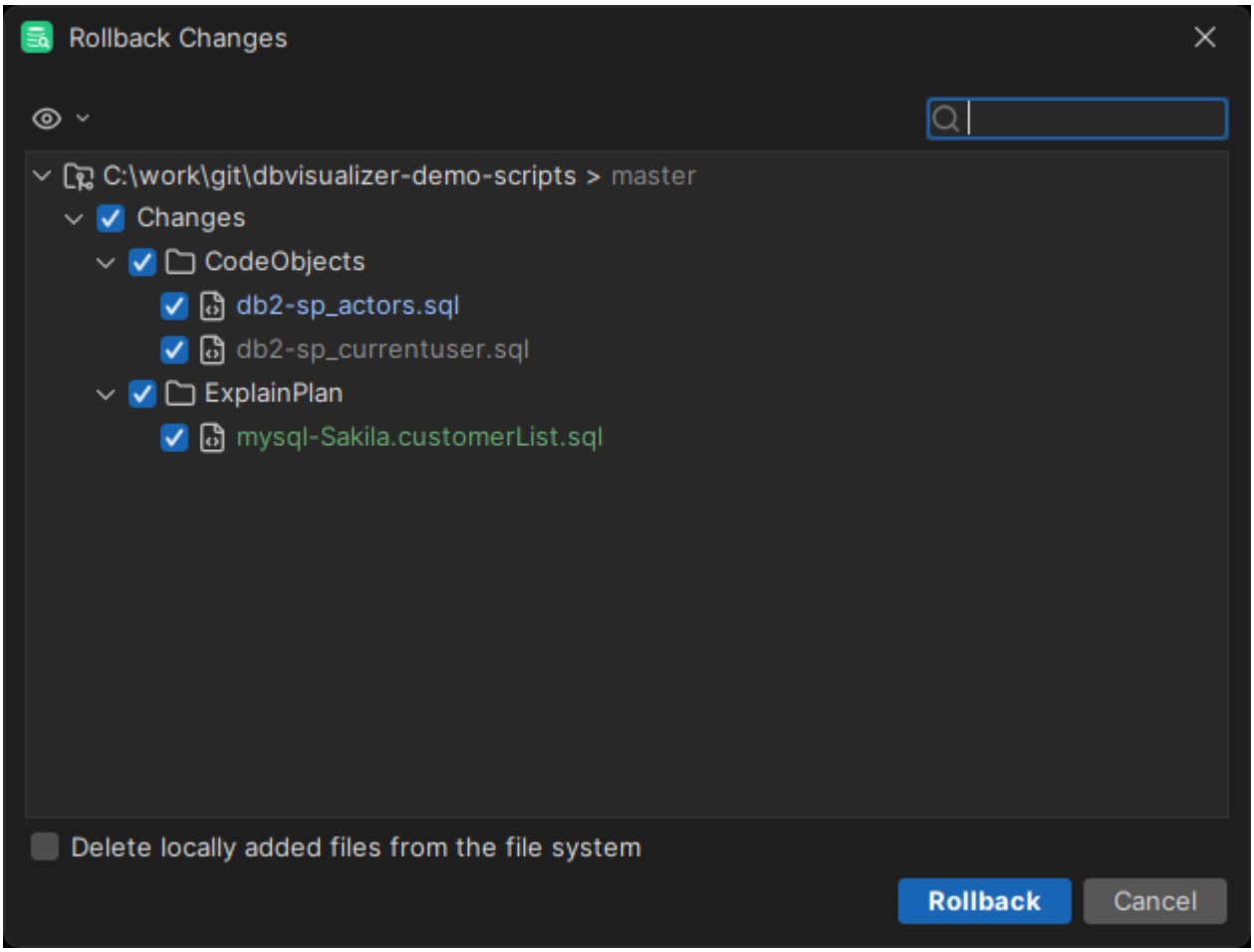


## Diff and rollback

To see the changes in a file, choose **Show Diff** in the context menu. This will open a diff view for the selected file, allowing you to examine the changes and make adjustments if necessary.

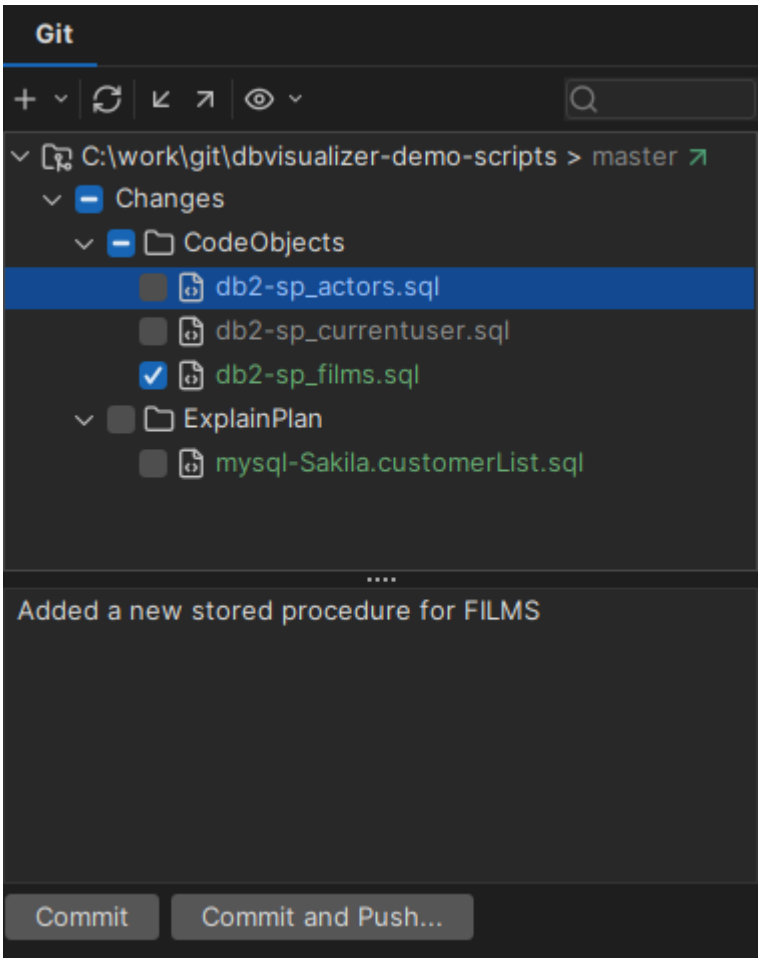



If you want to undo the changes in a file, choose **Rollback** in the context menu. This will revert the file to the state it was in when you last committed it. Choose the **Delete locally added files from the file system** option if you want to remove new files during rollback.



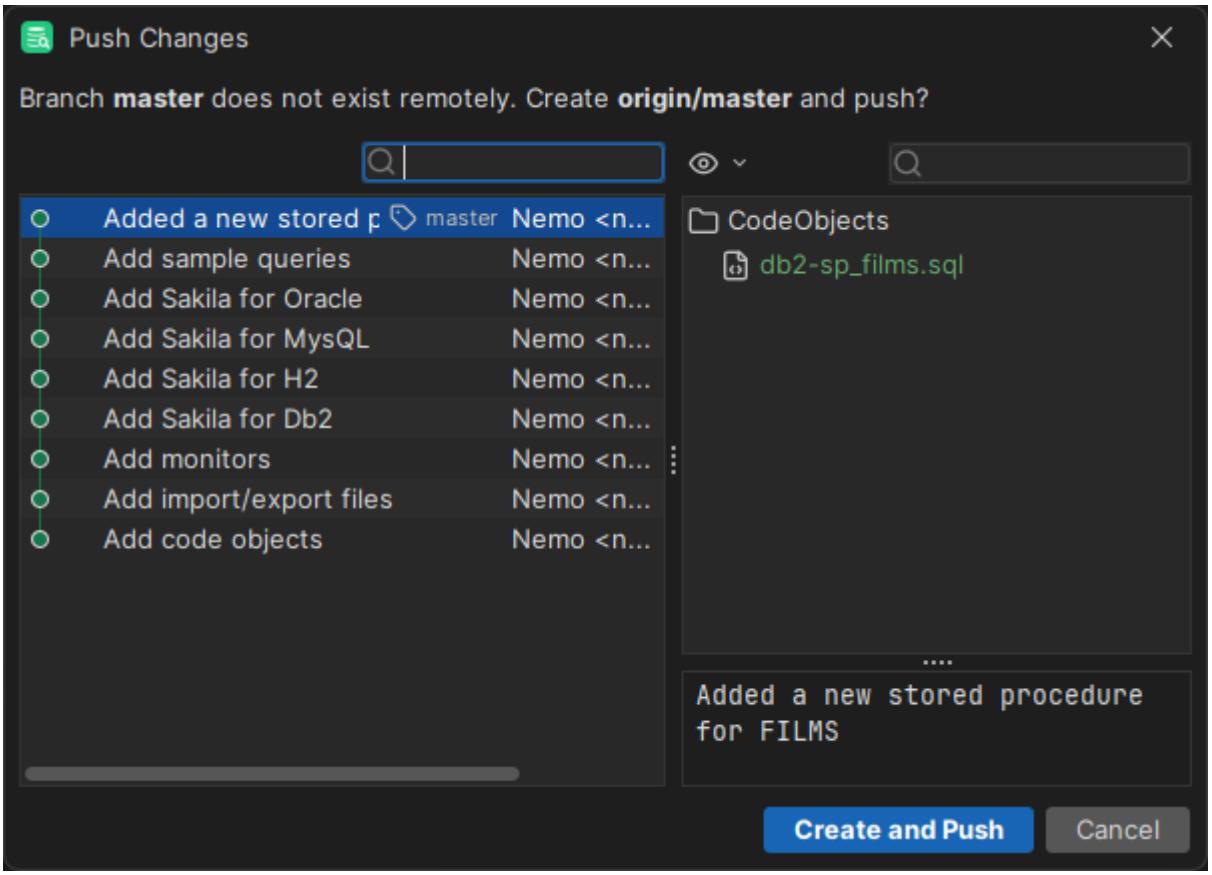

## Commit and push

When you are happy with your changes, you can commit them using the **Commit** action. Choose the files you want to include, and write a commit message that will describe the changes in the repository's commit history. Press **Commit** to create a local commit, or **Commit and Push** if you want to publish the changes to the remote repository at once.



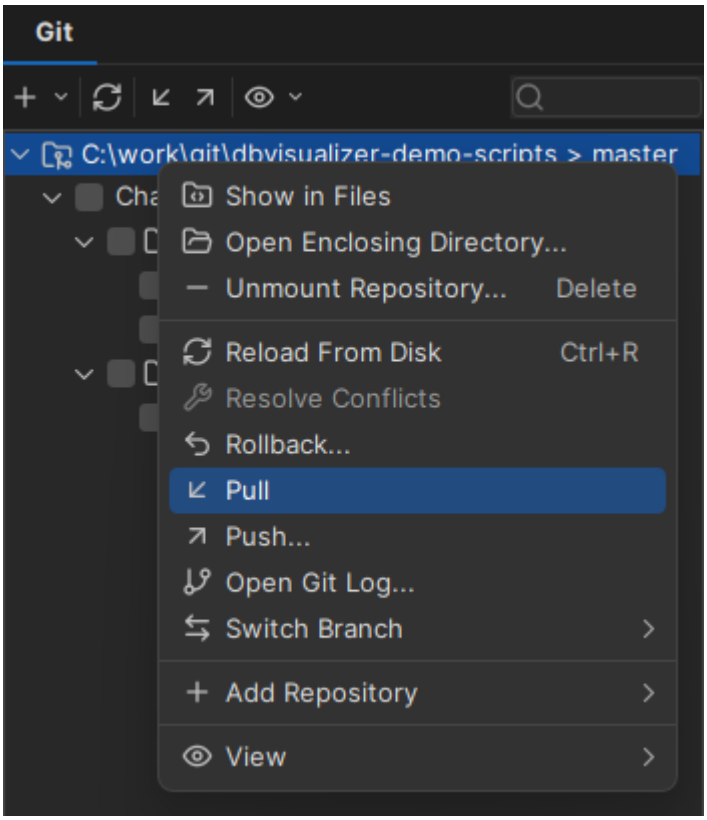

When you have local unpublished commits, an indicator will appear in the repository list. You can publish the changes using the **Push** action.

Screenshot: outgoing commits indicator and push dialog.




# Fetching remote changes

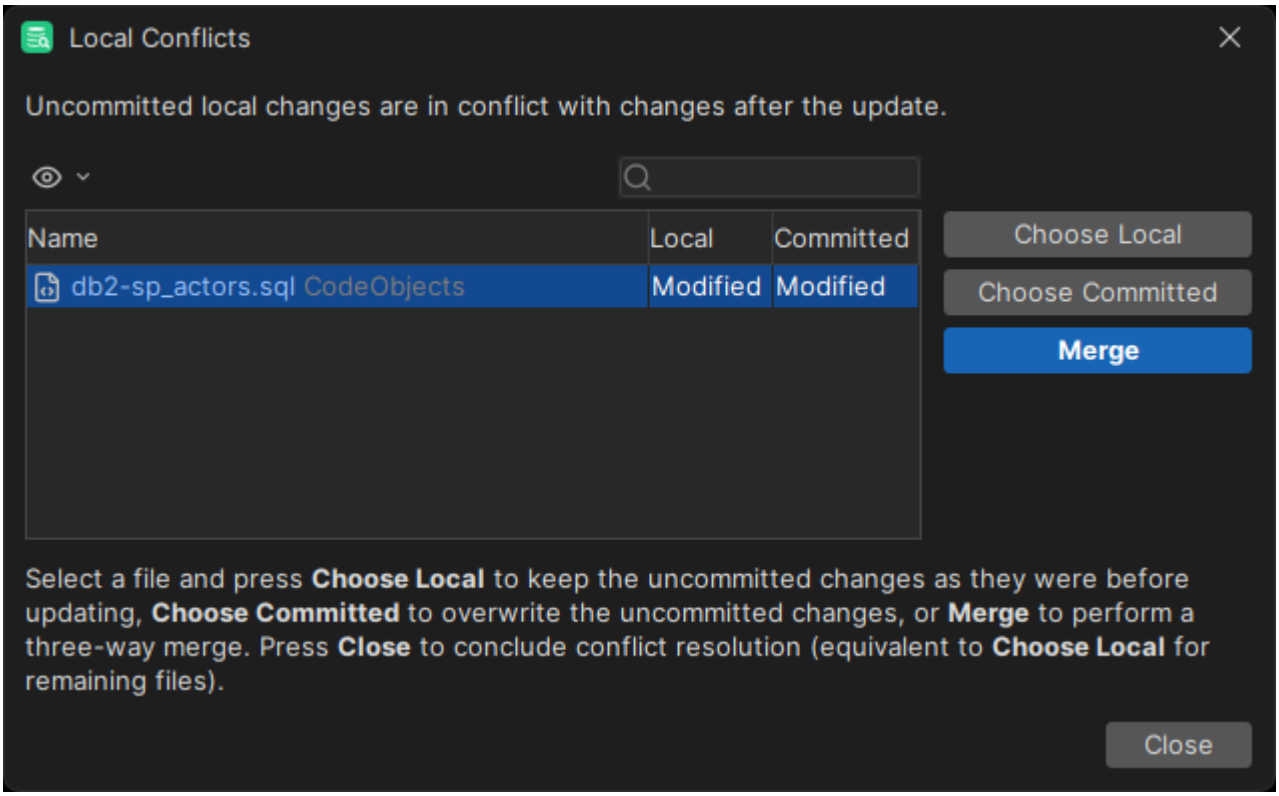
From time to time, you will want to fetch changes from the remote. This can be done using the **Pull** action.



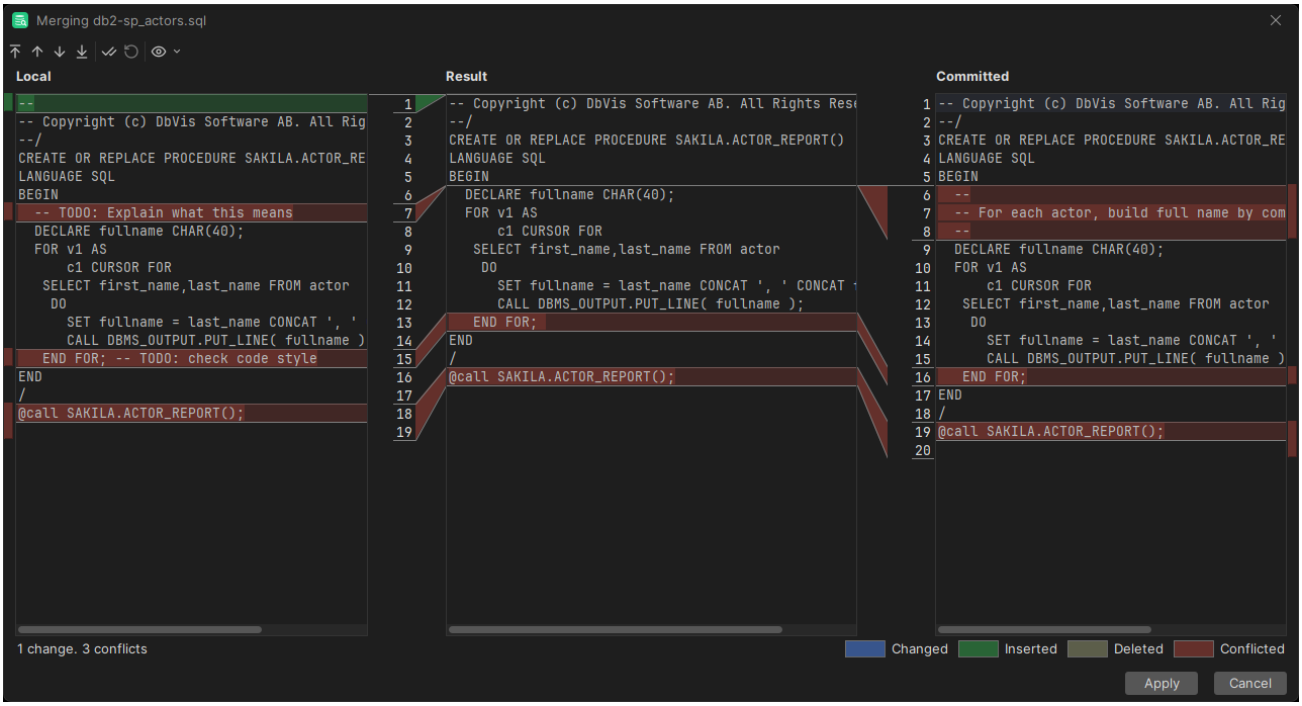

## Resolving conflicts

If there are no local changes, this is a simple operation. However, if you have uncommitted changes or unpublished commits, there may be conflicts between your changes and the changes in the remote repository. In this case, you will need to resolve the conflicts by **merging** the local and the incoming changes.

If conflicts are detected during pull, and they can't be handled automatically, a dialog listing the conflicting files will be displayed. For each conflict, you can double-click to see the diff, choose to accept the local or remote version, or to perform a manual merge of the conflicting changes.

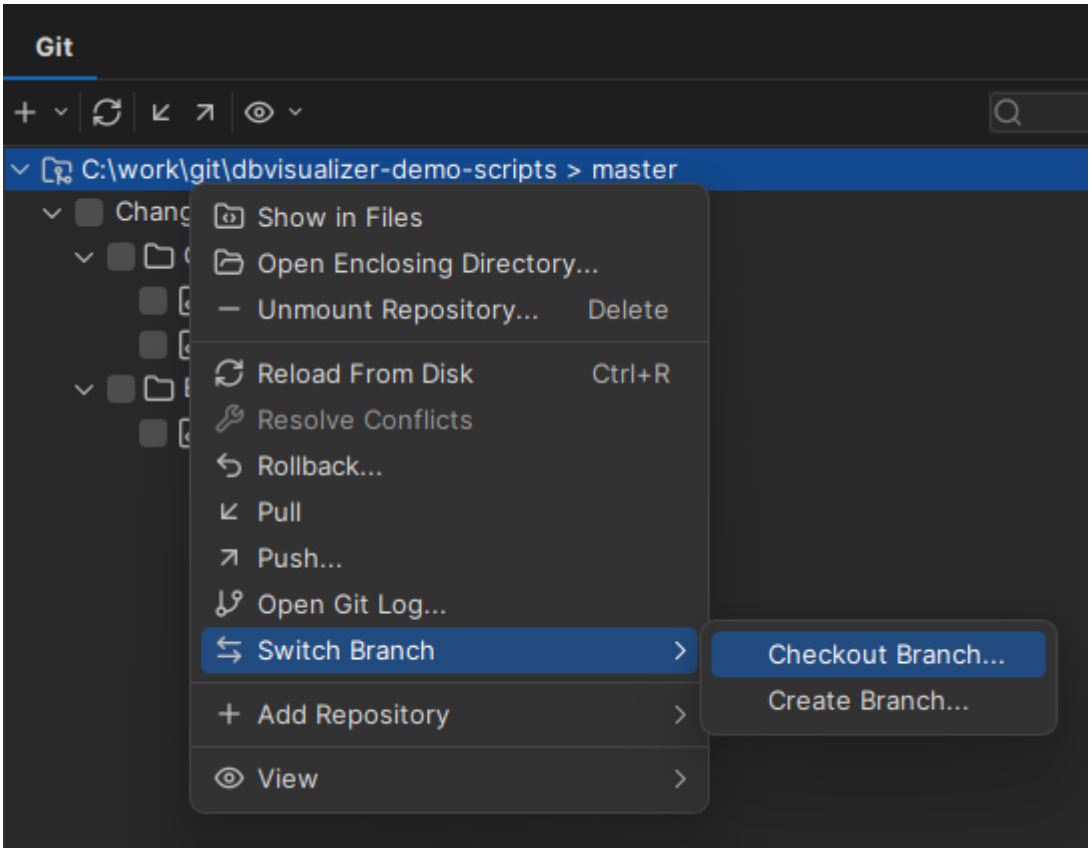



When merging a conflicting file, the two alternative versions are displayed on each side and the result of the merge is displayed in the middle. The changes are highlighted, and conflicts can be resolved one by one.

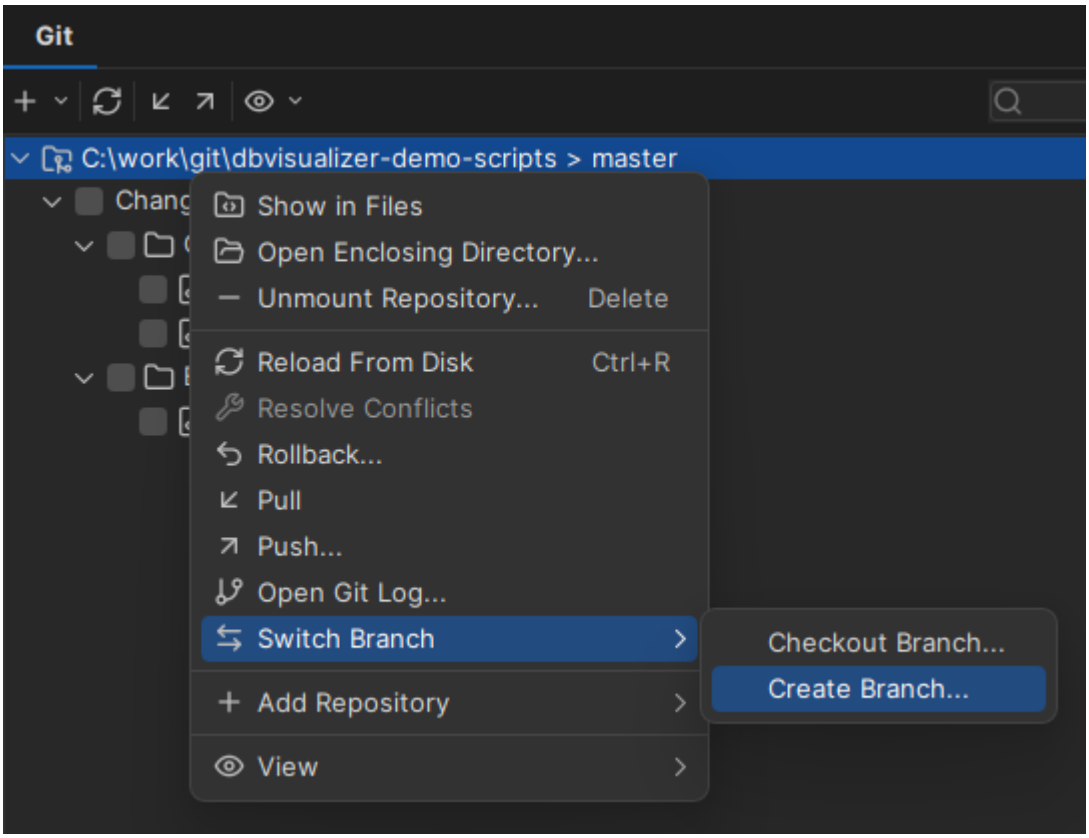


# Using branches

If you use branches in your workflow, you can switch between them using the **Checkout Branch** action. The list contains both local and remote branches. If you don't find the branch you are looking for, you can refresh the of branches from the remote in the same dialog.

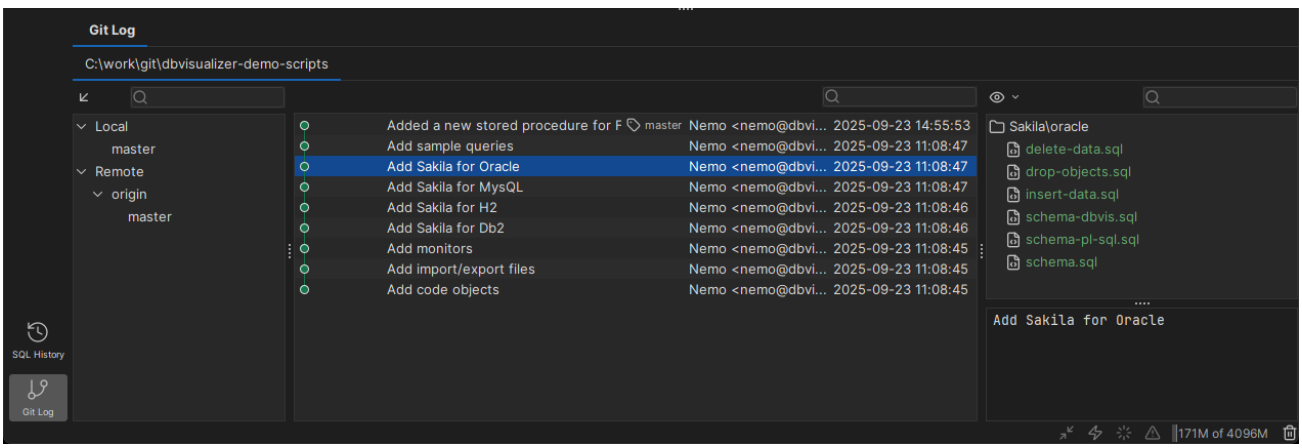



To create a new branch, click **Create Branch** and choose a name for the new branch.




# Browsing commit history

To examine the history of changes in a repository, choose **Show Log**, which will bring up the Git Log window. Here, you can choose among the existing branches add see the commit history of the selected branch, including the commit message, author, date, and any tags associated with the commit. For the selected commit, the list of changes and the full commit message are displayed on the right. Double-click on a change to see the corresponding diff.

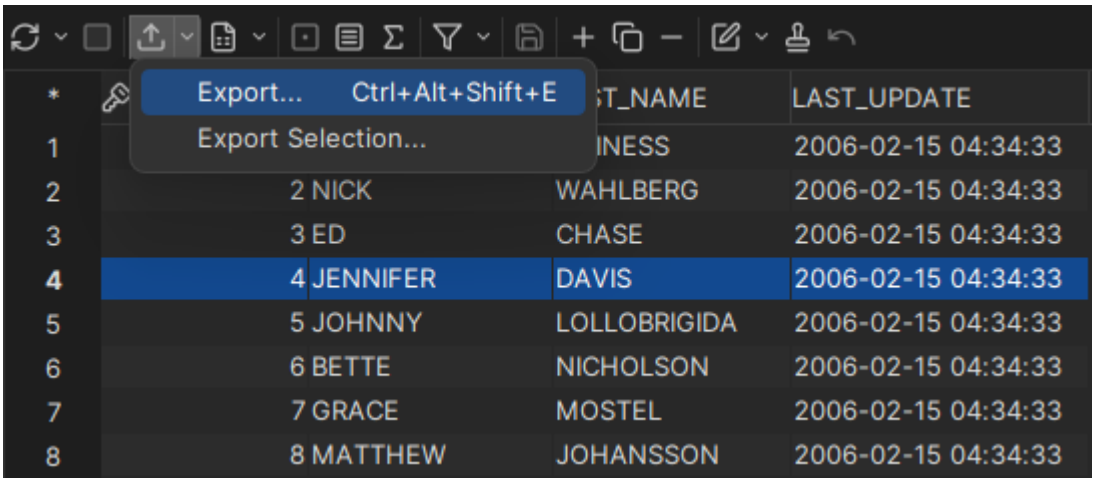


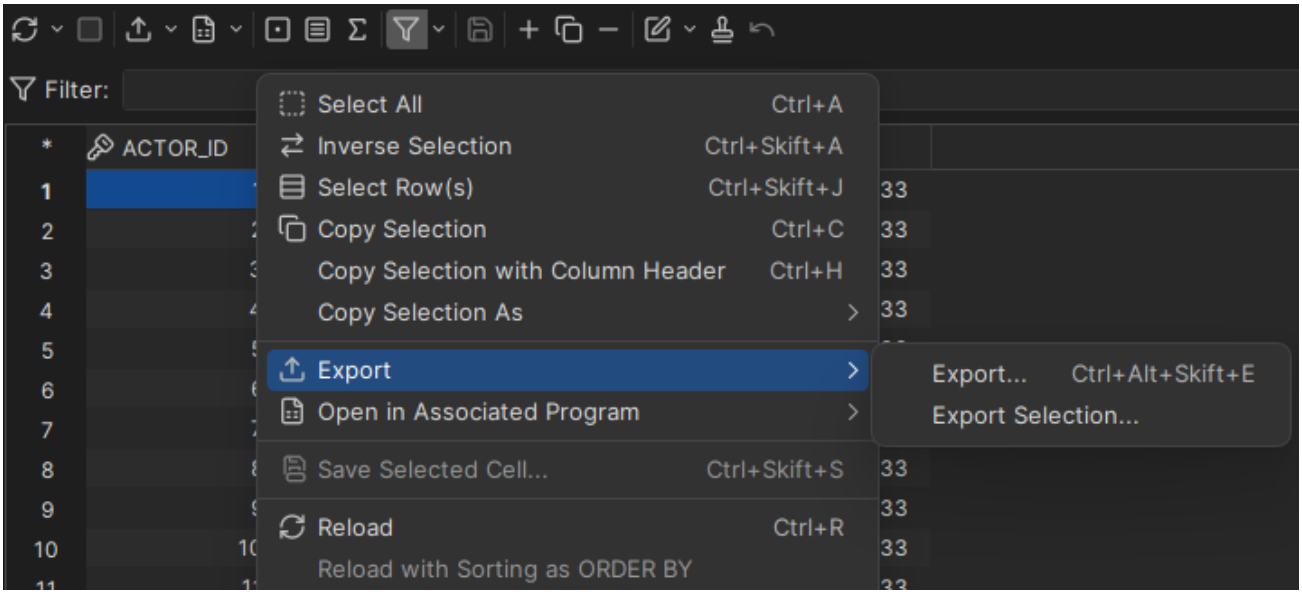
# Exporting a Grid

All grids in DbVisualizer can be exported to file, clipboard or into the SQL Commander using a number of formats.

The Export wizard gives you full control over the export (there is also an option to open the grid as a spreadsheet or in a web browser using predefined settings). The commands are available both on the right-click menu and on the toolbar and operate on either selected or all data:

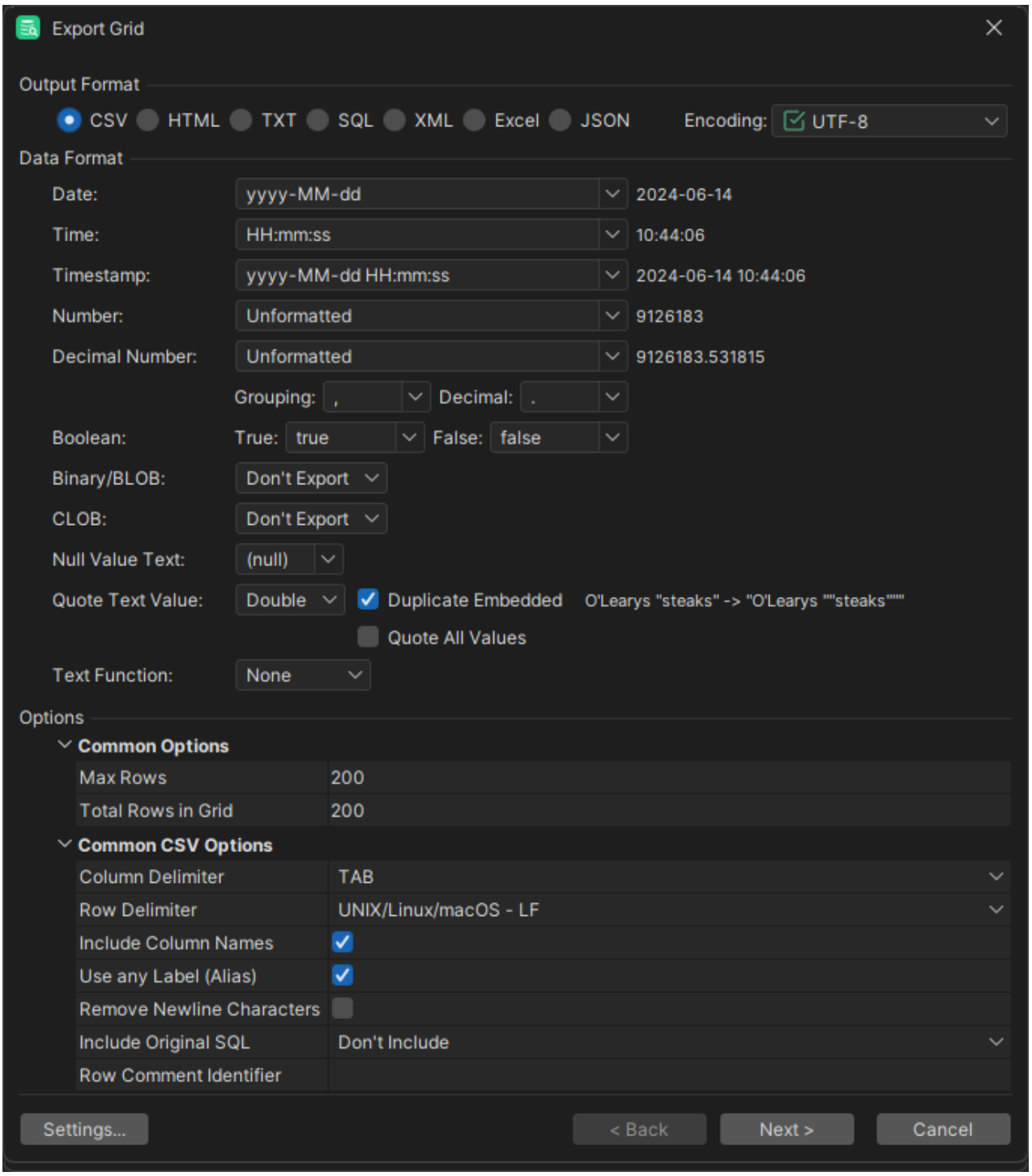
- **Export**: all data loaded into the grid (ignoring filters and sorting)
- **Export Selection**: data in selected cells



# Settings

The first wizard page is the **Settings** page, containing general properties for how the exported data should be formatted.



Select an output format, file encoding (it is also used to set the encoding in the HTML and XML headers, if you select one of those formats), and values for some different data formats (date, time, et.c.). For exporting to SQL, also see Exporting to SQL.

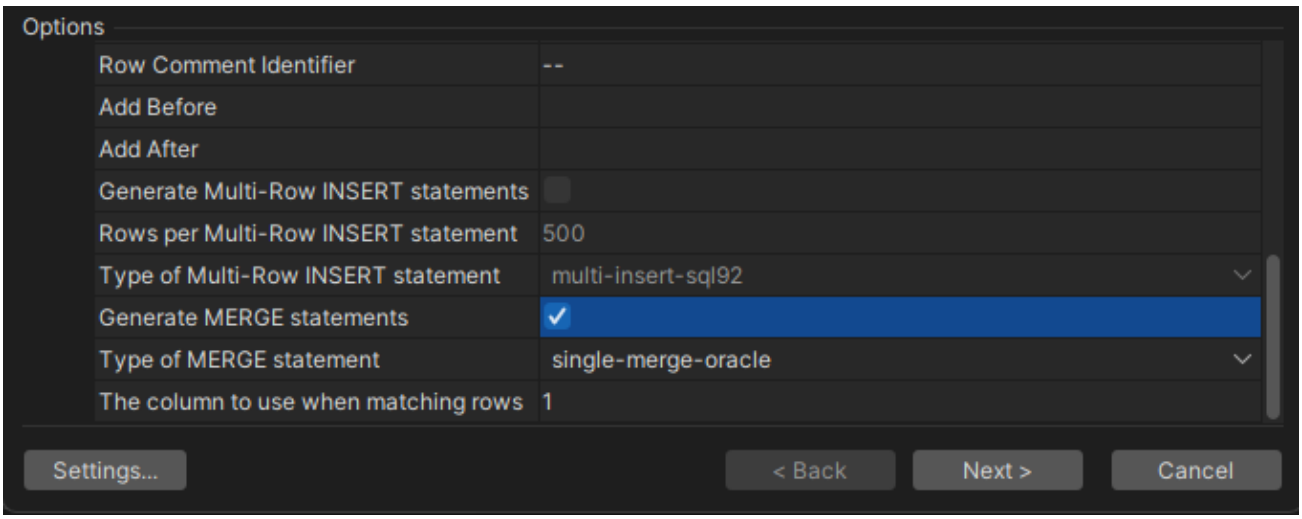
Note that the text to the right of the selections in the **Data Format** section shows an example of how the currently selected option would look like with real data.

**Excel** export do not support formatted date, time or timestamp. To get the correct date format in Excel you either have to export dates as text (check the **Export Date/Time as Text**) or set the format explicitly in Excel.

> **Only in DbVisualizer Pro**
>
> With the DbVisualizer Free edition, only the CSV and HTML formats are supported.

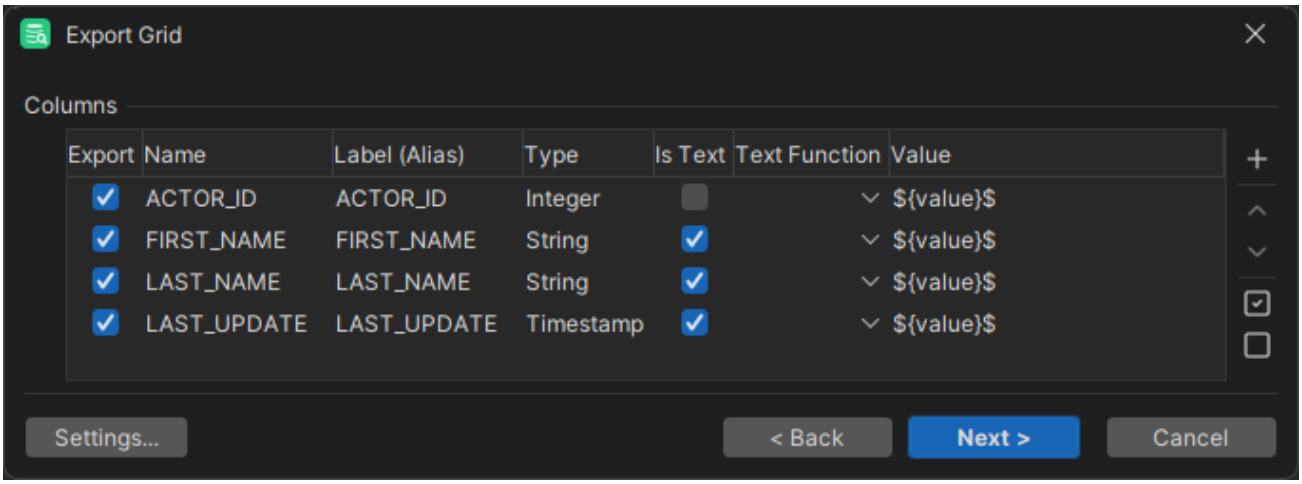
The **Options** section is used to define settings that are specific for the selected output format, for instance the column and row delimiters for the CSV format, or the Excel spreadsheet format.

**SQL** export has more complex options, like qualifiers and delimiters. Notably, there are options to set whether the **INSERT** statements should be separate or combined, or to instead generate **MERGE** statements.




# Data page

Clicking the **Next** button in the wizards moves you to the **Data** page. Use the columns list to control which columns to export and how to format the data for each columns. The list is exactly the same as the column headers in the original grid, i.e., if a column was manually removed from the grid before launching the Export Wizard, then it will not appear in this list.




The **Table Rows** fields show you how many rows are available and let you specify the number of rows to export. This setting along with the **Add Row** button is especially useful when you use the test data generation feature described in the next section.

The columns in this page's grid can be used like this.

| Column | Descriptions |
|---|---|
| **Export** | Defines whether the column will be exported or not. Uncheck it to ignore the column in the exported output. |
| **Name** | The name of the column. This is used if exporting in HTML, XML, XLS, JSON or SQL format. Column headers are optional in the CSV output format. |
| **Label (Alias)** | When you export a result set grid for a SELECT statement that uses column aliases, this column holds the alias. If you have also enabled **Use any Label (Alias)** in the **Options** section, this value is used in place of the name. |
| **Type** | The internal DbVisualizer type for the column. This type is used to determine if the column is a text column (i.e., if the data should be enclosed by quotes or not). |
| **Is Text** | Specifies if the column is considered to be a text column (this is determined based on the type) and so if the value should be enclosed in quotes. |
| **Value** | The default `${value}$` variable is simply be substituted with the column value in the exported output. You can enter additional static text in the value field. This is also the place where any test data generator variables are defined. |

## Generating Test Data

The test data generator is useful when you need to add random column data to the exported output.

The **Value** column in the **Data** page grid specifies the data to be in the exported output. By default, it contains the `${value}$` variable, which is simply replaced with the real column value during the export process. You can also add static values before and after the `${value}$` variable, to be exported as entered.

Alternatively, you can use test data generator variables in the **Value** column. The choices are available in the right-click menu when you edit the **Value** column.

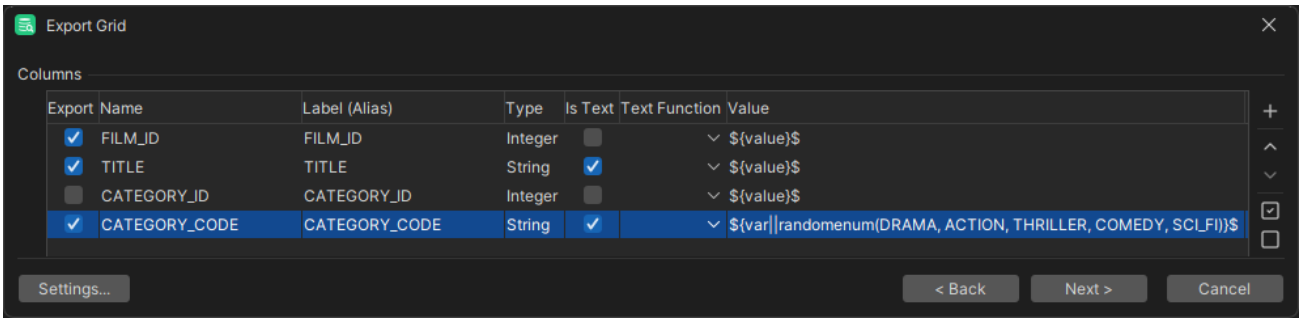
| Function Name | Function Call | Example |
|---|---|---|
| **Generate random number** | `${var\|\|randomnumber(1, 2147483647)}$` | Generates a random number between 1 and 2147483647 |
| **Generate random string of random size** | `${var\|\|randomtext(1, 10)}$` | Generates random text with a length between 1 an 10 characters |
| **Generate random value from a list of values** | `${var\|\|randomenum(v1, v2, v3, v4, v5)}$` | Picks one of the listed values in random order |
| **Generate sequential number** | `${var\|\|number(1, 2147483647, 1)}$` | Generates a sequential number starting from 1. The generator re-starts at 1 when 2147483647 is reached. The number is increased with 1 every time a new value is generated. |

Here is an example of how to use the test data generators to try out planned changes to the data. Consider this initial data:

| * | FILM_ID | TITLE | CATEGORY_ID |
|---|---|---|---|
| 1 | 1 | ACADEMY DINOSAUR | 6 |
| 2 | 2 | ACE GOLDFINGER | 11 |
| 3 | 3 | ADAPTATION HOLES | 6 |
| 4 | 4 | AFFAIR PREJUDICE | 11 |
| 5 | 5 | AFRICAN EGG | 8 |
| 6 | 6 | AGENT TRUMAN | 9 |
| 7 | 7 | AIRPLANE SIERRA | 5 |
| 8 | 8 | AIRPORT POLLOCK | 11 |
| 9 | 9 | ALABAMA DEVIL | 11 |
| 10 | 10 | ALADDIN CALENDAR | 15 |
| 11 | 11 | ALAMO VIDEOTAPE | 9 |
| 12 | 12 | ALASKA PHANTOM | 12 |
| 13 | 13 | ALI FOREVER | 11 |

After the changes, the **CATEGORY_ID** column should not appear in the output and the new **CATEGORY_CODE** should contain abbreviated category codes. To test this, we simply uncheck the **Export** checkbox for **CATEGORY_ID** entry and set the **Value** for the **CATEGORY_CODE** to use the **Generate random value from a list of values** function ( `${var||randomenum(DRAMA, ACTION, THRILLER, COMEDY, SCI_FI)}$` ).

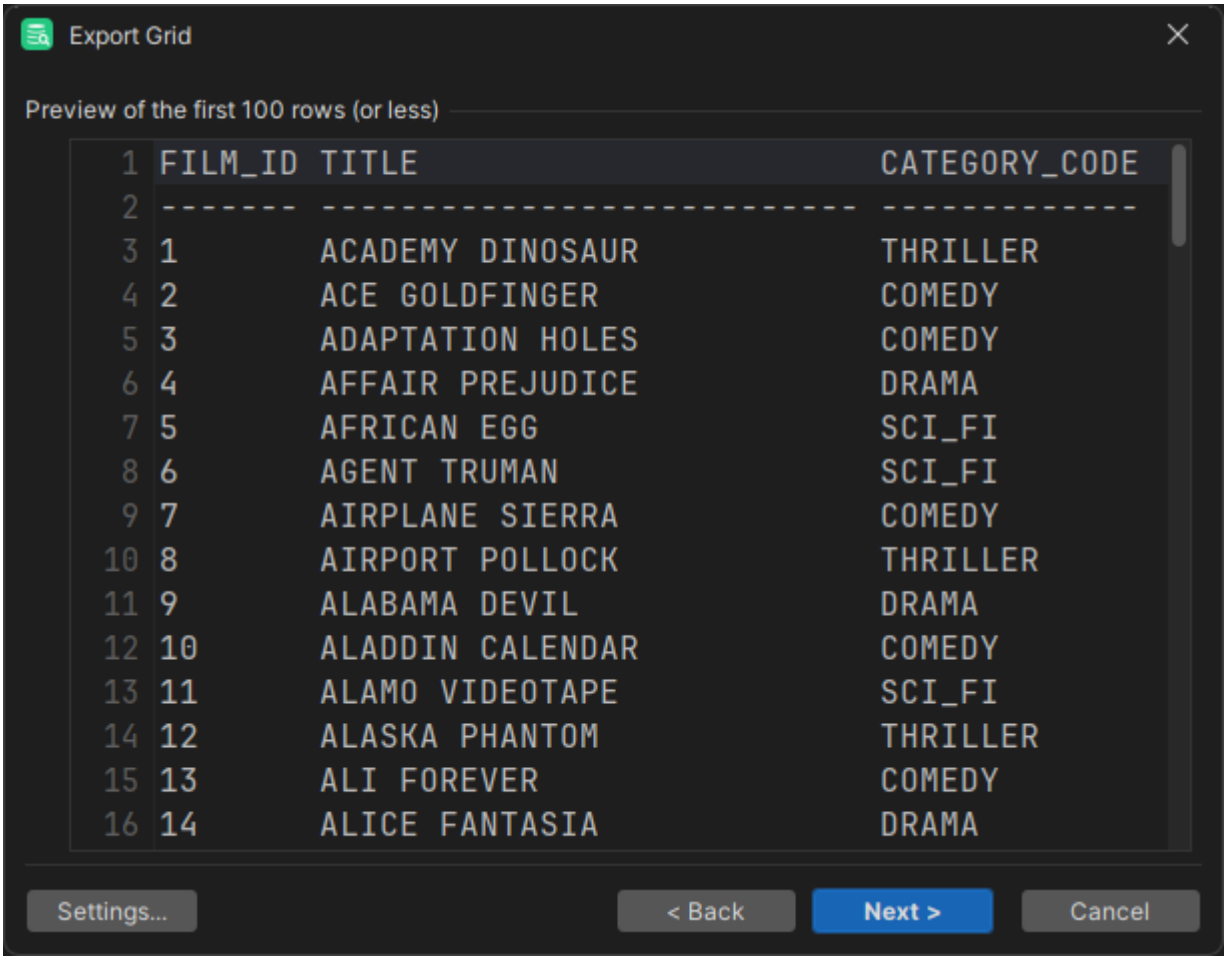
By unchecking the "Export" checkbox for the column **CATEGORY_ID** this column will not be included in the export.




# Preview

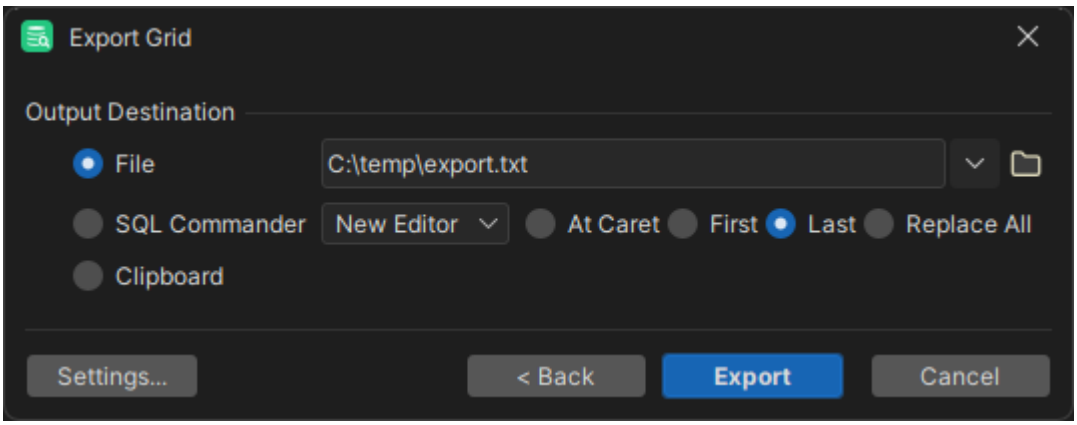
The third wizard page is the **Preview** page, showing the first 100 rows of the data as it will appear when it is finally exported. This is useful to verify the data before performing the export process. If the previewed data is not what you expected, just use the back button to modify the settings.

Previewing the data (or exporting it) in TXT format results in this:



# Output Destination

The final wizard page is the **Output Destination** page. The destination field specifies the target destination for the exported data, one of File, SQL Commander or Clipboard.




Click **Export** on this page to export the grid data to the selected destination.

# Settings Menu

If you often use the same settings, you can save them as the default settings for this assistant. If you use a number of common settings, you can save them to individual files that you can load as needed. Use the **Settings** drop-down button menu to accomplish this:

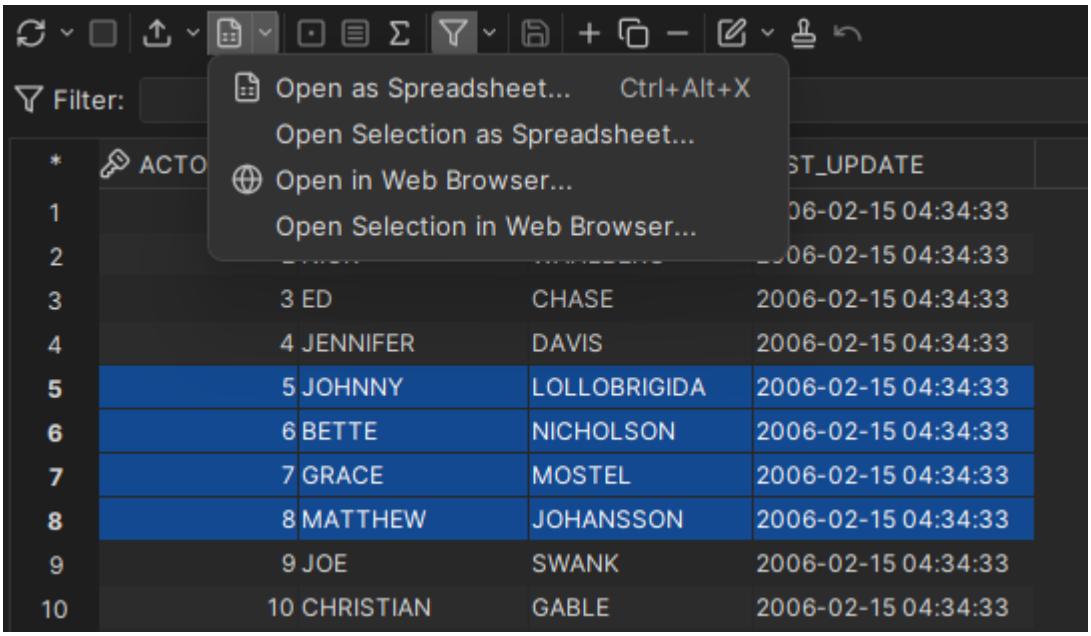
- **Save as Default Settings** Saves all format settings as default. These are then loaded automatically when open an Export Schema dialog
- **Use Default Settings** Use this choice to initialize the settings with default values
- **Remove Default Settings** Removes the saved defaults and restores the regular defaults
- **Load...** Use this choice to open the file chooser dialog, in which you can select a settings file
- **Save As...** Use this choice to save the settings to a file
- **Copy Settings to Clipboard** Copies the settings to the system clipboard

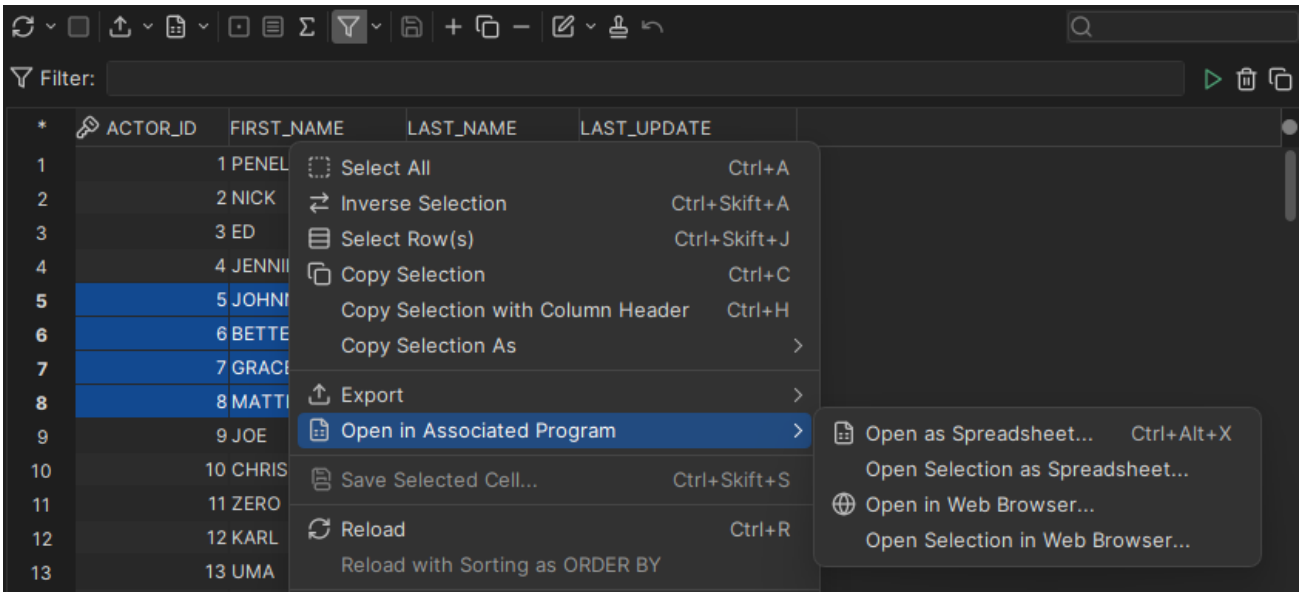
You can also use settings saved here with the @export client side command.

# Opening a Grid in an External Application

All grids in DbVisualizer offer options to open the grid data in an **external spreadsheet tool** (such as Excel) or a **web browser** using predefined settings (you can also export the grid to a number of formats with full control). The commands are available both on the right-click menu and on the toolbar and operate on either selected or all data:

- **Open as Spreadsheet**: all data loaded into the grid (ignoring filters and sorting)
- **Open Selection as Spreadsheet**: data in selected cells
- **Open in Web Browser**: all data loaded into the grid (ignoring filters and sorting)
- **Open Selection in Web Browser**: data in selected cells






When exporting to a spreadsheet, a separate sheet containing the SQL query used to generate the data will be included in the document.

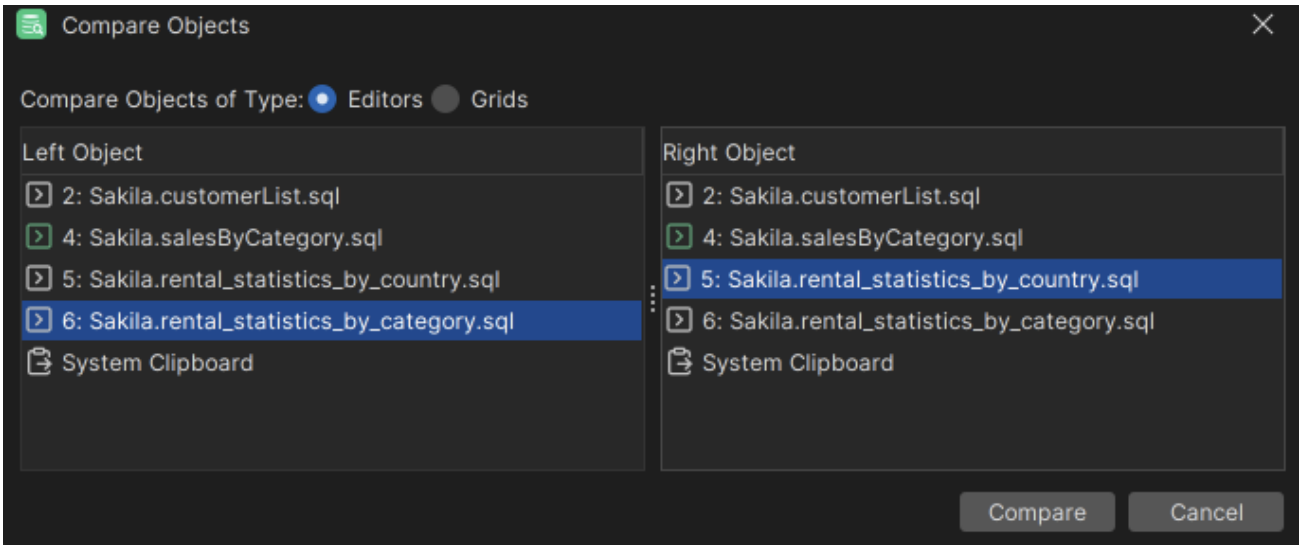
# Comparing Data

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

With DbVisualizer, you can compare grids and text data, such as scripts or the DDL for two tables or procedures.

## Selecting the Objects to Compare

You can open the **Compare Objects** object chooser via **Tools → Compare** and select two objects available for comparison.




If you select **Editor**, all **SQL Commander** editors and all **Object View** sub tabs that you have opened that contain text, such as **DDL** , **SQL Editor**, and **Procedure Editor** tabs, are listed. There is also an entry for the **System Clipboard**, holding the last text you copied.

Selecting **Grids** lists all **SQL Commander** Result Set tabs and all **Object View** sub tabs that you have opened that contain a grid, for instance the **Data** and **Columns** tabs for a table, or the **Tables** tab for a schema.

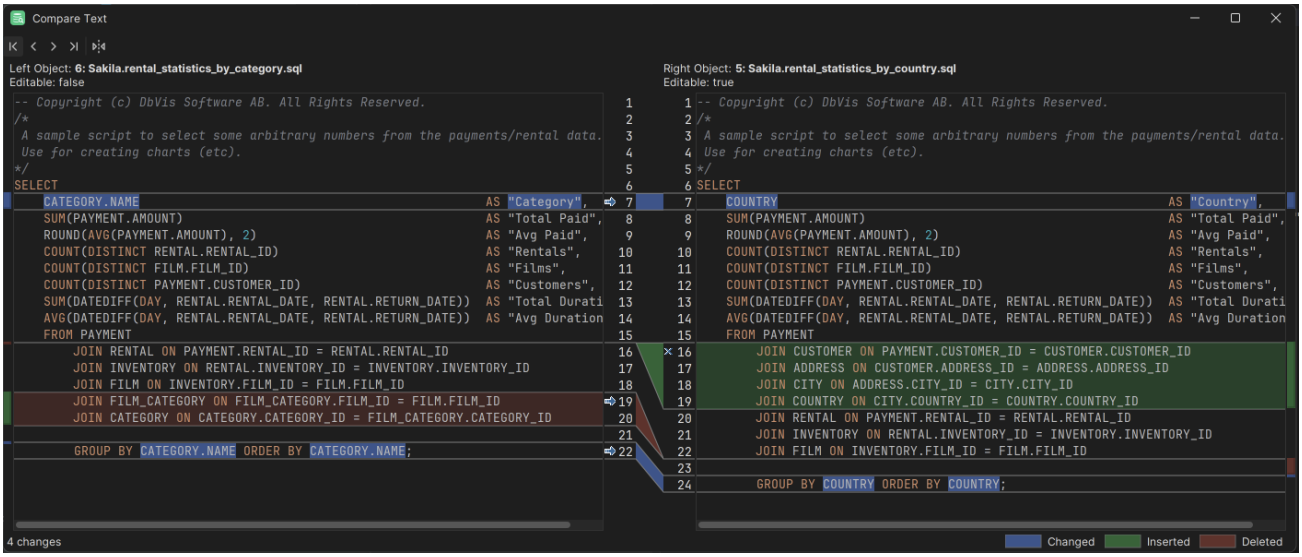
To compare two objects, select one each in the **Left Object** and **Right Object** columns and click **Compare** or double-click the entry.

You can also open the object selection dialog from the **Compare...** item in the right-click menu inside a tab that holds an object that can be compared. The object shown in that tab is then preselected as the **Right Object** so you only need to select the **Left Object** in the dialog.

The right-click menu for an **SQL Commander** editor also contains a **Compare to Saved** entry. This bypasses the object selection dialog and opens the **Compare** window directly, showing you how you have changed the script since loading it into the editor.

# Comparing Text Data

To compare text data, either select both text object from the **Tools → Compare** dialog or choose **Compare** from the right-click menu in one of them and select the one to compare to. The text compare window shows you how they differ.




The objects are shown side-by-side, with indications about how they differ in the divider between them. The margin areas outside the left and right objects have markings that also show sections of differences. You can navigate between the differences using the arrow buttons in the toolbar, or by clicking the markings to the left and right.

Comparing two texts is pretty straightforward. You can see what has been changed, inserted and deleted, with row level indications in the divider and details for changed text highlighted in the text panes. A difference may start on one line and span over multiple lines until a match is found again.

If the right object is editable, you can apply the changes needed to remove the differences, one by one or all at once.Along with the difference indications in the divider there are small icons: arrows and cross marks. Clicking on such an icon updates the modified object to match the original object, by inserting or deleting one or more rows or updating the text or column values.

If you want to apply all changes needed to make the right object match the left object, you can click on the **Sync** button in the toolbar (the second to rightmost button).

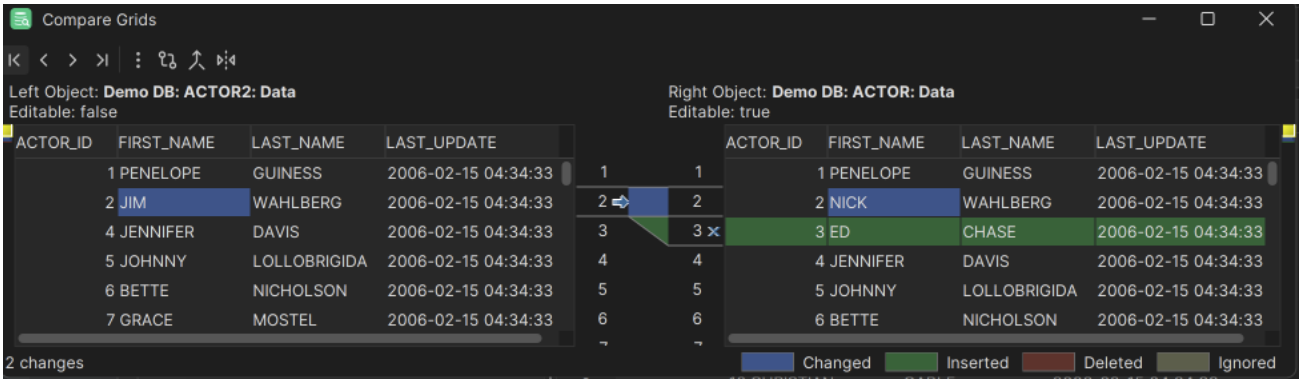
You can also edit the right object directly in the **Compare** window.

> No matter how you update the right object, all changes are also applied to the tab where the object was originally opened. To permanently save the changes, you need to use the **Save** button in that tab.

If you want to change sides when comparing objects, you can click the **Flip** button in the toolbar (the rightmost button). One reason for doing this may be that you want to update the object you originally used as the original instead of the modified object.

# Comparing Grids

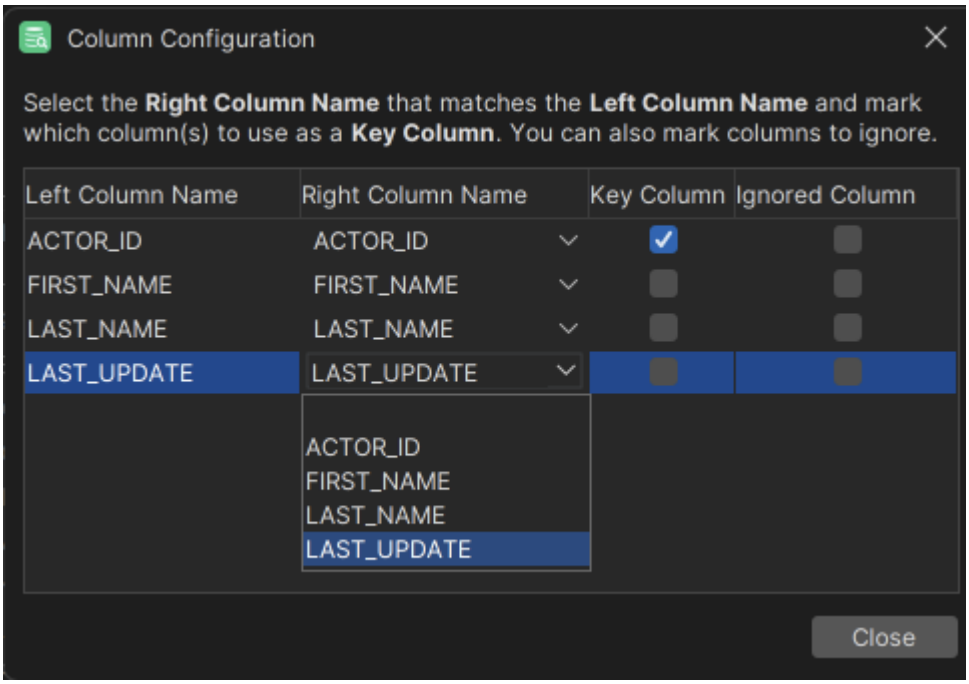
To compare grids, either select both grid object from the **Tools-** → **Compare** dialog or choose **Compare** from the right-click menu in one of them and select the one to compare to. The **Compare Grids** window shows you how they differ.




Comparing grids is a bit complicated, since each row is a unit in itself, with a unique identifier in the form of a Key. A difference can therefore not span rows. It is also important that both grids are sorted the same way and that the column in one grid is compared to the corresponding column in the other grid, otherwise the result is indeterminably. By default, columns are matched by name, or index if the names differ.

> Data is by default compared by its raw unformatted value meaning that any data type formatting set in **Tools-** → **Tool Properties** and in the **Data Formats** category are not considered. To compare using the visual formatted values, check **Compare Formatted Values** toolbar button.

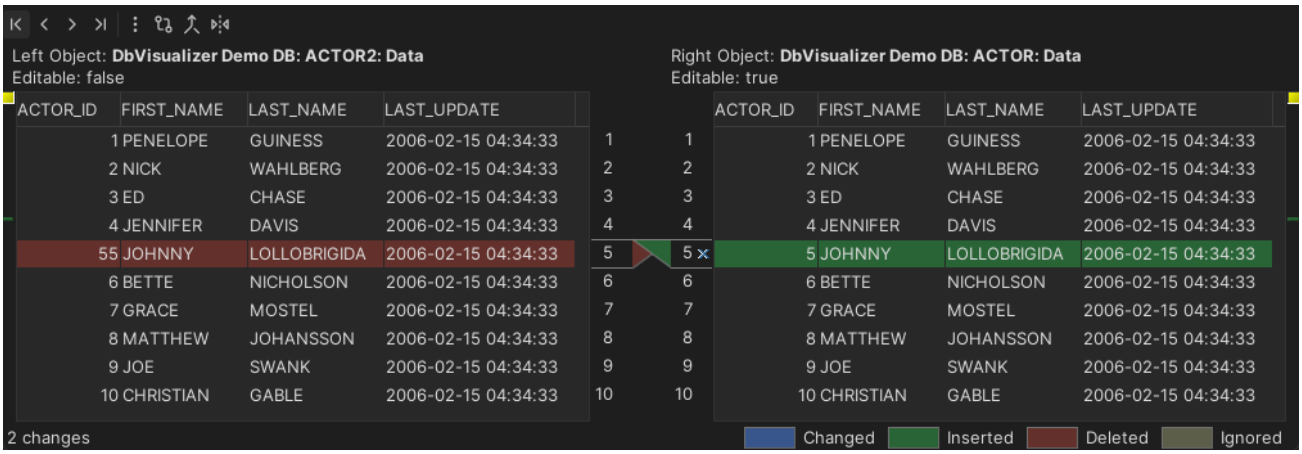
If the default matching is not correct, click the **Column Configuration** button in the toolbar to manually match the columns, and optionally select key columns and ignore some columns.




Use the **Right Column Name** drop down lists to select the column matching the **Left Column Name**. If a column does not match another, just leave it blank to exclude it from the comparison.

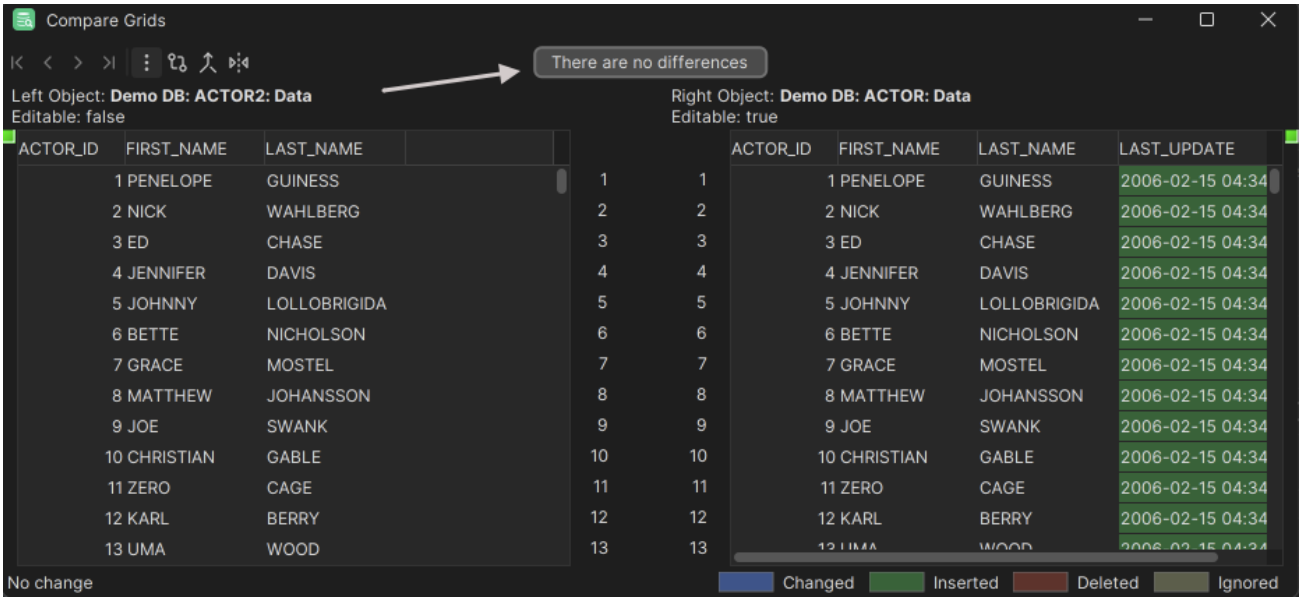
Use the **Key Column** check boxes to pick the columns that should be used as the key when comparing the grids. If you don't care about the value in some columns, e.g. a column that contains a timestamp that may vary between the grids without being an important difference, you can check the **Ignore Column** check box for that column to exclude it for the comparison.

It gets a bit more complicated if a key column value is changed.

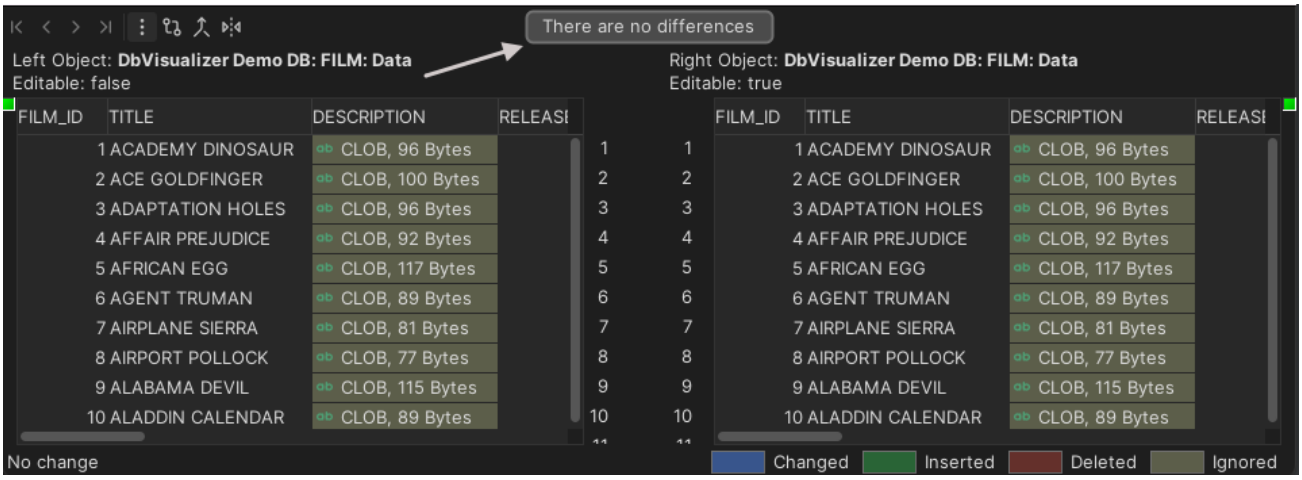



DbVisualizer considers a changed key value as one inserted row and one deleted row, as shown in the figure above. If the grids do not have any declared key columns, all columns are considered to be regular, non-key columns.

Two grids may also differ in the number of columns they contain. DbVisualizer finds columns that only exist in one of the grids and excludes their values when comparing the grids. If the column names do not match between the two grids, open the **Column Configuration** dialog to manually map the columns.




Similarly, DbVisualizer does not consider Binary/BLOB and CLOB columns when comparing, and marks them as ignored. You can manually specify that CLOB columns should be compared in the Column Configuration dialog.




Error rendering macro 'excerpt-include' : No location was provided.

# Comparing Cell Values

In a Data tab or a result set tab, you can also compare the values of two selected cells.

1. Select the two cells to compare,
2. Choose **Compare Two Selected Cells** from the right-click menu.

The values are compared and shown the same way as when comparing text data, with the exception that editing is disabled.

# Monitoring Data Changes

## Monitoring Data Changes

With the monitor feature, you can track changes in data over time, viewing the results of one or many SQL statements either as grids or graphs. Typically, you configure the monitor to run the statements automatically at certain intervals.

The monitoring feature combined with the charting capability in DbVisualizer Pro is really powerful, delivering real time charts of many result sets simultaneously. For example, you can use monitoring to spot trends in a production database, surveillance, statistics, database metrics, and so on.

Any SQL statement that produces a result set can be monitored, and when you monitor multiple statements, different statements may use different database connections concurrently.

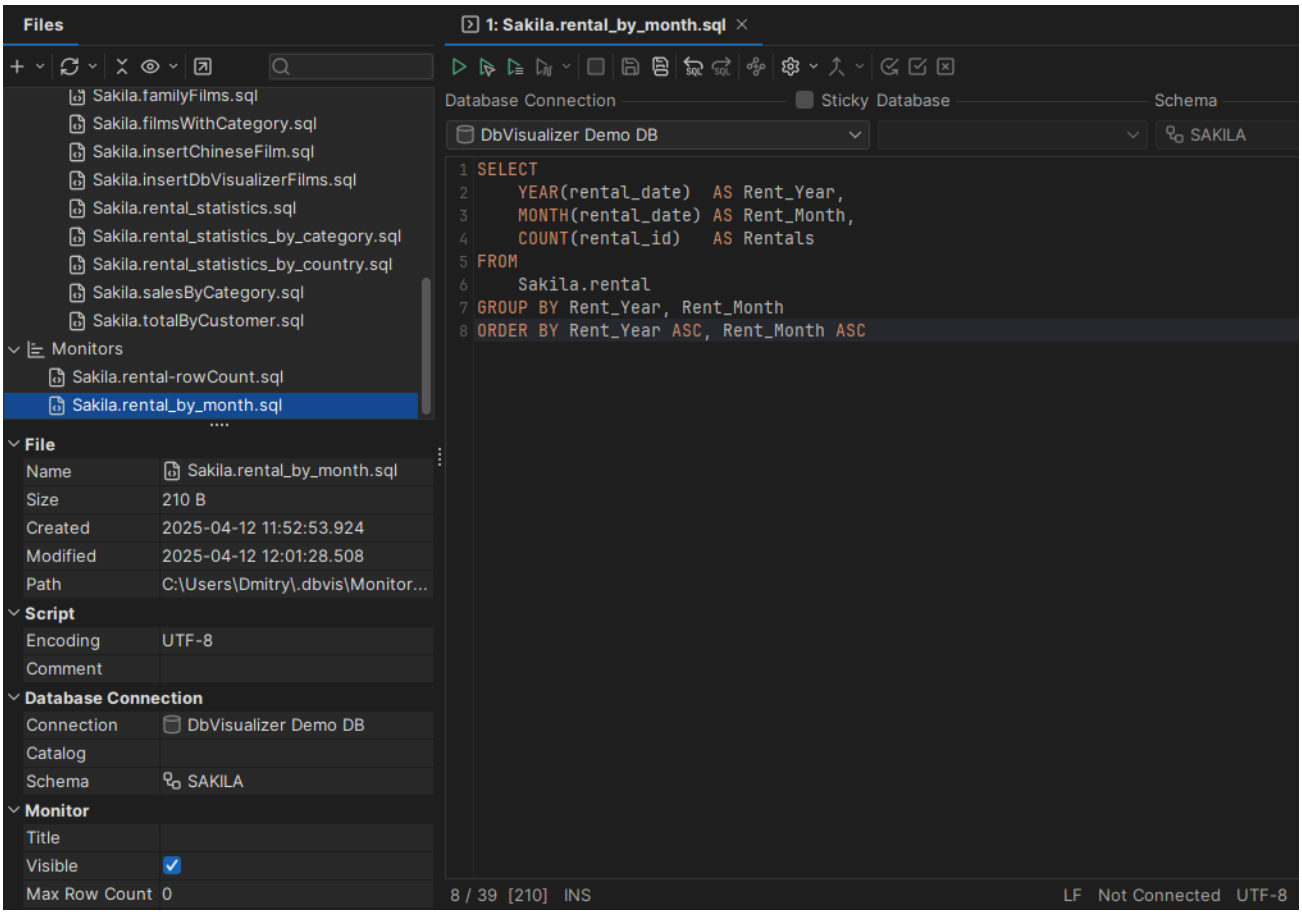
# Creating a Monitored Query

Monitored SQL statements are managed under the **Monitors** node in the **Files** tab in the tree area to the left in the main DbVisualizer window.

A monitor is basically a regular SQL script with some additional information and controls. You create it like any other script but place it in the **Monitors** folder.

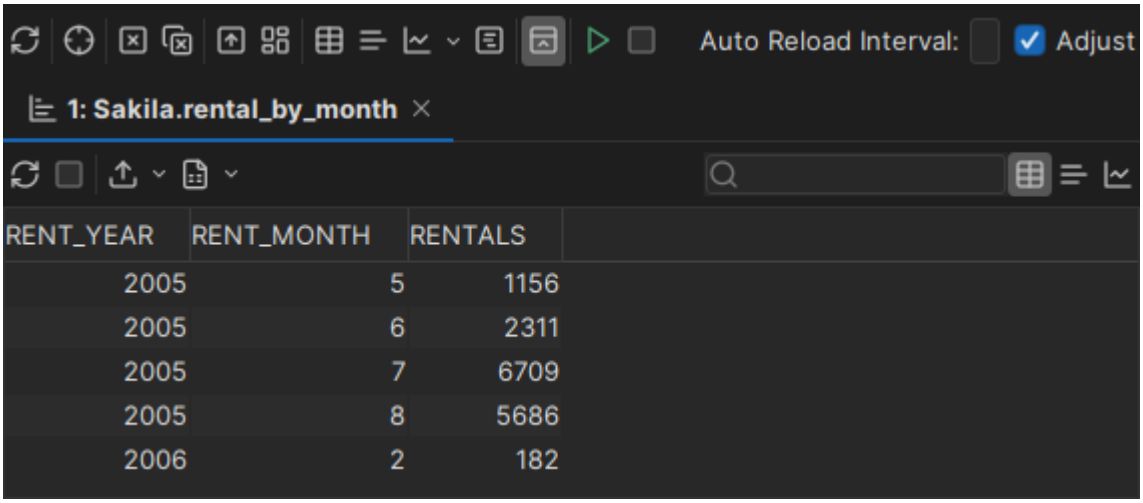
> Note that the monitor is not supposed to run scripts with multiple statements, and especially not client-side commands (@). Only statements returning a single result set (i.e. select statements) are supported.

A monitored SQL script is associated with information about the target database connection and (optionally) the catalog (the JDBC term which translates to a database for some databases, like Sybase, MySQL, SQL Server, etc) and schema. It also has a title, a maximum row count (how many results to keep track of) and a visibility status (whether the monitored statement result should be included in the Monitors windows, discussed below). This information is displayed, and can be edited, in the lower part of the **Files** tab, along with information about the file that holds the monitored statement. If you don't want to see these details, you can disable it with the **Show Details** toggle control in the right-click menu for a node.




The figure above shows the Rental/Month monitored statement and the SQL associated with it.

The following is an example of the result set produced by the statement:




You can create and work with monitored statements in the same way as with a Bookmark. The main difference is how they are used and a couple of additional ways monitored statements can be created. For information about how to manually create, manage and share monitored statements, please see the Managing Frequently Used SQL page.

The following sections describe how you can get help creating the bookmarks for a couple of cases that are commonly used for monitoring.

## Monitor table row count

It is very common to want to keep track of how the number of rows in a table varies over time. The right-click menu in the grid for a table or result set therefore has a **Create Row Count Data Monitor** operation that creates a monitored statement for you automatically.

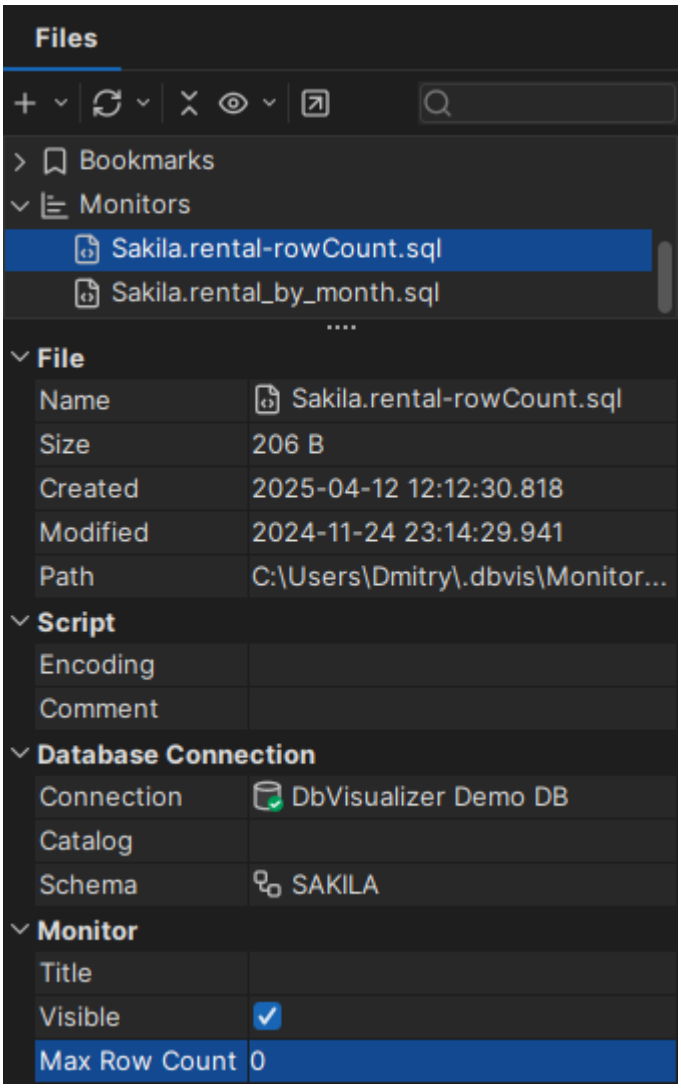
It creates a monitor with SQL for returning a single row with the timestamp for when the monitor was executed and the total number of rows in the table at that time. Every time the monitor is executed, a new row is added to the grid, up to a specified maximum number of rows. When the maximum row limit is reached, the oldest row is removed when a new row is added. Example:

| PollTime | NumRows |
|---|---|
| 2016-01-23 12:19:10 | 43123 |
| 2016-01-23 12:11:40 | 43139 |
| 2016-01-23 12:21:10 | 43143 |
| 2016-01-23 12:22:40 | 43184 |
| ... | ... |

The SQL for this monitor uses two variables, **dbvis-date** and **dbvis-time**. These variables are substituted with the current date and time, formatted according to the corresponding Tool Properties settings. The reason for using these variables instead of using SQL functions to retrieve the values is simply that it is almost impossible to get the values in a database-independent way. Another reason is that we want to see the client machine time rather than the database server time. You can, of course, modify the SQL any way you see fit, as long as the **PollTime** and **NumRows** labels are not changed.

```
SELECT '${dbvis-date}$ ${dbvis-time}$' AS PollTime,
       COUNT(*) AS NumRows
FROM RENTAL
```

DbVisualizer keeps the result for previous executions from the Monitor, up to the specified maximum number of rows, so that you can see how the result changes over time. You define the maximum number of rows in the **Max Row Count** field in the details area at the bottom of the **Files** tab. This property is initially set to 100 when you use **Create Row Count Data Monitor** to create the monitor.




You can change the value to limit or extend the number of rows that DbVisualizer should keep. Setting it to 0 or a negative number tells DbVisualizer to always clear the grid between executions of monitors.

## Monitor table row count difference

In addition to tracking the number of rows in a table over time, you may want to see by how many rows the value changes. You can create a monitor for this purpose with the **Create Row Count Diff Data Monitor** operation, available in the right-click menu for the grid.

In addition to the **Row Count Monitor**, the **Row Count Diff Monitor** reports the difference between the number of rows in the last two executions:

| PollTime | NumRows | NumRowsChange |
|---|---|---|
| 2016-01-23 12:19:10 | 43123 | 0 |
| 2016-01-23 12:11:40 | 43139 | 16 |
| 2016-01-23 12:21:10 | 43143 | 4 |
| 2016-01-23 12:22:40 | 43184 | 41 |
| ... | ... | ... |

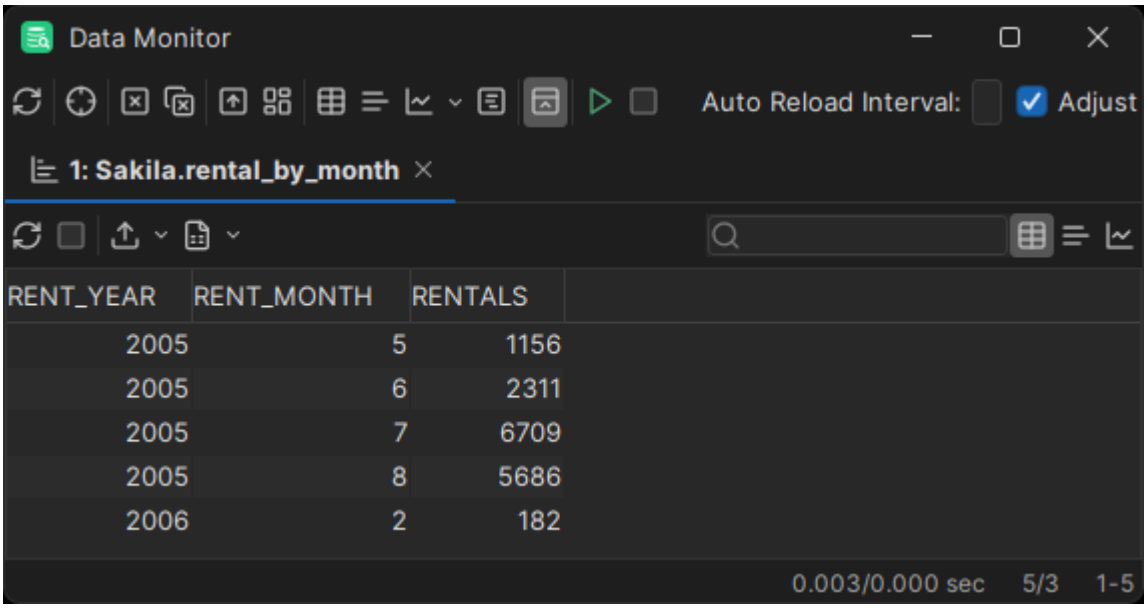
The SQL for this monitor adds a third column, named **NumRowsChange**. It utilizes the fact that DbVisualizer automatically creates variables for the columns in a monitor result set, holding the values from the previous execution. The **NumRowsChange** column is set to the value returned by the count(*) aggregate function for the current execution minus the value from the previous execution, held by the **NumRows** variable. All columns in a monitor result set can be used like this to reference values from the previous execution of the monitor.

```
SELECT '${dbvis-date}$ ${dbvis-time}$' AS PollTime,
       COUNT(*) AS NumRows,
       COUNT(*) - ${NumRows||count(*)}$ AS NumRowsChange
from RENTAL
```

# Running a Monitored Query

The Monitor window, launched via the **Tools → Data Monitor** menu option, is where you activate monitors and look at the results. The monitor tabs can be rearranged in the same way as all other tabs, pretty much any way you like. Please see Getting the Most Out of the GUI for details.

The monitor results can be viewed only as grids in DbVisualizer Free, while DbVisualizer Pro adds the capability to view them as charts or text.




The Monitor window has a toolbar at the top with an **Auto Reload Interval** field and a **Adjust** box. The **Auto Reload Interval** field is used to control how often, in seconds, to execute the monitors when auto update is running. The specified number of seconds may be increased automatically by DbVisualizer if the total execution time for all monitors is longer than the specified value. Check the **Adjust** box and the Monitor feature will automatically increase the number of seconds so that all monitors will complete before next auto-update.

The rest of the window holds result areas for each monitored statement with the **Visible** attribute enabled. Each individual monitor result tab or window may also have a toolbar with controls that apply just to that result. The screenshot is from DbVisualizer Pro, with **View** buttons in the toolbar for the selected monitor; these buttons are not included in DbVisualizer Free.

When auto reload is running, there is a progress bar shown to the left of the **Auto Reload Interval** field. This progress bar is only shown when auto update is running.

The main toolbar buttons have the following functions:

| Toolbar Button | Description |
|---|---|
| **Close** | Closes the Monitor window |
| **Reload** | Reloads all results (i.e., executes all monitors and updates the result sets) |
| **Locate Current** | Locates and select the monitor node in the Files tab corresponding to the currently selected result |
| **Clear Current** | Clears the currently selected result |
| **Clear All** | Clears all results |
| **Show as Tabs** | Shows the results as collapsed tabs |
| **Show as Windows** | Shows the results as tiled tabs |
| **Show Grids** | Shows all results as grids |
| **Show Text** | Shows all results as text |
| **Show Chart** | Shows all results as graph in the selected chart type |
| **Show/Hide Chart Legends** | Toggle this to show/hide chart legends |
| **Show/Hide Monitor Toolbars** | Toggle this to show/hide toolbars for each monitor |
| **Start Monitors** | Starts auto-update of all monitors, repeatedly executing all statements at the intervals specified by the Auto Reload Interval field |
| **Stop Monitors** | Stops the auto-update |

In the Tool Properties dialog, you can enable **Show Monitor Window at Startup** and **Start Monitors Automatically**, in the **Monitor** category under the General tab.

# Accessing Frequently Used Objects

## Accessing Frequently Used Objects

When you work on many different tasks, it is important to easily find and use the data and scripts you need.

DbVisualizer helps you by keeping the tabs you use open between sessions and letting you organize references to objects and scripts.


- Keeping Tabs Open Between Sessions
- Using Favorites
- Using the Files Tab

# Keeping Tabs Open Between Sessions

If you often work with the same objects and a few scripts, you can ensure that the Object View and SQL Commander tabs for these objects remain open between DbVisualizer sessions.

1. Open **Tools → Tool Properties**,
2. Select the **Tabs** category,
3. Enable one or both of **Preserve SQL Commander tabs between Sessions** and **Preserve Object View tabs between Sessions**,
4. Click **Apply** or **OK** to apply the new settings.

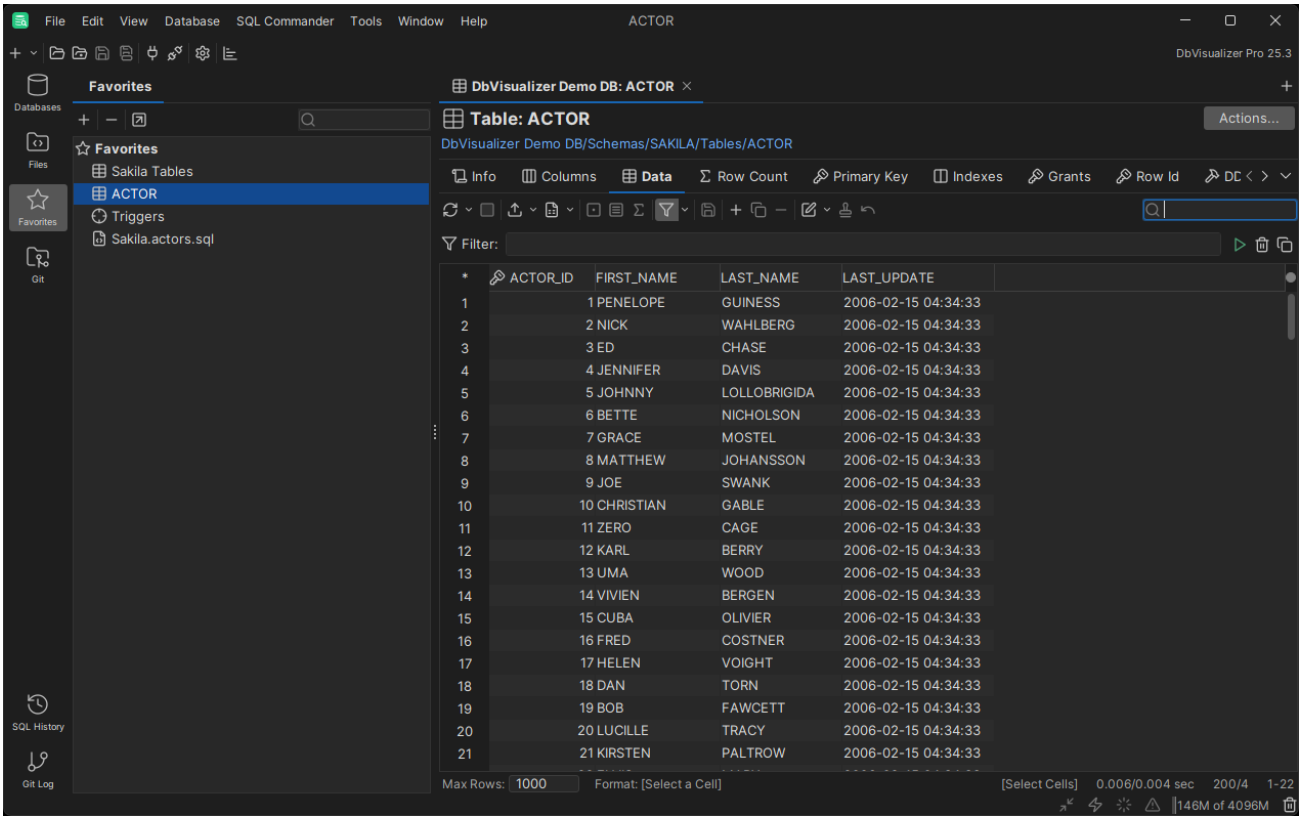
This feature is enabled by default for SQL Commander tabs but not for Object View tabs.

The content of the SQL Commander tabs is saved at regular intervals so when you restart DbVisualizer, the content is the same as where you left off.

For Object View tabs, you can also enable **Preserve Object View tabs at Disconnect**. By default, Object View tabs for objects that belong to a connection are closed when it is disconnected.

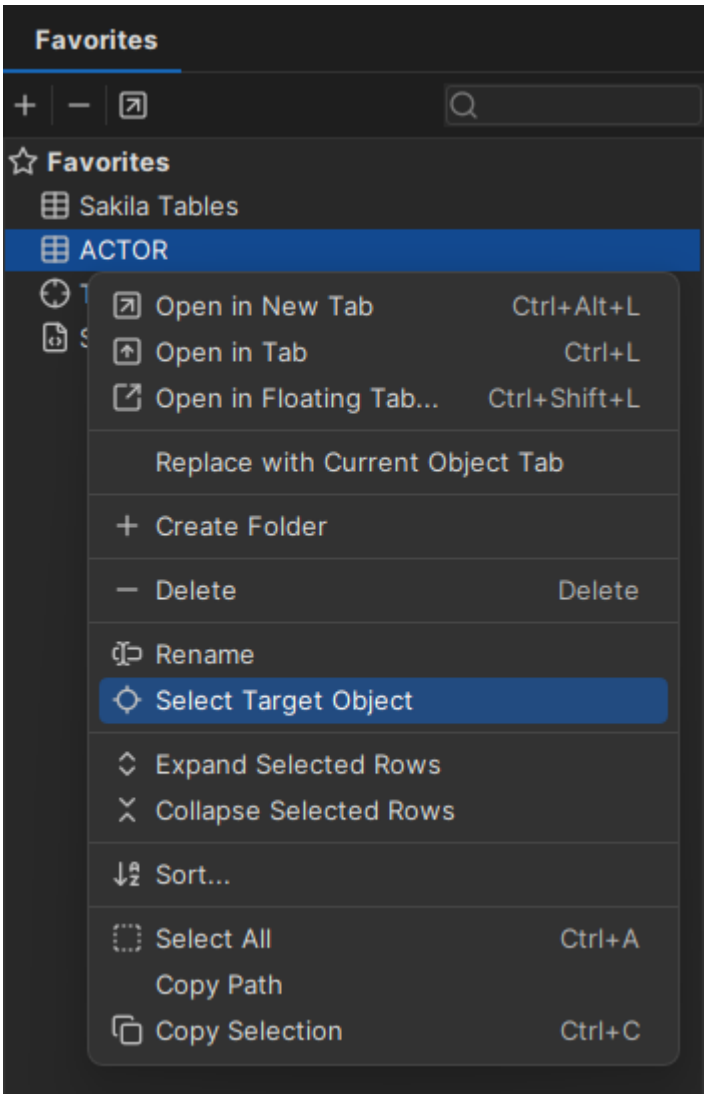
# Using Favorites

Navigating the **Databases** tab tree down to a specific object (such as a table) can be quite time-consuming for database objects that you work with often. By adding these objects to the Favorites tab, you have one-click access to them instead. In addition to database objects, you can also add script files to the Favorites tab.



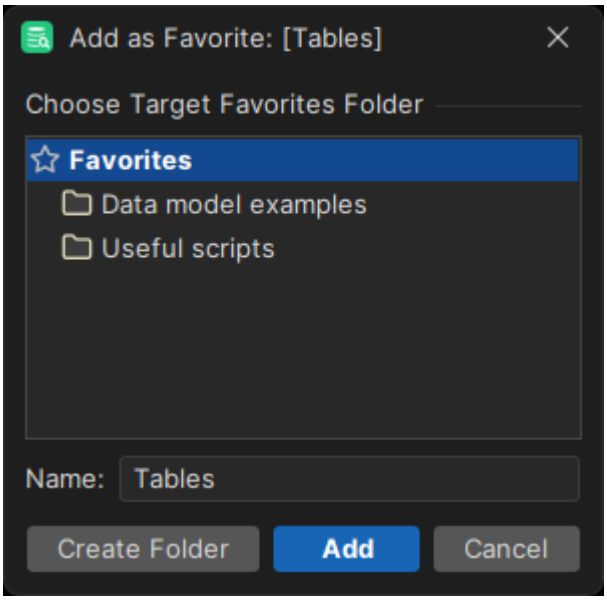

When you click on a database object in the Favorites tab, the corresponding object is opened in an Object View tab. If you're not connected to the database the object belongs to, a connection is automatically established. Clicking on a script in the Favorites tab opens it in an SQL Commander tab.

You can also easily find the corresponding object in the **Databases** tab or **Files** tab tree using the **Select Target Object** right-click menu item.




## Adding objects to Favorites

To add an object to Favorites, select it in the **Databases** or **Files** tab and choose the **Add to Favorites** option in the context menu. A dialog will be displayed where you will be able to choose whether to add the new item at the top level or in an existing or new Favorite folder.

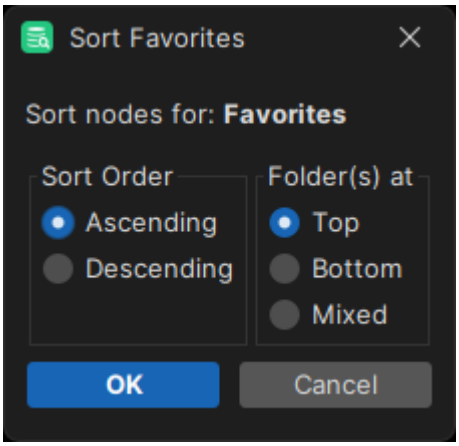



## Organizing Favorites

In the Favorites tab, you can reorder entries, organize them into folders, rename or delete them.

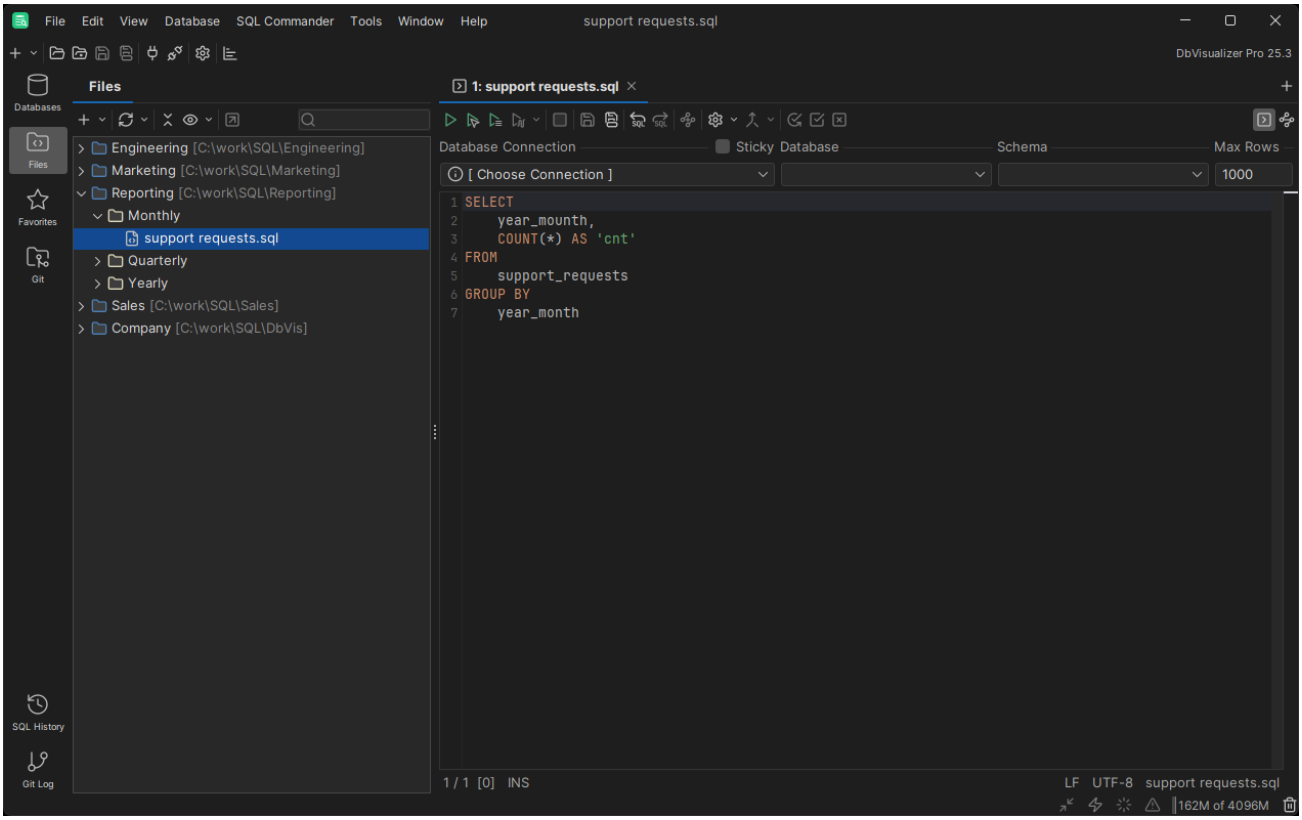
The right-click menu for an entry also contains entry-type dependent operations, such as executing a Bookmark and open a database object in a separate window.

To sort the favorites, select **Sort...** from the right-click menu for any favorite. The sorting criteria can be defined in the dialog that pops up.



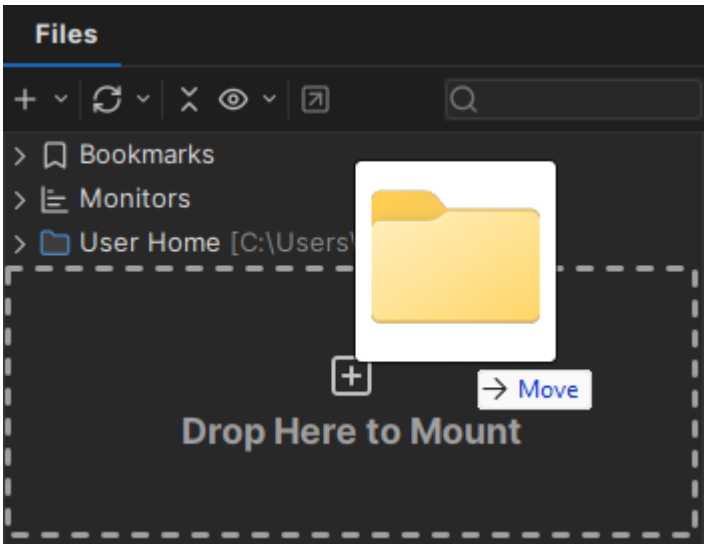
# Using the Files Tab

A script is a file with one or more SQL statements that you can edit and execute in an SQL Commander tab. Scripts may be kept anywhere in the file system and loaded into SQL Commander when needed, but using the Files tab makes it easier to organize and access your scripts.



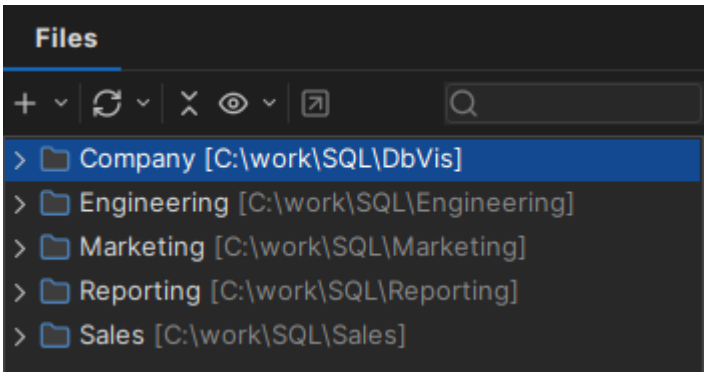

## Mounting folders

The Files tab gives you quick access to file system folders of your choice. Adding a relevant folder to the Files tab is called **mounting**. To mount a folder, drag and drop it from the file system to an empty area in the Files tab, or use the corresponding menu action to choose the folder to mount.




Mounted folders can be rearranged using drag and drop in the Files tab.

You can choose an arbitrary display name for your mounted folders. To do so, right-click on the folder and choose Rename. In addition to the display name, the folder's actual path is displayed within square brackets.

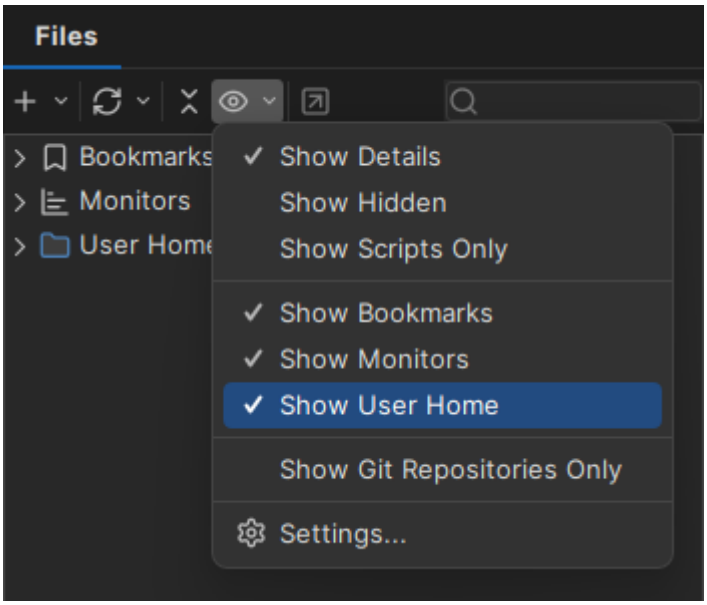



### Pre-mounted folders

By default, the Files tab contains three pre-mounted locations.

- **Bookmarks** is a pre-defined location for scripts stored within the DbVisualizer file structure. See Managing Frequently Used SQL for more details.
- **Monitors** contains monitored queries in DbVisualizer. See Monitoring Data Changes for more information.
- User's home directory is also pre-mounted for your convenience.

You can remove and add them back using the toolbar, and add your own folders by mounting them in addition to or instead of the default locations.




### Expanded state

For convenience, any currently expanded folders will be re-expanded upon the next start of DbVisualizer.

## Working with files

When expanded, the contents of the folder will be displayed including any subfolders and files. Double-click or use the context menu to **open** a file in SQL Commander.

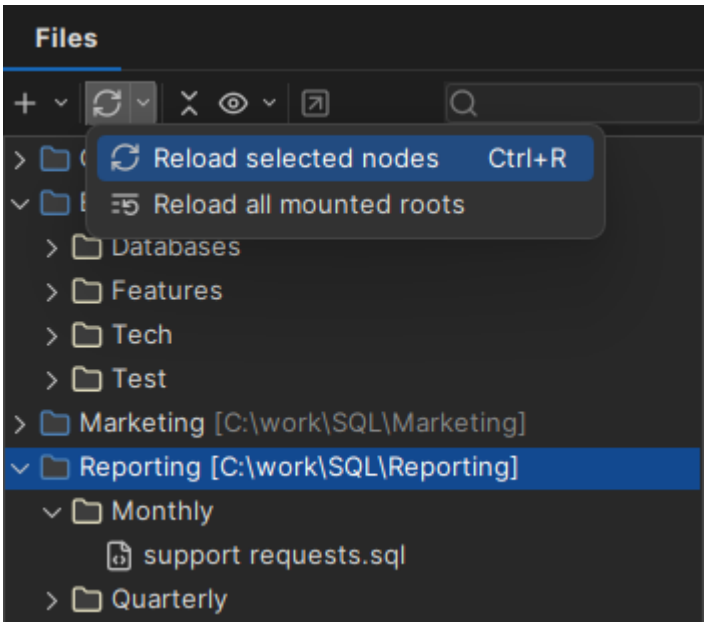
Use the toolbar or the context menu to **create** new files and folders or **rename** existing ones. To **move** files and folders, use drag and drop. To **copy**, use drag and drop with a modifier key or the corresponding keyboard shortcuts. There's also a handy context menu action for **duplicating** a file.

To **delete** a file or a folder, use the Delete action in the context menu. If supported on your operating system, this will move the corresponding file to the recycle bin.

## Reloading the tree

When possible, the nodes in the tree are kept in sync with the file system by monitoring changes in the corresponding physical locations. For example, if a new file has been created in an expanded folder outside DbVisualizer, it will appear in the tree automatically.

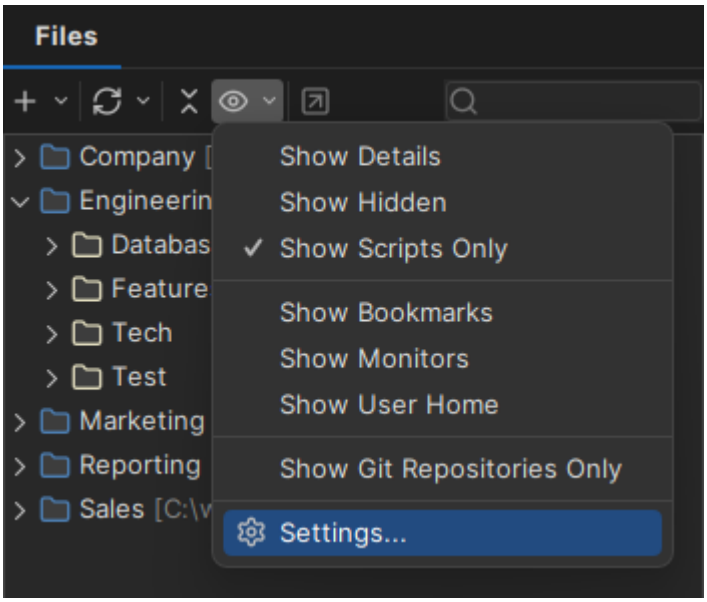
In situations when the tree gets out of sync, a manual reload of either selected nodes or the whole tree can be triggered using the toolbar.




## Choosing what to see

By default, the Files tab shows all file types but doesn't include hidden files. This can be changed using the corresponding toolbar options.

**Show Scripts Only** will hide non-script files from the tree. The **Settings...** action will open the corresponding section in **Tool Properties** where you can choose which file extensions will be considered to be scripts.




If you use Git, you may find the **Show Git Repositories Only** option useful. When active, only mounted folders that are repositories will be displayed. For more information about Git integration in DbVisualizer, please see Working with Git.

## Quick search

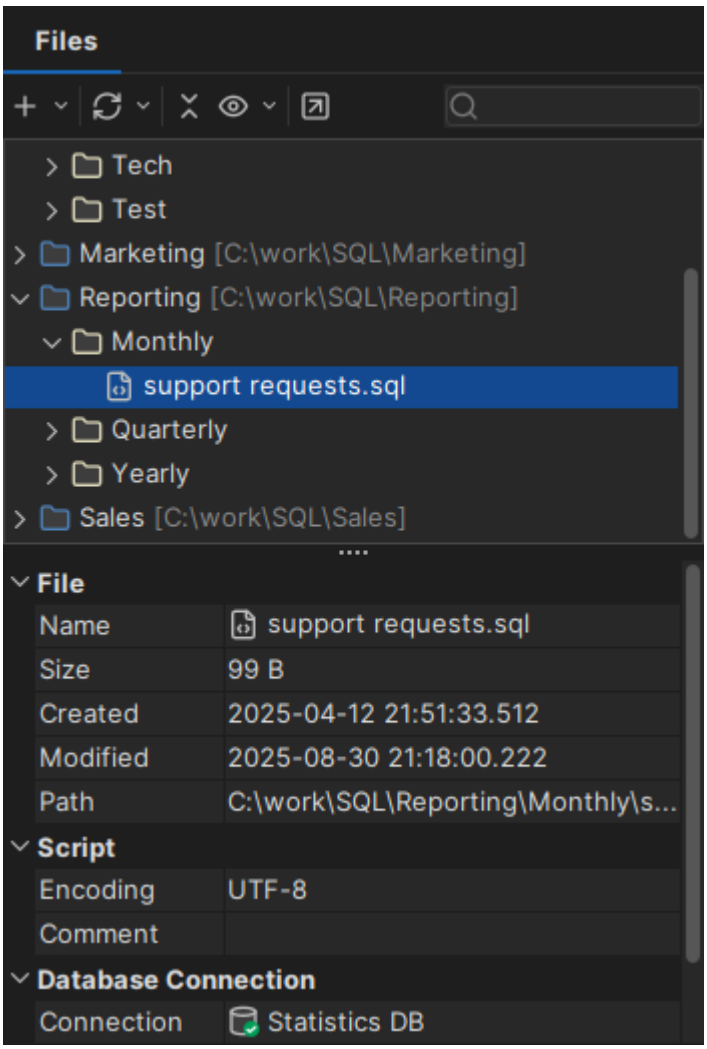
The quick search field can be used to find matching entries in the tree. As usual, pressing on the magnifying glass icon will bring up the menu with advanced search options.

Please note that quick search only affects those parts of the tree that have already been loaded. Bookmarks and Monitors are fully searchable from the start, but other folders are only searchable in those parts that have been expanded since the last reload.

## The details section

When activated in the toolbar, a details section will be displayed in the bottom part of the Files tab displaying information about the currently selected item.

Among other things, DbVisualizer keeps track of the latest used connection, catalog and schema for script files. This information is also accessible in the details section.



# Delimited Identifiers and Qualifiers

Delimited Identifiers must generally be used for database object with names that contain special characters, reserved words or mixed case. Qualifiers are needed when referring to an object in a different schema/catalog than the current one. DbVisualizer uses delimited, qualified identifiers in all SQL it automatically executes in response to user interaction, such as when loading the Data tab for a table, dropping a stored procedure or getting metadata for a DDL statement.

For the features where DbVisualizer generates SQL that you can then execute, you can control if you want to use delimited identifiers and/or qualified object names. In the **Properties** tab for a connection, or in the **Tool Properties** dialog for a database type under the **Database** tab, you find the **Delimited Identifiers** and **Qualifiers** categories.

In the Delimited Identifiers category you can see which delimiter characters that is used, e.q. double-quotes or square brackets, and set for which features to use them: **Scripting** (all SQL generating features), **Auto Completion/Query Builder**, **Export**, and **Actions.** The syntax for Delimited Identifiers is defined by the database and can only be changed for the **Generic** database type.

In the Qualifiers category you can specify for which features to use them, and also if column names should be qualified with the table name for the Scripting and the Auto Completion/Query Builder features. For database types that supports fully qualified named (both database and schema qualifiers), you can enable fully qualified names in the References/Navigator Graphs and for the Auto Completion/Query Builder features.

# Handling Transactions

## Handling Transactions

Database transactions are intended to make sure operations performed simultaneously do not cause data integrity problems.

By default, DbVisualizer commits all changes immediately but you can disable this to have full control over the transactions. You can also set an appropriate transaction isolation level for your connections.

- Setting Transaction Isolation
- Changing the Auto Commit Setting

# Setting Transaction Isolation

When you connect to a database that is concurrently modified by other users and processes, the Transaction Isolation Level specifies how changes made by others will affect you and how your changes will affect others.

To set the Transaction Isolation Level for a connection,

1. Double-click the connection node in the **Databases** tree to open an **Object View** tab for it,
2. Select the **Properties** tab,
3. Select the **Transaction** category,
4. Pick an appropriate **Transaction Isolation Level** from the drop-down list.

The levels that are supported depends on the database you are connecting to. Many databases support the levels as defined in the JDBC specification, which are described in the table below. But be aware that there are also many databases that have their own levels and/or terminology.

| Level | Transactions | Dirty Reads | Non-Repeatable Reads | Phantom Reads |
|---|---|---|---|---|
| `TRANSACTION_NONE` | Not supported | N/A | N/A | N/A |
| `TRANSACTION_READ_COMMITTED` | Supported | Prevented | Allowed | Allowed |
| `TRANSACTION_READ_UNCOMMITTED` | Supported | Allowed | Allowed | Allowed |
| `TRANSACTION_REPEATABLE_READ` | Supported | Prevented | Prevented | Allowed |
| `TRANSACTION_SERIALIZABLE` | Supported | Prevented | Prevented | Prevented |

A dirty read occurs when transaction A reading a value before transaction B has made permanent, i.e. before it has been committed.

A non-repeatable read occurs when transaction A retrieves a row, transaction B subsequently updates the row, and transaction A later retrieves the same row again. Transaction A retrieves the same row twice but sees different data.

A phantom read occurs when transaction A retrieves a set of rows satisfying a given condition, transaction B subsequently inserts or updates a row such that the row now meets the condition in transaction A, and transaction A later repeats the conditional retrieval. Transaction A now sees an additional row. This row is referred to as a phantom.

Please see the database documentation for the description of transaction isolation values supported by your database.

# Changing the Auto Commit Setting

Auto Commit means that each SQL statement successfully executed in an SQL Commander is committed automatically, permanently changing the database. This is the default for a connection, but you can change it at different levels.

## Changing Auto-Commit for a Database Type

1. Open **Tools → Tool Properties**,
2. Select the **Database** tab,
3. Expand the node for the database type, e.g. **Oracle,**
4. Select the **Transaction** category,
5. Uncheck the **Auto Commit** checkbox.

## Changing Auto-Commit for a Connection

1. Double-click the connection node in the **Databases** tab tree to open an **Object View** tab,
2. Select the **Properties** tab,
3. Select the **Transaction** category under the node for the database type, e.g. **Oracle**,
4. Uncheck the **Auto Commit** checkbox.

## Changing Auto-Commit for an SQL Commander tab

- Use the **SQL Commander → Transaction → Turn On/Off Auto Commit** toggle item, or,
- Use the corresponding toggle button to the right in the SQL Commander toolbar.

## Changing Auto-Commit for a Statement Block

You can use the `@set autocommit` command in a script to enable or disable auto commit for different blocks:

```
@set autocommit off;
insert into SAKILA.FILM (FILM_ID, LANGUAGE_ID, TITLE) values(1201, 1, 'Mission:
Impossible - DbVisualizer');
insert into SAKILA.FILM (FILM_ID, LANGUAGE_ID, TITLE) values(1202, 1, 'Harry
Potter and the DbVisualizer');
insert into SAKILA.FILM (FILM_ID, LANGUAGE_ID, TITLE) values(1203, 1,
'DbVisualizer: Episode 12');
@set autocommit on;
```

# Database Connection Options

## Database Connection Options

The database connection is a central concept in DbVisualizer.

Learn how to configure it to your needs, how to use special features like connecting through an SSH tunnel, using Single-Sign-On, organizing the connections, and much more.

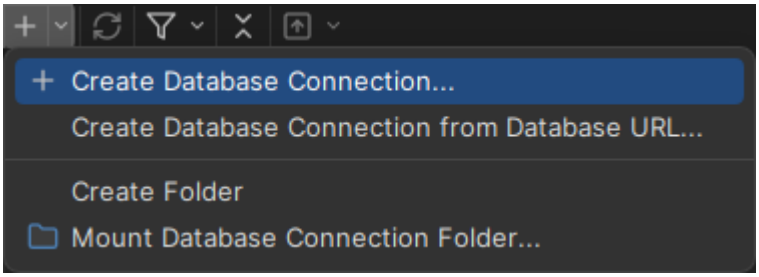

- Create a New Database Connection
- Configuring Connection Properties
- Copying an Existing Connection
- Edit Multiple Database Connections
- Removing a Connection
- Organizing Connections in Folders
- Rearranging Connections and Folders
- Setting Common Authentication Options
- Setting a Master Password
- Using Connection Keep-Alive
- Read-Only Connections
- Using Oracle TNS Names
- Changing Database Password
- Using Variables in Connection Fields
- Automatically Connecting at Startup
- Executing SQL at Connect and Disconnect
- Using a Single Shared Physical Connection
- Connection Permissions - Confirming connect
- Managing the Metadata Cache
- Sharing Database Connections

# Create a New Database Connection

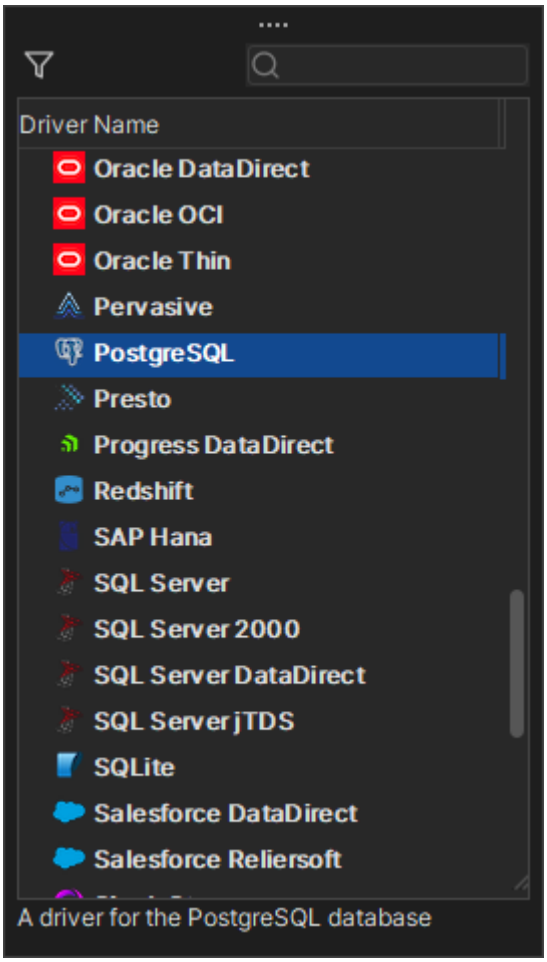
To connect to a database, you must first create and setup a Database Connection.

## Create a database connection

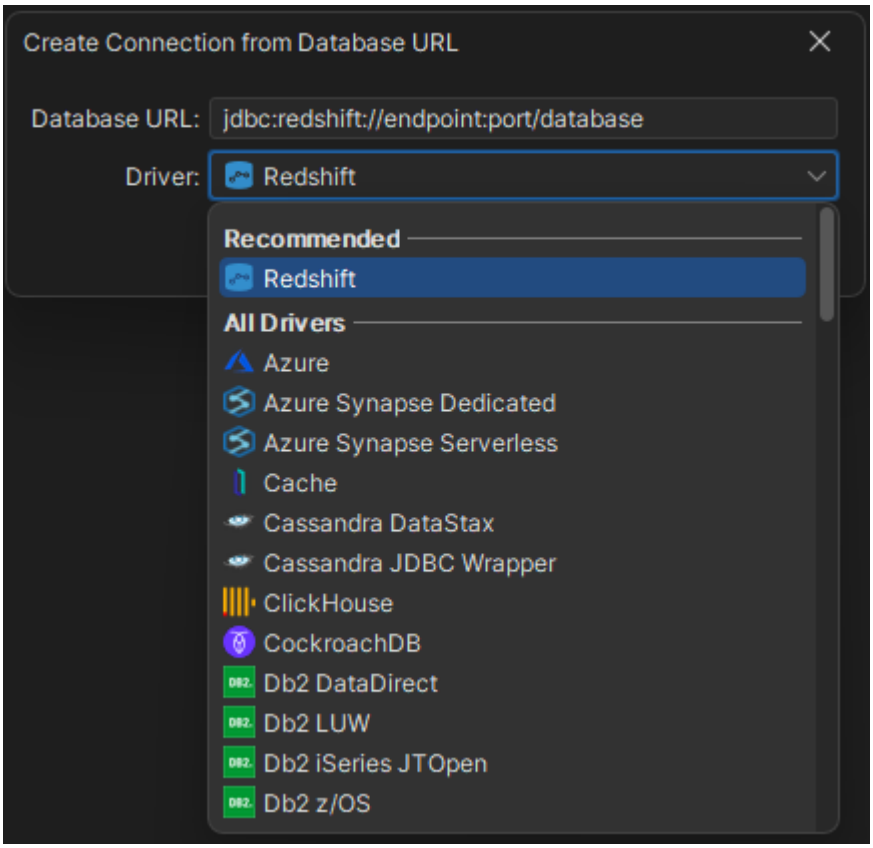
Database connections are created using the **Database → Create Database Connection** main menu or using the plus icon in the **Databases** tab toolbar.



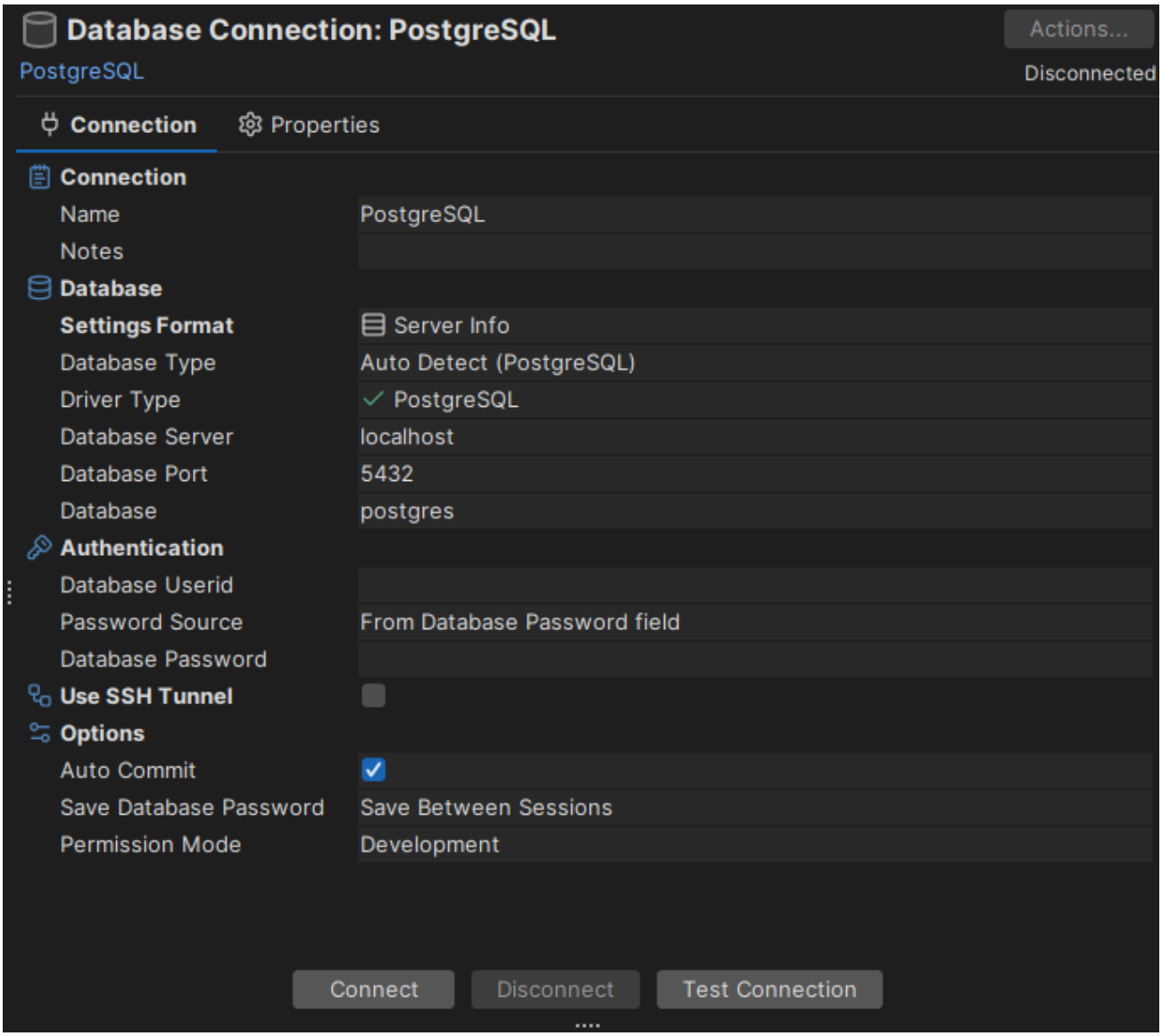

The first choice, **Create Database Connection**, opens a window in which the driver type should be selected by double-click:




The second choice, **Create Database Connection from Database URL** shows a dialog in which a JDBC URL is edited or pasted from clipboard. Based on the URL, the driver type is selected automatically:




Next step is to setup the connection details:




1. Enter a name for the connection in the **Name** field, and optionally enter a description of the connection in the **Notes** field,
2. Leave the **Database Type** as **Auto Detect,**
3. If the selected driver in **Driver Type** is marked with a green checkmark then it is ready to use. If it is not marked with a green checkmark, you may have to configure the driver in the **Driver Manager** (see Installing a JDBC Driver how to install a JDBC driver),
4. Enter information about the database server in the remaining fields,
5. Verify that a connection can be established by clicking the **Test Connection** button,
6. Click **Connect** to connect to the database server.

> See Fixing Connection Issues for some tips if you have problems connecting to the database.

Alternatively, you can set the **Settings Format** to **Database URL** (this is the only choice for some JDBC drivers). This replaces the fields for information about the database server with a single **Database URL** field, where you can enter the JDBC URL.

The information about the database server that needs to be entered depends on the which JDBC driver you use. For most drivers, you need to specify:

| Field | Description |
|---|---|
| **Database Server** | The IP address or DNS name for the server where the database runs. |
| **Database Port** | The TCP/IP port used by the database. |
| **Database Userid** | The database user account name. Enter `(null)` to not send an account name. |
| **Database Password** | The database user account password. Enter `(null)` to not send a password. |

For some database such as Oracle, you may use a TNS name instead of specifying the server and port. Other drivers may add more fields that are driver specific.

You may also optionally specify SSH tunneling information and Options, such as:

| Option | Description |
|---|---|
| **Auto Commit** | Check if you want to enable auto commit in the SQL Commander by default for the connection. |
| **Save Database Password** | Check if you want the password to be saved (encrypted) during the session, between sessions, or cleared when you disconnect. |
| **Permission Mode** | One of **Development**, **Test** or or **Production** to select which set of Permissions to use. |

See the Configuring Connection Properties page for related topics.

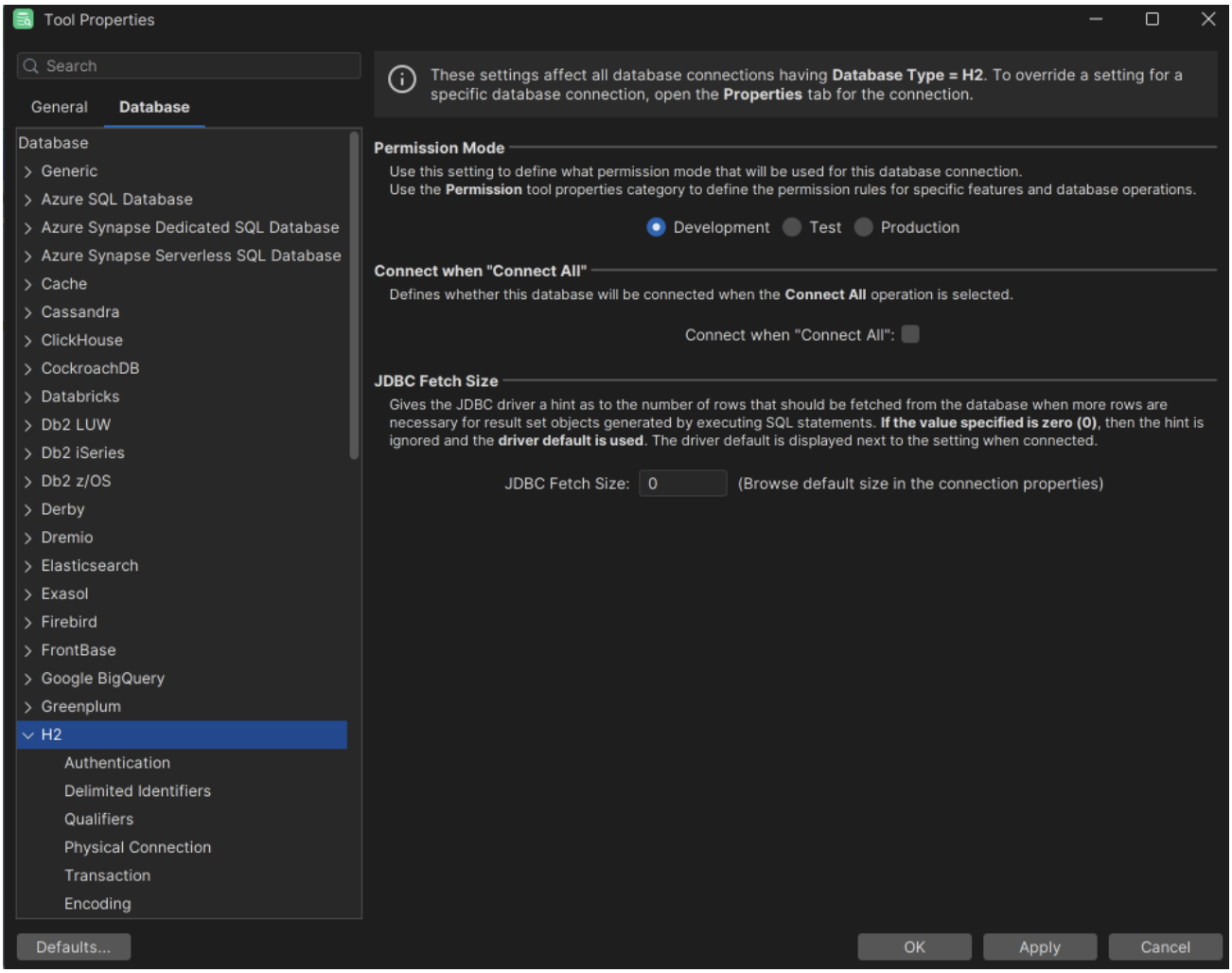
# Configuring Connection Properties

In addition to the basic connection information in the **Connection** tab, there is also a collection of connection properties. Which properties are available depends on the **Database Type** selected for the database connection in the Connection tab. Some database types have more properties than others. Which edition of DbVisualizer you use also affects which connection properties are available.

Properties for a connection can be defined at two different levels: **Tool Properties (Database tab)** and **Connection Properties**.

## Tool Properties

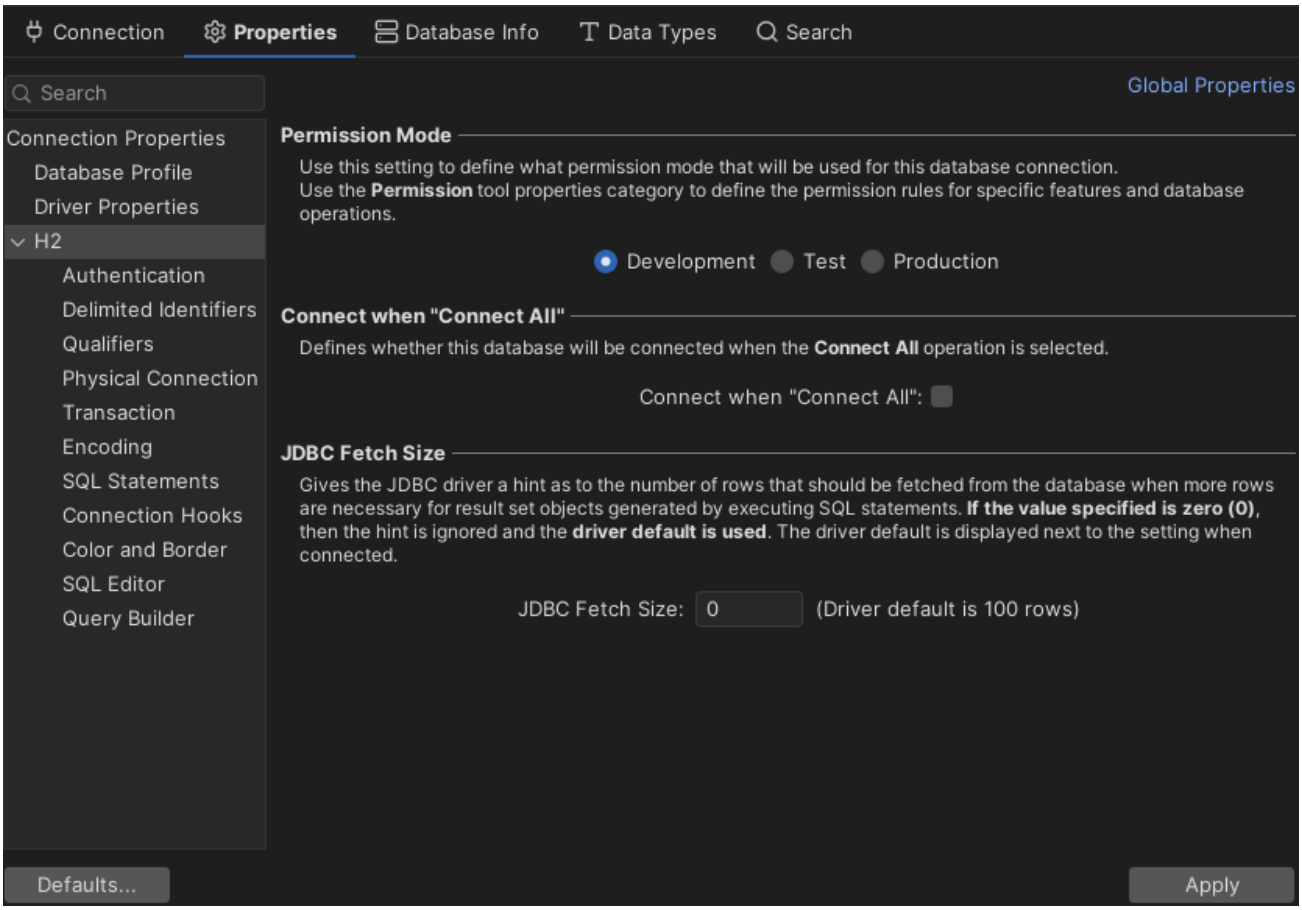
The Database tab in **Tool Properties** defines settings for all connections of the specific database type. All supported database types (Oracle, Informix, SQL Server, Db2, MySQL, etc.) are listed, and for each database type, there are a number of properties that are applied to any database connection of that type. This means, for instance, that a database connection defined as being a PostgreSQL database type will use the PostgreSQL properties defined in **Tool Properties**.




## Connection Properties

The global properties can be overridden for individual connections. The advantage with this inheritance model is that property changes that apply to all connections can be made in one place, instead of having to apply a common setting for every database connection of a specific database type.

The Connection Properties are available in the **Properties** sub tab in the the database connection's **Object View** tab.



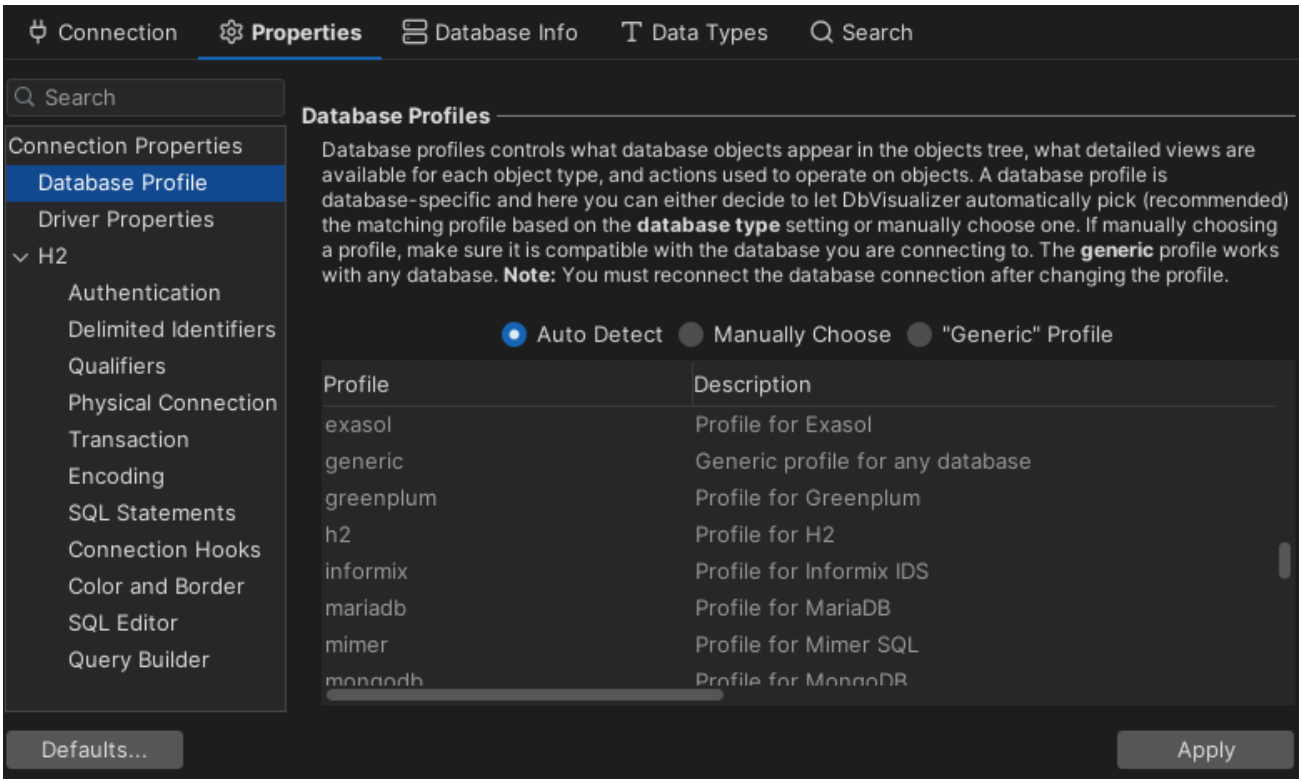

The **Properties** tab is organized basically the same way as the **Tool Properties** window. The main difference is that the list contains only the categories that are applicable to this database connection. The **Database Profile** and **Driver Properties** categories are available only in the **Properties** tab and not in **Tool Properties**. The page explains the **Database Profile** and **Driver Properties** categories, while the other categories are described in pages that describe feature the property applies to.

Additional categories may appear in the connection properties depending on the type of database. An example is the category for Explain Plan for the databases where this feature is supported.

At the top right corner, you will also find a link to set the **Global Properties** for all connections of this database type.

## Database Profile

The Database Profile category is used to select whether a profile should be automatically detected and loaded by DbVisualizer, or if a specific one should be used for the database connection. The default strategy is to **Auto Detect** a database profile.

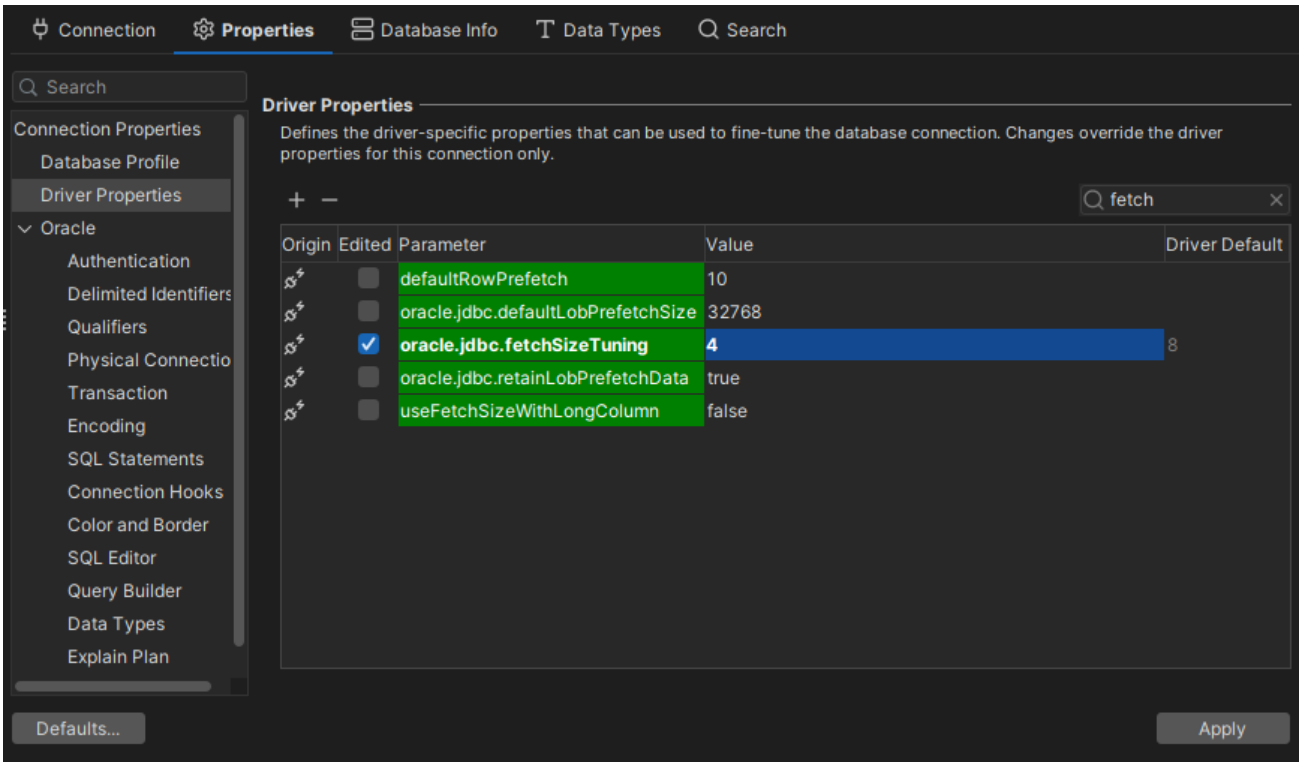



> The way DbVisualizer auto detects a profile is based on the setting of **Database Type** in the **Connection** tab. If the **Database Type** is also set to **Auto Detect**, DbVisualizer first detects the database type based on the JDBC information, and then detects the profile based on the database type.

There is rarely a reason to use another setting than **Auto Detect**, but if you manually choose a database profile, this choice will be saved between invocations of DbVisualizer.

## Driver Properties

The **Driver Properties** category is used to fine tune a JDBC driver before the database connection is established. You can use the filter to quickly find a specific property.




The list of parameters, their default values and parameter descriptions are determined by the JDBC driver used for the connection. Not all drivers support additional driver properties. To change a value, just modify it in the list. The second column in the list indicates whether the property has been modified or not, and so, whether DbVisualizer will pass that parameter and value onto the driver at connect time.

New parameters can be added using the plus button to the right of the list. The minus button removes user added properties or resets the property to the default value.

## Invoke Java methods in the JDBC driver

In addition to driver properties, it is also possible to invoke low-level Java methods in the JDBC driver classes, java.sql.Connection and java.sql.Statement. These are edited in the driver properties list.

You may run single argument methods taking one of **String.class**, **Integer.class**, or **Boolean.class** Java types as argument. The method name should be specified as **Parameter** which must be fully qualified with the all-lowercase class name. The argument is specified in the **Value** field in the driver properties list.

Here are a few examples:

| Parameter | Value |
| --- | --- |
| java.sql.connection.setReadOnly | `true` |
| java.sql.statement.setFetchSize | `1000` |

**java.sql.connection** methods are invoked just after a physical connection with the database has been established.
**java.sql.statement** methods are invoked just after a statement has been created and is ready for execution.

# Copying an Existing Connection

To copy an existing connection and use as the basis for a new:

1. Select the original connection node in the **Databases** tab tree,
2. Use the **Database → Duplicate Database Connection** to create a copy,

The Object View tab for the copied connection is opened where you can make adjustments.

# Edit Multiple Database Connections

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

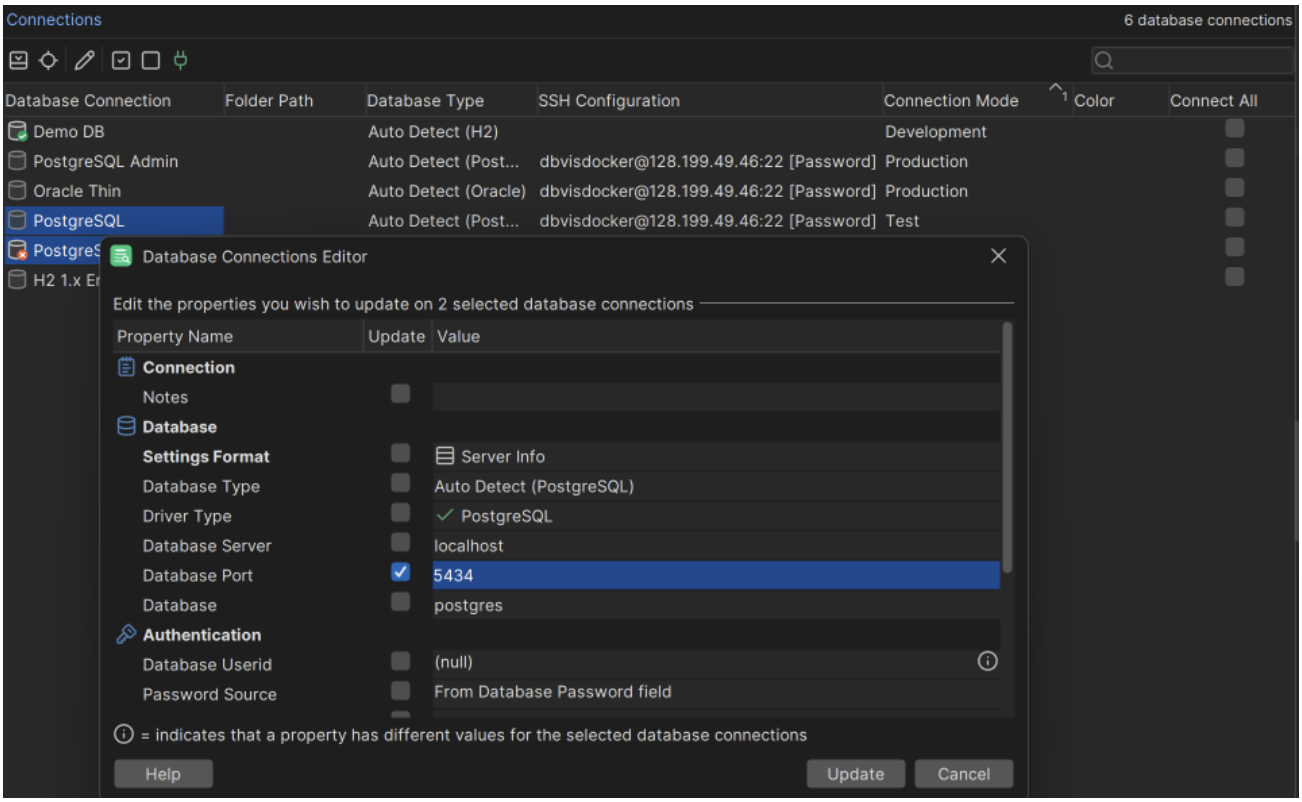
There are situation when you may need to change one or several connection settings for multiple database connections at the same time. Instead of doing this one connection at a time, the connections editor come in handy.

There are multiple paths to edit database connections:

1. Selecting multiple database connections in the **Databases** tab, right-click and choose **Edit Database Connection(s)...**
2. Selecting multiple database connections in the connections list shown when opening the object view for the **Connections** node. Right-click and choose **Edit Database Connection(s)...**
3. Selecting a **Folder** object in the **Databases** tab will show all included database connections in the right-pane. Select the ones you want to edit, right-click and choose **Edit Database Connection(s)...**

Note that all database connections that are to be edited must be disconnected.

The screenshot below shows how the **Connections** tab is listing all the connections. The list has been sorted on **Connection Mode** in order to make it easy to select and edit all **Test** databases. As an alternative, you could have used the filter to show only these connections.




The connections editor will show only those fields that are commonly available for the database connections being edited.

The window has the following columns:

- **Update**: Indicates if the value is edited. You may manually uncheck this to indicate the setting should not be updated
- **Value**: Enter the new value here. Once the field is being edited its background will change and the Update box is checked.

## Changing the database driver

The **Driver Type** setting is the main setting that controls what properties are available for a database connection. Once Driver Type has been edited its **Update** checkbox cannot be unchecked.

# Removing a Connection

To remove a connection,

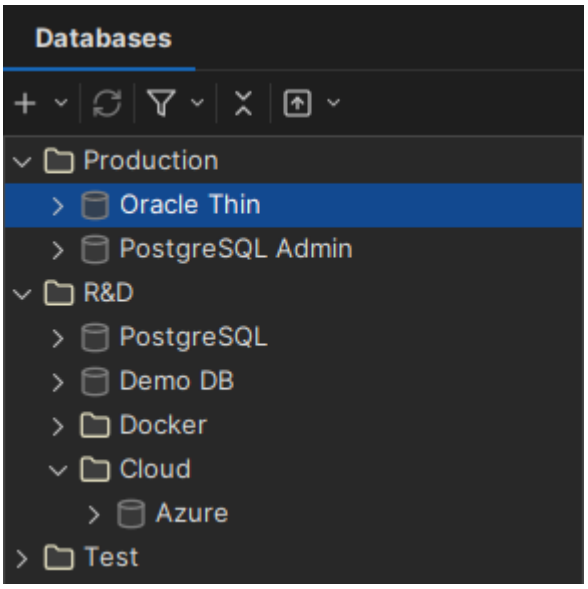
1. Select the connection node in the Database tab tree,
2. Use the **Database → Remove Database Connection(s)** menu choice to remove the connection.

# Organizing Connections in Folders

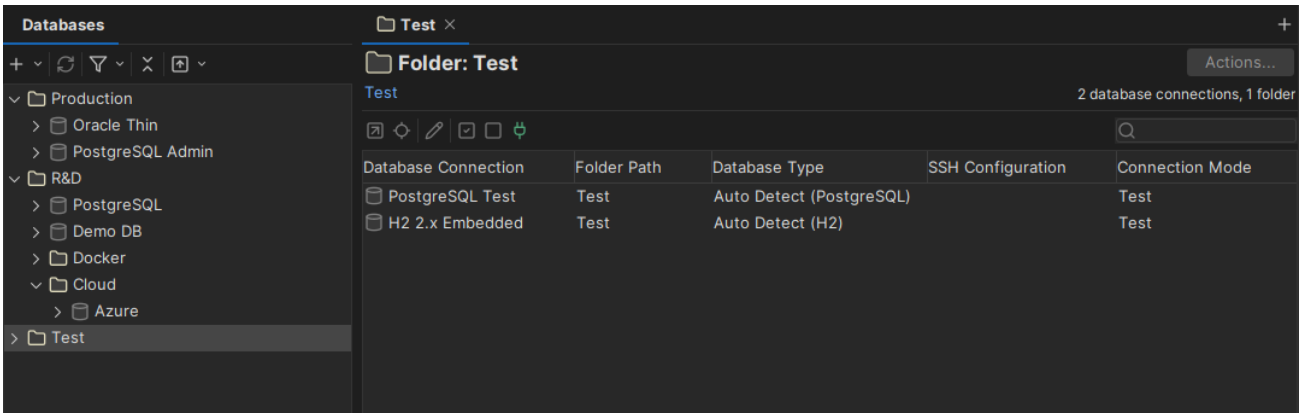
> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

If you work with many database connections, you can use folder objects to organize and group them in the tree. Folder objects can have child folder objects in an unlimited hierarchy.

Use the **Database → Create Folder**, **Database → Rename Folder**, and **Database → Remove Folder** menu choices to manage folder objects. You can also drag and drop folders and other objects to move them to a new location in the **Databases/Connections** tree.




By double clicking the Folder node a tab is opened listing all the connections recursively in the folder. This is handy when you need to edit properties for multiple database connection. See the chapter Edit Multiple Database Connections for details about this.


Databases
Test
Production
Oracle Thin
PostgreSQL Admin
R&D
PostgreSQL
Demo DB
Docker
Cloud
Azure
Test
Folder: Test
Actions...
Test
2 database connections, 1 folder
Database Connection
Folder Path
Database Type
SSH Configuration
Connection Mode
PostgreSQL Test
Test
Auto Detect (PostgreSQL)
Test
H2 2.x Embedded
Test
Auto Detect (H2)
Test

# Rearranging Connections and Folders

You can sort the nodes under the **Connections** node or a folder node in the Databases tab:

1. Select the Connections node or a folder node,
2. Open the **Sort...** dialog from the right-click menu,
3. Select the sort order and where to place nestled folder,
4. Click **OK**.

To move an individual connection node or folder node:

1. Select one or more nodes,
2. Drag the node(s) to the new location,
3. Drop the node.

# Setting Common Authentication Options

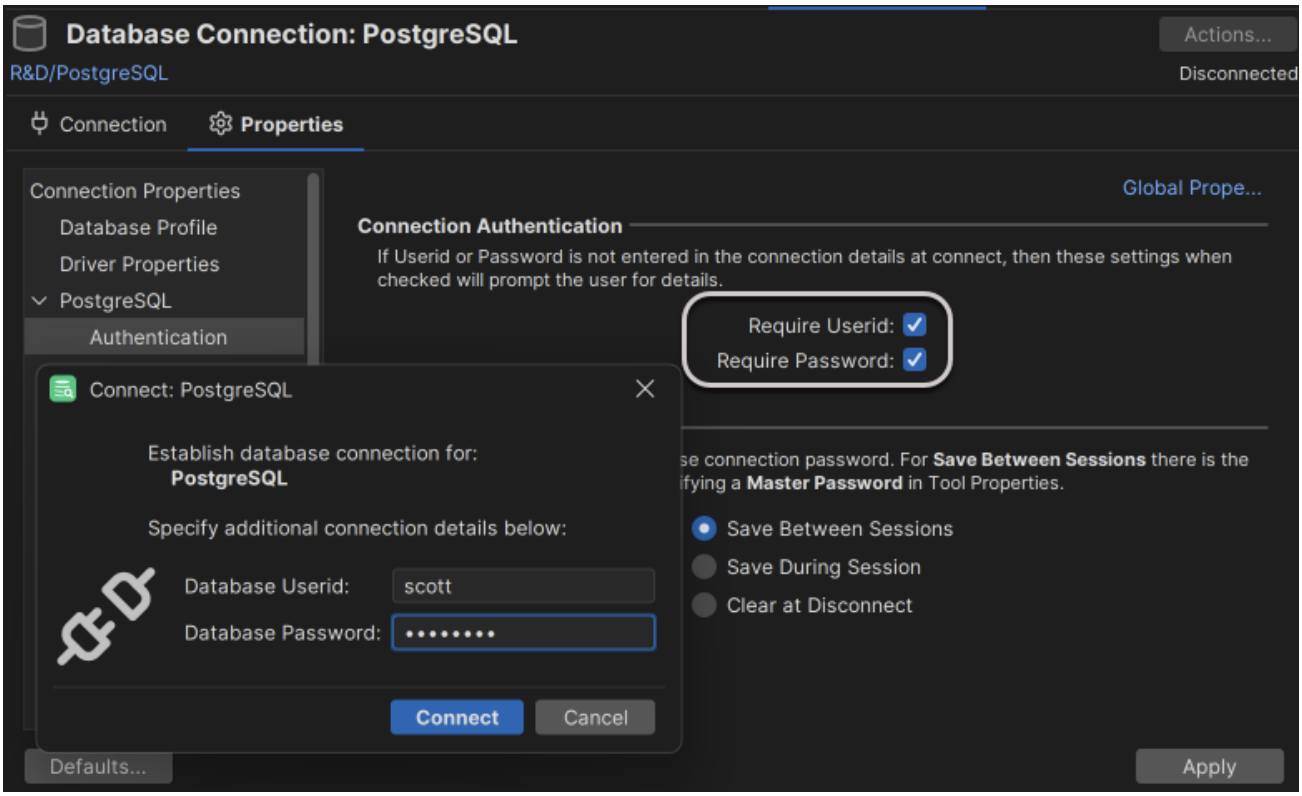
DbVisualizer lets you handle authentication in a manner that suits your balance between security and convenience.

## Authentication settings in Connection Properties

Userid and password information is generally information that should be handled with great care. By default, DbVisualizer saves both the Database Userid and Database Password (encrypted) for each database connection. The default for SSH is to save the SSH Userid but not the SSH Password (or private key passphrase). You can change this behavior to fit your preferences.

You specify how to handle the Database Userid and Password in the **Authentication** category of the **Properties** tab.

The **Require Userid** and **Require Password** properties can be enabled to tell DbVisualizer to automatically prompt for userid and/or password when a connection is to be established if they are not specified for the connection. The following dialog is displayed if requiring both userid and password.



# Setting a Master Password

When you use **Save Between Sessions** for database passwords and SSH passwords or passphrases, they are by default encrypted using an auto-generated key. The same is true for the Proxy password if you have specified one. For better security, you can specify a Master Password that is then used instead to encrypt all passwords and passphrases. This way, only you know the information needed to decrypt the data. The algorithms used for encryption with a Master Password are also more advanced, minimizing the risk that the data can be decrypted by brute force.

Using a Master Password does, however, mean that if you forget it, there is no way to retrieve it and therefore no way to decrypt the saved passwords.

> ⚠ If you forget the Master Password, it cannot be recovered. The only way forward is to reset the Master Password, which also clears all passwords encrypted with it.

## Specifying a Master Password

To use a Master Password for encoding of passwords saved between sessions:

1. Open **Tools → Tool Properties** and select the **General / Master Password** category,
2. Enter a password matching the described rules in both the **New Password** and **Confirm New Password** fields,
3. Click **Apply** and then confirm that you want to do this after reading the warning about what it implies.

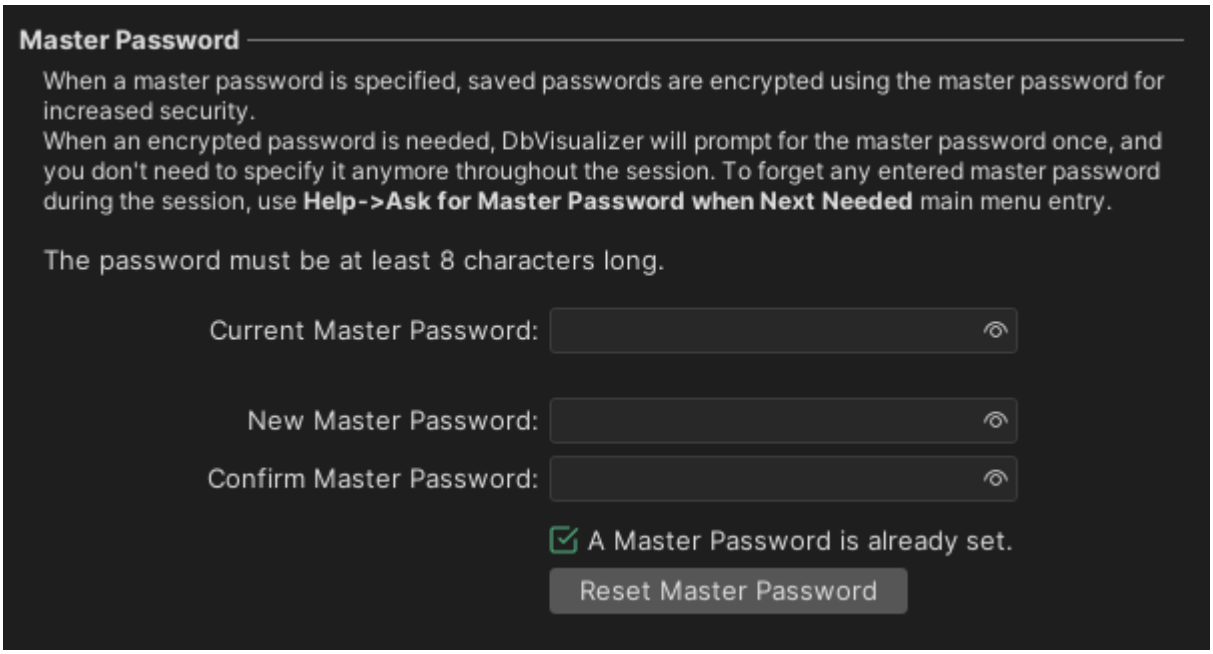



The passwords for all connections with **Save Between Sessions** chosen for the password are now encrypted with the Master Password. The same goes for the SSH passwords/passphrases if you have selected to have them saved between sessions, as well as the proxy password, if any.

## Changing a Master Password

If you want to change the Master Password:

1. Open **Tools → Tool Properties** and select the **General / Master Password** category,
2. Enter the current password in the **Current Password** field and the new password in both the **New Password** and **Confirm New Password** fields,
3. Click Apply.




The saved passwords are then decrypted with the current Master Password and re-encrypted with the new.

## Resetting the Master Password

If you have forgotten the Master Password, or simply no longer want to use one, you can reset it:

1. Open **Tools → Tool Properties** and select the **General / Master Password** category,
2. Click the **Reset Master Password** button and confirm that you want to do this.

> ⚠ Note that all passwords encoded with the Master Password are then immediately cleared and there is no way to recover them.

## Prompted for Master Password

When you have a Master Password specified, you will be prompted for it whenever DbVisualizer needs you to authenticate. This happens for example when connecting to a database or modifying credentials. From then on, you can make other connections with saved passwords without being prompted until you restart DbVisualizer.

## Manually Requesting the Master Password for New Connections

You can manually require being prompted for the Master Password again after entering it once within a DbVisualizer session:

- Select **Help → Ask for Master Password when Next Needed**

## Showing the Encrypted Password in Cleartext

When you have specified a Master Password, you can view the saved database password or SSH password/passphrase in cleartext.

1. Right-click on the password field label and select **Show Password**,
2. Enter the Master Password when prompted.

## Declaring a Master Password Rule

A Master Password must have at least eight characters of any kind by default, but you can declare your own rule using a regular expression in an installation configuration file:

1. Open the DBVIS-HOME/resources/dbvis-custom.prefs file,
2. Enter a regular expression as the value of the `dbvis.-MasterPasswordRule` property,
3. Enter a description of the rule for showing the user in Tool Properties as the value of the `dbvis.-MasterPasswordRuleDescr` property.

The regular expression for the default rule is `.{8,}`. It is easy to change the number in this expression to any number you want. There are regular expressions that can describe pretty much any rule you can come up with. For instance, this rule requires at least nine characters, with at least one symbol, one digit, one uppercase character, and one lowercase character:

```
(?=.{9,})(?=.*?[^\\w\\s])(?=.*?[0-9])(?=.*?[A-Z]).*?[a-z].*
```

If you cannot adopt these examples to your own policy, you can search the Internet for other examples of "regular expressions for password validations".

# Using Connection Keep-Alive

Databases can be configured to terminate sessions that have been idle for some time, and networks often does the same with TCP/IP connection. The **Connection Keep-Alive** feature helps preventing connections to be closed due to time-outs of this kind by periodically executing a simple SELECT statement.

> ⚠ Network connections may be terminated for other reasons than a time-out in the database or at the network layer, e.g due to a restart of the database or a network element. The Connection Keep-Alive feature does not help in those cases. Also note that connections that are busy, e.g. actively used to run a script, are not "pinged". If a SELECT statement or stored procedure takes a very long time to complete, it is therefore possible that a time-out happens at the network level. In this case, the network configuration must be tuned to handle long running statements without timing out.

To enable Keep-Alive for a connection:

1. Open the **Object View** tab for the connection,
2. Open its **Properties** tab,
3. Select the **Physical Connection** category,
4. Modify or enter a simple SQL statement in the **Validation and Keep-Alive SQL** field, if needed (see note below),
5. Enable **Connection Keep-Alive** and optionally change the **Connection Idle Time**,
6. Click the **Apply** button.

> ✎ The SELECT statement used for Connection Keep-Alive can also be specified in the properties pane. For supported databases, it is set to a SELECT statement that has been verified to work for the database type but for connections that use the Generic profile, you must specify a valid SELECT statement in order for this feature to work. For many databases, `SELECT 1` or `SELECT 1 FROM aSmallTable WHERE 1 = 0` should work.

# Security

## Security

When DbVisualizer communicates with a database server, all communication (except passwords) is done in plain text; by intercepting the communication, someone can access the transmitted data and even modify it while in transit. To protect your communication you need to encrypt the communication between DbVisualizer and the server. Techniques like Kerberos, LDAP, Radius, security mechanisms, external authentication options, SSH server forwarding, and a lot of other database specific support are not directly features covered in DbVisualizer and needs to be managed specifically for each database. Read more in the security section in the support portal for additional information.

> You find additional information on security related to specific databases in our support portal. Do also check the database/driver specific information.

# Using SSH

This section documents how to define, configure, and test SSH configurations. Starting with DbVisualizer 14.0, managing SSH configurations is done in **Tool Properties**. A single SSH configuration can be shared with multiple database connections making the administration a lot easier than (as previously) having multiple replicas of the same SSH configuration associated with different database connections.

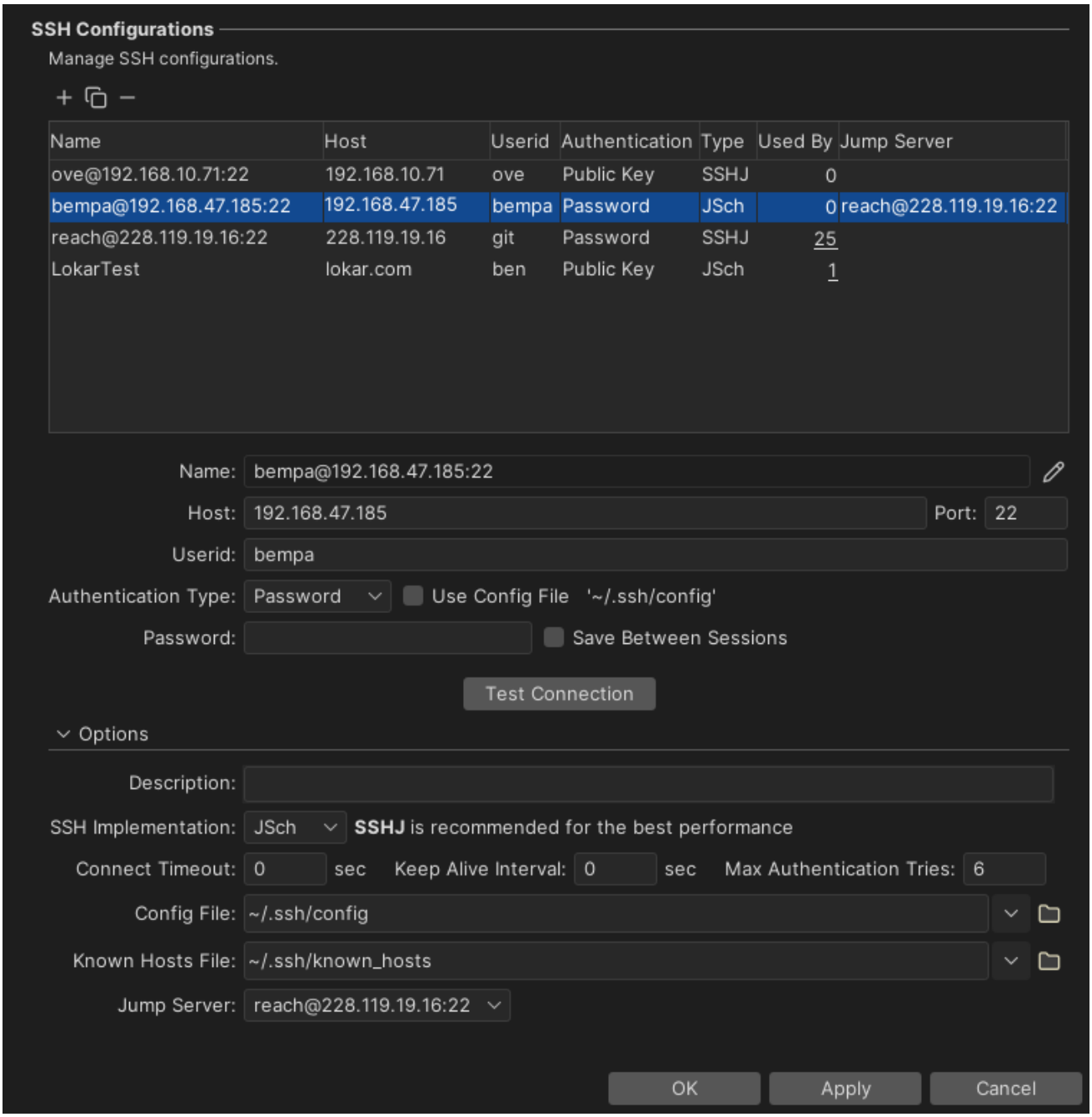
When SSH tunneling is enabled, a tunnel is established when you connect to the database and the connection is then made through the secure tunnel.

> You find additional information on security related to specific databases in our support portal. Do also check the database/driver specific information.

## Configuring SSH

To manage a SSH configuration, any of the following options many be used:

- Open **Tools → Tool Properties** and select **General / SSH Configurations**
- In the database **Connection** tab, select **Use SSH Tunnel**, click in the **SSH Configuration** drop-down and select **Create/Edit SSH Configurations**
- For a defined SSH configuration in the Connection tab, right-click it, and select **Open SSH Configurator**




The upper grid shows the SSH configurations where the **Name** column is either auto generated or manually set. **Type** is one of Password, Public Key or SSH Agent, and the last **Used By** column shows when clicked what database connections are using an SSH configuration.

Selecting an SSH configuration in the grid shows its details and the additional **Options** section that can be made visible or hidden. The following sections explains the different **Authentication Types**.

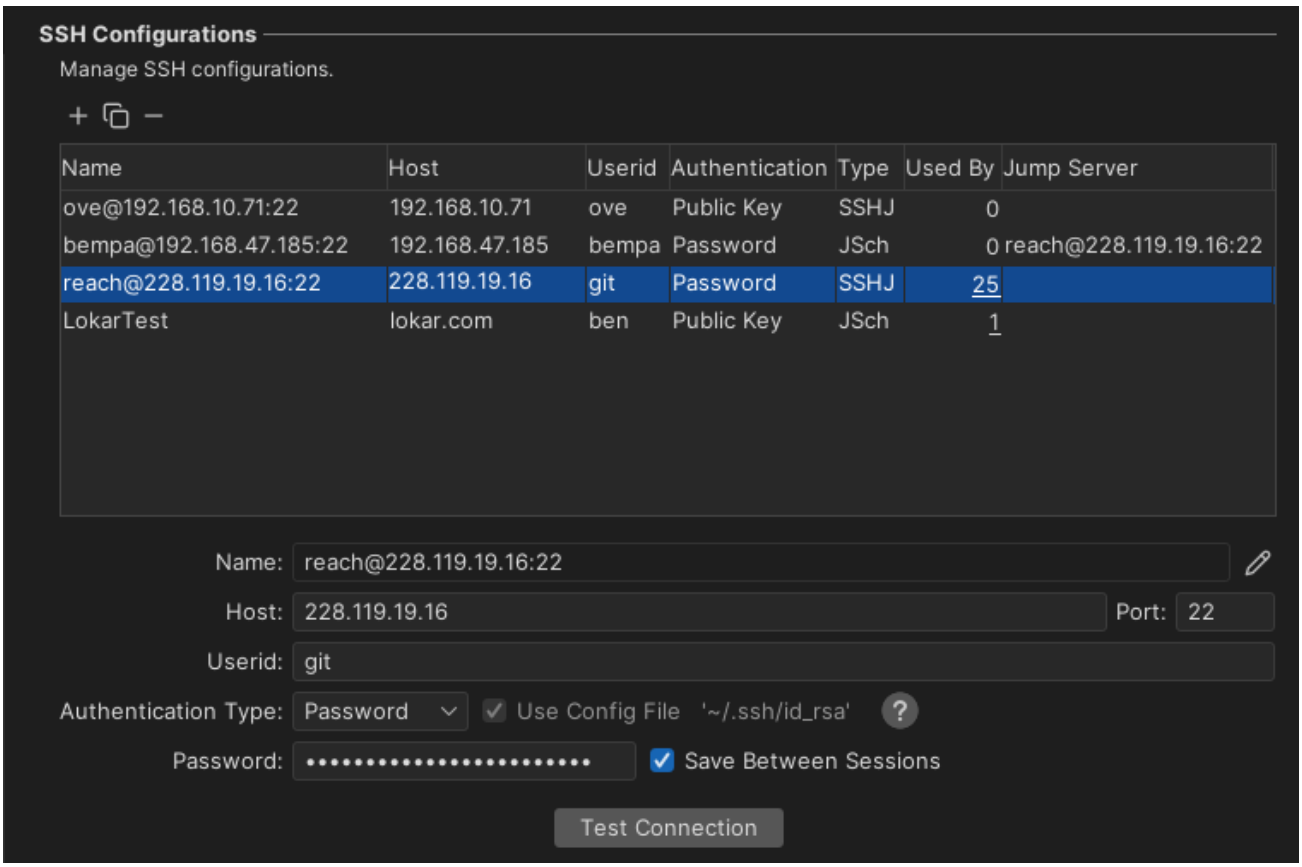
AUTHENTICATION TYPES

There are three different authentication types when establishing an SSH tunnel.

Password

Uses a userid/password combination to authenticate with the SSH server.

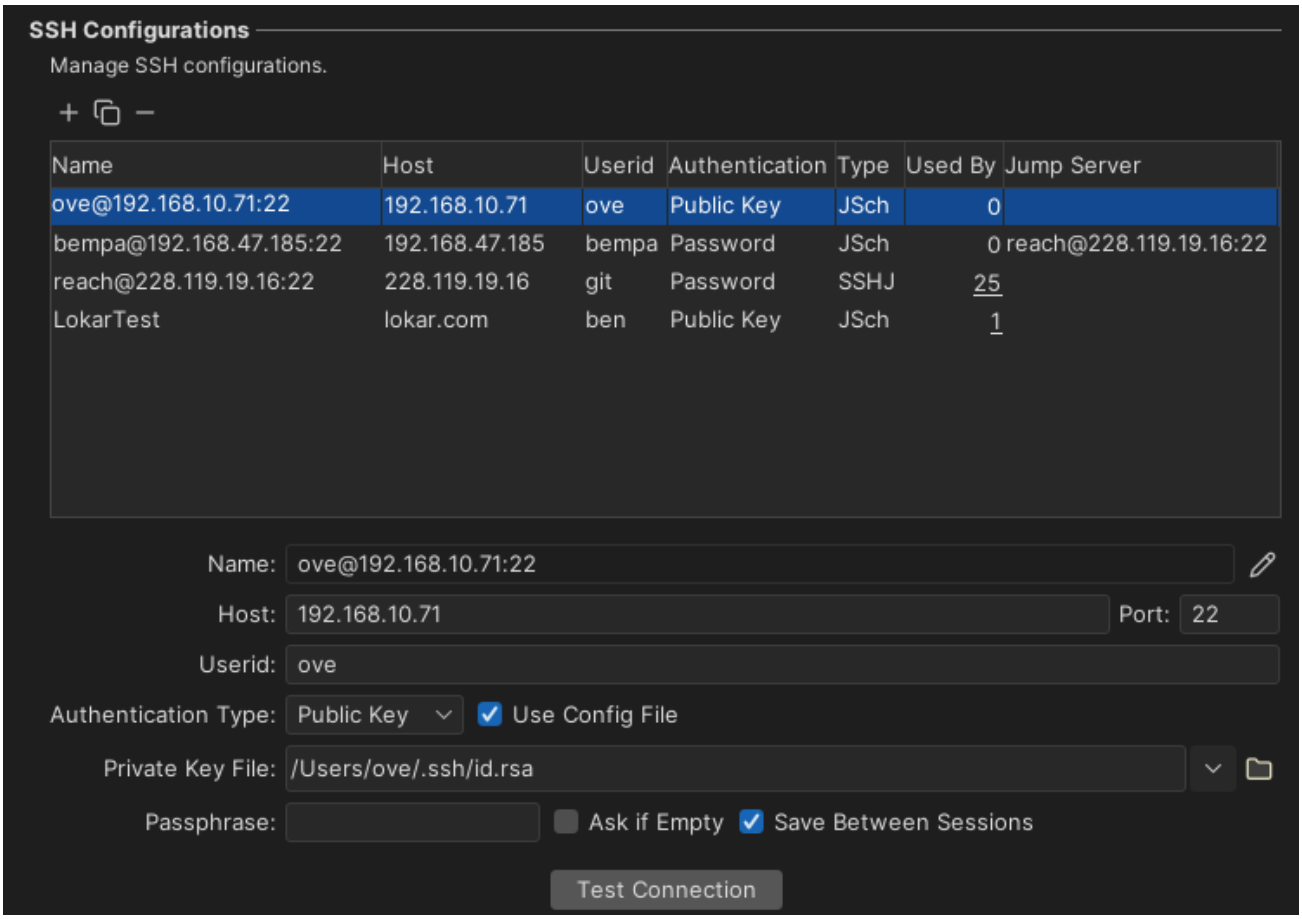
The **Host** is the name or IP address for the SSH host, or if using a SSH config file, the name of the host entry in the file. The target SSH host server is typically the same as the Database Server. Enter the port for SSH connections in the **Port** field. The default value is 22. If **Password** is not specified, you will be prompted for it when connecting to the server. To keep the password between sessions, select **Save Between Sessions**. You should also look into enabling a Master Password for strong security.




Public Key

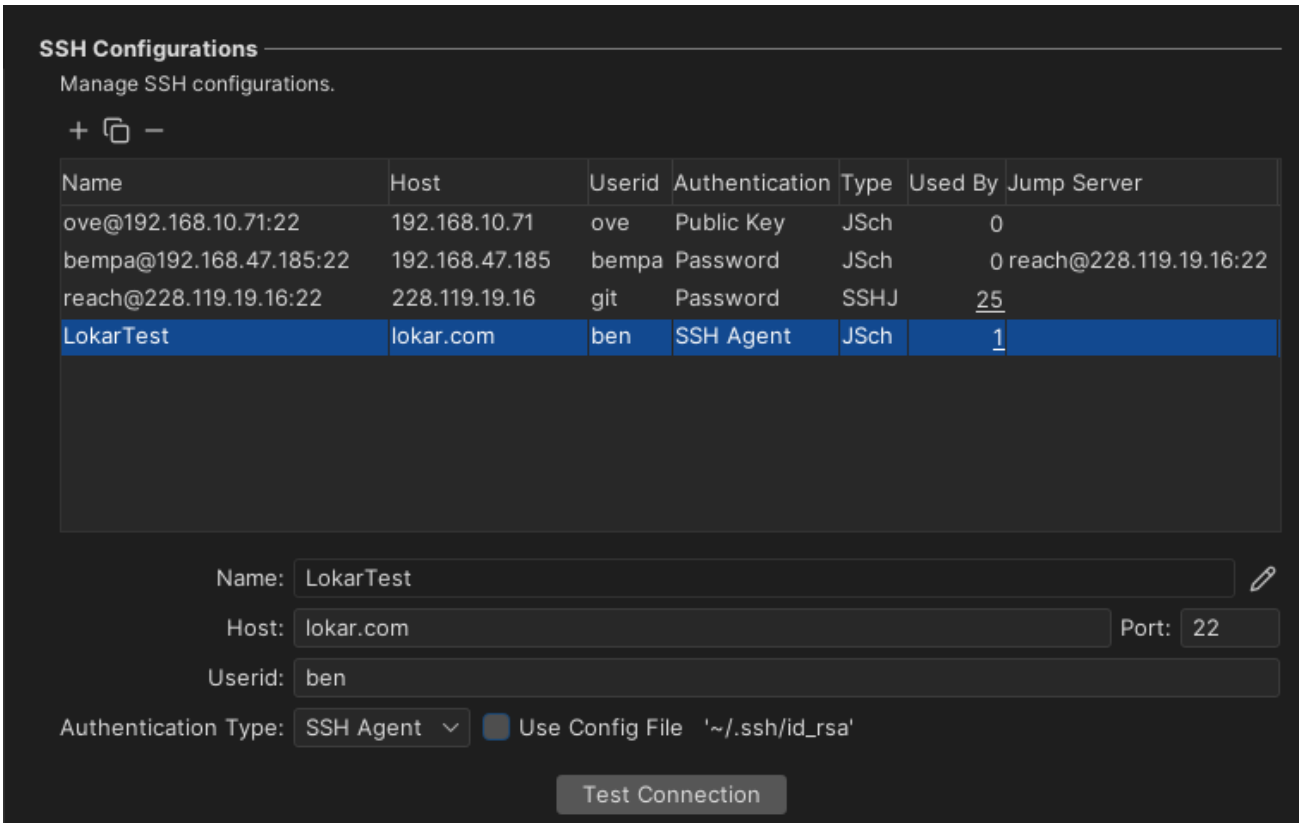
Uses a public/private key pair (using either RSA or DSA algorithms) to authenticate with the server. Enter the **Private Key File** for the SSH configuration and upload the public key, typically the file with extension **.pub** in the <User home>/.ssh/authorized_keys file in the server account.

An optional passphrase may be set for the key. If passphrase is not set you can set that it should be prompted at connect with the **Ask if Empty** setting. To keep the passphrase between sessions, select **Save Between Sessions**. You should also look into enabling a Master Password for strong security.



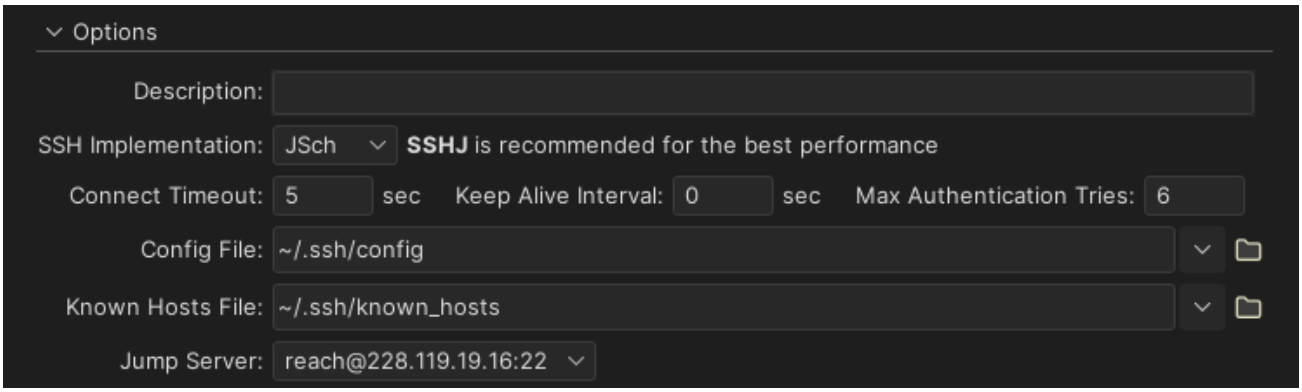

SSH Agent

On macOS, Linux and other Unix based operating systems the **SSH Agent** authentication type is available. When this option is selected ssh-agent will be used for authentication, removing the need to provide a private key or password. In order to use ssh-agent for authentication, the identity used for authentication needs to be added to ssh-agent by running `ssh-add` in a terminal. Read more about setting up SSH Agent.




SSH OPTIONS

These are the generic SSH options independent of authentication type:




- **Description** use this to optionally describe the SSH configuration
- **SSH Implementation** chose from two different client-side toolkits; SSHJ or JSch. The most obvious difference is that SSHJ is faster and due to this it is also the recommended SSH Implementation. There might be differences in supported SSH algorithms and if any problems connecting with one of them, try the other one.
- **Connect Timeout** number of seconds to prevent endlessly waiting for an SSH host to respond
- **Keep Alive Interval** to minimize the risk that the tunnel is disconnected due to inactivity, specify number of seconds how often a keep-alive request should be made
- **Max Authentication Tries** the number of allowed connection attempts
- **SSH Config File** containing optional SSH configuration. Using config file is not supported with authentication type = SSH Agent
- **SSH Known Hosts File** identifies the known hosts file. The location of this file is typically <User home>/.ssh/config.
- **Jump Server** read more in the Jump Server section

SSH Config File

OpenSSH supports defining some parts of the SSH configuration in an **SSH config file**. This file is typically named <User home>/.ssh/config. **To use a config file identify the file in the** SSH Config File** field for the SSH configuration. Read more about OpenSSH/Client Configuration Files.

> Please note that while most of the OpenSSH config file options are supported, some aren't such as **ProxyCommand**. For a list of supported keywords please see the documentation of OpenSSHConfig and Configuration Options.

Here is a sample of the config file:

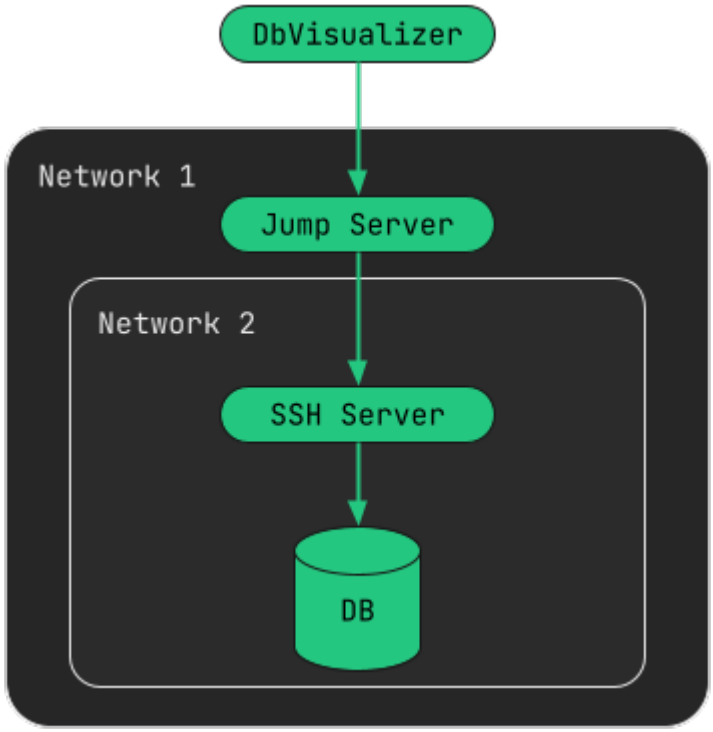
```
Host arc
        Port 2022
        HostName arc.example.org
        User fred
        IdentityFile ~/.ssh/id_rsa_arc
```

When using a config file, the **Host** field in DbVisualizer should refer the **Host <host>** defined in the config file. Having options in the config file that are already in the SSH configuration setup in DbVisualizer, then the config file options will be used except for Userid which when set in DbVisualizer will override any **User** option set in the config file. To see what actual options are used, run Test Connection.

### Jump Server

Jump Server (also known as **Jump Host**, **Remote Host** or **Bastion Host)** is a gateway server that gives access to instances within a private network using the SSH protocol. You can log into the instances of your private network via a single, centralized, entry point. The following image illustrates that with a Jump Server set it is invoked as the first server from DbVisualizer and then a connection with the target SSH Server is established. The jump server and the target SSH server may use different authentication types and implementation types.

> There may be only one Jump Server in a chain of SSH servers.

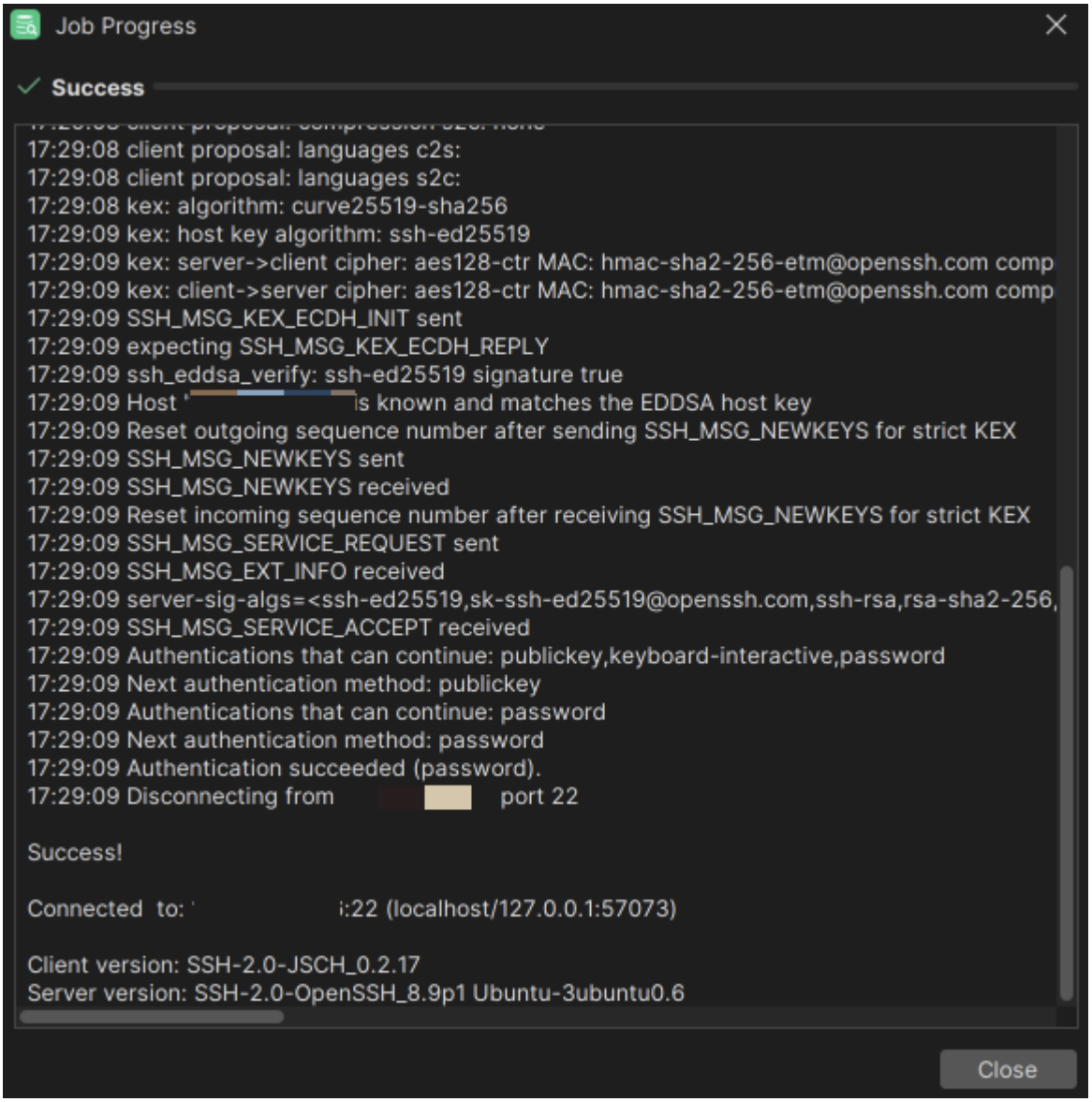



A jump server is just another SSH server configuration in DbVisualizer. Create, test it, and then use it as a jump server for one of the other defined SSH servers.

### TESTING AN SSH CONFIGURATION

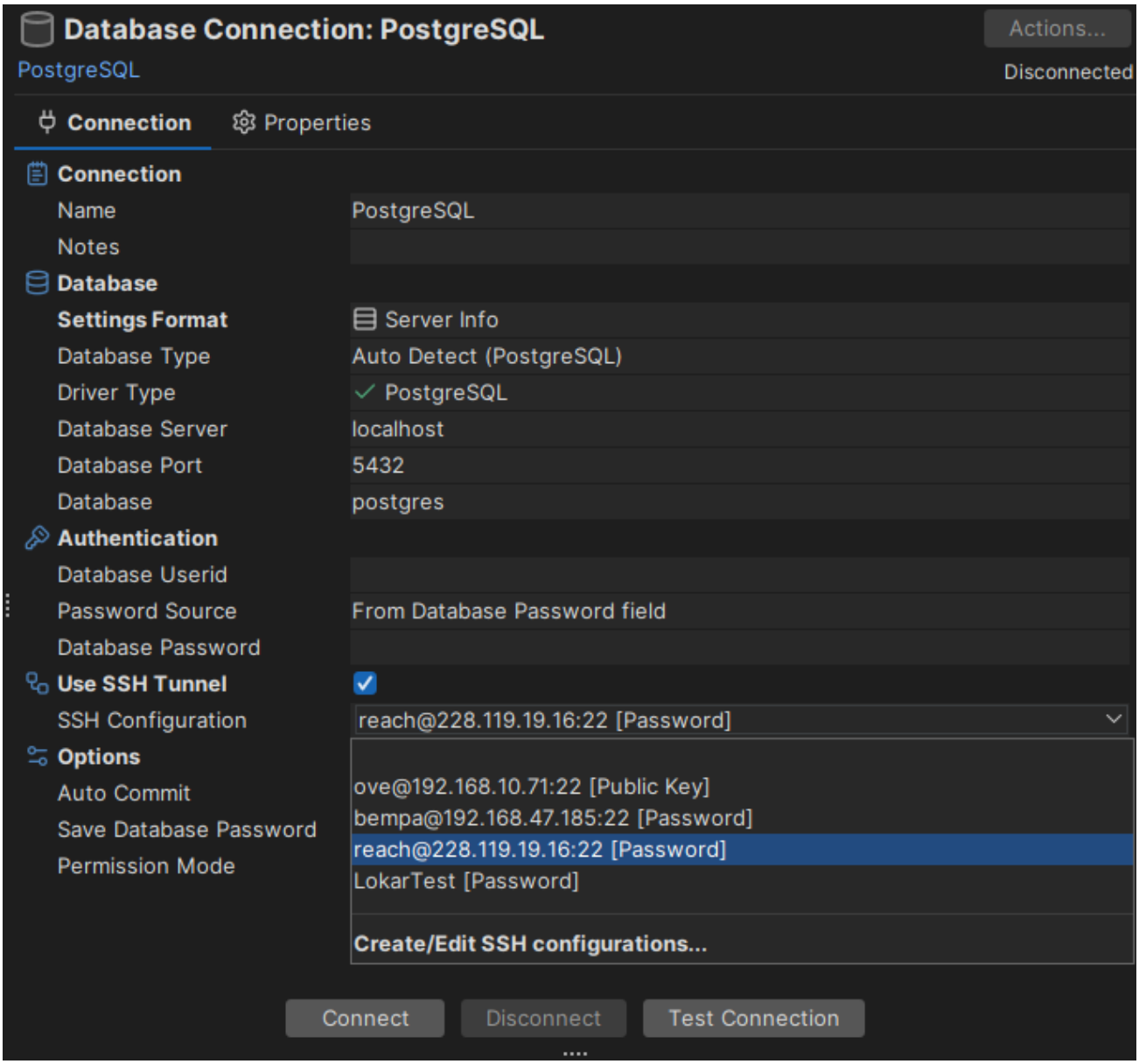
Using an SSH configuration when connecting to a data source and there are connection issues, it might be difficult to distinguish if the problem is related to establishing the SSH tunnel or if it is a problem getting connected with the target database. Testing just the SSH tunnel is really useful and the **Test Connection** is handy. Once you get the tunnel working and later experience connection problems with the target database, those kinds of connection problems are related with the database only.

Click the **Test Connection** button to verify that you are able to connect and to detect any problems.

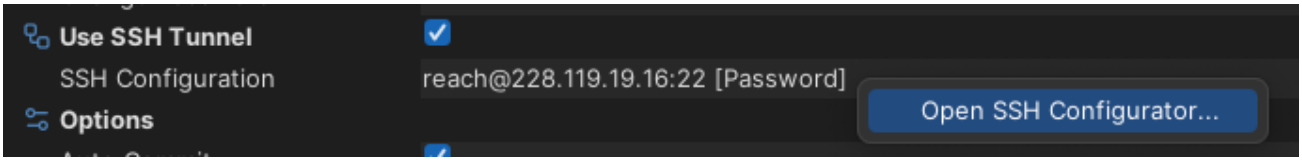



## Enabling SSH for a database connection

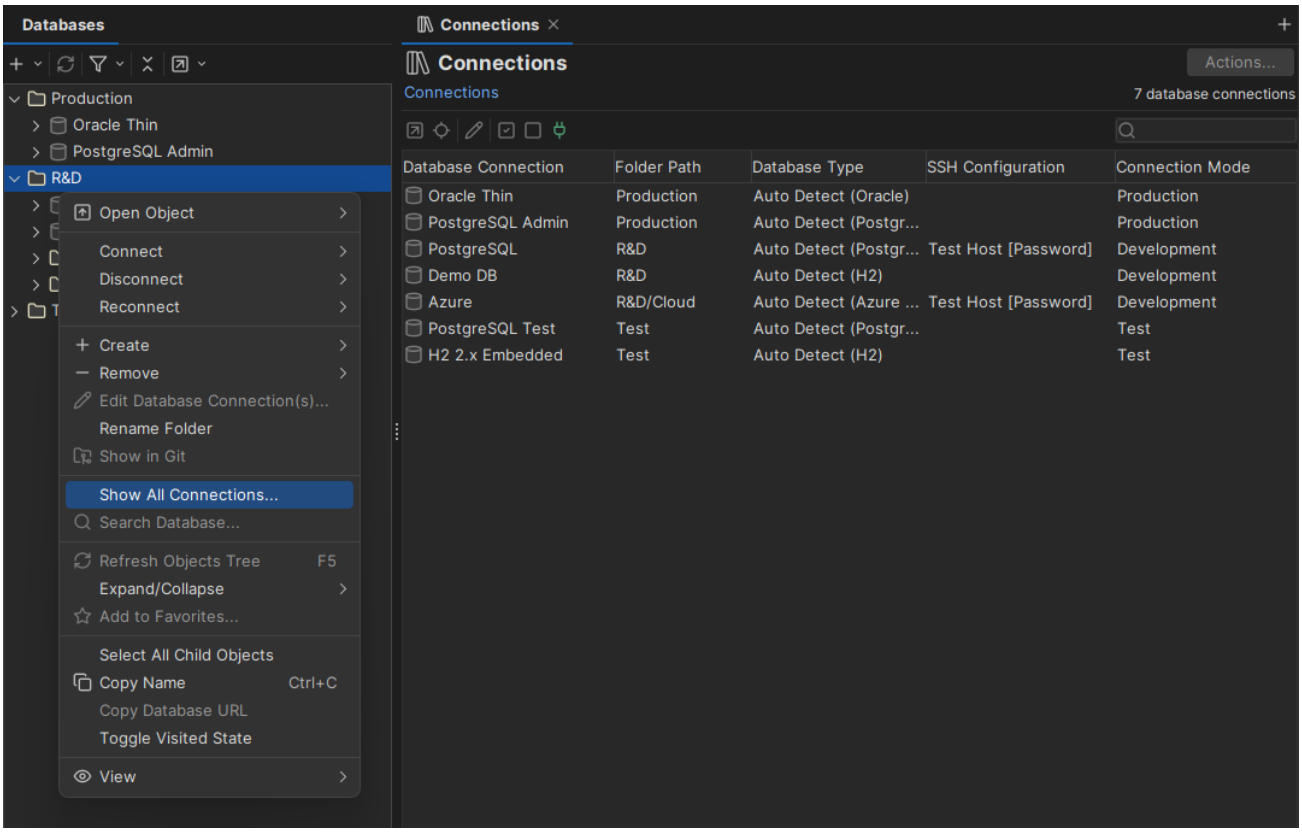
To use a SSH tunnel for a database connection, click **Use SSH Tunnel** and select a SSH configuration in the dropdown:




To edit an SSH configuration that is associated with a database connection, right-click the SSH configuration field and select **Open SSH Configurator**:




Another option to browse and set the SSH configuration is to open the **Connections** Tab from the right-click menu in the **Databases** tab to list all database connections.




To clear out the use of an SSH configuration, just choose the first empty entry in the drop-down listing all SSH configurations.

> The **Use SSH Tunnel** checkbox is only shown when the **Server Info** settings format is selected, and only for databases identified by at least a **Database Server** and a **Database Port** (i.e. not for embedded databases or when using the TNS Connections Type for an Oracle database, or similar).

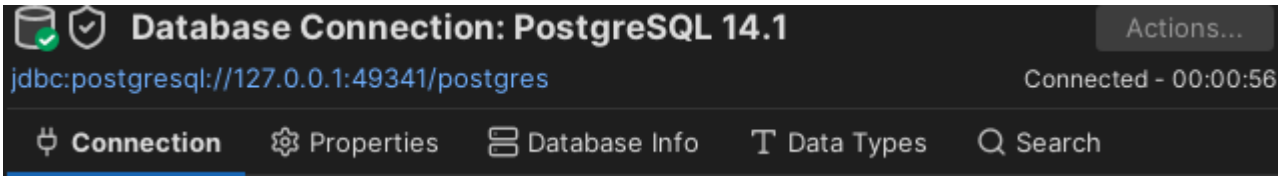
When connecting to the database, you might be prompted for SSH password or passphrase depending on the settings for the SSH configuration.

The JDBC URL is constructed using 127.0.0.1 as the Database Server portion and <LocalPort> as the Database Port portion, e.g. like this for the Oracle Thin driver when <LocalPort> is 50538:

```
jdbc:oracle:thin@127.0.0.1:50538/XE
```

In other words, the JDBC driver connects to the SSH tunnel's local port, which then forwards all communication to the database server.

The URL that is used for the connection is shown at the top of the **Object View** tab for the database connection when a connection is established, along with a certificate icon if the connection is made through an SSH tunnel.

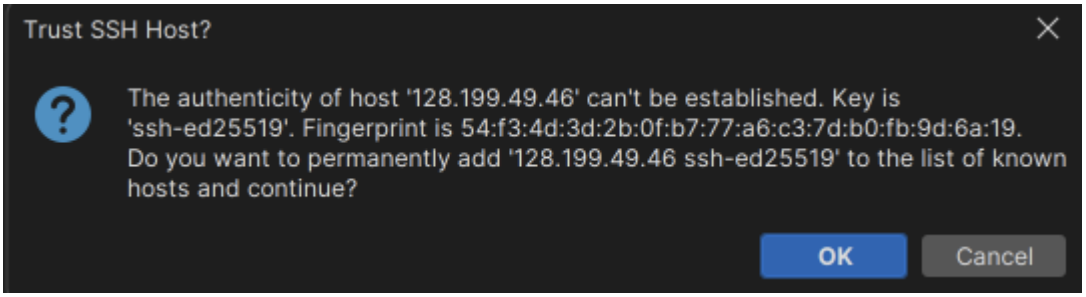



## Common problems

Establishing an SSH tunnel may result in various errors and problems due to algorithms, key lengths, and much more. Here we list a few of them and possible solutions.

THE AUTHENTICITY OF HOST 'X.X.X.X' CAN'T BE ESTABLISHED

This question is displayed when connecting to notify that the host is not currently a known host.




If you recognize the host as a "good" host - click **Yes** to add it to the **Known Hosts File** for the SSH configuration. Accepting (i.e. click **Yes**) not known hosts will then be collected in the file and there will be no future prompts for the host. Note: The dialog text may differ slightly depending on SSH Implementation.

"INVALID PRIVATEKEY: [B@....."

This error is reported when using a private key file that is not in a valid format. If you are sure it is a valid private key this error may also occur when using keys that are in the "new OpenSSH" format rather than classic. One potential fix on the OpenSSH problem is to ensure you generate the keys with the **-t PEM** option to ssh-keygen. The following example shows the command used to create new keys and the other how to convert existing keys.

```
# Create new keys with the -m PEM option
ssh-keygen -t rsa -m PEM

# Converting existing keys with -m PEM option
ssh-keygen -p -f /home/me/.ssh/id_rsa -m PEM
```

Another possible solution if your private key is generated by PuTTY (file extension .ppk) is to convert the key from a PuTTY private key to an ssh private key. This can be done in PuTTYGen as follows:

1. Create a backup of your private key file.
2. In PuTTYGen's main interface, click the **Load** button.
3. Select your private key (.ppk) file and click **Open**.
4. Go to **Conversions->Export OpenSSH key.**
5. **Select location and file name for your new key and click Save.**

# SSL and TLS

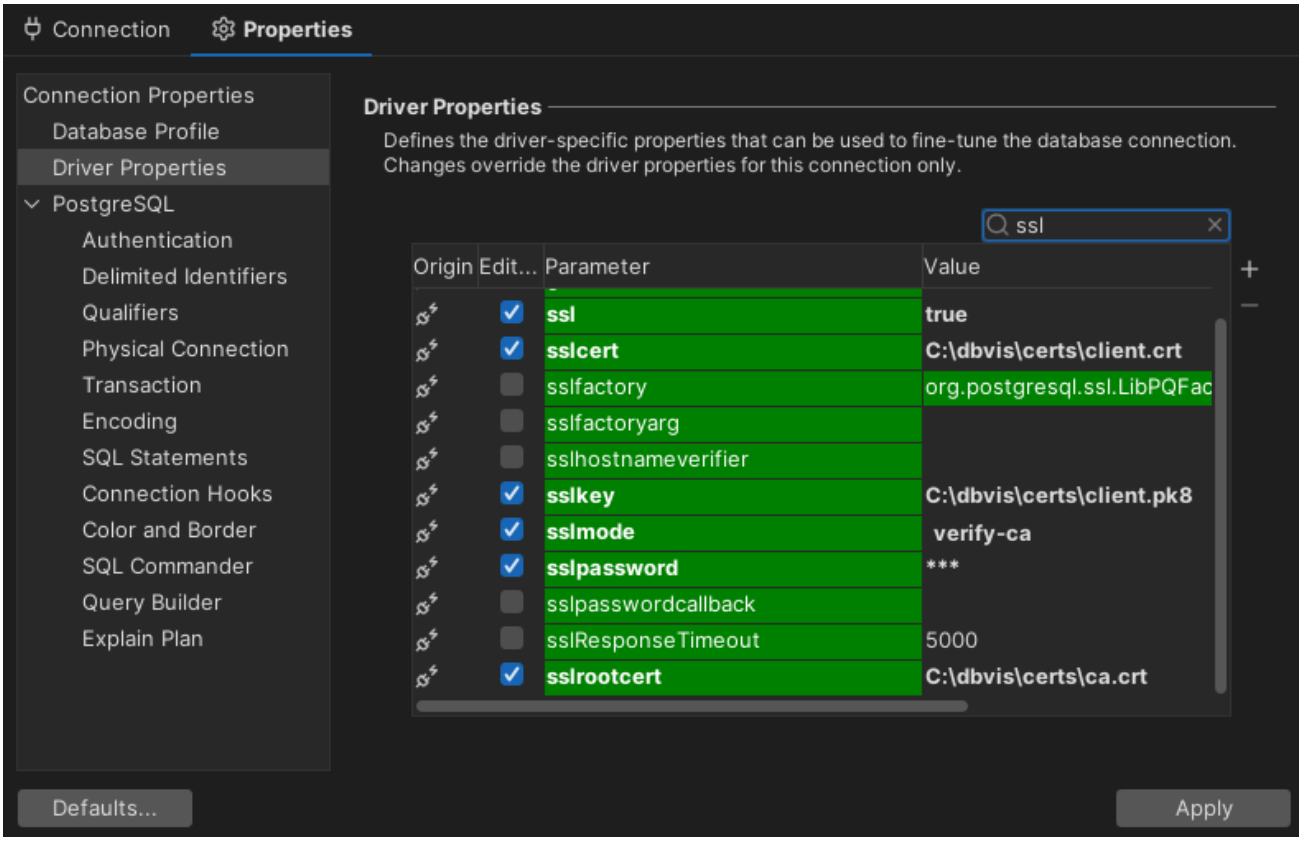
When DbVisualizer communicates with a database server, all communication (except passwords) is done in plain text; by intercepting the communication, someone can access the transmitted data and even modify it while in transit. To protect your communication you need to encrypt the communication between DbVisualizer and the server. Kerberos, LDAP, Radius, security mechanisms, external authentication options, SSH server forwarding, and a lot of other database specific features are not directly related to DbVisualizer, but we will do our best to assist in these scenarios.

> You find additional information on security related to specific databases in our support portal.

## Using SSL/TLS

Depending on the database and the JDBC driver, you may be able to use SSL (Secure Socket Layer) to encrypt client/server communications and securely authenticate client and server.

In case the JDBC driver supports SSL, you usually define the SSL settings as Driver Properties for your connection according to the documentation for the JDBC driver. The exact details depend on the versions of the database and the driver, but using **PostgreSQL as an example**, the settings can look like this:



## Certificates

For ensuring security of the data being transferred between a client and server, SSL can be implemented either one-way or two-way (aka mutual authentication). In one way SSL, only client validates the server to ensure that it receives data from the intended server. In case of two-way SSL, both client and server authenticate each other to ensure that both parties involved in the communication are trusted.

#### TRUSTING THE SERVER – ONE WAY AUTHENTICATION

A server certificate that is signed by a trusted Certificate Authority (CA) should always work fine, since the Java distribution includes a truststore with all the CA public keys.

When a self-signed server certificate is used, some additional configuration is needed. Depending on the actual JDBC driver this may include importing certificates to a truststore using the Java keytool. Some drivers allow this truststore to be configured per connection instead of for the whole Java VM. When using a truststore that affects the whole JVM special considerations must be taken.

#### USING A TRUSTSTORE FOR THE WHOLE JAVA VM

Many forums on the net suggests using the Java keytool to import the certificate into the Java VM's default truststore. The drawback with that solution is that it does not survive a Java upgrade; when the Java VM bundled with DbVisualizer is used, upgrading DbVisualizer effectively causes SSL connections to fail. We therefore recommend creating a truststore separate from the Java VM (e.g. in /Users/me/MyTrustStore) and import the certificate to that Trust Store.

Note that setting the truststore on Java VM level may affect other functionality of DbVisualizer as well as other database connections. E.g. if the certificate of dbvis.com cannot be verified neither **Help → Contact Support** nor **Help → Check for Updates** will work.

When creating the truststore you should always start with a trust store containing the needed certificates (E.g. the default java Trust Store) and add/import your custom certificate to it.

Do the following:

1. Copy the Java default truststore to your location (e.g. /Users/me/MyTrustStore). The location of the Java default truststore is in most cases <Java Home>/lib/security/cacerts
2. Import your server certificate to the truststore using keytool. The password of the Java VM truststore is in most cases changeit. Following is an example of using the keytool when importing a certificate.

```
keytool -importcert -alias mycert -file ca.pem -keystore /Users/me/MyTrustStore -
storepass changeit
```

(Replace the paths in the samples accordingly.)

The next step is to associate the custom truststore with Java. This is done in **Tools → Tool Properties** and in the **General** category. The following Java properties should be added in **Specify overridden Java VM Properties here** input field.

```
-Djavax.net.ssl.trustStore=/Users/me/MyTrustStore
-Djavax.net.ssl.trustStorePassword=mytruststore_password
```

MUTUAL AUTHENTICATION – TWO WAY AUTHENTICATION

Some database servers can be configured to require clients connecting to authenticate using a certificate.

The configuration on the client side (DbVisualizer) resembles the way a truststore is configured. In this case you may create a keystore containing a single certificate. DbVisualizer functionality is not depending on any keys in the JVM keystore.

The next step is to associate the custom keystore with Java. This is done in **Tools → Tool Properties** and in the **General** category. The following Java properties should be added in **Specify overridden Java VM Properties here** input field.

```
-Djavax.net.ssl.keyStore=/Users/me/MyKeyStore
-Djavax.net.ssl.keyStorePassword=mykeystore_password
```

Again, there may be drivers supporting configuration of keystores (and truststores) per connection. If this is supported this is the preferred way of configuration.

Example

For an example on how such a certificate can be created, see description below. Note that this and the following commands are just examples and should only be seen as a guideline.

Create your local keystore directory, for example as /users/me/MyKeyStore. Go to this directory and create your keystore and key pair with a command like below:

```
keytool -genkey -alias dbvisuser -keyalg RSA -validity 365 -keystore
client.keystore -storetype pkcs12
```

Give a password to be used for the keystore and answer the questions. As a result you will finally have the keystore in a created file with name as given: client.keystore

You can now view your keystore with command:

```
keytool -list -v -keystore client.keystore
```

And, you can create a certificate (in the example given the name mydomain.crt) to be used by a database server with command:

```
keytool -export -alias dbvisuser -file mydomain.crt -keystore client.keystore
```

This new certificate file can then be given to the DBA(s) for the database that you connect to.

VERBOSE DEBUGGING OUTPUT

If you run into problems and need more verbose logging (displayed in **Tools → Debug Window**), add this property:

```
-Djavax.net.debug=all
```

## Single sign-on (SSO), 2FA, MFA

Availability and configuration of SSO, 2FA/MFA, and other security options depend on the database and JDBC driver. Please refer to our support portal for more details.

# Read only connections

There are situations where you may want to prevent accidental or intentional write operations to a database.

> There is no way to securely prevent write operations from the client. A proper security solution must be implemented by the DBA at the server using roles and permission.

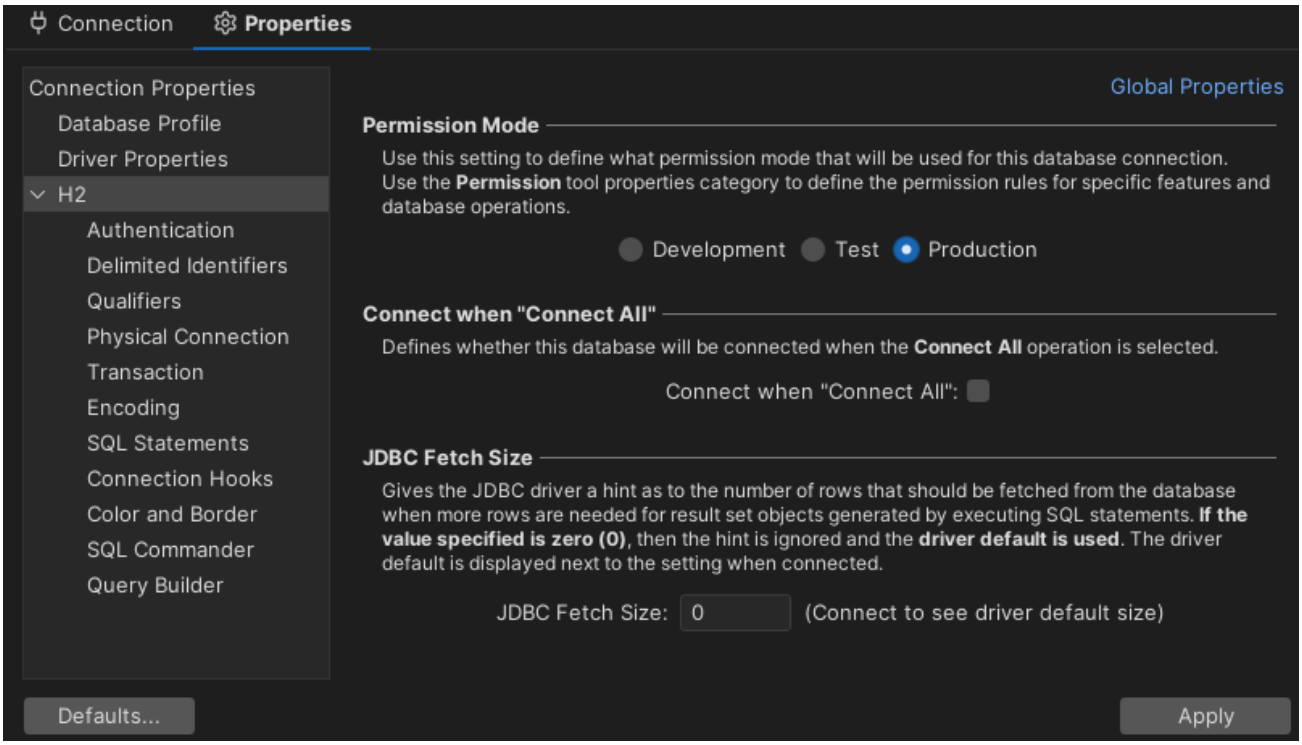
This said, depending on your JDBC Driver and the server you connect to, DbVisualizer can offer some help.

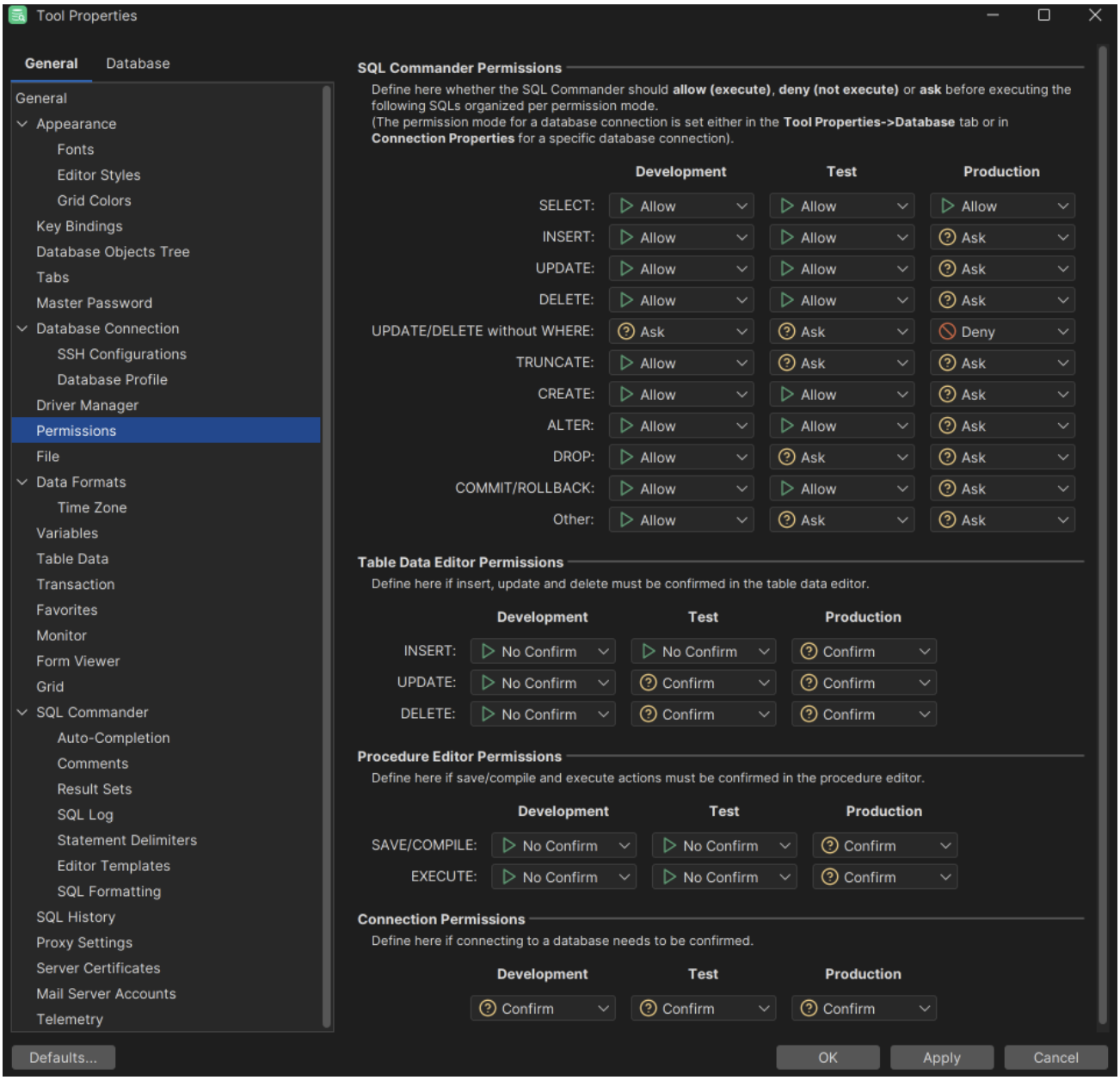
## Permission Mode

Modify the connection properties and set the desired type (**Development**/**Test**/**Production**) and then define permissions in the SQL Commander to prevent all write operations.

The default settings for the **Production** type will prompt for confirmation before doing any write operations. You can change this to **Deny** for extra safety.

**Note:** this will only affect SQL Commander and will not prevent operations like "Alter Table" when called from a menu.





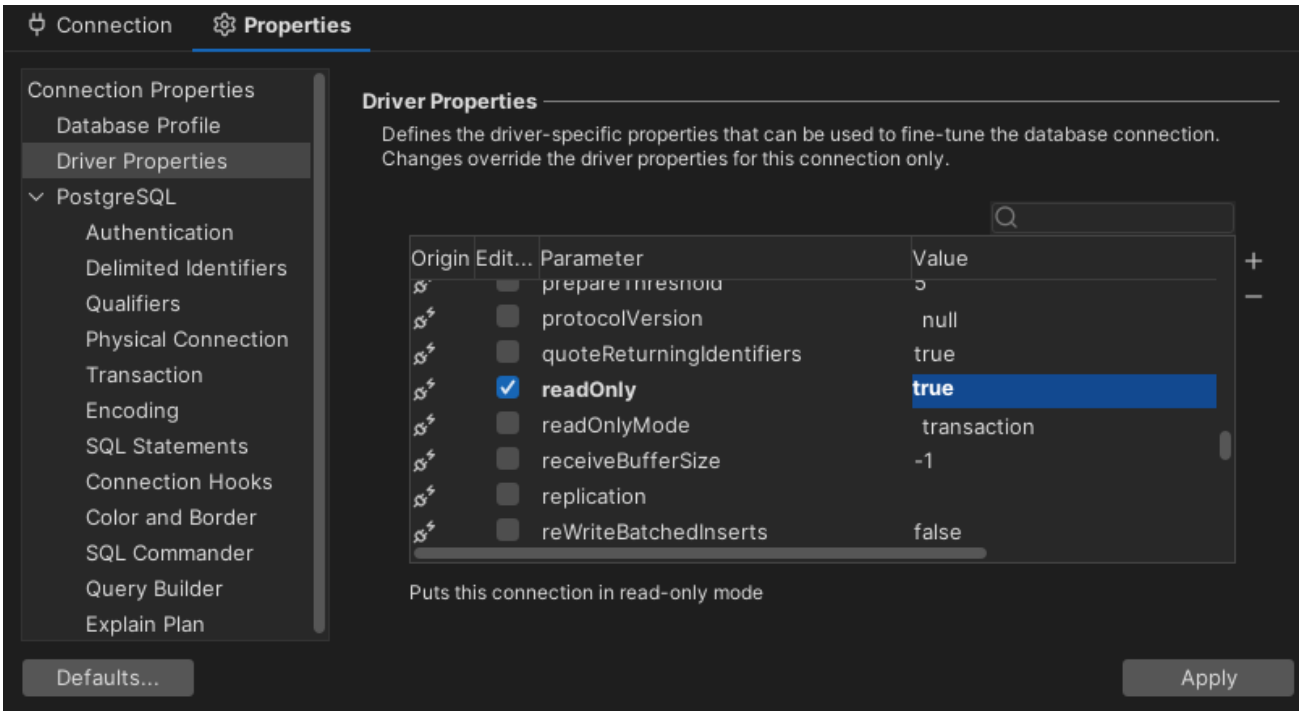

## java.sql.connection.setReadOnly

Depending on the database and JDBC driver, you may be able to set the session in read-only mode by defining the connection property **java.sql.connection.setReadOnly = true** (see Configuring Connection Properties).

**The effect of this is not guaranteed**; the server may silently ignore this setting and still operate in read/write mode.

## Setting a Driver Property

Some drivers offer properties for setting read-only mode (see example below). Please refer to the driver documentation for more information.




## Connection Hook

Finally, you can create a connection hook, i.e. an SQL statement that is run whenever you connect or disconnect to a database. Exactly what statement to run depends on the database.

# Using Oracle TNS Names

All information for connecting to an Oracle database may be stored in a tnsnames.ora file, with each database instance defined by a TNS alias. If you want to create a connection in DbVisualizer that uses the information from this file, you must first tell DbVisualizer where it is stored by setting either the `TNS_ADMIN` environment variable to the path of the folder holding the file, or making sure it is located in the ORACLE_HOME/network/admin folder and that the `ORACLE_HOME` environment variable is set.

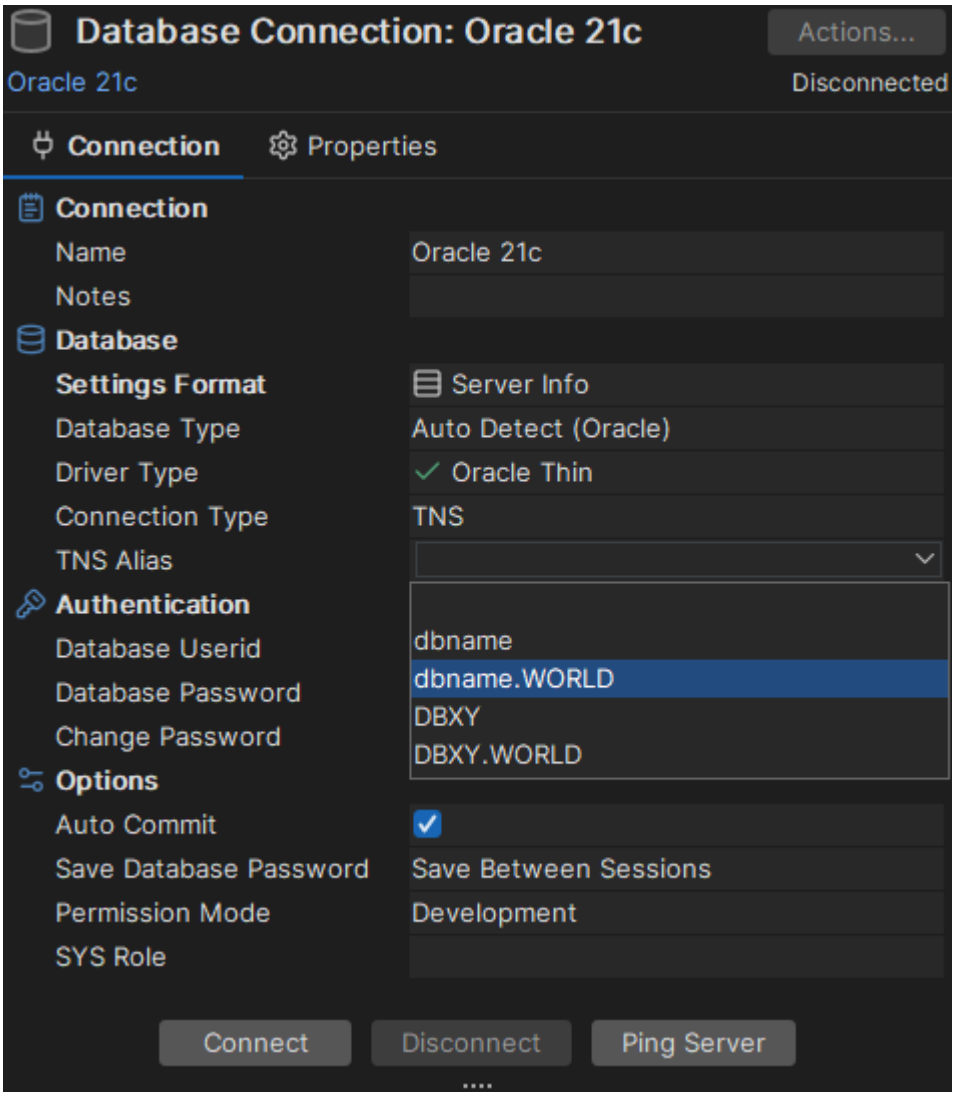
If a **tnsnames.ora** file cannot be found when clicking the **TNS Alias** drop-down, the following is displayed:

"If you like to use TNS aliases there are 2 options:

1. The tnsnames.ora file must be located either in the ORACLE_HOME/network/admin directory or in a directory identified by the TNS_ADMIN environment variable
2. Specify the `-Doracle.net.tns_admin=<folder-containing-tnsnames.ora>` Java VM Property in Tools → Tool Properties under the General tab

For general information about TNS in Oracle, see the Local Naming Parameters (tnsnames.ora) chapter in the Oracle Database Net Services Reference."

When this configuration is done, you can select **TNS** from the **Connection Type** list for the Oracle connection and then pick the TNS alias from a list of all aliases found in the file.




⚠ On macOS, environment variables set in `.bash_profile` or similar are not available to applications started via Spotlight or by double-clicking an app icon. The `launchctl` command can be used to set these environment variables instead, for instance from the `.bash_profile` script. This thread discusses an alternative solutions for Yosemite and El Capitan:

https://stackoverflow.com/questions/25385934/setting-environment-variables-via-launchd-conf-no-longer-works-in-os-x-yosemite

# Changing Database Password

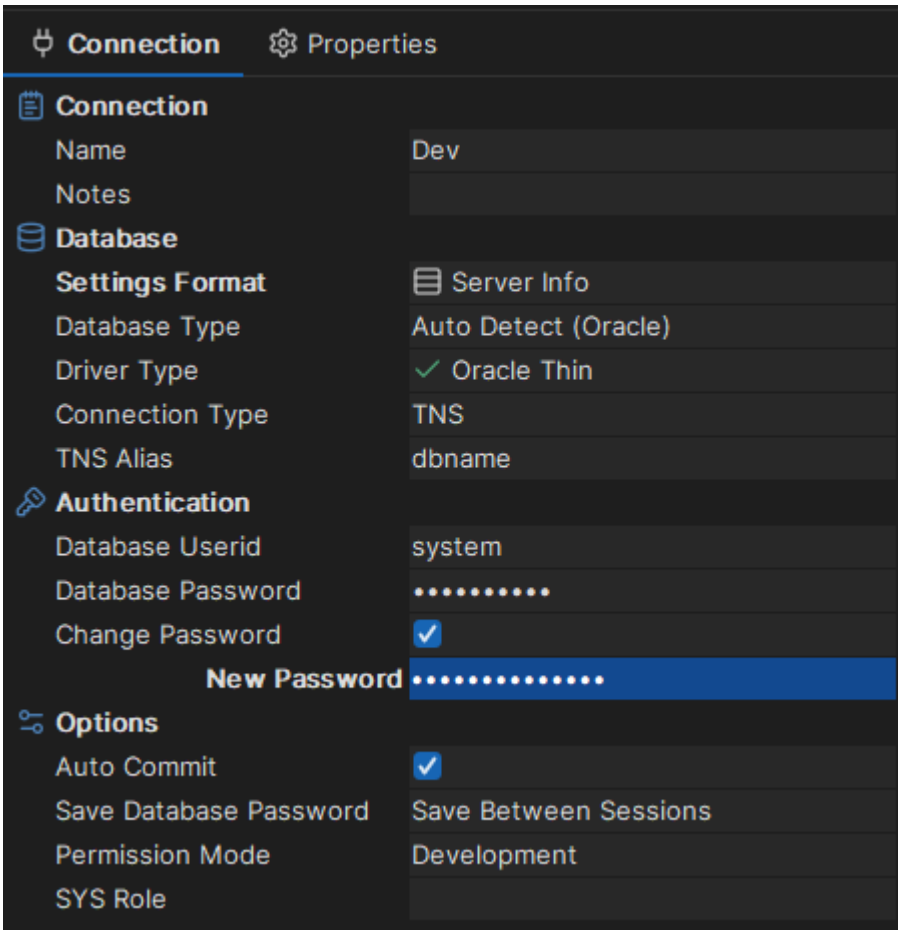
> This is supported for: Oracle 12c, MySQL, and MariaDB

If your password has expired, you get a message about this in the **Connection Message** area for the database connection when you try to connect. For Oracle 12c (and later versions), MySQL, and MariaDB, you can change the password in the Connection tab:

1. Open the Object View tab for the connection if it is not already open,
2. Check the **Change Password** checkbox in the Authentication area,
3. Enter the new password in the **New Password** field,
4. Click Connect.

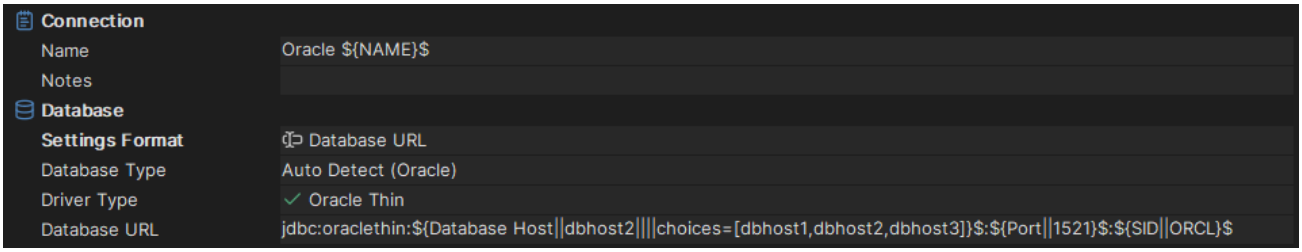
The connection is then established and the password is changed. The **Database Password** field is updated with the new password automatically.

You can also change the password as described at any time, even if it hasn't expired.


Connection
Properties
Connection
Name
Dev
Notes
Database
Settings Format
Server Info
Database Type
Auto Detect (Oracle)
Driver Type
Oracle Thin
Connection Type
TNS
TNS Alias
dbname
Authentication
Database Userid
system
Database Password
Change Password
New Password
Options
Auto Commit
Save Database Password
Save Between Sessions
Permission Mode
Development
SYS Role

# Using Variables in Connection Fields

Variables can be used in some of the **Connection** tab fields. You can use variables in the **Name**, **Userid** and **Password** (both Database and SSH) fields with the **Server Info** settings format, or in the **Database URL** field when using this settings format. This can be a useful alternative to having a lot of similar database connection objects. Several variables can be defined in a single field, and default values can be set for each variable. The following figure shows an example with variables, described in more detail in the Using DbVisualizer Variables page.

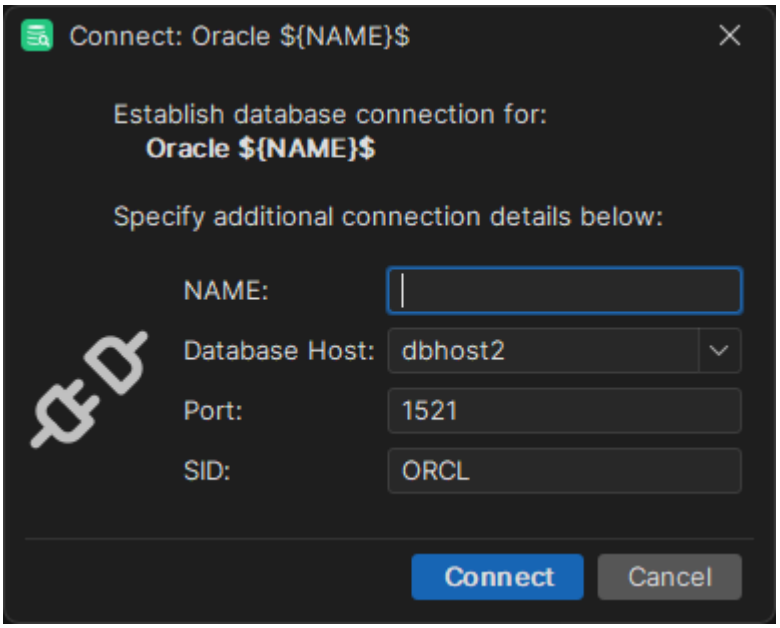



The following variables appear in the figure:

- `${Name}$`
- `${Database Host||dbhost2||||choices=[dbhost1,dbhost2,dbhost3]}$`
- `${Port||1521}$`
- `${SID||ORCL}$`

All of these variables define a default value after the "||" delimiter, except for the `${Name}$` variable, which has no default value. The default values appear in the connect dialog when you ask for a connection to be established. The `${Database Host}$` variable includes the choices option, with a comma separated list of choices that should appear in a drop-down list. The drop-down list is editable, so the user is not locked into the choices from the list.

The following figure shows the connect dialog based on the connection definition shown above.




Enter the appropriate information in the fields and then press the **Connect** button to establish the connection. When the connection is established, DbVisualizer automatically substitutes the variables in the Connection tab with the values entered in the connect dialog. At disconnect from the database, they revert back to the original variable definitions.

# Automatically Connecting at Startup

If you want to automatically connect to a database when you start DbVisualizer there are three ways to achieve this:

The first way will hold for one or more database connections and is:

1. From the **Databases** tree, select the top node and open the **Connections** Object View. See Edit Multiple Database Connections.
2. For each Database Connection that you want to automatically connect, set the **Connect All** check box

The second option will have the same result but for one database connection at the time. Then you do:

1. From the **Databases** tree, select a specific database connection node and open the **Database Connection** Object View
2. Select the **Properties** tab
3. In the left tree, select the database node and set the **Connect when "Connect All"** check box in the right panel

The third option will hold for all database connections of a certain type. Then you do:

1. Open the **Tool Properties** window
2. Under the **Database** tab, select the specific database
3. Set the **Connect when “Connect All”** check box.

Finally after one or more of the alternatives above, the auto connect function must be activated at startup with the following steps:

1. Open **Tools → Tool Properties** and select the **Database Connection** category under the **General** tab,
2. Set the **Run "Connect All" at Startup** check box.

If you enable **Connect when "Connect All"** but do not enable **Run "Connect All" at Startup**, you can instead use the **Database → Connect All** main menu choice to manually connect all connections marked this way.

Note that while the **Database → Connect All** main menu choice, will only connect all database that have **Connect All** selected, the **Database → Disconnect All** main menu choice, will disconnect all database connections.

# Executing SQL at Connect and Disconnect

Connection hooks defines optional SQL commands that are sent to the database at connect and just before disconnect. They are typically used to initialize the database session with custom settings and to clean up various resources at disconnect.

You can enter the SQL you want to execute in the **Properties** tab for the connection, in the **Connection Hooks** category.

You can also set this for all database connection of a certain type. Then you instead do.

1. Open the **Tool Properties** window
2. Under the **Database** tab, select the specific database and the **Connection Hooks** category from the tree
3. Enter the SQL in the shown **Database Connection Hooks** panel.

# Using a Single Shared Physical Connection

By default, DbVisualizer uses multiple physical connections to the database. Each SQL Commander tab is allocated its own connection. Other processes, which update the database, such as saving grid edits or importing data in a table, use their own connections. Finally, most read-only operations, such as navigating the database objects tree, use a separate shared connection.

This is normally the most efficient way to access the database, but in certain circumstances it is important to instead use a single shared physical connection for all operations. Some examples are:

- Only one session per account is allowed in the target database,
- Locking issues when modifying the same table in the Data tab and in an SQL Commander (when a pending transaction locks the whole table)
- When using one-time passwords, new physical connections cannot be established without prompting for a new password.

For situations like these, you can force DbVisualizer to use a single shared physical connection.

## MFA/2FA and browser-based authentication

Turning on "Single Shared Physical Connection" is especially useful for database connections established with any of MFA/2FA, token or browser-based authentication. The reason is that you otherwise will need to authenticate whenever DbVisualizer opens a new physical connection with the server, which may be a bit overwhelming and annoying. The downside with a single shared physical connection is that running parallel actions in DbVisualizer may be limited.

## Selecting the Single Shared Physical Connection Mode

To use a single shared physical connection:

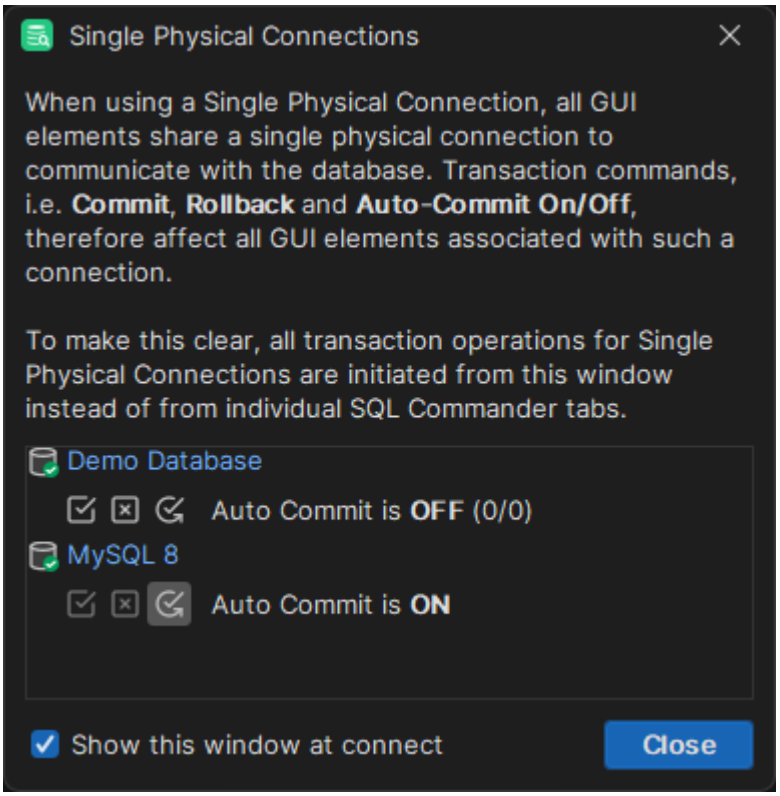
1. Open the Object View tab for the database connection,
2. Select the **Properties** tab,
3. Select the **Physical Connection** category and enable **Use a Single Shared Physical Connection**.

## Data Manipulation with a Single Shared Physical Connection

Executing a script in an SQL Commander tab, using an Action, editing a table in a grid and importing data to a table are all operations that (potentially) modify data in the database. When a single shared physical connection is used, only one such operation may be performed at a time. If you try to start an operation like this while another one is already being processed, a dialog will pop up asking you to try again later.

## Transaction Handling with a Single Shared Physical Connection

If you have **Auto-Commit** disabled with Single Shared Physical Connection enabled, commits or rollbacks done in one part of the GUI affect changes done in any other part of the GUI. An example is if you have executed UPDATE or INSERT statements in an SQL Commander tab, and then edit a table in its Data tab and commit those changes, you are also committing the changes made by the UPDATE or INSERT statements. To make this clear, all GUI controls for transaction handling for shared physical connections are shown in a separate Single Physical Connections window.

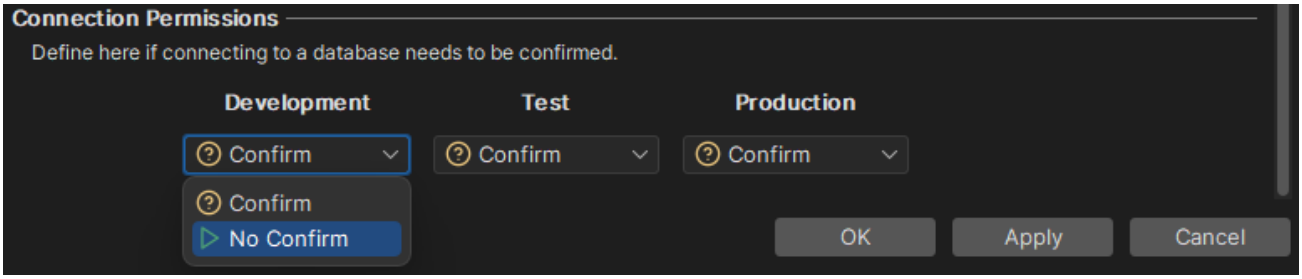



This window pops up when you connect to a database with Single Shared Physical Connection enabled, or when clicking any of the transaction control buttons in an SQL Commander tab for such a database. You can also bring it up by clicking on the corresponding icon in the DbVisualizer status bar, or by choosing **Help → Single Physical Connections** in the application menu. From this window, you can enable or disable Auto-Commit and manually commit or rollback a pending transaction.

You also get prompted to commit, rollback or continue working within the same transaction every time an operation results in data changes. Before potentially making lots of changes, you get prompted to enable Auto-Commit, since making lots of changes (e.g., importing lots of data) may fill up redo logs if running with Auto-Commit disabled.

# Connection Permissions - Confirming connect

DbVisualizer will in certain cases launch a dialog to confirm a new connection. The **Connection Permissions** in **Tool Properties → General → Permissions** let you specify if a connection should be confirmed or not depending on the connection mode.



# Metadata Cache

The metadata cache is used internally in DbVisualizer to speed up various features such as auto-completion, table data editing, and for functions that gather information about columns, keys, tables, etc. The cache keeps only metadata such as schema, table, and column names. It does not store table data or other sensitive information.

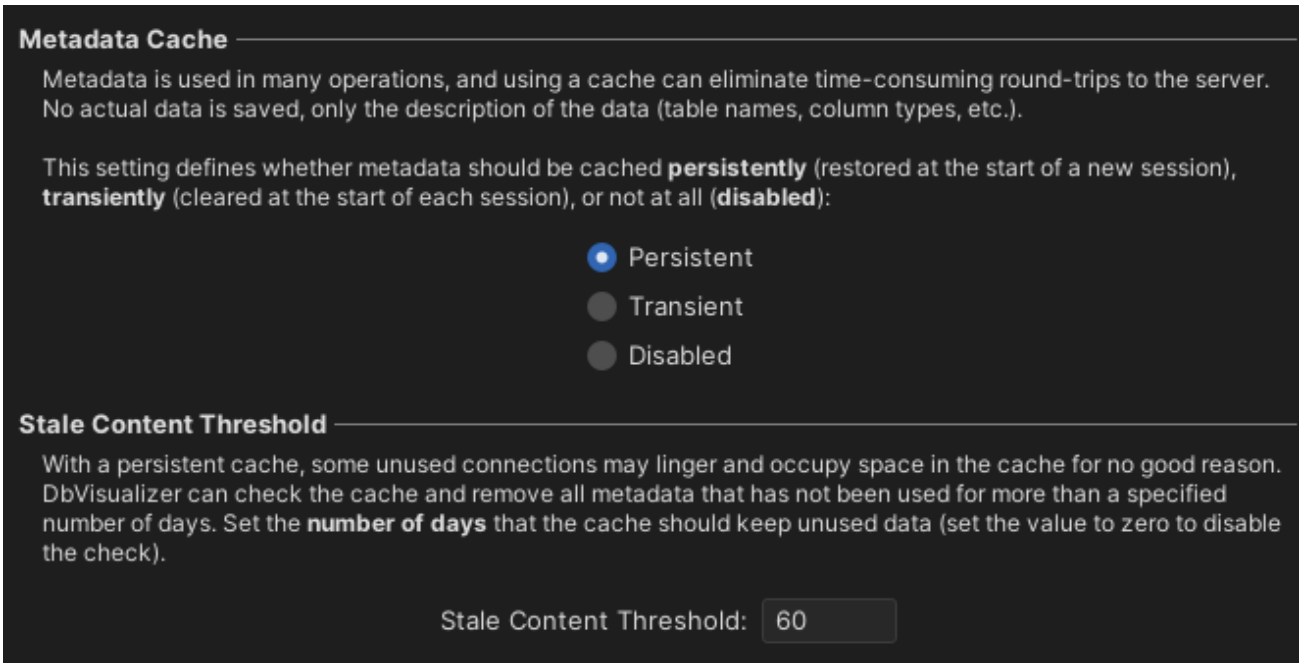
## Configuring the Metadata Cache

The metadata cache can be set to any of these modes:

- **Persistent** (default)
  the metadata is stored in a local cache and is reused for new sessions
- **Transient**
  metadata is cleared for each new session
- **Disabled**
  the cache is not used at all

When the cache is Persistent, you can specify the number of days to keep stale (unused) contents.

You find the setting for the metadata cache in **Tools → Tool Properties** and in the **Database Connection / Metadata** category.

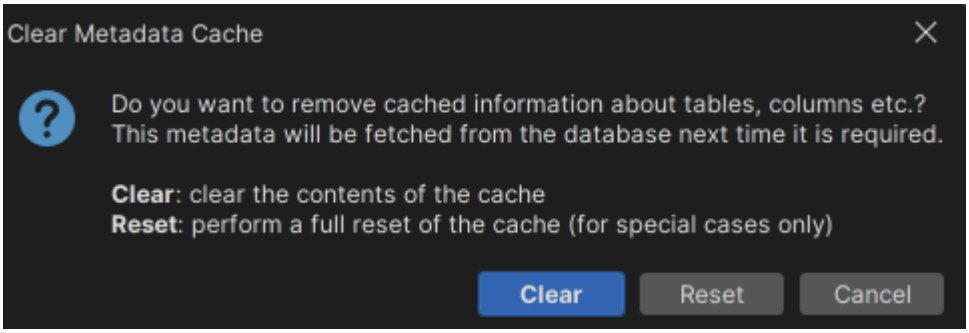


# Refresh the Metadata Cache

When metadata has been changed outside DbVisualizer or via scripts, you may need to refresh the cache and reload metadata. This can be done for all connections, for a single connection, or for some elements of a connection.

## Using the File Menu

Use **File → Clear Cache** to clear any of the listed connections, or the entire cache (**Clear All**). When prompted, you should normally select to **Clear** the cache; the choice **Reset** will remove and recreate the metadata cache and should only be used when there are issues with the cache.



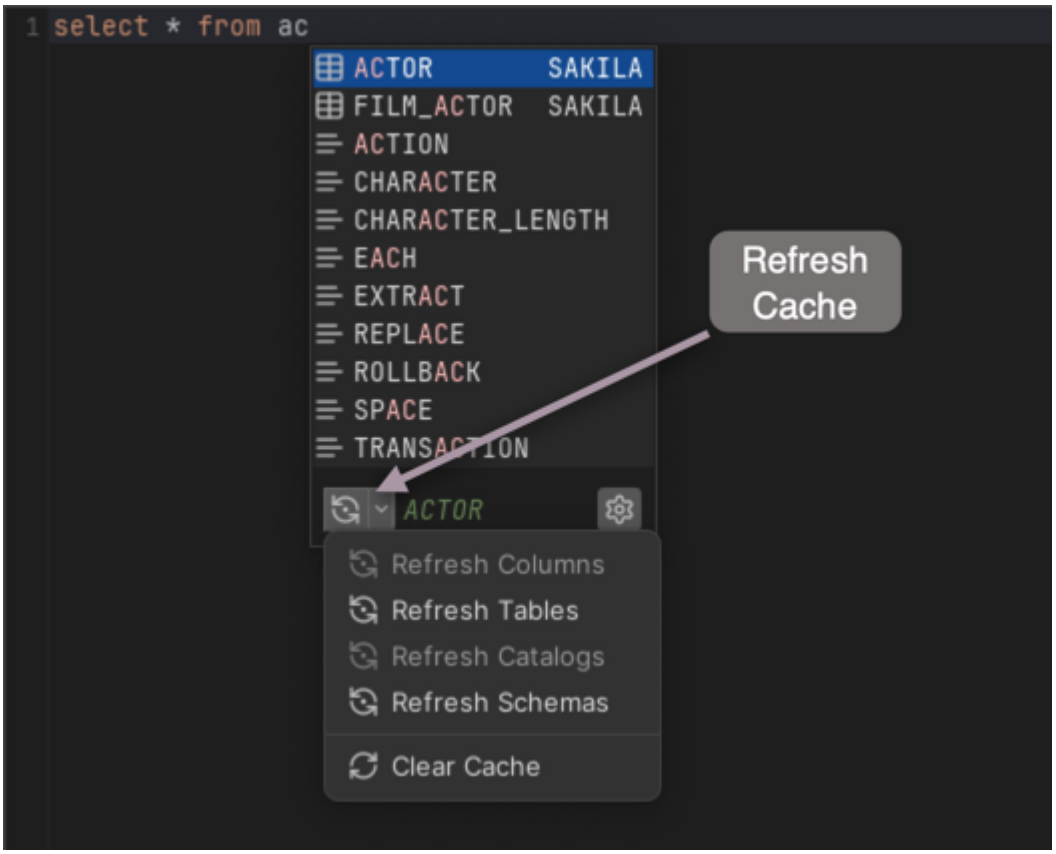

## Using the Object Tree

Qualifiers (catalogs and schemas) are refreshed when you select **Refresh Object Tree** from the right-click menu; this will also update the database/schema drop-downs in the SQL Commander.

## Using the Auto-Complete Popup

The auto-complete window in an SQL Commander has a button to refresh the metadata cache for the actual database. This might be handy in case there is a discrepancy between what is displayed in the auto-completion window and the actual data source, and you want to keep the popup open.

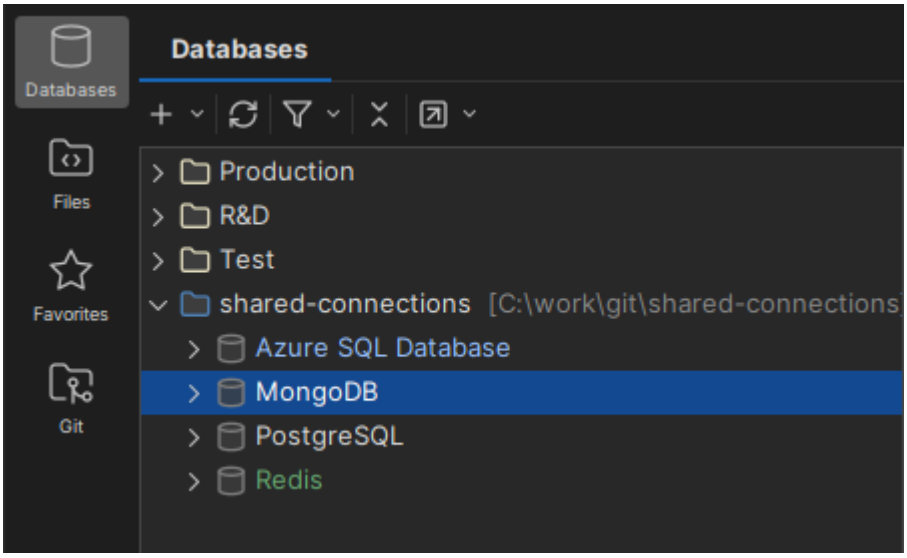
You can refresh metadata in several ways; by default the refresh button will refresh the contents currently showing in the popup (e.g. tables or columns), but using the drop-down you can choose to refresh tables, columns, catalogs or schemas, or to clear the cache (the same way as on the **File** menu). Items are enabled/disabled depending on the cursor position, the connected database, and whether the cache is enabled.


1 select * from ac
ACTOR SAKILA
FILM_ACTOR SAKILA
ACTION
CHARACTER
CHARACTER_LENGTH
EACH
EXTRACT
REPLACE
ROLLBACK
SPACE
TRANSACTION
ACTOR
Refresh Cache
Refresh Columns
Refresh Tables
Refresh Catalogs
Refresh Schemas
Clear Cache

# Sharing Database Connections

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

When working in a team, chances are you will be using a common set of database connections that is the same for all members of your team. Such connections can be created and managed manually for each user, or you can export and import settings to share them between the team members. A more powerful approach is to use **mounted database connection folders**.

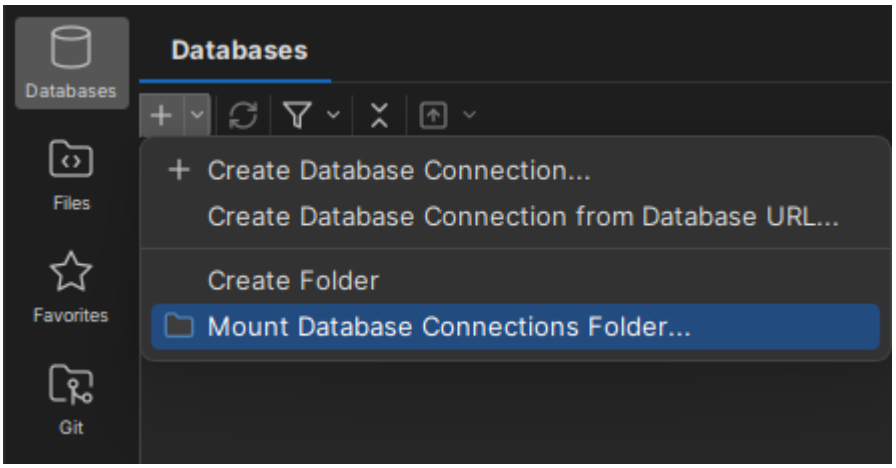



Connections placed in a mounted connection folder are stored in that location, and anyone who mounts the same folder in DbVisualizer will be able to use the included connections. Changes made to shared connections will be visible to other users, except for attributes that are defined as private and are not shared.

Typical use cases include storing shared connections on a shared network drive, or checking them into a shared Git repository.

## Mounting a database connection folder

To add a shared connection folder to DbVisualizer, open the Databases tab and choose **Mount Database Connection Folder** in the context menu or in the toolbar. Any contained connections will appear under the corresponding node in the Databases tab.




As with normal connections and folders, you can use drag and drop to **move** the mounted folder in the tree. Note that this will only change the logical presentation and will not actually move the corresponding folder on disk. Similarly, a mounted connection folder can be **renamed** to give it a more suitable display name.

To **unmount** a shared connection folder, choose the corresponding action in the context menu after right-clicking on the folder you want to remove.

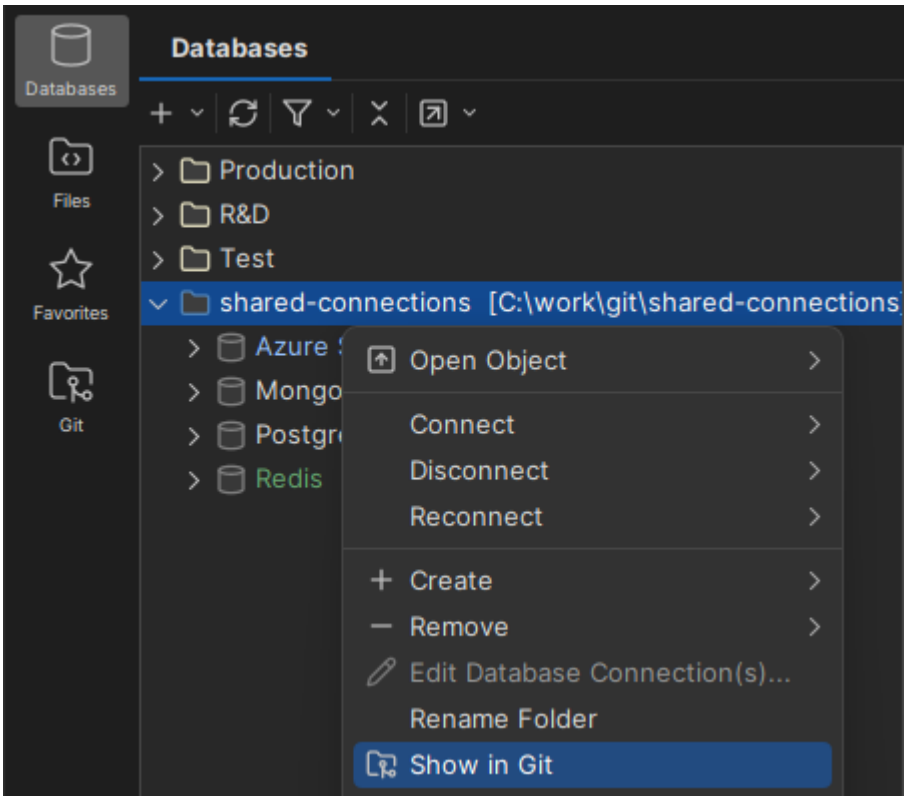
## Creating, changing and removing connections

To add a database connection to a mounted connection folder, either drag an existing connection into the folder or create a new one using the context menu or the toolbar. Shared connections can be reordered, organized into folders, renamed and deleted similarly to normal connections. To unshare a connection, drag it out of the corresponding mounted folder.

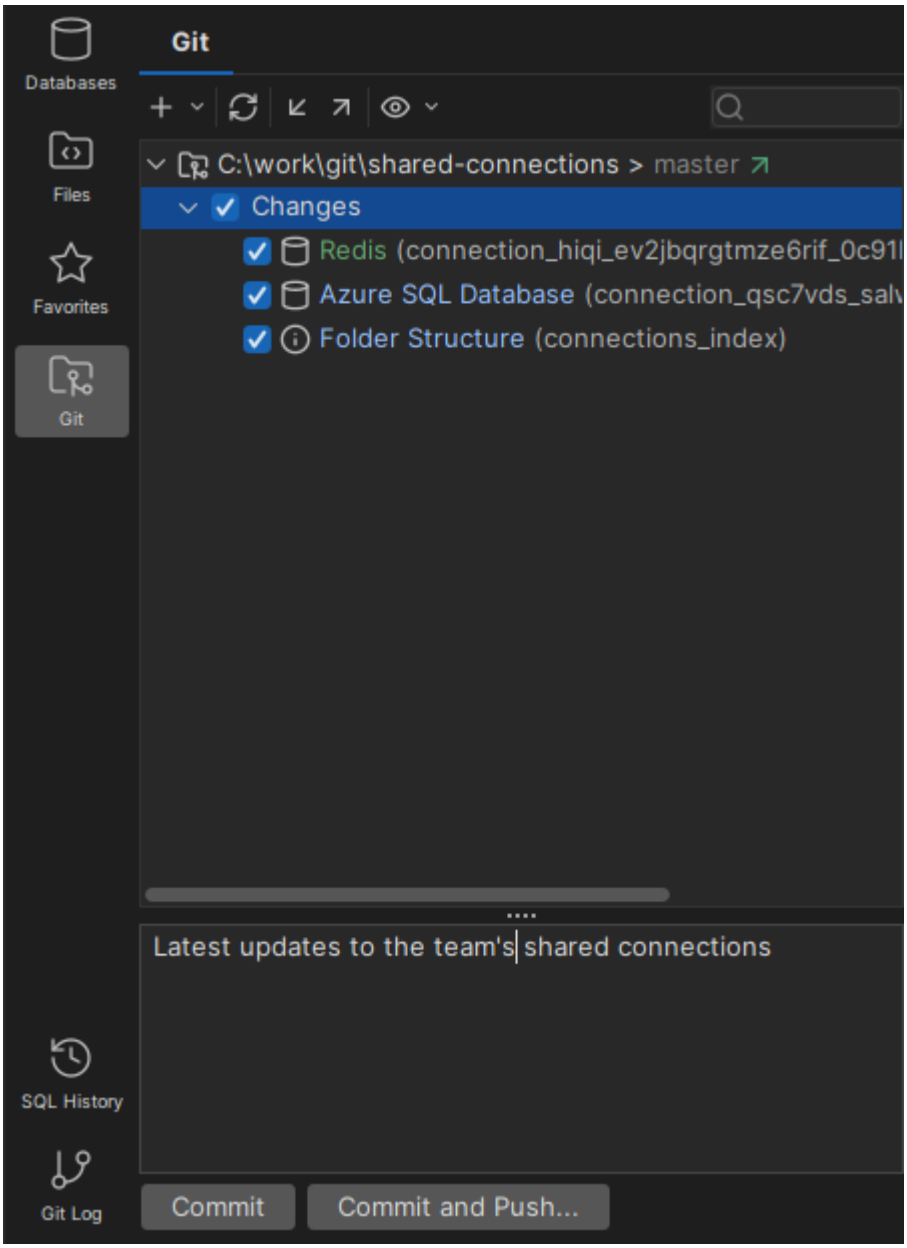
The next user to mount this folder will load the same connections, and any changes can be received by **refreshing** the folder. This can be done by selecting the folder and choosing **Refresh Objects Tree** in the context menu or in the toolbar.

## Integration with Git

Database connections can be shared using Git if the corresponding mounted connection folder is contained within a Git repository. To enable this, the repository needs to be added to the Git tab. To do so, right-click and choose **Show in Git**. You can also start by mounting the repository in the Git tab first and choosing **Show in Databases** instead.




When changes are made to connections in a mounted connection folder that is Git-backed, the corresponding changes will be marked in the tree and will also appear in the Git tab. From there, you can publish your changes and fetch incoming updates in the same way as you do with Git-backed scripts.




## Shared and private settings

While most of the important database connection settings are shared with other users, there are some that are considered private. Private settings must be manually set up as needed for each user and are stored locally.

The following connection settings are **shared** with other users:

- Connection tab: Name, Notes, Settings Format, Database Type, Driver Type, Database Server, Database Port, Database URL, Database, Instance, Use SSH Tunnel, SSH Configuration.
- Properties tab: all settings in the Driver Properties section, except those recognized as passwords.

The following settings are **private**:

- Connection tab: Database Userid, Database Password, Auto Commit, Save Database Password, Permission Mode.
- Properties tab: all settings except Driver Properties that are not recognized as passwords.

> **User credentials are private and are not included in the shared connection definition.**
>
> Being able to choose which settings will be shared and which will be private would allow more flexibility when setting up shared connections. Support for this is planned for a future version of DbVisualizer.

## Drivers and SSH settings

Drivers and SSH settings are not shared between users. However, a set of requirements that the driver or SSH configuration must adhere to is shared, making the setup seamless in most cases.

While different users have their own independent drivers, all users sharing a connection will use drivers based on the same **Template Driver**. When a shared connection is added to a user's environment, DbVisualizer will automatically find a matching driver or create a new one if necessary.

As with drivers, SSH configurations are also individual for each user. The values that all users must agree on are **Host** and **Port**. When a new shared connection is added, DbVisualizer will attempt to find a matching SSH configuration and use it. If none can be found, the user will have to set up a corresponding configuration to be able to use the connection or update the connection to use a different SSH configuration. In the latter case, the change will be propagated to other users as well.

# Finding Database Objects and Data

## Finding Database Objects and Data

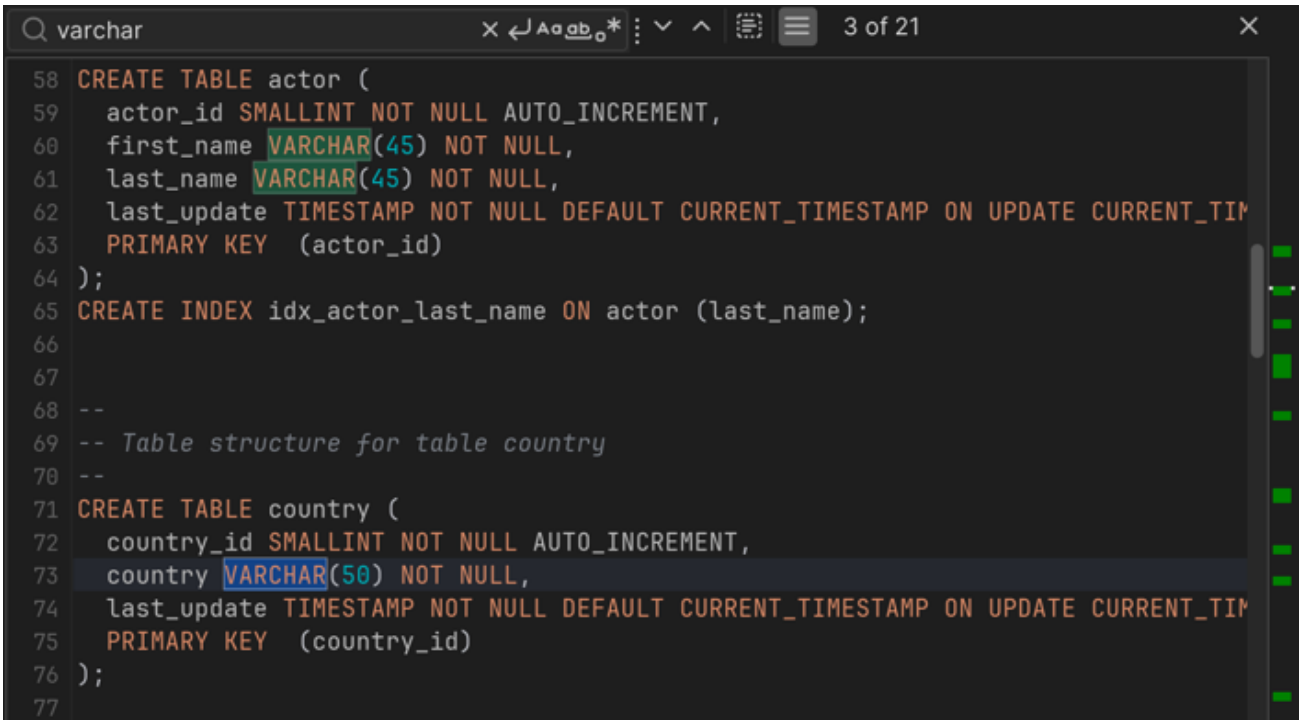
DbVisualizer provides ways to find all kinds of things, from parts of a script and data in a grid to objects in a connection tree.


- Finding and Replace in the Editor
- Finding Data in a Grid
- Locating an Object in an SQL Statement
- Locating an Object in the Databases tab
- Searching Database Objects
- Synchronizing object tab selection and selection in the tree
- Search in all open editors
- Search in Tool Properties

# Find and Replace in the Editor

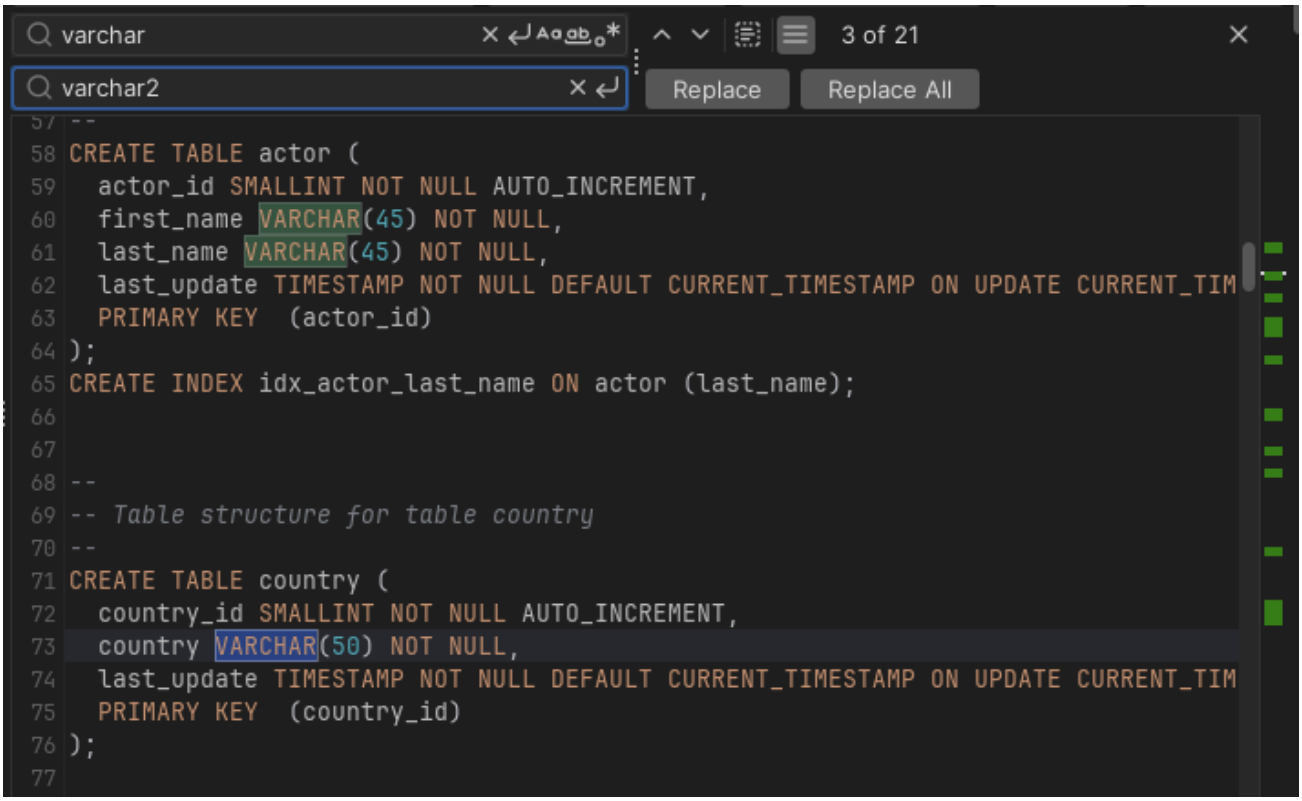
The **Edit** main menu and the editor right-click menu contains the **Find** (Ctrl+F) and **Replace** (Ctrl+R) actions.

**Find** shows a pane at top of the editor where you can enter what to look for, either as text, a regular expression or using wildcards. You can also limit the search to the current selection and use other options for a more precise search. Use **Find Next** and **Find Previous** to navigate to other matches, by default mapped to the **F3** and **Shift-F3** keys, or **Down** and **Up** keys.




Matches are highlighted while typing in the search field.

When in **Replace** mode, a second field is displayed used to enter the replacement text.




When you use a regular expression with group expressions in the **Find what** field, you can reference the captured text in the **Replace with** field using the **dollar sign** (**$**) plus the group number (e.g. `$1` for the first group).

## Regular Expression Example:

**Text in editor:**

```
SELECT
    firstname,
    lastname
FROM
    person;
```

**Find what:**

```
^(\s+)(\w+)(,?)$
```

**Replace with:**

```
$1person.$2$3
```

Deselect **Match whole word**, select **Entire scope** and **Use Regular Expression**.

When you click **Find** and choose **All** in the following dialog, you get the following result:

**Resulting text:**

```
SELECT
    person.firstname,
    person.lastname
FROM
    person;
```

# Finding Data in a Grid

The right-click menu for a grid contains the **Find Data** and **Find Column** choices.

**Find Data** shows a Quick Find field where you can type text to look for, and use the **Up** and **Down** keys to find the next or previous occurrence. Use the **Escape** key to close the field.

**Find Column** works the same, except it locates a column with the name you type.

# Locating an Object in an SQL Statement

To open an **Object View** tab for an object named in an SQL statement (e.g. a table in a SELECT statement) in the SQL Commander tab, place the caret in or next to the name and choose **Show Object at Cursor** from the right-click menu.

# Locating an Object in the Databases tab

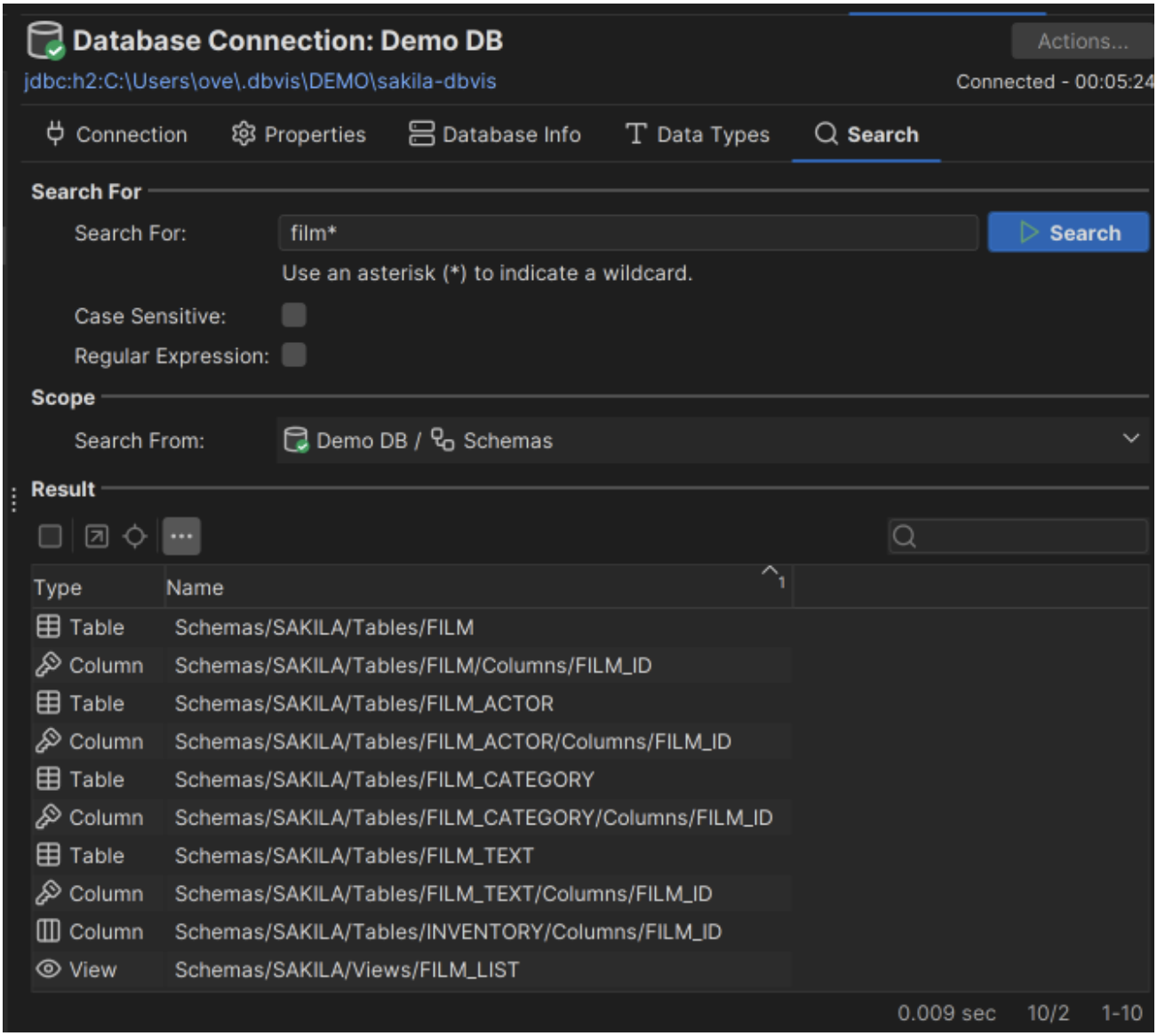
With a node selected in the Databases tab, typing any character shows a Quick Find field where you can type the name of an object you want to locate. Use the **Escape** key to close the field.

> Note that only the visible, expanded, nodes are searched. To search among all nodes for a connection, see Searching Database Objects.

# Searching Database Objects

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

The **Search** tab in the **Object View** tab for a connection is used to search among the objects in the tree by object name. The types of objects that are searchable depends on the database you are connected to. For instance, columns are included in the tree for some databases but not for others.



Search by specifying the name of the object, or name pattern, and press the **Search** button. You can use asterisk (*) as a wildcard in a pattern, or you can use a regular expression pattern if you enable it by checking the **Regular Expression** checkbox. You can also specify where in the tree to start the search, and whether to do a case-sensitive search.

You can interrupt a search operation when the Search button displays **Stop**, or by using the **Stop** button in the grid toolbar. Use the **Show Object Path** toolbar toggle button to include or exclude a column for the complete path for each found object in the grid. This path is the same as if navigating to each object manually in the objects tree. Other grid toolbar buttons let you export and print the search result grid.

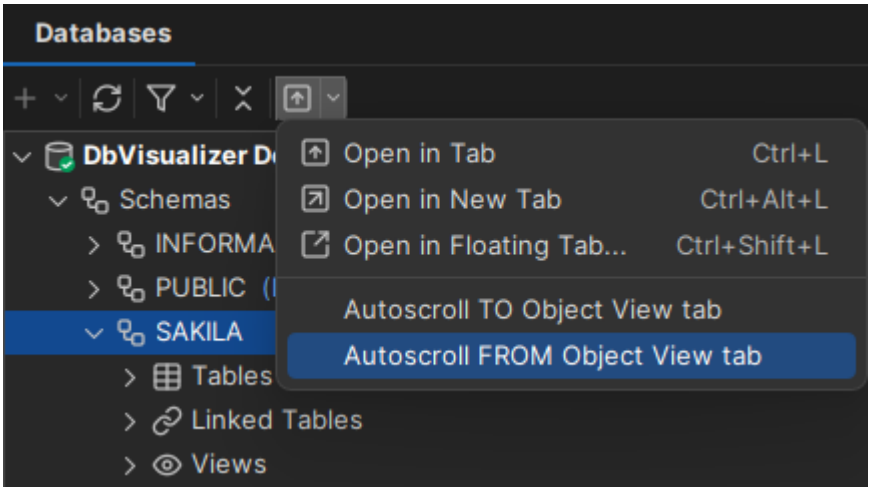
**Double-click** on a row to switch to the Object View to see detailed information about an object.

> Note that if you have tree filters or any other property that limits the content of the tree enabled, the search is performed only for those objects that match the filters.

# Synchronizing object tab selection and selection in the tree

By use of this functionality it is possible to set up DbVisualizer to automatically select the database tree node when the corresponding object view tab is selected and vice versa.

This is done through the **Open selected object(s)** dropdown in the toolbar of the Databases tab tree.



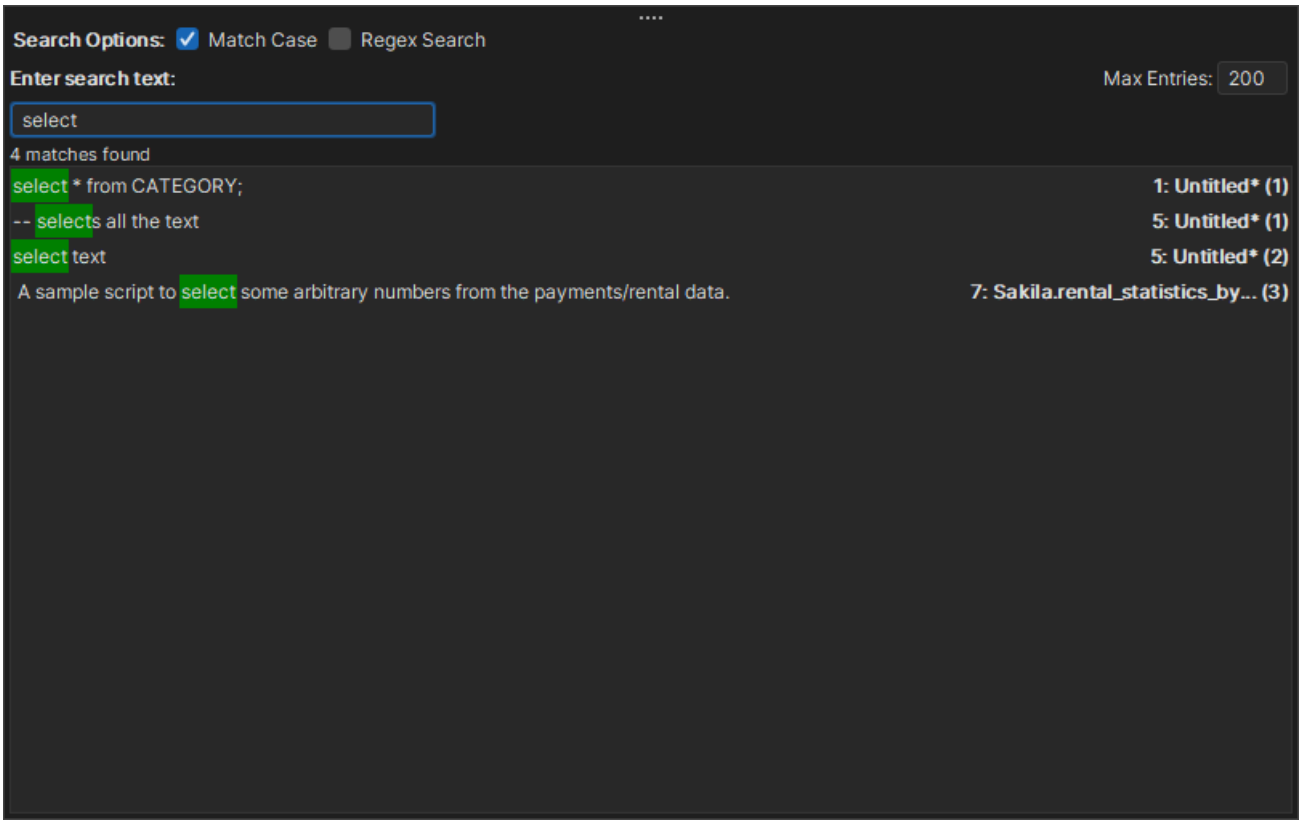

The two options controlling this are

- **Autoscroll TO Object View tab** Automatically open the object in an object view tab when the tree node is selected
- **Autoscroll FROM Object View tab** Automatically select the tree node whenever its object view tab is clicked

The default values for **Autoscroll** can be configured in **Tools → Tool Properties** and in the ** General / Database Objects Tree** category.

# Search in all open editors

**Search Open Editors** is available in the **File** menu and in the editor right-click menu under **Find**.



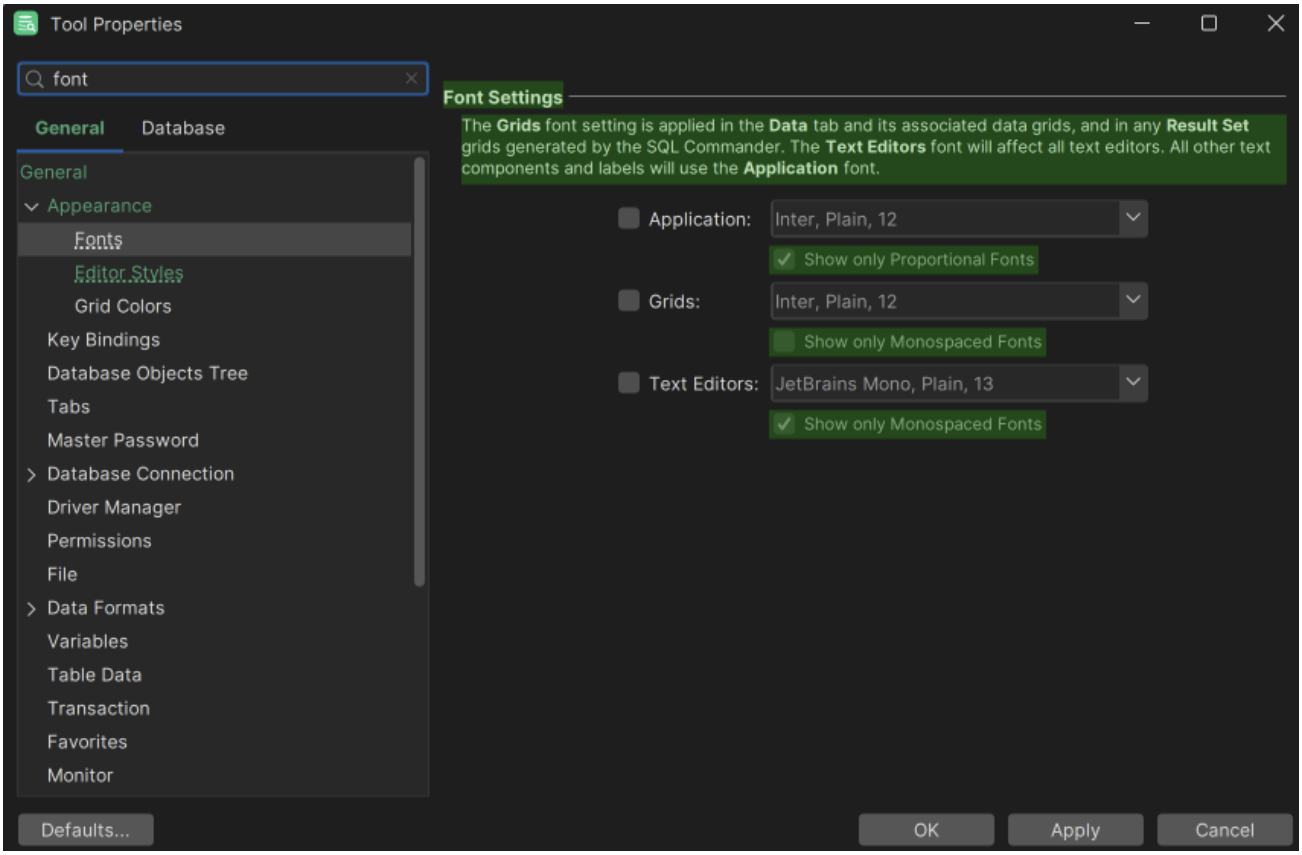

In the search dialog the following options are available:

- **Match Case -** determines whether or not the search will be case-insensitive. This setting applies even when searching using a regular expression.
- **Regex Search** - if this checkbox is checked the search string will be parsed and interpreted as a regular expression.
- **Max Entries** - the maximum number of search results which will be displayed. If this is set to **-** or to any negative number the maximum number of search results will be unlimited.

Each search result in the list shows the line where the result was found as well as which editor contains the result and the line number. Double clicking on a search result will display the editor containing that result, scroll to the line and select the search result. The highlighting color can be configured under **Tools → Tool Properties** by changing the Quick Filter background color in the **Appearance/Grid Colors** category under the General tab.

# Searching In Tool Properties

The **Search** field in the **Tool Properties** and **Connection Properties** tab for a connection can be used to search for properties. You can use keywords or parts of text that appear in the property names or descriptions. All matched properties and descriptions will be marked and the first matched occurrence is displayed.



# Transfer DbVisualizer settings

## Transfer DbVisualizer settings to new environment

To move your DbVisualizer settings from one machine to another, do as follows:

1. Install DbVisualizer on your new machine.
2. Use **File → Export Settings** in the old machine
3. Transfer the **settings.jar** file to your new machine
4. Open DbVisualizer on your new machine and choose **File → Import Settings** to import the **settings.**jar file
5. Once done, re-start DbVisualizer. (Your settings from your old machine are now migrated)

Read more about the Export and Import Settings

## Transfer the DbVisualizer Pro license to new machine

First make sure you have followed the steps in Transfer DbVisualizer settings to new environment.

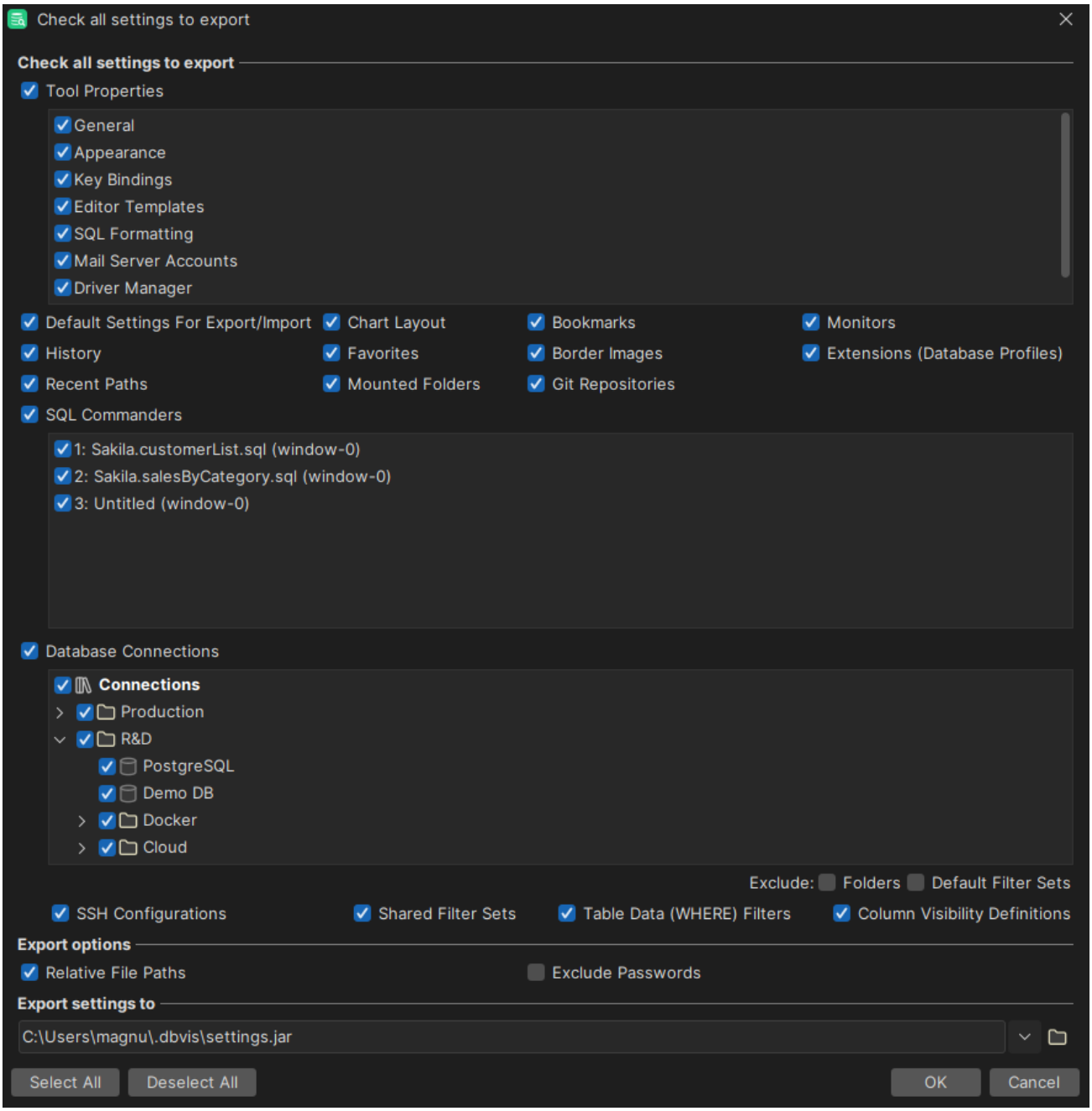
1. The location for the license file is listed in DbVisualizer Pro, license file location
2. Copy the dbvis.license file from the old machine to the **Download** folder on your new machine
3. Start DbVisualizer on your new machine and open **Help → License Key**
4. Select the **dbvis.license** file in your **Download** folder
5. Re-start DbVisualizer when prompted
6. Remove the dbvis.license file in the old machine

# Exporting and Importing Settings

Sometimes it may be necessary to migrate all your settings for DbVisualizer and import them in second DbVisualizer installation. This is very handy if you are migrating from one machine to another, or if you want to setup an exact copy on your home computer, or if you would like to share your settings with other users. Another key reason is for backup purposes. Loosing all database connection configurations can be really frustrating.

## Export Settings

The Export Settings feature is available from the **File → Export Settings** main window menu choice.




The default settings ensures that all settings are exported, but you can selectively exclude certain items. Once you've made the adjustments you want, press OK and the settings will be saved in the specified file. The structure of this JAR file is the same as the content in your DbVisualizer preferences directory. Note that when you export a Connection it may be wise to export the Driver Definition it uses if you have some specific configurations done for the driver. This ensures that the connection works as specified (if Driver Definition is not exported, the import will try to assign a matching driver).

The **Relative File Paths** option will transform all path definitions in the exported file to be relative to the DbVisualizer installation directory and your personal settings directory. This is useful if you will import the settings on another machine or share it with other users. Note that the DbVisualizer version importing relative file paths must be 7.1 or later to work properly (importing in earlier versions than 7.1 will not fail but path information will be erroneous for things such as drivers, favorites, etc.)

If Master Password is enabled for DbVisualizer and passwords are not excluded in the export you must specify an export password. The export password is used to protect the passwords that are exported. This export password is used later when importing the settings.

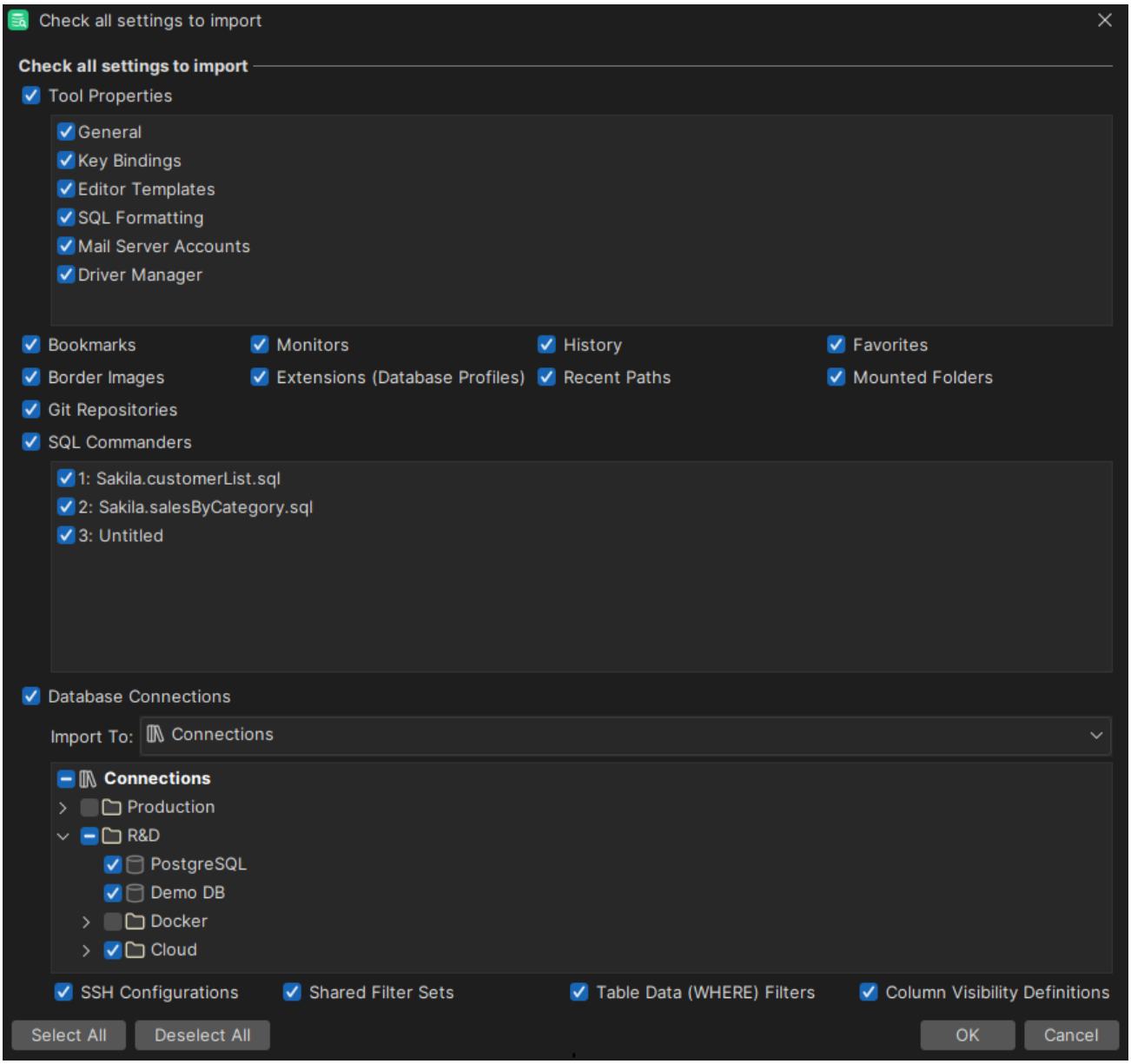



You should **not** use your master password as the export password.

# Import Settings

The Import Settings dialog is launched from the **File → Import Settings** main window menu choice.

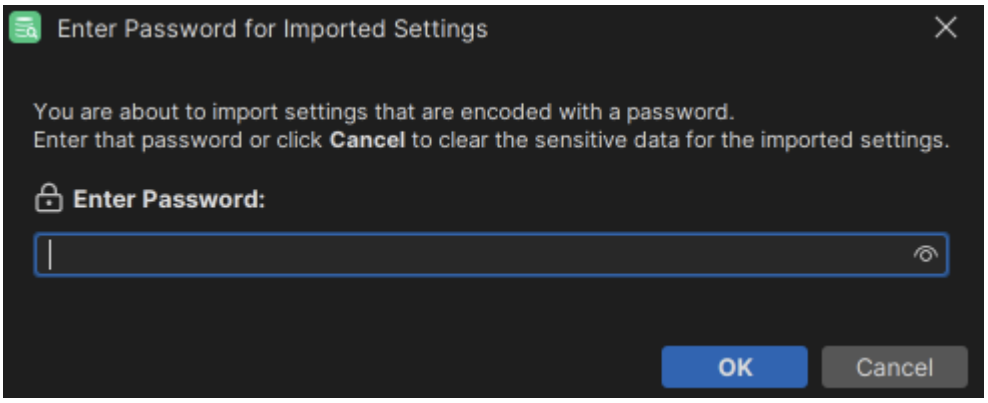
Import Settings is used to import settings as previously exported with Export Settings. Import examines the content of the specified file and present the choices available.




Use the **Import To** list to set where the imported database connections will appear in the objects tree.

Note how some of the databases and folders are pre-selected. DBVisualizer will automatically pre-select databases and folders depending on differences between the current connection settings and the imported connection settings. In this example, the icon decorations for the pre-selected connections show that the **Oracle** connection already exist but the imported settings are different, and that the **DbVisualizer Demo DB** and **MySQL 8** connections exists only in the imported settings. All the other connections exists with the same settings as in the imported settings.

If the imported data has been exported with an export password you will be prompted to enter it. You can optionally click Cancel to clear the passwords for all the connections.




Note: if the Driver Definition you import already exist, it will be reused and not imported.

# Command Line Interface (CLI)

> **Only in DbVisualizer Pro**
>
> This feature is only available in the DbVisualizer Pro edition.

In addition to the DbVisualizer GUI tool, there is also a pure command line interface for running scripts. We recommend that you use this interface for tasks that you schedule via the operating system's scheduling tool, or when you need to include database tasks in a command script for a larger job. It is also the right tool for execution of large scripts, such as a script generated by the DbVisualizer Export Schema feature.

> Please note that all client-side commands that can also be used in the scripts executed with the command-line interface.

> When a new DbVisualizer version is installed you first need to open the DbVisualizer tool and let it migrate all your old settings before using the command-line interface.

On Windows and Linux/Unix, you find this command as a BAT file (dbviscmd.bat) or a shell script (dbviscmd.sh) in the DbVisualizer installation directory. For macOS, the shell script is located in /Application/DbVisualizer-<Version>.app/Contents/Resources/app.

# Command Line Options

The command line interface supports the following options:

```
Usage: dbviscmd     -connection <name> |
                    -url <URL> -drivername <name> |
                    -url <URL> -driverclass <name> -driverpath <p1:p2...> )
                   [-userid <userid>] [-password <password>]
                   [-createconnection]
                   [-modifyconnection]
                   [-masterpw <password>]
                   (-sql <statements> |
                    -sqlfile <filename> [-encoding <encoding>])
                   [-catalog <catalog>] [-schema <schema>]
                   [-maxrows <max>] [-maxchars <max>]
                   [-stoponerror] [-stoponsqlwarning] [-stoponnorows]
                   [-stripcomments (true | false)]
                   [-processvariables]
                   [-emptypromptvalue <string>]
                   [-listconnections]
                   [-output (all | none | log | result)] [-outputfile <filename>]
                   [-debug [-debugfile <filename>]]
                   [-errordir <directory>]
                   [-workspace <directory>]
                   [-reconnectpolicy (reconnectandexecute | noreconnect |
reconnect)]
                   [-help] [-version]

Options:
 -connection <name>                  Database connection name
 -url <URL>                          Database URL
 -drivername <name>                  Database driver name
 -driverclass <name>                 Full name of the JDBC driver class name
 -driverpath <p1:p2...>              Paths to the jar files constituting the JDBC
driver.
                                     Each path separated by a ":"
 -userid <userid>                    Userid to connect as
 -password <password>                Password for userid
 -createconnection                   Creates a new database connection in the
workspace using the specified parameters.
                                     Required options:
                                     -connection <name>       Name of the
connection to be created.
                                     -url <URL>               JDBC URL for the
target database.
                                     -drivername <name>       (OR)
                                     -driverclass <name> -driverpath
<path1[:path2...]>  JDBC driver details for the database.
                                     -userid <userid>         Database userid for
the connection.
                                     -password <password>     Database password
for the connection.
                                     The command connects to the specified
database after creating the connection to validate credentials.
                                     Example:
                                     -connection "H2 1.x Embedded with password" -
```

```
createconnection -url "jdbc:h2:/path/to/db" -drivername "H2 1.x Embedded" -userid
"sa" -password "secret"
 -modifyconnection                    Modifies an existing database connection by
updating its user ID and/or password.
                                      Required options:
                                      -connection <name>       Name of the existing
connection to modify.
                                      Optional options:
                                      -userid <userid>         New userid for the
connection.
                                      -password <password>     New password for the
connection.
                                      At least one of -userid or -password must be
provided.
                                      Example:
                                      -connection "MyDatabase" -modifyconnection -
userid "newuser" -password "newpass"
 -masterpw <password>                 Master Password for encrypted database
passwords
 -sql <statements>                    One or more delimited SQL statements
 -sqlfile <filename>                  SQL script file to execute
 -encoding <encoding>                 Encoding for the SQL script file
 -catalog <catalog>                   Catalog to use for unqualified identifiers
 -schema <schema>                     Schema to use for unqualified identifiers
 -maxrows <max>                       Maximum number of rows to display for a
result set
 -maxchars <max>                      Maximum number of characters to display for a
column
 -reconnectpolicy <reconnect policy> Specify how to handle connection lost when
executing scripts
 -stoponerror                         Stop execution when getting an error
 -stoponsqlwarning                    Stop execution when getting an SQL warning
 -stoponnorows                        Stop execution when an empty result set or no
affected rows
 -stripcomments <true/false>          Strip comments before sending to database.
                                      Default is the setting made in the GUI
 -processvariables                    Process variables
 -emptypromptvalue <string>           String to use when entering an empty value
when
                                      prompted for variable(s)
 -listconnections                     Lists all database connections
 -output <out>                        "all" (default), output both log messages and
result sets
                                      "none", suppress both log messages and result
sets
                                      "log", output only log messages
                                      "result", output only result sets
 -outputfile <filename>               Script execution output file. Default is
stdout
 -debug                               Write debug messages
 -debugfile <filename>                File for debug messages. Default is stderr
 -errordir <directory>                Use an alternate location for error logs
 -workspace <directory>               Load preferences from an alternate workspace
```

```
  -help                                 Display this help
  -version                              Show version info
```

There are two options to specify which database to connect to:

- Using a connection already defined by the DbVisualizer tool. This is done by using the `-connection` parameter. If you have forgot the connection name, use the `-listconnections` option to get a list of all existing names.
- Specifying the connection properties by using the parameter `-url`. The `-url` parameter specifies the JDBC URL for the database to connect to. For information about the JDBC url see Create a New Database Connection. There are also some examples below showing how to specify connection properties using the `-url` parameter

# Examples

## Executing single statements

You can use the command line interface to execute a single SQL statement:

```
> dbviscmd.bat -connection "Oracle" -sql "select * from hr.countries"
14:34:48  START Executing Command Line, Database Connection: Oracle Database Type:
ORACLE Catalog: null Schema: SYSTEM
14:34:48  INFO  Physical database connection acquired for: Oracle
COUNTRY_ID  COUNTRY_NAME              REGION_ID
----------  ------------------------  ---------
AR          Argentina                 2
AU          Australia                 3
BE          Belgium                   1
BR          Brazil                    2
CA          Canada                    2
CH          Switzerland               1
CN          China                     3
DE          Germany                   1
DK          Denmark                   1
EG          Egypt                     4
FR          France                    1
HK          HongKong                  3
IL          Israel                    4
IN          India                     3
IT          Italy                     1
JP          Japan                     3
KW          Kuwait                    4
MX          Mexico                    2
NG          Nigeria                   4
NL          Netherlands               1
SG          Singapore                 3
UK          United Kingdom            1
US          United States of America  2
ZM          Zambia                    4
ZW          Zimbabwe                  4
14:34:48  SUCCESS  [SELECT - 25 rows, 0.007 secs]  Result set fetched
select * from hr.countries;
14:34:48  END Execution 1 statement(s) executed, 25 row(s) affected, exec/fetch
time: 0.007/0.005 secs   [1 successful, 0 errors]
```

If you like to execute just a few statements, you can pass in a list of statements:

```
> dbviscmd.bat -connection "Oracle" -sql "select * from hr.countries; select *
from hr.regions"
14:42:21  START Executing Command Line, Database Connection: Oracle Database Type:
ORACLE Catalog: null Schema: SYSTEM
14:42:21  INFO  Physical database connection acquired for: Oracle
COUNTRY_ID  COUNTRY_NAME              REGION_ID
----------  ------------------------  ---------
AR          Argentina                 2
AU          Australia                 3
BE          Belgium                   1
BR          Brazil                    2
CA          Canada                    2
CH          Switzerland               1
CN          China                     3
DE          Germany                   1
DK          Denmark                   1
EG          Egypt                     4
FR          France                    1
HK          HongKong                  3
IL          Israel                    4
IN          India                     3
IT          Italy                     1
JP          Japan                     3
KW          Kuwait                    4
MX          Mexico                    2
NG          Nigeria                   4
NL          Netherlands               1
SG          Singapore                 3
UK          United Kingdom            1
US          United States of America  2
ZM          Zambia                    4
ZW          Zimbabwe                  4
14:42:21  SUCCESS  [SELECT - 25 rows, 0.004 secs]  Result set fetched
select * from hr.countries;
REGION_ID  REGION_NAME
---------  ----------------------
5          Australia
6          South America
1          Europe
2          Americas
3          Asia
4          Middle East and Africa
14:42:21  SUCCESS  [SELECT - 6 rows, 0.003 secs]  Result set fetched
select * from hr.regions;
14:42:21  END Execution 2 statement(s) executed, 31 row(s) affected, exec/fetch
time: 0.007/0.002 secs   [2 successful, 0 errors]
```

## Executing scripts

If you frequently want to execute a number of statements, it's best to put them into a script file. Here's how to execute a script that contains the two statements from the example above:

```
> dbviscmd.bat -connection "Oracle" -sqlfile "myscript.sql"

14:45:11  START Executing Command Line, Database Connection: Oracle Database Type:
ORACLE Catalog: null Schema: SYSTEM
14:45:11  INFO  Physical database connection acquired for: Oracle
COUNTRY_ID  COUNTRY_NAME              REGION_ID
----------  ------------------------  ---------
AR          Argentina                 2
AU          Australia                 3
BE          Belgium                   1
BR          Brazil                    2
CA          Canada                    2
CH          Switzerland               1
CN          China                     3
DE          Germany                   1
DK          Denmark                   1
EG          Egypt                     4
FR          France                    1
HK          HongKong                  3
IL          Israel                    4
IN          India                     3
IT          Italy                     1
JP          Japan                     3
KW          Kuwait                    4
MX          Mexico                    2
NG          Nigeria                   4
NL          Netherlands               1
SG          Singapore                 3
UK          United Kingdom            1
US          United States of America  2
ZM          Zambia                    4
ZW          Zimbabwe                  4
14:45:11  SUCCESS  [SELECT - 25 rows, 0.004 secs]  Result set fetched
select * from hr.countries;
REGION_ID  REGION_NAME
---------  ----------------------
5          Australia
6          South America
1          Europe
2          Americas
3          Asia
4          Middle East and Africa
14:45:11  SUCCESS  [SELECT - 6 rows, 0.003 secs]  Result set fetched
select * from hr.regions;
14:45:11  END Execution 2 statement(s) executed, 31 row(s) affected, exec/fetch
time: 0.007/0.002 secs   [2 successful, 0 errors]
```

## Controlling the output

You can use options to control how much output to generate. If you only want to see the results, use the `-output` option with the result keyword:

```
> dbviscmd.bat -connection "Oracle" -sqlfile "myscript.sql" -output result
COUNTRY_ID  COUNTRY_NAME              REGION_ID
----------  ------------------------  ---------
AR          Argentina                 2
AU          Australia                 3
BE          Belgium                   1
BR          Brazil                    2
CA          Canada                    2
CH          Switzerland               1
CN          China                     3
DE          Germany                   1
DK          Denmark                   1
EG          Egypt                     4
FR          France                    1
HK          HongKong                  3
IL          Israel                    4
IN          India                     3
IT          Italy                     1
JP          Japan                     3
KW          Kuwait                    4
MX          Mexico                    2
NG          Nigeria                   4
NL          Netherlands               1
SG          Singapore                 3
UK          United Kingdom            1
US          United States of America  2
ZM          Zambia                    4
ZW          Zimbabwe                  4
REGION_ID  REGION_NAME
---------  ----------------------
1          Europe
2          Americas
3          Asia
4          Middle East and Africa
```

For other scripts, for instance a script containing INSERT statements, you may only want to see the log messages:

```
> dbviscmd.bat -connection "Oracle" -sqlfile "myscript.sql" -output log
14:25:29  START Executing Command Line, Database Connection: Oracle Database Type:
ORACLE Catalog: null Schema: SYSTEM
14:25:30  INFO  Physical database connection acquired for: Oracle
14:25:30  SUCCESS  [SELECT - 25 rows, 0.012 secs]  Result set fetched
select * from hr.countries;
14:25:30  SUCCESS  [SELECT - 4 rows, 0.009 secs]  Result set fetched
select * from hr.regions;
14:25:30  END Execution 2 statement(s) executed, 29 row(s) affected, exec/fetch
time: 0.021/0.004 secs   [2 successful, 0 errors]
```

## Using variables - prompting for values

The DbVisualizer command line execution supports the DbVisualizer Variables as described in Using DbVisualizer Variables. To enable this you will need to use the option `-processvariables`.

```
> dbviscmd.bat -connection "Oracle" -sql "SELECT FIRST_NAME FROM HR.EMPLOYEES
where FIRST_NAME LIKE ${Name||A%||String||}$;" -processvariables
dbviscmd: Valid inputs: Enter '_B_' for empty, '(null)' for null
dbviscmd: Variable 'Name' (Literal) [A%]: B%
11:25:26  START Executing Command Line for: 'Oracle' [Oracle], Schema: SYSTEM
11:25:26  INFO  Physical database connection acquired for: Oracle
FIRST_NAME
----------
Bruce
Britney
11:25:26  SUCCESS  [SELECT - 2 rows, 0.047 secs]  Result set fetched
SELECT FIRST_NAME, LAST_NAME, PHONE_NUMBER FROM HR.EMPLOYEES where
EMPLOYEES.FIRST_NAME LIKE B%;
11:25:26  END Execution 1 statement(s) executed, 2 row(s) affected, exec/fetch
time: 0.047/0.000 secs   [1 successful, 0 errors]
```

## Combining OS scripts, the command line interface and DbVisualizer variables

For more complex tasks, you can call the command line interface from a shell script, for instance a Bourne shell script on Unix or a BAT file on Windows. You can also use DbVisualizer variables to pass information between the shell script and the SQL script. In this example, we have a simple SQL script (cmdtest.sql) that contains a SELECT statement with a variable in place for the table name:

**cmdtest.sql**

```
select * from ${table}$
```

A text file (tables.txt) contains the table names we want to execute the SQL script with:

**tables.txt**

```
hr.countries
hr.regions
```

In a command shell (Bourne or Bash), we can then execute the script using the table names from the text file:

```
for name in `cat tables.txt`;
  do ./dbviscmd.sh -connection "oracle" -sql "@run cmdtest.sql \${table||$name||||
nobind}\$; ";
done

15:01:19  START Executing Command Line, Database Connection: oracle Database Type:
ORACLE Catalog: null Schema: SYSTEM
15:01:20  INFO  Physical database connection acquired for: oracle
15:01:20  RUNNING  [@run ...ntries||||nobind}$ - - secs]
@run cmdtest.sql ${table||hr.countries||||nobind}$;
COUNTRY_ID  COUNTRY_NAME              REGION_ID
----------  ------------------------  ---------
AR          Argentina                 2
AU          Australia                 3
BE          Belgium                   1
BR          Brazil                    2
CA          Canada                    2
CH          Switzerland               1
CN          China                     3
DE          Germany                   1
DK          Denmark                   1
EG          Egypt                     4
FR          France                    1
HK          HongKong                  3
IL          Israel                    4
IN          India                     3
IT          Italy                     1
JP          Japan                     3
KW          Kuwait                    4
MX          Mexico                    2
NG          Nigeria                   4
NL          Netherlands               1
SG          Singapore                 3
UK          United Kingdom            1
US          United States of America  2
ZM          Zambia                    4
ZW          Zimbabwe                  4
15:01:20  SUCCESS  [SELECT - 25 rows, 0.016 secs]  Result set fetched
select * from hr.countries;
15:01:20  SUCCESS  [@run ...ntries||||nobind}$ - 0.016 secs]  Script processed
@run cmdtest.sql ${table||hr.countries||||nobind}$;
15:01:20  END Execution 1 statement(s) executed, 25 row(s) affected, exec/fetch
time: 0.016/0.012 secs   [1 successful, 0 errors]
java -cp /Users/ulf/work/github/dbvis/trunk/pureit/apps/dbvis/classes:/Users/ulf/
work/github/dbvis/trunk/pureit/apps/dbvis/resources:/Users/ulf/work/github/dbvis/
trunk/pureit/apps/dbvis/external/* -Xmx512M -Djava.awt.headless=true -
Ddbvis.home=/Users/ulf/work/github/dbvis/trunk/pureit/apps/dbvis
com.onseven.dbvis.DbVisualizerCmd -masterpw stairway -connection oracle -sql @run
cmdtest.sql ${table||hr.regions||||nobind}$;
15:01:23  START Executing Command Line, Database Connection: oracle Database Type:
ORACLE Catalog: null Schema: SYSTEM
15:01:23  INFO  Physical database connection acquired for: oracle
```

```
15:01:23  RUNNING  [@run ...egions||||nobind}$ - - secs]
@run cmdtest.sql ${table||hr.regions||||nobind}$;
REGION_ID  REGION_NAME
---------  ----------------------
5          Australia
6          South America
1          Europe
2          Americas
3          Asia
4          Middle East and Africa
15:01:23  SUCCESS  [SELECT - 6 rows, 0.001 secs]  Result set fetched
select * from hr.regions;
15:01:23  SUCCESS  [@run ...egions||||nobind}$ - 0.001 secs]  Script processed
@run cmdtest.sql ${table||hr.regions||||nobind}$;
15:01:23  END Execution 1 statement(s) executed, 6 row(s) affected, exec/fetch
time: 0.001/0.000 secs   [1 successful, 0 errors]
```

The command-line interface is called with the `-sql` option, specifying the client-side command @run. A DbVisualizer variable is passed to the `@run` command with the value taken from the shell variable. This DbVisualizer variable value is then available to the script executed by the `@run` command.

Note that you may need to escape certain characters that the shell would otherwise interpret, like the dollar signs that are part of the DbVisualizer variable delimiters.

# Setting up the connection properties on the command line

As an alternative to using a connection already set-up through the DbVisualizer tool you may use the `-url` parameter. In combination with the parameters `-drivername`, `-driverclass`, and `-driverpath` these parameters enables you connect without prior specification using the DbVisualizer tool.

Following are some examples.

Executing SQL towards a MySQL instance running on `localhost` port `3306`. The parameter `"-drivername MYSQL"` specifies that we are using a JDBC driver specified in the DbVisualizer tool named `MYSQL`. For listing of the existing drivers use the **Tools → Driver Manager** in the DbVisualizer tool. Read more in Installing a JDBC Driver.

Note that if a user driver is specified using Maven or Remote artifacts, and the driver files has not yet been downloaded, the download will be performed when running dbviscmd.

**Using -url and -drivername parameters**

```
./dbviscmd.sh -url jdbc:mysql://localhost:3306/
   -drivername MYSQL
   -sql "select * from sakila.actor"
   -userid root
```

An alternative to use the `-drivername` you may use the parameters `-driverclass` and `-driverpath` to specify the JDBC driver.

**Using -driverclass and -driverpath parameters**

```
./dbviscmd.sh -url jdbc:oracle:thin:@localhost:1521/ORCL
   -driverclass oracle.jdbc.OracleDriver
   -driverpath "ojdbc6.jar:orai18n.jar:xdb.jar:xmlparserv2.jar"
   -sql "select * from HR.COUNTRIES"
   -userid system
   -password oracle
```

The above example connects to an Oracle instance on `localhost` and port `1521`. Note that the separator character **':'** between the different jar files is platform dependant. On Windows-based desktop platforms, the value of this field is the semicolon **";"**.

# Exit codes from dbviscmd

These are the exit codes when running dbviscmd

| Code | Meaning |
|---|---|
| 0 | OK |
| 1 | Other error |
| 2 | Connect error |
| 3 | Script execution resulted in an error. Execution stopped |
| 4 | Script execution resulted in errors |
| 5 | Script execution failed unexpectedly |

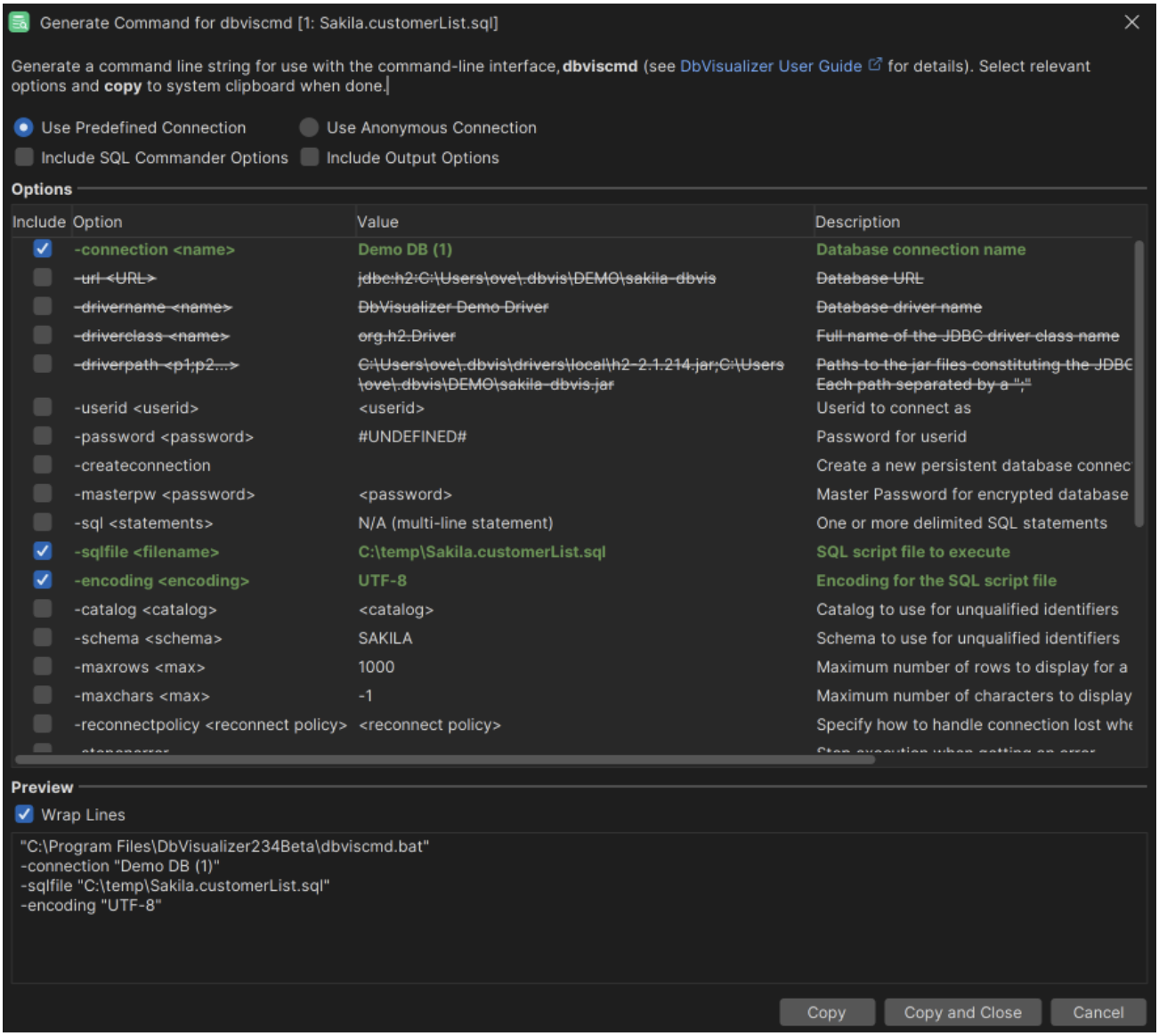
# Generating a Command From SQL Commander

From time to time, you may want to use some SQL that you have tested in the GUI and run it in the command line interface. You can generate the string with all parameters and paste that string on the command line or a script. Use menu option **SQL Commander → Generate Command for dbviscmd** to open a dialog where you can specify the options. Some options are selected and some are disabled depending on the editor contents.

If you generate a command from an anonymous script (a script that has not been saved to a file) with more than one line, you will be prompted to save the file or generate a temporary one. If you generate the command from a file that is modified, you will be prompted to save the file before generating the command.

In addition to the Command Line Options, there are a few settings in the dialog:

- You can use a predefined (named) or an anonymous connection; this will enable and disable connection options accordingly.
- You can choose whether or not to **Include SQL Commander Options**; this is a convenient way to enable or disable all corresponding options. You can still enable or disable individual options as desired.
- You can choose whether or not to **Include Output Options**; this is a convenient way to enable or disable all corresponding options. You can still enable or disable individual options as desired.



# Telemetry

DbVisualizer can automatically collect diagnostics and anonymous usage data, which will help us understand what features are most valuable and where we can improve.

Telemetry settings can be changed at any time using the **Telemetry** tab in **Tools → Tool Properties**.

- When **Collect Anonymous Usage Statistics** is enabled, DbVisualizer will collect information about what features are used. This will give us important insights into how the different features are used so we can make them even more useful for you. Examples of the kind of information that is included in usage statistics:
- Application startup time.
- Query builder has been used.
- "Export Schema" action has been used.
- A database connection to a H2 database has been established.
- When **Report Application Errors** in enabled, unexpected errors in the application will be collected. This will help us detect issues in the application and correct them faster. Examples of application errors that can be reported:
- NullPointerException when opening a dialog.
- ClassCastException when loading a database driver.
- IndexOutOfBoundsException when reordering table columns.

No restart is necessary when changing these settings.

> Your privacy is important to us. We won't track queries, table data, or anything that identifies you.

# Database Profiles

## Database Profiles

A Database Profile is the foundation for database specific support in DbVisualizer. Technically the database profile is a single XML file declaring what object types, actions and viewers/editors should be available in the DbVisualizer user interface for each specific database.


- Creating a Database Profile
- Extending a Database Profile
- Icons
- Conditional Processing
- Database Profile Utilities
- Database Profile Changes in 23.3
- Database Profile Changes in 14.0
- Database Profile changes in 13.0
- Database Profile changes in 11.0
- Database Profile changes in 9.5

# Understanding Database Profiles

## Understanding Database Profiles

A database profile is, somewhat simplified, a definition of the kind of information that is presented in the database objects tree and in the various object views for a specific database engine. In addition, the profile defines the actions for the object types defined in the profile. DbVisualizer loads the matching database profile when you connect to a database. If no matching profile is found, DbVisualizer uses a **Generic** profile with just the general database information and actions included.

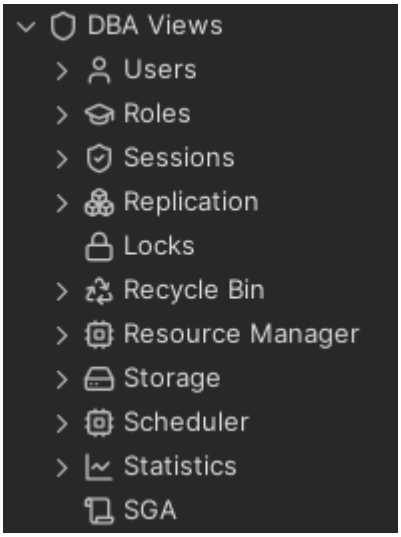
DbVisualizer currently offer database specific support (database profiles) for the following databases (click links for details):

- Azure SQL Database
- Azure Synapse Dedicated
- Azure Synapse Serverless
- Cassandra
- ClickHouse
- Databricks
- Db2 LUW
- Elasticsearch
- Exasol
- Google BigQuery
- Greenplum
- H2
- Informix
- JavaDB/Derby
- MariaDB
- Microsoft Access
- Mimer SQL
- MongoDB
- MySQL
- Netezza
- NuoDB
- Oracle
- PostgreSQL
- Presto
- Redshift
- SingleStore
- Snowflake
- SQL Server
- SQLite
- Sybase ASE
- Trino
- Vertica
- Yellowbrick

The specialized database profiles define different object types, so the database objects tree may look different depending on which database you are connected to. The structure and organization of a database profile is also something that may impact the layout of the objects tree, even though the provided ones are similar in their structure. There are two root nodes in the majority of the provided profiles:

- **Schema Objects**
- **DBA objects**

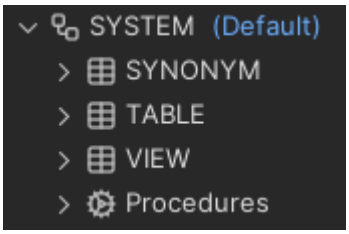
Schema objects are, for example, **tables**, **views**, **triggers**, and **functions**, while DBA objects most often are objects that require administration privileges in the database in order to access them. The convention in DbVisualizer is to put all DBA objects under the **DBA Views** tree node. If you connect to a database using an account with insufficient privileges to access a DBA object, you may see error messages if you try to select nodes under the DBA Views node. The following is an example of the DBA sub tree for Oracle:




For databases that have no specific profile, DbVisualizer uses the **Generic** profile. DbVisualizer supports a wide range of databases. The nature of the databases and what they support differ from vendor to vendor, so the appearance and structure of the tree below the database connection objects for different databases differ as well. The generic database profile displays objects based on what JDBC offers in terms of database information (aka metadata information). DbVisualizer asks the JDBC driver for all schemas, databases, tables and procedures, and then builds the tree based on what the driver returns.

The advantage of using JDBC to get database metadata is that it is a standard way to access the information, independent of the database engine type; the JDBC driver layer hides the proprietary details about where and how the information is really stored. The drawback with using JDBC is that JDBC doesn't offer access to all metadata a database may hold. While the information presented by the generic profile, with its reliance on JDBC, is sufficient for many tasks, a database specific profile offers far more details as well as more features.

The generic database profile when used for an Oracle connection look as follows:




The appearance of the generic database profile may include schema objects and/or catalog objects depending on whether the database supports these objects. The Procedures object always appear in the tree, regardless if the database connection supports procedures or not. There is no DBA Views node in the generic profile.

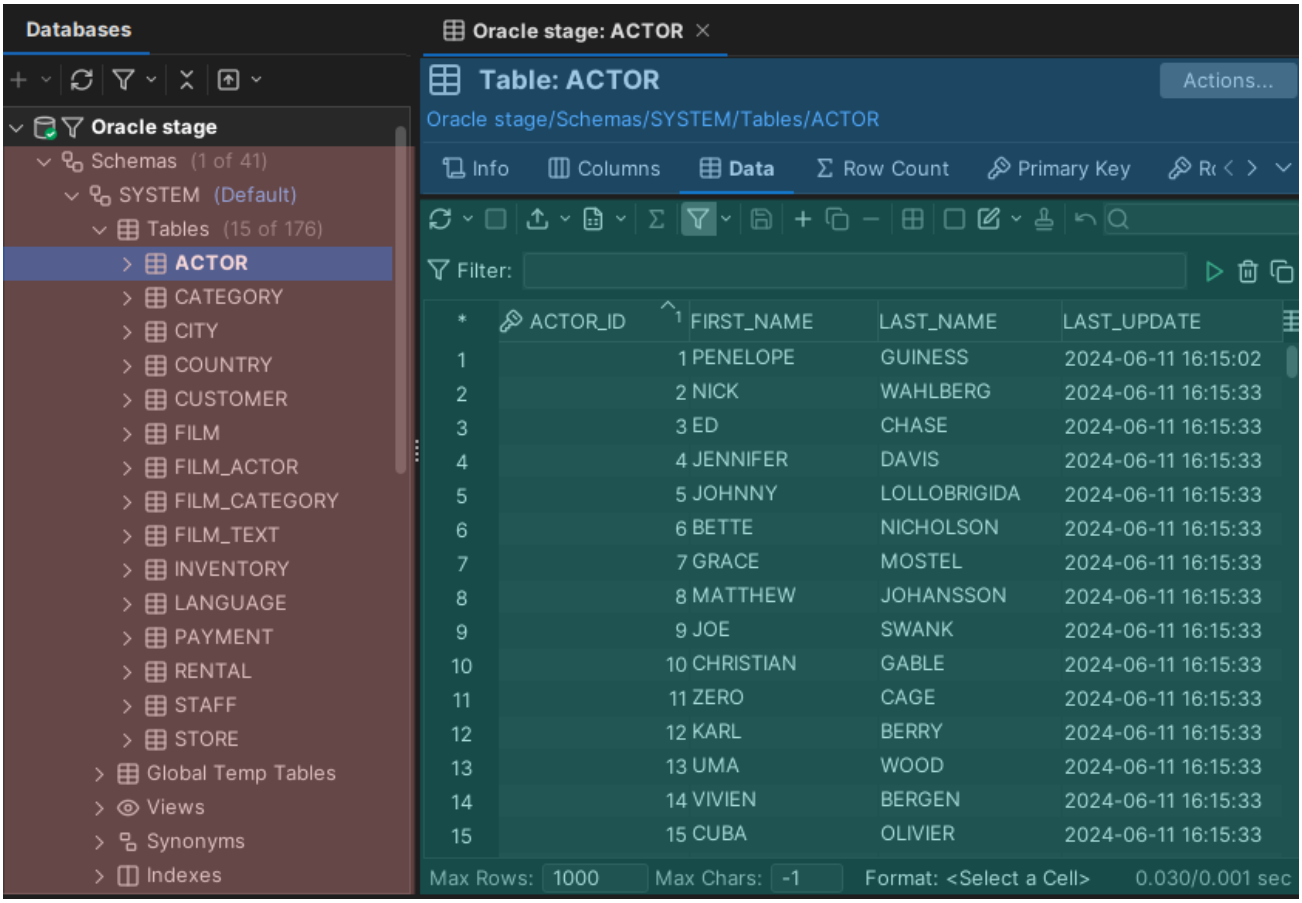
## Affected DbVisualizer features

One of the most important and central features in DbVisualizer is the **database objects tree**, used to navigate databases, and the **object view**, showing details about specific objects. The general problem exploring any database is that they are all different with respect to the information describing what's in the database (also called system tables or database meta data). This basically means that it's rather complex to implement a multi-database support product, such as DbVisualizer, since each database must be handled specifically. All databases also support different object types, apart from the most common ones, such as table, view, index, etc.

The database profile framework is used to simplify the process of defining what information DbVisualizer will display and operate on for a specific database. Technically, a database profile is an **XML** file with all of the logic, structure and actions mapped to the visual components in DbVisualizer. Another great benefit of separating the database specific logic from the implementation of DbVisualizer is that anyone with some degree of domain knowledge can create a database profile. All that is needed is a text editor (preferably with XML support) and some ideas of what should be the final result.

A great source for inspiration (except for related sections in the user guide) is all the existing database profiles that comes with DbVisualizer. All database profiles that comes with DbVisualizer are stored in the `DBVIS-HOME/resources/profiles` directory (exact path is OS dependent).

The following figure illustrates which features in DbVisualizer are controlled by the database profile.



The **red box** at the left shows the **database objects tree**. This tree is used to navigate the objects in the database. Selecting an object in the tree shows the **object view** (**blue box**) for the selected object type. An object view may have several **data views** (**green**), showing object information. DbVisualizer shows these as labeled tabs. The green box in the screenshot shows the content of the data view labeled Columns. The type of viewer that is presenting the data in the screenshot is the grid viewer. Read more about all data viewers in the Viewers section.

Common to both the database objects tree and the object view are the SQL **commands** that are used to fetch the information from the database. The associated SQL is executed by DbVisualizer whenever a node in the tree is expanded (to expose any child objects) or when a node is selected (to fill the object data views).

Right-clicking the mouse on an object in the tree or clicking the **Actions** button in the object view shows a menu with all valid **actions** for the selected object. These are also defined per database profile and object type. Read more about the capabilities of actions in the definition of user actions section.

The mapping from the visual components in the user interface described earlier and the element definitions in the XML file is, briefly, as follows:

- The database objects tree (green box) is described by the ObjectsTreeDef root element,
- The object views (green and blue boxes) are described by the ObjectsViewDef root element, The commands used to execute the SQL to get the information for ObjectsTreeDef, ObjectsViewDef and ObjectsActionDef definitions are defined by the Commands root element,
- All Actions for an object are defined by the ObjectsActionDef root element.

## How a Database Profile is loaded

DbVisualizer automatically detect what database profile to use based on the **Database Type** setting for a database connection. A database profile can also be manually specified in the connection properties.

A database profile is located using the search paths defined in **Connection Properties** tab and the **Database Profile** category. The standard directories in the search path are:

1. `WORKSPACE\ext\profile`
2. `DBVIS-HOME\resources\profiles`

**WORKSPACE** is the workspace directory keeping all the user settings for DbVisualizer. By default, it's located in the `.dbvis` directory in the user's home directory.

The list of paths may be reorganized and directories can be added. Profile files are searched in the specified order.

If the actual database profile is found in the search path it is loaded and any parent profile(s) it extends are also loaded and finally merged.

If there is no matching profile then the **generic** profile is automatically used. This is very basic profile and shows only basic information about the objects in the database and should support most databases with a JDBC driver.

A database profile other than the generic is built for a specific database. Manually selecting for example a database profile for Oracle while connecting to Db2 will result in all sorts of errors.

# Creating a Database Profile

At a first glance creating and developing a database profile in XML may seem difficult. However, all definitions forming the functionality for a specific database are expressed in a single file and the XML elements are well formed. The general recommendation is to use one of the existing database profiles as base (copy/paste) and then step-by-step modify it for the actual database. You may use an external text editor (preferably with XML support) or the SQL editor in DbVisualizer to edit the file. Once new changes has been made, save the file, reconnect the database connection in DbVisualizer to test the layout and functionality changes.

Creating a new database profile should only be made for databases with no database profile available. If you are looking into changing one of the existing database profiles by adding or modifying existing functionality, it should be extended.

> For information where custom database profiles should be saved check the How a Database Profile is loaded is section.

For more information how to **create a new database profile** or **extending an existing database profile** check the Extending a Database Profile section.

# Extending a Database Profile

All database profiles must extend the **generic** database profile. The generic profile handles the very basic object types in a relational database such as **Tables**, **Columns**, **Indexes** and **Procedures**. Its implementation is based entirely on what the **JDBC** driver provide in terms of database meta data. Due to the tight connection between the generic profile and the JDBC driver, the generic profile can be used to access almost any database with a JDBC driver.

The selection on what database profile should be used is determined with the **Database Type** setting for the database connection. Some of the database types that can be picked have a dedicated database profile with extended support while others have not. For databases with no database profile available, the generic one is used. It is also possible to manually choose the generic profile in the **Connection Properties / Database Profiles** settings.

The most important area in the database profile as seen from the DbVisualizer user interface is the section describing the **database objects tree**. This is the browser or navigator showing database objects. This is also the place that connects **actions** used to operate on database objects and **views** (not database views) used to display detailed information.

This section of the user guide is mainly focused on extending an existing database profile (not the generic profile) rather than creating a completely new profile (which should extend the generic database profile).

> Extending a database profile is not only about adding functionality to an existing profile but also the process of changing and removing existing definitions in any of the profiles that are extended.

## Extending Commands

Extending the **Commands** and **InitCommands** elements is simple as every command should be uniquely identified. To add a **Command** just insert the new command.

```
<Commands extends="true">
      <Command id="sample.getLoginSchema">
         <SQL>
            <![CDATA[
select '${schema}' as schema from dual
            ]]>
         </SQL>
      </Command>
   </Commands>
```

To override the definition of an existing command in the parent profile, just make sure the **id** of the new command match the id of the parent profile command. It will then be replaced.

## Extending Database Objects Tree

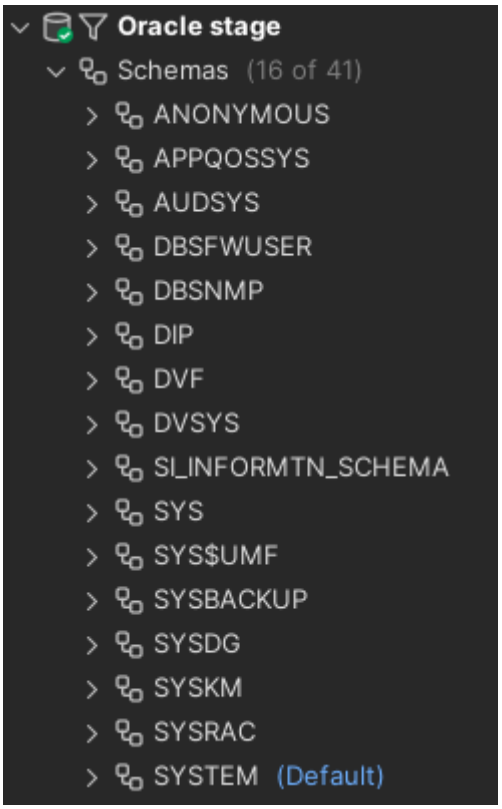
Extending or modifying the database objects tree (ObjectsTreeDef) require some attention since the modifications must match the exact object paths as defined in the parent profile. The object path is determined by the **GroupNode** and **DataNode** structure in the ObjectsTreeDef with the addition of the **type** attribute. The following is an example of the object path to the **Columns** sub node for a **Table** node:

```
GroupNode[@type='Schemas']/DataNode[@type='Schema']/GroupNode[@type='Tables']/
DataNode[@type='Table']/GroupNode[@type='Columns']
```

(The analyze database profile utility will report object paths in the above xpath format).

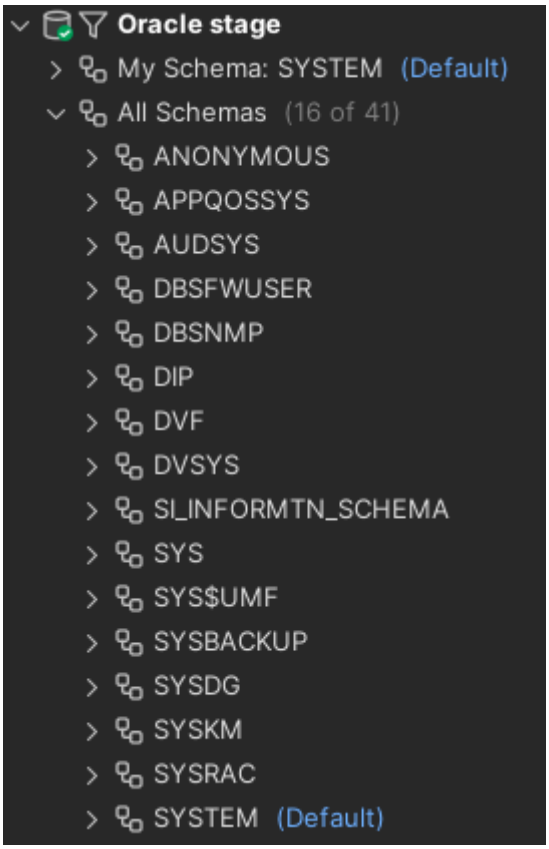
> The hierarchy of GroupNode and DataNode is important when extending the database objects tree since the exact same hierarchy must be implemented in the extended profile. This also involve any conditional elements such as If/Else/ElseIf that are used in the parent profile.

Consider the following example showing the objects tree (for Oracle) with the **Schemas** node being expanded to show all schemas in the database:




Instead of showing all schema objects in the database we want to adjust so that only the default schema is displayed at the top level (below the Oracle database connection node). The default schema node should in addition

only show **table** objects rather than all 20 (or so) object types being displayed in the standard Oracle database profile.




The previous screenshot shows the new **My Schema: SYSTEM** node at the top while the **Schemas** node has been renamed **All Schemas**. To accomplish the above a custom database profile has been created in the $

{dbvis.prefsdir}/ext/profiles/sample-ext-oracle.xml file with the following content required for the **ObjectsTreeDef** definition:

```
<!--Commands used in this profile-->
<Commands extends="true">
   <Command id="sample.getLoginSchema">
      <SQL>
         <![CDATA[
select '${schema}' as schema from dual
         ]]>
      </SQL>
   </Command>
</Commands>

<ObjectsTreeDef extends="true">
   <!--The following "Schema" definition shows the login schema directly below-->
   <!--the Database Connection for faster access. It is limited to only show-->
   <!--tables (by setting the "Table" DataNode to isLeaf="true")-->
   <DataNode type="Schema" label="My Schema: ${sample.getLoginSchema.SCHEMA}"
order-before="0">
      <SetVar name="schema" value="${sample.getLoginSchema.SCHEMA}"/>
      <Command idref="sample.getLoginSchema">
         <Input name="schema" value="${#db.loginSchema}"/>
      </Command>

      <DataNode type="Table" label="${oracle.getTables.TABLE_NAME}" isLeaf="true">
         <SetVar name="objectname" value="${oracle.getTables.TABLE_NAME}"/>
         <SetVar name="rowcount" value="true"/>
         <SetVar name="acceptInQB" value="true"/>
         <Command idref="oracle.getTables">
            <Input name="owner" value="${schema}"/>
            <Input name="temporary" value="N"/>
            <ProcessDataSet action="sortcolumn" index="getTables.TABLE_NAME"/>
            <Filter index="getTables.TABLE_NAME" label="Table"/>
         </Command>

         <!--These are needed for the viewers defined in the parent profile-->
         <!--associated with the "Table" type-->
         <SetVar name="theTableName" value="${objectname}"/>
         <SetVar name="theParentName" value="${objectname}"/>
         <SetVar name="triggersCondition" value="and table_name = '$
{theTableName}'"/>

      </DataNode>
   </DataNode>

   <!--Renaming the standard Schemas node to "All Schemas"-->
   <GroupNode type="Schemas" label="All Schemas"/>
</ObjectsTreeDef>
```

In the **Commands** section there is a new **Command** that run a dummy SQL SELECT only to create a result set containing a single row/column with the value of the **${schema}** variable. The value for the **${schema}** variable is

provided in the Command element for **DataNode type="Schema"** using the **${#db.loginSchema}** variable value as input. This variable is maintained by DbVisualizer and contain the login schema as specified in the connection setup. For Oracle this is the **userid**.

```
<Command idref="sample.getLoginSchema">
   <Input name="schema" value="${#db.loginSchema}"/>
</Command>
```

The command above is used to present the default schema as in the following **DataNode** declaration.

```
<DataNode type="Schema" label="My Schema: ${sample.getLoginSchema.SCHEMA}" order-
before="0">
   <SetVar name="schema" value="${sample.getLoginSchema.SCHEMA}"/>
   <Command idref="sample.getLoginSchema">
      <Input name="schema" value="${#db.loginSchema}"/>
   </Command>

   <DataNode type="Table">
      ...
   </DataNode>
</DataNode>
```

The label for this Schema type is **label="My Schema: ${sample.getLoginSchema.SCHEMA}"**. The **${sample.getLoginSchema.SCHEMA}** variable name consists of two parts, the name of the command: **sample.getLoginSchema** and the column name: **SCHEMA** in the result set the produced by the command.

As a sub node to the **My Schema** node there is the **DataNode type="Table"** definition for the Table object type. The complete declaration for the Table element and its sub elements has been copied from the parent profile:

```
<DataNode type="Table" label="${oracle.getTables.TABLE_NAME}" isLeaf="true">
   <SetVar name="objectname" value="${oracle.getTables.TABLE_NAME}"/>
   <SetVar name="rowcount" value="true"/>
   <SetVar name="acceptInQB" value="true"/>
   <Command idref="oracle.getTables">
      <Input name="owner" value="${schema}"/>
      <Input name="temporary" value="N"/>
      <ProcessDataSet action="sortcolumn" index="TABLE_NAME"/>
      <Filter index="TABLE_NAME" label="Table"/>
   </Command>

   <!--These are needed for the viewers defined in the parent profile-->
   <!--associated with the "Table" type-->
   <SetVar name="theTableName" value="${objectname}"/>
   <SetVar name="theParentName" value="${objectname}"/>
   <SetVar name="triggersCondition" value="and table_name = '${theTableName}'"/>
</DataNode>
```

The **Table** objects in the extended profile should not show any sub nodes such as columns or triggers and these declarations are then removed in the copied DataNode for the Table object. The **isLeaf="true"** attribute specifies that there will be no child nodes.

The new **All Schemas** node in the extended profile is supposed to have the exact same content and definition as in the parent profile when it was called **Schemas**. The following definition will just rename the label of the existing node.

```
<GroupNode type="Schemas" label="All Schemas"/>
```

This example show adding a new **Role** node under all **DBA / Users / User** objects.

In the parent Oracle profile there are no child nodes below **User**. To handle this all nodes from **DBA** down to **User** must be specified (aka the object type path). The only requirement is that the type attribute is specified and that it match the type in the parent profile. In addition, this example specify the **label** attribute for some of the nodes just to show that **overridden attributes** will replace any parent equvivalent node attributes.

> Only attributes for **GroupNode** and **DataNode** can be overridden. If you need to override for example a SetVar in a DataNode then all of the attributes in the DataNode and all its sub elements must be specified.

```
<GroupNode type="DBA" label="Database Administration">
   <GroupNode type="Users">
      <!--The "User" type don't allow child nodes in the parent profile.-->
      <!--Setting isLeaf="false" is needed to override this and allow the-->
      <!--new "Role" child node-->
      <DataNode type="User" isLeaf="false">
         <!--Here comes the new child "Role" DataNode-->
         <DataNode type="Role" isLeaf="true" label="$
{oracle.getGranteeRoles.GRANTED_ROLE} - ext">
            <SetVar name="objectname" value="$
{oracle.getGranteeRoles.GRANTED_ROLE}"/>
            <Command idref="oracle.getGranteeRoles">
               <Input name="grantee" value="${objectname}"/>
            </Command>
         </DataNode>
      </DataNode>
   </GroupNode>
</GroupNode>
```

The following are specified only to redefine the position of the **Locks** and **Sessions** nodes. One of **order-before** and **order-after** attributes are used to either identify a type for which the node should be positioned before or after, or an index. The index is the fixed position or 0 which means first or a somewhat high number means last. The following will move the **Sessions** first among the **DBA** child nodes.

```
<GroupNode type="Sessions" order-before="0"/>
<!--The following will move the "Locks" node before "Sessions"-->
<GroupNode type="Locks" order-before="Sessions"/>
```

## Extending Actions

Extending ActionGroup and Action elements follow the same rules as for extending ObjectsTreeDef section. The following example show removing the **oracle-schema-stringsearch** action for the **Schema** object type in the parent profile. A new **ActionGroup**: **Extended Schema Actions** is added with a single new **Action**: **sample-schema-sample-action**.

```
<ObjectsActionDef extends="true">
   <ActionGroup type="Schema">
      <!--Remove action from parent profile for "Schema"-->
      <Action id="oracle-schema-stringsearch" action="drop"/>
      <!--Adds an "Extended Schema Actions" sub menu in the "Schema" actions
menu-->
      <ActionGroup type="sample-schema-test" label="Extended Schema Actions">
         <!--Sample action that does nothing-->
         <Action id="sample-schema-sample-action" label="Sample Action"
                 reload="true" resetcatalogs="true" icon="remove">
            <Input label="Text Field" name="textField" style="text"
editable="true"/>
            <Command>
               <SQL><![CDATA[Sample Action "${textField}"]]></SQL>
            </Command>
            <Confirm>
               Really run Sample Action "${textField}"?
            </Confirm>
            <Result>
               Sample Action "${textField}" processed!
            </Result>
         </Action>
      </ActionGroup>
   </ActionGroup>
</ObjectsActionDef
```

## Extending Object Views

Extending **<ObjectView> and <DataView>** elements follow the same rules as previously explained.

```
<ObjectsViewDef extends="true">
   <!--Schema is dropped in the Oracle profile. Redefine it here and show the-->
   <!--dictionary views. That data is really not associated with the single
schema-->
   <!--defined in this profile but is a way to have it quickly accessed from-->
   <!--a single node.-->
   <ObjectView type="Schema">
      <DataView id="sample-schema-dict" label="Dictionary"
                icon="sample-schema-dict" viewer="grid">
         <Command idref="sample.getDict"/>
         <Message>
            <![CDATA[
<html>
Simple viewer showing all dictionary tables with description. Easily accessed
by opening the Schema viewer since that is empty anyway in the
parent <b>oracle</b> profile.
</html>
            ]]>
         </Message>
      </DataView>
   </ObjectView>
</ObjectsViewDef>
```

## Remove an Element

Removing an object in the parent profile is easy, just add the **action="drop"** attribute to any of **GroupNode**, **DataNode**, **ObjectView**, **DataView**, **ActionGroup** and **Action** elements. If there are any sub elements for the object being dropped, these are also removed.

## Complete sample Database Profile

This document describe the different parts of a extended sample database profile for an Oracle database. Here follow the complete sample database profile. If you would like to test it with your Oracle database, then just copy and paste it to your $HOME\.dbvis\ext\profiles folder as sample-oracle.xml. In DbVisualizer open **Connection**

**Properties** for your Oracle connection and click **Database Profiles** category. In the list of profiles choose **sample-oracle**, click **Apply** and then connect.

```
<?xml version="1.0" encoding="UTF-8" ?>


<!--
   Copyright (c) DbVis Software AB. All Rights Reserved.
-->

<DatabaseProfile
        desc="Sample profile extending the Oracle profile"
        extends="oracle"
        minver="9.5">

   <!--Commands used in this profile-->
   <Commands extends="true">
      <Command id="sample.getLoginSchema">
         <SQL>
            <![CDATA[
select '${schema}' as schema from dual
            ]]>
         </SQL>
      </Command>

      <Command id="sample.getDict">
         <SQL>
            <![CDATA[
select * from dict order by table_name
            ]]>
         </SQL>
      </Command>
   </Commands>

   <ObjectsActionDef extends="true">
      <ActionGroup type="Schema">
         <!--Remove action from parent profile for "Schema"-->
         <Action id="oracle-schema-stringsearch" action="drop"/>

         <!--Adds an "Extended Schema Actions" sub menu in the "Schema" actions
menu-->
         <ActionGroup type="sample-schema-test" label="Extended Schema Actions">
            <!--Sample action that does nothing-->
            <Action id="sample-schema-sample-action" label="Sample Action"
reload="true"
                    resetcatalogs="true" icon="remove">
               <Input label="Text Field" name="textField" style="text"
editable="true"/>
               <Command>
                  <SQL><![CDATA[Sample Action "${textField}"]]></SQL>
               </Command>
               <Confirm>
                  Really run Sample Action "${textField}"?
               </Confirm>
               <Result>
```

```
                  Sample Action "${textField}" processed!
               </Result>
            </Action>
         </ActionGroup>
      </ActionGroup>
   </ObjectsActionDef>

   <ObjectsTreeDef extends="true">
      <!--The following "Schema" definition shows the login schema directly
below-->
      <!--the Database Connection for faster access. It is limited to only show-->
      <!--tables (by setting the "Table" DataNode to isLeaf="true")-->
      <DataNode type="Schema" label="My Schema: ${sample.getLoginSchema.SCHEMA}"
order-before="0">
         <SetVar name="schema" value="${sample.getLoginSchema.SCHEMA}"/>
         <Command idref="sample.getLoginSchema">
            <Input name="schema" value="${#db.loginSchema}"/>
         </Command>

         <DataNode type="Table" label="${oracle.getTables.TABLE_NAME}"
isLeaf="true">
            <SetVar name="objectname" value="${oracle.getTables.TABLE_NAME}"/>
            <SetVar name="rowcount" value="true"/>
            <SetVar name="acceptInQB" value="true"/>
            <Command idref="oracle.getTables">
               <Input name="owner" value="${schema}"/>
               <Input name="temporary" value="N"/>
               <ProcessDataSet action="sortcolumn" index="TABLE_NAME"/>
               <Filter index="TABLE_NAME" label="Table"/>
            </Command>

            <!--These are needed for the viewers defined in the parent profile-->
            <!--associated with the "Table" type-->
            <SetVar name="theTableName" value="${objectname}"/>
            <SetVar name="theParentName" value="${objectname}"/>
            <SetVar name="triggersCondition" value="and table_name = '$
{theTableName}'"/>

         </DataNode>
      </DataNode>

      <!--Renaming the standard Schemas node to "All Schemas"-->
      <GroupNode type="Schemas" label="All Schemas"/>

      <!--The main purpose with the following is to add a "Role" child DataNode --
>
      <!--for each "User". In the parent Oracle profile there are no child-->
      <!--nodes below "User". To handle this all nodes from "DBA" down to "User"--
>
      <!--must be specified. The only requirement is that the type attribute is-->
      <!--specified and that it match the type in the parent profile.-->

      <!--In addition, this example specify the label attribute for some of the-->
```

```
      <!--nodes just to show that overridden attributes will replace any parent-->
      <!--equvivalent node attributes.-->
      <GroupNode type="DBA" label="Database Administration">
         <GroupNode type="Users">
            <!--The "User" type don't allow child nodes in the parent profile.-->
            <!--Setting isLeaf="false" is needed to override this and allow the-->
            <!--new "Role" child node-->
            <DataNode type="User" isLeaf="false">
               <!--Here comes the new child "Role" DataNode-->
               <DataNode type="Role"
label="${oracle.getGranteeRoles.GRANTED_ROLE} - ext" isLeaf="true">
                  <SetVar name="objectname" value="$
{oracle.getGranteeRoles.GRANTED_ROLE}"/>
                  <Command idref="oracle.getGranteeRoles">
                     <Input name="grantee" value="${objectname}"/>
                  </Command>
               </DataNode>
            </DataNode>
         </GroupNode>

         <!--The following are specified only to re-define the position of the-->

<!--"Locks" and "Sessions" nodes. One of "order-before" and "order-after" -->
         <!--attributes are used to either identify a type for which the node
should-->
         <!--be positioned before or after, or an index. The index is the fixed
position-->
         <!--or 0 which means first or a somewhat high number means last.-->

         <!--The following will move the "Sessions" first among the "DBA"-->
         <!--child nodes-->
         <GroupNode type="Sessions" order-before="0"/>

         <!--The following will move the "Locks" node before "Sessions"-->
         <GroupNode type="Locks" order-before="Sessions"/>
      </GroupNode>
   </ObjectsTreeDef>

   <ObjectsViewDef extends="true">
      <!--Schema is dropped in parent profile. Re-define it here and show the-->
      <!--dictionary views. That data is really not associated with the single
schema-->
      <!--defined in this profile but is a way to have it quickly accessed from-->
      <!--a single node.-->
      <ObjectView type="Schema">
         <DataView id="sample-schema-dict" label="Dictionary" icon="sample-schema-
dict" viewer="grid">
            <Command idref="sample.getDict"/>
            <Message>
                  <![CDATA[
<html>
Simple viewer showing all dictionary tables with description. Easily accessed
by opening the Schema viewer since that is empty anyway in the
```

```
parent <b>oracle</b> profile.
</html>
                ]]>
             </Message>
          </DataView>
       </ObjectView>
    </ObjectsViewDef>

</DatabaseProfile>
```

# Top level XML Elements

## Top level XML Elements

The top level XML elements in a database profile is as follows:

- InitCommands (optional) Defines SQLs that are executed before the profile is being loaded,
- Commands Defines the SQLs for the ObjectsTreeDef, ObjectsViewDef and ObjectsActionDef,
- ObjectsActionDef (optional) Defines actions for object types,
- ObjectsTreeDef Defines the structure and what objects should be visible in the objects tree,
- ObjectsViewDef Defines the object views for a specific object type.

All database connections loads a database profile from an XML file and if there is no matching database profile, the **generic** profile is used. This profile use standard JDBC metadata requests to obtain information about the (some) objects in the database. The generic profile is located in `DBVIS-HOME\resources\profiles\generic.xml`.

## XML template

The following show an overview of a database profile with the top level XML elements.

```
<?xml version="1.0" encoding="UTF-8"?>


<DatabaseProfile desc="Profile for Sybase ASE"
                 version="$Revision: 20369 $"
                 date="$Date: 2016-05-24 17:29:25 +0200 (Tue, 24 May 2016) $"
                 minver="9.5"
                 extends="generic">

  <InitCommands extends="true">
    ...
  </InitCommands>

  <Commands extends="true">
    ...
  </Commands>

  <ObjectsActionDef extends="true">
    ...
  </ObjectsActionDef>

  <ObjectsTreeDef extends="false">
    ...
  </ObjectsTreeDef>

  <ObjectsViewDef extends="true">
    ...
  </ObjectsViewDef>

</DatabaseProfile>
```

The DOCTYPE identifier at row 2 defines the DTD that is used to validate the XML.

The root element for the database profile framework is the DatabaseProfile element setting various attributes. Continue to the next sections for information about XML elements forming a database profile.

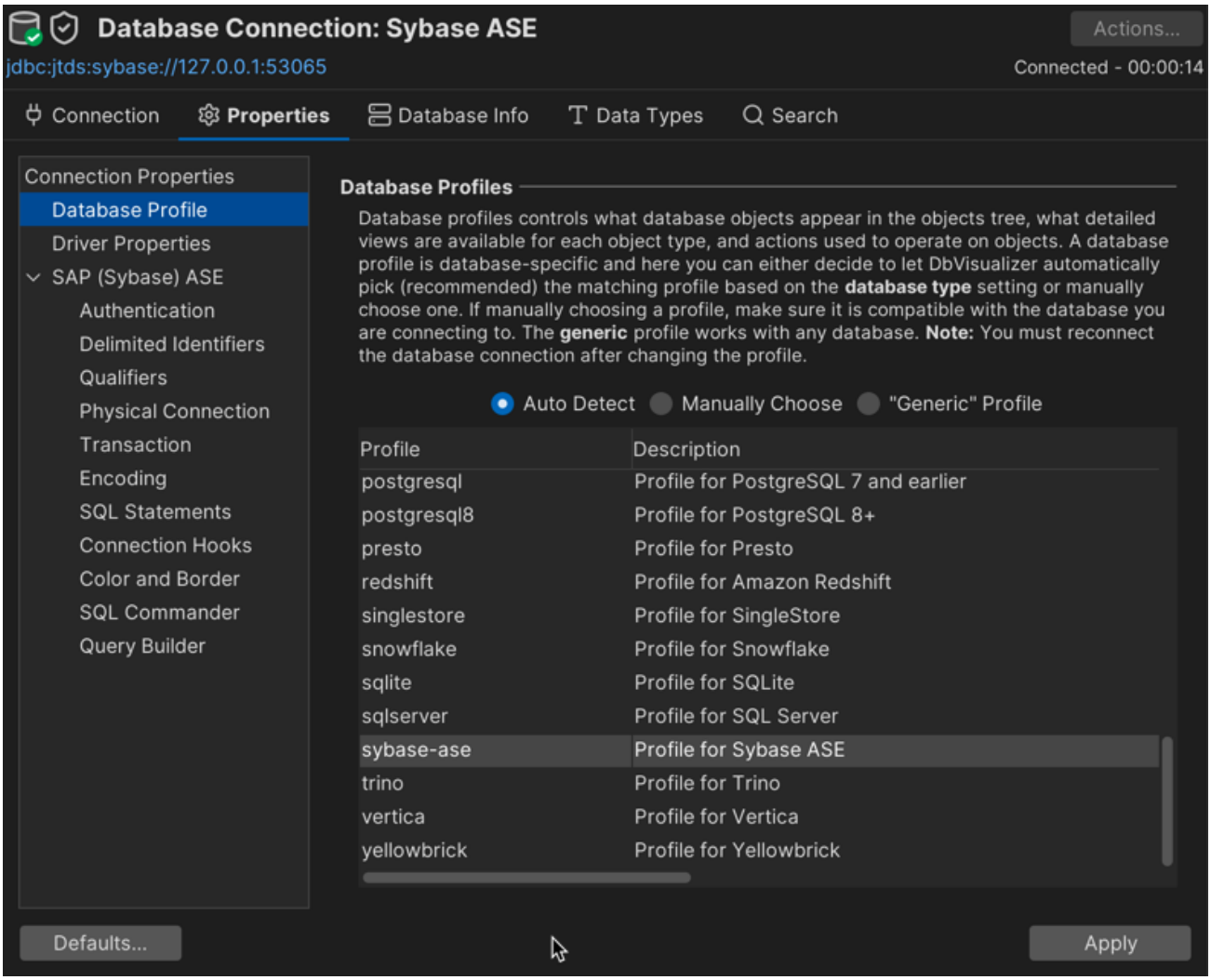
## XML element - DatabaseProfile

The DatabaseProfile is the root element in the XML file. It is required and have the following attributes:

```
<DatabaseProfile desc="Profile for Sybase ASE"
                 minver="9.5"
                 extends="generic">
  ...
</DatabaseProfile>
```

| Attribute | Description |
|---|---|
| desc | The description of the profile |
| minver | The minimum version of DbVisualizer that is required for the profile to work |
| abstract | Used in a sub profile to indicate that the profile should not be selectable in DbVisualizer. An example is the **mysql-base** which can only be selected using the extended profiles for MySQL and MariaDB. The default is false |
| **extends** | The parent profile that is extended. In most situations at least **generic** should be specified |

Most attributes except **extends** are informative and are displayed in the **Database Profile** list when selecting the connection properties for a database connection:


Database Connection: Sybase ASE
Actions...
jdbc:jtds:sybase://127.0.0.1:53065
Connected - 00:00:14
Connection
Properties
Database Info
Data Types
Search
Connection Properties
Database Profile
Driver Properties
SAP (Sybase) ASE
Authentication
Delimited Identifiers
Qualifiers
Physical Connection
Transaction
Encoding
SQL Statements
Connection Hooks
Color and Border
SQL Commander
Query Builder
Database Profiles
Database profiles controls what database objects appear in the objects tree, what detailed views are available for each object type, and actions used to operate on objects. A database profile is database-specific and here you can either decide to let DbVisualizer automatically pick (recommended) the matching profile based on the database type setting or manually choose one. If manually choosing a profile, make sure it is compatible with the database you are connecting to. The generic profile works with any database. Note: You must reconnect the database connection after changing the profile.
Auto Detect
Manually Choose
"Generic" Profile
Profile
Description
postgresql
Profile for PostgreSQL 7 and earlier
postgresql8
Profile for PostgreSQL 8+
presto
Profile for Presto
redshift
Profile for Amazon Redshift
singlestore
Profile for SingleStore
snowflake
Profile for Snowflake
sqlite
Profile for SQLite
sqlserver
Profile for SQL Server
sybase-ase
Profile for Sybase ASE
trino
Profile for Trino
vertica
Profile for Vertica
yellowbrick
Profile for Yellowbrick
Defaults...
Apply

## XML element - InitCommands

The **InitCommands** element define initialization commands that are executed just before the rest of the database profile is loaded. These commands are typically used to determine characteristics of the target database and database session. The result of these commands are stored in **variables** that can be used in conditions that are evaluated when the rest of the profile is loaded. A common use case is to find out the authorization level of the current user as defined in the database. If the user have limited privileges then some object types, views and actions should be disabled.

Multiple commands may be defined in the InitCommands element and these are executed in order.

The main purpose with the following sample and the commands is to first determine the database version by querying a system table. Based on the database version a condition controls which of two queries will be executed to find out another property from the database. The result of the executed query is stored in a the **METACAT** variable.

```
<InitCommands extends="true">
   <Command id="neoview.getDbVersion" method="runBeforeConditionsEval">
      <SQL>
         <![CDATA[
SELECT SUBSTRING(SYSTEM_VERSION FROM 10)
FROM (GET VERSION OF SYSTEM) V(SYSTEM_VERSION)
         ]]>
      </SQL>
      <Output id="DBVERSION" index="1"/>
   </Command>

   <Command id="neoview.getMaster">
      <If test="#DBVERSION gte 2400">
         <SQL>
            <![CDATA[
SELECT MIN(SYSTEM_CATALOGS) AS MASTER_CAT
FROM (GET SYSTEM CATALOGS) V(SYSTEM_CATALOGS)
WHERE SYSTEM_CATALOGS LIKE _ISO88591'NONSTOP_SQLMX_%'
           ]]>
         </SQL>
      </If>
      <Else>
         <SQL>
            <![CDATA[
SELECT 'NONSTOP_SQLMX_${#dp.METACAT}'
FROM (VALUES(1)) AS T1
           ]]>
         </SQL>
      </Else>
      <Output id="METACAT" index="1"/>
   </Command>
</InitCommands>
```

The **extends="true"** attribute specifies that the list of commands will extend the list of commands defined in the profile being extended.

Initialization commands are processed in two stages:

1. First stage is to execute all commands having the attribute **method="runBeforeConditionsEval"** set. As the attribute reveal, these commands are execute before any conditions are evaluated,
2. The second and last stage will execute all commands **with no** method="runBeforeConditionsEval" set. This time any conditions are evaluated.

The reason for these stages is that the processing of initialization commands may also rely on conditions.

Here is an example how the new **METACAT** variable is used in the rest of the database profile:

```
<Command id="neoview.getCatalogs">
   <SQL>
      <![CDATA[
SELECT
    TRIM(CAT_NAME) AS CATALOG_NAME
FROM
${METACAT}.SYSTEM_SCHEMA.CATSYS C
WHERE
    CAT_NAME     NOT LIKE _ISO88591'NONSTOP_SQLMX_%'
    AND CAT_NAME NOT IN (_ISO88591'NSMWEB', _ISO88591'NVSCRIPT',
_ISO88591'METRIC',
                         _ISO88591'MATRIX', _ISO88591'GENUSCAT',
_ISO88591'MANAGEABILITY')
ORDER BY
    CATALOG_NAME
FOR READ UNCOMMITTED ACCESS
      ]]>
   </SQL>
</Command>
```

Here is another example for Oracle getting the **instance_type** property from the **v$parameter** table and put the value in the **INSTANCE_TYPE** variable.

```
<InitCommands extends="true">
   <Command id="oracle.initGetInstanceType" method="runBeforeConditionsEval">
      <SQL>
         <![CDATA[
select value from v$parameter where name = 'instance_type';
         ]]>
      </SQL>
      <Output id="INSTANCE_TYPE" index="1"/>
   </Command>
</InitCommands>
```

Below show that only if **INSTANCE_TYPE** have the value **RDBMS** schema objects should be displayed in the database objects tree:

```
<ObjectsTreeDef extends="false">
   <If test="#INSTANCE_TYPE eq 'RDBMS'">
      <GroupNode type="Schemas" label="Schemas">
         ...
      </GroupNode>
   </If>

   <GroupNode type="Properties" label="Session Properties" isLeaf="true"/>
   <GroupNode type="DBA" label="DBA Views">
      ...
   </GroupNode>
</ObjectsTreeDef>
```

One more example showing the use of **<OnError>** and **<Message>** element **<InitCommands>**. Based on the database major version it runs one of two SQLs. If any **924** error (table not found) is raised the message will be logged in the DbVisualizer log. The `type="info"` defines that the log entry should be logged without further notification. Leaving the type out or setting it to `type="warning"` will also raise the error balloon in the DbVisualizer tool. Use this only when really necessary.

```
<InitCommands extends="true">
   <Command id="oracle.initInstanceNumber">
      <If test="#util.isDatabaseVersionLT(9)">
         <SQL><![CDATA[select -1 from dual;]]></SQL>
         <Output name="OWN_INSTANCE_NUMBER" index="1"/>
      </If>
      <Else>
         <SQL><![CDATA[select instance_number from v$instance;]]></SQL>
         <Output name="OWN_INSTANCE_NUMBER" index="1"/>
      </Else>
      <OnError>
         <If test="#result.getErrorCode() eq 942" context="runtime">
            <Message type="info">
               <![CDATA[
Could not to retrieve own INST_ID (probably due to missing privileges) which is
needed to browse DBA->Sessions, etc. DbVisualizer will work fine but related
features will be disabled
               ]]>
            </Message>
         </If>
      </OnError>
   </Command>
</InitCommands>
```

Click these links for more information about the command element and conditions.

## XML element - Commands

The **Commands** element is a simple grouping element for **Command** elements.

```
<Commands extends="true">
  <Command id="profileName.xxx">
    ...
  </Command>
</Commands>
```

The **extends="true"** attribute specifies that the list of commands will extend the list of commands defined in the profile being extended.

### XML element - Command

The main purpose with the **Command** element is to run a single **SQL statement** or a **script** of SQL statements. In most cases, the script should return a result set with 0 or multiple rows with the exception for actions which not

necessarily need to return a result set, e.g., a "drop" action). The following show the command element, its attributes with default values, and valid sub elements.

```
<Command     id="sybase-ase.getLogins"
         method="dynamic"
       exectype="script"
 processmarkers="true"
     autocommit="true"
    whensuccess="commit"
      whenerror="rollback">

   <SQL>
      ...
   </SQL>

   <Input>
      ...
   </Input>

   <Output>
      ...
   </Output>

   <Filter>
      ...
   </Filter>

   <ProcessDataSet>
      ...
   </ProcessDataSet>

   <ProcessSQLWarning>
      ...
   </ProcessSQLWarning>

</Command>
```

| Attribute | Description |
| --- | --- |
| id | The command element is identified with a unique **id** attribute. This id is referred in ObjectsTreeDef, ObjectsViewDef and ObjectsActionDef definitions using the **idref** attribute. The id naming convention for command elements is to prefix with the name of the profile and a dot. Example **oracle.xxx**, **sqlserver.xxx**, and so on. |
| method | • **dynamic**<br>this is the default value and define that the SQL is a dynamic SQL statement as opposed to setting it to JDBC which defines that the SQL is really a JDBC meta data call rather than SQL,<br>• jdbc<br>See description for dynamic,<br>• runBeforeConditionsEval<br>this value is only considered if the command is define in the InitCommands section. |
| exectype | • **script**<br>• asis<br>• explain<br>The default behavior is that the SQL may contains multiple SQL statements each delimited by a semi colon (';'). Set this attribute to false to disable multiple SQL statements. |
| autocommit | • **true**<br>• false |
| whensuccess | • commit<br>• rollback<br>• ask |
| whenerror | • commit<br>• rollback<br>• ask<br>• ignore |

The following command queries login information in Sybase ASE.

```
<Command id="sybase-ase.getLogins">
   <SQL>
      <![CDATA[
SELECT name "Name", suid "SUID", dbname "Default Database", fullname "Full Name",
language "Default Language", totcpu "CPU Time", totio "I/O Time", pwdate "Password
Set"
FROM master.dbo.syslogins ORDER BY 1
      ]]>
   </SQL>
</Command>
```

The id for this command is **sybase-ase.getLogins**. The reason for prefixing the id with the name of the profile is that profiles can be extended and id's need to be unique.

> This SQL example show a command with a **SELECT** statement using **column aliases**. If no aliases are specified the column names should be used to refer the data.

RESULT SET

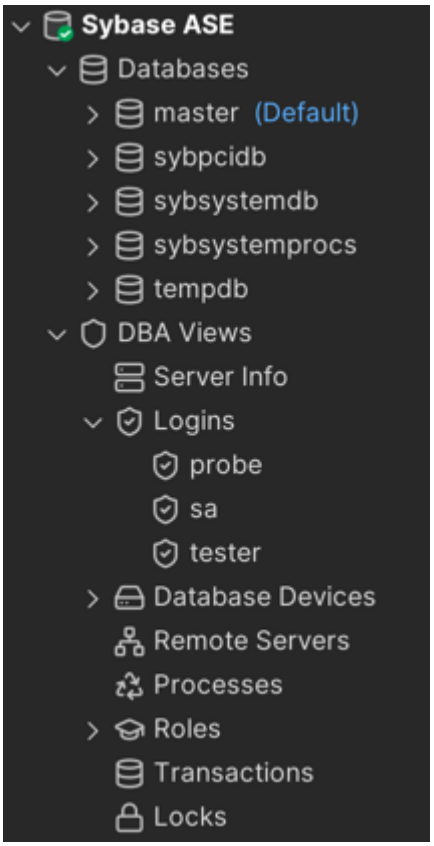
This is the result set for the previous query:

| Name | SUID | Default Database | Full Name | Default Language | CPU Time | I/O Time | Password Set |
|---|---|---|---|---|---|---|---|
| probe | 3 | master | (null) | (null) | 0 | 10 | 2009-12-22 09:53:50 |
| sa | 2 | subsystem db | (null) | (null) | 0 | 0 | 2009-12-22 08:37:35 |
| tester | 1 | master | (null) | (null) | 182 | 168723 | 2009-12-22 08:36:54 |

How DbVisualizer handle the result set depends on whether the command is executed as a request in the database objects tree (ObjectsTreeDef) or in the object view (ObjectsViewDef). If executed in the database objects tree, each row in the result set will be represented by a node in the tree. If executed in the object view, it is the viewer component that decide how the result will be displayed.

Another important difference between the database objects tree and the object view is that the tree is a hierarchical structure of objects while the object view presents information about a specific object. An object that is inserted in the database objects tree is a 1..1 mapping to a row from the result set. The end user will see these objects (nodes) by some descriptive label, as defined in the ObjectsTreeDef. All data for the row from the original result set is stored with the object in the tree and may be used in the **label**, **variables**, **conditions**, etc. This is not the case in the ObjectsViewDef.

The following example put some light on this. Consider the previous result set and that it is used to create objects in the database objects tree. The end user will see the following in DbVisualizer. (The label for each row is the name column in the result set.):




Each of the **probe**, **sa,** and **tester** nodes have all their respective data from the result set associated with the nodes. The data is referenced as **commandId.columnName**, i.e., **sybase-ase.getLogins.Name**, **sybase-ase.getLogins.Default Database,** etc. All associated data for the **sa** node in the example is listed next:

```
sybase-ase.getLogins.Name = sa
sybase-ase.getLogins.suid = 1
sybase-ase.getLogins.Default Database = master
sybase-ase.getLogins.Full Name = (null)
sybase-ase.getLogins.Default Language = (null)
sybase-ase.getLogins.CPU Time = 182
sybase-ase.getLogins.I/O Time = 168716
sybase-ase.getLogins.Password Set = 2009-12-22 08:36:54.576
```

The DataNode element definition presenting **probe**, **sa,** and **tester** nodes in the previous screenshot use the associated data for the **label** as follows:

```
<DataNode type="Login" label="${sybasease.getLogins.Name}" isLeaf="true">
   <SetVar name="objectname" value="${sybasease.getLogins.Name}"/>
   <Command idref="sybasease.getLogins">
      <Output id="sybasease.getLogins.Name" index="1"/>
      <Output id="sybasease.getLogins.suid" index="2"/>
   </Command>
</DataNode>
```

> The **Input** sub element for a Command is only used when a command is being referred with the **idref** attribute in any of ObjectsActionDef, ObjectsTreeDef or ObjectsViewDef. It has no effect specifying it for a Command in the Commands section.

There are two types of commands, with and without dynamic input. The difference is that dynamic commands accepts input data that is typically used to form the **WHERE** clause in SELECT SQLs. The previous example illustrates a static SELECT statement (without dynamic data).

To allow for dynamic input, just add variables at the positions (can be anywhere) in the SQL statement that should be replaced with dynamic values. The following is an extension of the previous example that allows for dynamic input.

```
<Command id="sybase-ase.getLogins">
   <SQL>
      <![CDATA[
SELECT name "Name", suid "SUID", dbname "Default Database", fullname "Full Name",
language "Default Language", totcpu "CPU Time", totio "I/O Time", pwdate "Password
Set"
FROM master.dbo.syslogins WHERE name = '${name}' and suid = '${suid}' ORDER BY 1
      ]]>
   </SQL>
</Command>
```

This example add two input variables: **${name}** and **${suid}**. Values for these variables should then be supplied wherever the command is referred for execution via the Input element.

The following is an example from the ObjectsTreeDef that specify the **Input** sub elements to **map values** to the variables defined in the SQL.

```
<GroupNode type="Logins" label="Logins">
  <DataNode type="Login" label="${sybase-ase.getLogins.Name} isLeaf="true">
    <SetVar name="objectname" value="${sybase-ase.getLogins.Name}">
    <Command idref="sybase-ase.getLogins">
      <Input name="name" value="sa">
      <Input name="suid" value="${sybase-ase.getProcesses.suid}">
    </Command>
  </DataNode>
</GroupNode>
```

(Note that the Command element refer the command via the **idref** attribute which is then matched with the corresponding **id** for the Command).

The **${name}** variable in the SQL will be replaced with string **sa**.

The value for the **${suid}** variable will in this case get the value of another variable, **sybase-ase.getProcesses.suid**. So where is this variable defined? As explained in the Result Set section, all the data for a row in the result set is

associated with the corresponding node in the database objects tree. In addition, it is possible to use all the data kept by the node and even its parent nodes (as presented in the objects tree) in the input to commands. So to evaluate the **${sybase-ase.getProcesses.suid}** variable, DbVisualizer first look for the variable in the current node. If it doesn't exist, it continues to look through the parent nodes until it reaches the root, which is the **Connections** node in the objects tree. If the variable is not found, it will be set to the string representation for null, which is **(null)** by default. Whenever a matching variable is found, DbVisualizer use its value and stops searching.

XML ELEMENT - OUTPUT

As mentioned earlier, a specific column value in a result set row is referenced by the name of the column and then prefixed by the command id. Sometimes this is not desirable and the **Output** definition can be used to change this behavior. The following identifies a column in the result set by its **index number**, starting from 1, and then force its name to be set.

```
<Output index="1" name="sybase-ase.getLogins.Name"/>
<Output index="2" name="sybase-ase.getLogins.suid"/>
```

(The index attribute accepts either the name of the column or index number in the result set starting from the first column at index 1).

FILTER

The Filter element assists the Database Objects Tree filtering what fields are available for the user to filter on. The label attribute for DataNode is always available for filtering. Declaring a specific Filter is useful if for example using the label1 attribute to show additional information about an object and it should be possible to filter its label.

Here is an example of the Column sub-node to Table objects showing a filter definition:

```
<GroupNode type="Columns" label="Columns">
   <DataNode type="Column" label="${getColumnDefinitions.COLUMN_NAME}"
             label1="${getColumnDefinitions.TYPE_NAME}" isLeaf="true"
             icon="#dataMap.get('getColumnDefinitions.IS_PRIMARY_KEY') eq true ?
'PrimaryKey' : 'Column'">
      <SetVar name="objectname" value="${getColumnDefinitions.COLUMN_NAME}"/>
      <Command idref="getColumnDefinitions">
         <Input name="schema" value="${schema}"/>
         <Input name="objectname" value="${theTableName}"/>
         <Input name="tableType" value="Table"/>
         <Filter index="TYPE_NAME" label="Type"/>
      </Command>
   </DataNode>
</GroupNode>
```

Above is filter is defined for the **TYPE_NAME** column in the result set. The label for it as it will appear in the filtering pane in DbVisualizer is **Type**.

The **Show Default Database/Schema** pre-defined filter in DbVisualizer requires a Filter for the **DataNode type="schema"** command. The label for this filter must be **Name**:

```
<DataNode type="Schema" label="${oracle.getSchemas.TABLE_SCHEM}">
   <SetVar name="schema" value="${oracle.getSchemas.TABLE_SCHEM}"/>
   <Command idref="oracle.getSchemas">
      <Filter index="TABLE_SCHEM" label="Name"/>
   </Command>
 ...
```

PROCESSDATASET

The ProcessDataSet element is used to process the result set generated by a command. This process is performed just after the result set has been retrieved and before any **Output** elements are handled. Examples:

```
<ProcessDataSet action="addcolumn"  index="Added Column" value="Add column with
value 'Static string ${dbvis-jdbcURL}'"/>
<ProcessDataSet action="addrow" index="first" value="${#db.loginDatabase}"/>
<ProcessDataSet action="convertnullvalues" index="1" value=""/>
<ProcessDataSet action="convertsqlwarningtodataset"/>
<ProcessDataSet action="copycolumn" index="argument_types"
value="label='plain_args' matcher='[\[\]]' replace=''"/>
<ProcessDataSet action="distinct" index="Country"/>
<ProcessDataSet action="distinctignorecase" index="Country"/>
<ProcessDataSet action="dropcolumn" index="AUTO_INCREMENT"/>
<ProcessDataSet action="dropidenticalcolumns" name="datname"/>
<ProcessDataSet action="expanddataset" index="replication"
value="columns='replication' labels='replication_key,replication_value'
json_object='true'"/>
<ProcessDataSet action="mapvalues" index="ASC_OR_DESC" value="A='ASC' D='DSC'"/>
<ProcessDataSet action="movecolumn" index="datname" value="first"/>
<ProcessDataSet action="printdataset"/>
<ProcessDataSet action="reducedataset"
index="INDEX_CATALOG,INDEX_SCHEMA,INDEX_NAME"
value="columns='COLUMN_NAME,ORDERING_SPECIFICATION' pattern='${COLUMN_NAME} $
{ORDERING_SPECIFICATION}' label='COLUMNS' separator=', '"/>
<ProcessDataSet action="removeisnullrows" index="VALUE"/>
<ProcessDataSet action="removerowsifequalto" index="DATA_LENGTH" value="16384"/>
<ProcessDataSet action="removerowsifnotequalto" index="DATA_LENGTH" value="16384"/
>
<ProcessDataSet action="renamecolumn" index="2" name="NewName"/>
<ProcessDataSet action="sortcolumn" index="LAST_NAME, FIRST_NAME"/>
<ProcessDataSet action="splitline" index="VALUE" value="splitter='&#xD;?&#xA;'"/>
<ProcessDataSet action="trimcolumn" index="PAD_COLUMN"/>
<ProcessDataSet action="truncatedataset" value="keeplast 5"/>
<ProcessDataSet action="usedataset" value="keeplast"/>
```

Explanation of the actions. The index attribute for actions that process columns, the start index is 1, while actions processing rows start with index 0.

| action | Description |
| --- | --- |
| **addcolumn** | Adds a new column to all rows with the value specified in the value attribute. The value may contain variables using the ${...} notation |
| **addrow** | Adds a new row in the result set at the specified position which can be "first", "last" or a specific row index starting at 0. The value attribute is a TAB separated list of column values. The first element will go into column 0 in the result set, the second in column 1, and so on. Variables may be used in the value. |
| **convertnullvalues** | Converts any null values in the specified column to the literal specified in the value attribute. |
| **convertsqlwarningtodataset** | Converts any SQL Warnings generated by the command to a result set |
| **copycolumn** | Create a new column and copies the data from the the source column, possibly transformed using a regexp.<br>• **label**<br>The name of the new column<br>• **matcher** (optional)<br>regex to match data on each row<br>• **replace**<br>(optional) what to replace the match with - default empty string |
| **distinct** | This action is used to remove all duplicates identified by the specified column index. What rows are removed may be different from time to time since matching is done on a single column value |
| **distinctignorecase** | Same as **distinct** but case insensitive. |
| **dropcolumn** | Drops the specified column |
| **dropidenticalcolumns** | Drops identically named columns (keeps first) |
| **expanddataset** | Splits data in specified columns to new columns. The existing columns are not removed.You can specify how to split the data in several ways:<br>• **separator**<br>the character between values e.g separator=','<br>• **range_marker**<br>characters enclosing the values that should be remove (used with separator) e.g. range_marker='[]'<br>• **no_split_in_brackets**<br>to avoid split inside when separator is a bracket (used with separator) e.g. no_split_in_brackets='true'<br>• **json_array**<br>use instead of separator to split json arrays - e.g. json_array='true'<br>• **json_object**<br>use instead of separator to split a plain json with key value e.g. json_object='true' |
| **mapvalues** | For the specific columns, converts any values. Example: `index="ASC_OR_DESC" value="A='ASC' D='DSC'"` will convert any 'A' in the ASC_OR_DESC column to 'ASC' and any 'D' values to 'DSC' |

| action | Description |
| --- | --- |
| **movecolumn** | Moves the column specified by the index attribute by index or name to the position specified by the value attribute. The latter can be expressed as "first", "last" or a specific column index. |
| **printdataset** | Prints the result set to the debug log which is useful during profile development. Debug DbVisualizer must be enabled in the **Tools → Debug Window**. |
| **reducedataset** | Groups the data in specified columns, and adds a new column with values from other columns. The grouped columns are then removed. Example:<br>`COL1 COL2 COL3 COL4`<br>`---- ---- ---- ----`<br>`A  B  C  D`<br>`A  B  E  F`<br><br>`index="COL1,COL2"`<br>`value="columns='COL3,COL4'`<br>`pattern='${COL4, COL3}'`<br>`label='NEWCOLUMN'`<br>`separator=', '"`<br><br>`COL3 COL4 NEWCOLUMN`<br>`---- ---- ---------`<br>`C  D  D, C`<br>`E  F  F, E` |
| **removeisnullrows** | Removes the row if the value in the specified result set column is null |
| **removerowsifequalto** | Removes the row if the data in the specified column is equal to the specified value |
| **removerowsifnotequalto** | Removes the row if the data in the specified column is **not** equal to the specified value |
| **renamecolumn** | Renames the result set column identified by the index |
| **sortcolumn** | Enter one or several column names (or indexes) separated with command (',') which are used to sort the result set |
| **splitline** | Replaces each row with multiple rows by splitting the value of a specified column using the given regular expression (specified in the **splitter** parameter, default is newline). |
| **trimcolumn** | Trims the data in the specified column index by removing all leading and trailing whitespaces |
| **truncatedataset** | Truncates, i.e. removes a number of rows, from the start or the end of the result set:<br>• keepfirst <n><br>• keeplast <n><br>• removefirst <n><br>• removelast <n><br>truncatedataset is useful in for example **<DataNode ... viewer="graph">** uses. |

| action | Description |
| --- | --- |
| **usedataset** | Specifies which data set to use if the result set contains multiple data sets.<br>• keepfirst<br>• keeplast<br>• removefirst<br>• removelast |

CONVERT SQL WARNING TO DATASET

The **convertsqlwarningtodataset** element will look for any SQLWarning raised during execution of the command and convert it to a result set.

```
<InitCommands extends="true">
   <Command id="netezza.supportsMultipleSchemas">
      <SQL>
         <![CDATA[
show enable_schema_dbo_check
         ]]>
      </SQL>
      <ProcessDataSet action="convertsqlwarningtodataset"/>
      <Output index="1" name="SUPPORTS_MULTIPLE_SCHEMAS"/>
   </Command>
</InitCommands>
```

**Note:** The SQLWarning support is quite brute as it will remove any other data sets produced by the command, and insert the SQLWarning alone as the only data set having a single 0,0 cell with the complete warning string.

The <If> condition based on the output from the previous InitCommand is:

```
<If test="#SUPPORTS_MULTIPLE_SCHEMAS.matches('^.*ENABLE_SCHEMA_DBO_CHECK is
[12]$')">
   ...
</If>
```

## XML element - ObjectsTreeDef

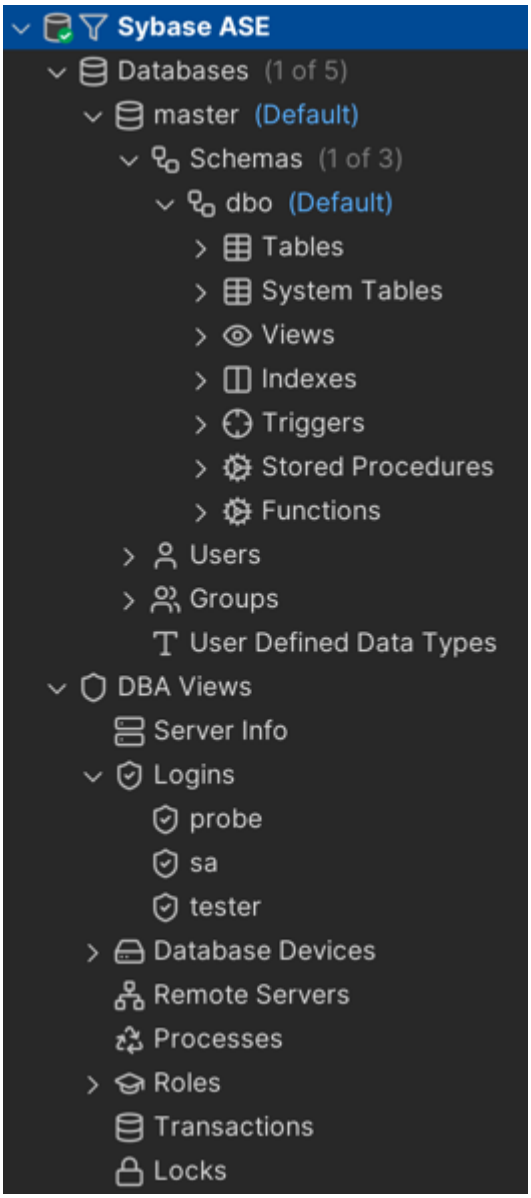
The **ObjectsTreeDef** element section controls how the database objects tree should be presented and which commands should be executed to form its content (nodes).

```
<ObjectsTreeDef extends="false">
   <GroupNode type="xxx" label="xxx">
      <DataNode type="yyy" label="yyy">
         ...
      </DataNode>
   </GroupNode>
</ObjectsTreeDef>
```

Setting the **extends** attribute to **"true"** attribute specifies that the ObjectsTreeDef will extend the ObjectsTreeDef definition in the profile being extended.

The mapping between the graphical representation in DbVisualizer and its ObejctsTreeDef XML is as quite straight forward:

**Representation in DbVisualizer**


Sybase ASE
Databases (1 of 5)
master (Default)
Schemas (1 of 3)
dbo (Default)
Tables
System Tables
Views
Indexes
Triggers
Stored Procedures
Functions
Users
Groups
User Defined Data Types
DBA Views
Server Info
Logins
probe
sa
tester
Database Devices
Remote Servers
Processes
Roles
Transactions
Locks

XML Specification for DbVisualizer Objects Tree

```
<ObjectsTreeDef extends="false">
    <GroupNode type="Databases">
        <DataNode type="Catalog">
            <GroupNode type="Schemas">
                <DataNode type="Schema">
                    <GroupNode type="Tables">
                        <DataNode type="Table"/>
                    </GroupNode>
                    <GroupNode type="SystemTables">
                        <DataNode type="SystemTable"/>
                    </GroupNode>
                    <GroupNode type="Views">
                        <DataNode type="View"/>
                    </GroupNode>
                    <GroupNode type="Indexes">
                        <DataNode type="Index"/>
                    </GroupNode>
                    <GroupNode type="Triggers">
                        <DataNode type="Trigger"/>
                    </GroupNode>
                    <GroupNode type="Procedures">
                        <DataNode type="Procedure"/>
                    </GroupNode>
                    <GroupNode type="Functions">
                        <DataNode type="Function"/>
                    </GroupNode>
                </DataNode>
            </GroupNode>
            <GroupNode type="Users">
                <DataNode type="User"/>
            </GroupNode>
            <GroupNode type="Groups">
                <DataNode type="Group"/>
            </GroupNode>
            <GroupNode type="Types"/>
        </DataNode>
    </GroupNode>
    <GroupNode type="DBA">
        <GroupNode type="ServerInfo"/>
        <GroupNode type="Logins">
            <DataNode type="Login"/>
        </GroupNode>
        <GroupNode type="Devices">
            <DataNode type="Device"/>
        </GroupNode>
        <GroupNode type="RemoteServers"/>
        <GroupNode type="Processes"/>
        <GroupNode type="ServerRoles">
            <DataNode type="ServerRole"/>
        </GroupNode>
        <GroupNode type="Transactions"/>
        <GroupNode type="Locks"/>
```

```
    </GroupNode>
</ObjectsTreeDef>
```

The screenshot in the above example show all **nodes** representing the **GroupNode** definitions in the ObjectsTreeDef. One exception is the **Logins** object, which has been expanded (**jstask**, **probe** and **sa** child objects) to illustrate how **DataNode** objects look like. The ObjectsTreeDef in the example has been simplified to show only the **type** attribute. (The label of the nodes as they appear in the screenshot is not listed in the example XML). The difference between a GroupNode and a DataNode is that GroupNode represent a static object in the tree while DataNode is dynamically created based on result sets produced by running a SQL statement. See GroupNode as a container holding other GroupNodes and DataNodes.

The database objects tree in DbVisualizer is the core visual component and it is the place where the user open object details and launch actions. To connect object actions and object views with a node in the objects tree, the **type** attribute is used. The type should be a descriptive word for a node such as **Table**, **Schemas**, **MaterializedQueryTable**, and so on. The type also map to a predefined icon. Check the database profile utilities for information how to show all bundled icons and their type mappings.

There is no limitation on the number of levels in the objects tree expressed by nesting GroupNode and DataNode elements. A good rule is as always to keep it intuitive, simple and clean.

## XML element - GroupNode

The GroupNode element represents a static object in the tree. A GroupNode do not have any associated SQL and appear only once where they are defined. A GroupNode is primarily used for structural and grouping purposes. The GroupNode element have the following attributes.

```
<GroupNode type="SystemTables" label="System Tables" isLeaf="false">
  ...
</GroupNode>
```

The **isLeaf** attribute is optional (default is **false**) and controls whether the GroupNode may have any child objects or not. It can always be set to false, the effect in the visual database objects tree is then that an expand handle always will be visible next to the icon, even if the node don't have any child nodes.

> If isLeaf is set to true and there are child Group and/or Data -nodes, these will not appear. The result may cause some frustration during the design of the database profile.

The complete set of attributes for the DataNode element:

| Attribute | Value | Description |
|---|---|---|
| **type** | | The type of node |
| **label** | | The label attribute must identify the object apart from other objects. This label should uniquely identify the object in the list of objects for the same parent node. It is used as part of the object identifier when opening object view tabs. |
| label1 | | Optional label which shows additional information about the object |
| isLeaf | true/**false** | Specifies if the node cannot have child objects |
| icon | | Icons are typically mapped using the type attribute with an icon name in the icon.prefs file(s). The **icon** attribute for a GroupNode can be set to specify an alternative icon. Conditions can be used to identify alternate icons (see below) |
| drop-label-not-equal | | Do not add the node if the label is not equal to this value or variable |
| drop-on-condition | | Read more in **drop-on-condition attribute** |
| order-before | | Specifies the order of this GroupNode among a collection of nodes having the same parent node. It can either be an index starting at 0 (first) or a node type. Ex. **order-before="Views"** will order this GroupNode before nodes defined by the **type="Views"** attribute |
| order-after | | Specifies the order of this GroupNode among a collection of nodes having the same parent node. It can either be an index starting at 0 (first) or a node type. Ex. **order-after="0"** will order this GroupNode after the first node definition |

## XML element - DataNode

The DataNode element feeds the tree with nodes produced by a **Command**. The example in the Command section querying for all logins in Sybase ASE look as follow in the ObjectsTreeDef:

```
<GroupNode type="Logins" label="Logins">
  <DataNode type="Login" label="${sybase-ase.getLogins.Name}" isLeaf="true">
    <Command idref="sybase-ase.getLogins"/>
  </DataNode>
</GroupNode>
```

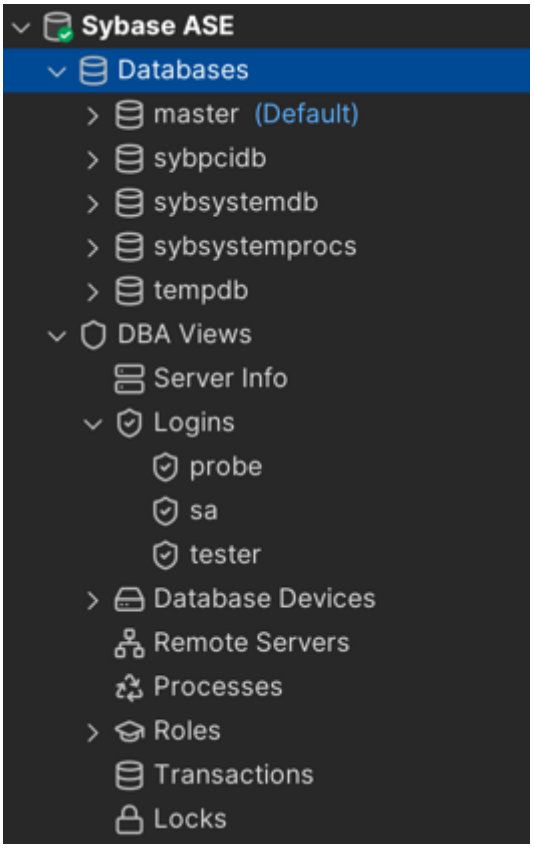
First, there is a **GroupNode** element with the purpose to group all child objects in a Logins node. The **DataNode** in this example have the same attributes as the GroupNode, the type is however singular **Login** instead of plural **Logins**

(as it is for the GroupNode). This difference is important when the user decide to open one of the nodes, since the object view will show the matching views based on the object type. For Logins a list of all logins is displayed while opening a Login, details for that specific login is displayed.

The DataNode definition can be seen as a template, as the associated command fetches rows of data from the database and DbVisualizer uses the DataNode definition to create one node per row in the result set.

The **label** attribute for the data node introduce the use a variable. The real value for the label will, in this example, be the value in the **Name** column produced by the **sybase-ase.getLogins** command (variable names are automatically prefixed with the command id that produced them).

The **Command** element uses the **idref** attribute to identify the command that should be executed. The command in this case produce a result set with 3 rows and 8 columns. The result will be two nodes for each row, with the label of the **Name** column in the result set.




The label1 attribute can be set to any other valid variable, a combination of several variables or even static text:

```
label1="${sybase-ase.getLogins.Default Database}"
```

The example above results in the following labels:

```
jstask  master
probe  subsystemdb
sa  master
```

The complete set of attributes for the DataNode element:

| Attribute | Value | Description |
|---|---|---|
| **type** | | The type of node |
| **label** | | The label attribute must identify the object apart from other objects. This label should uniquely identify the object in the list of objects for the same parent node. It is used as part of the object identifier when opening object view tabs. |
| label1 | | label1 should show useful additional information about the object. For column objects, it typically shows generic information about the columns. For index objects it typically shows the source table name. |
| icon | | Icons are typically mapped using the type attribute with an icon name in the icon.prefs file(s). The **icon** attribute for a DataNode can be set to specify an alternative icon. A condition can be specified which is used to choose an icon based on evaluating other variables for the node. Ex: `icon="#dataMap.get('getColumnDefinitio ns.IS_PRIMARY_KEY') eq true ? 'PrimaryKey' : 'Column'"` |
| tip | | An optional description that is displayed when hovering the node. This is useful to describe why a different icon is displayed for the node using the **icon** attribute. Ex: `tip="#dataMap.get('getColumnDefinition s.IS_PRIMARY_KEY') eq true ? 'This is a primary key column' : ''"` |
| actiontype | | Object type used for object actions |
| isLeaf | true/**false** | Specifies if the node cannot have child objects |
| drop-label-not-equal | | Do not add the node if the label is not equal to this value or variable |
| stop-label-not-equal | | The node will be a leaf if the label doesn't match the specified value or variable value |
| warnstate | | A condition expression returning either true or false. For true, show a warning overlay icon for the node. Ex: `errorState="! #dataMap.get('oracle.getTriggers.STATU S'). equals('ENABLED')"` |
| errorstate | | A condition expression returning either true or false. For true, show an error overlay icon for the node. Ex: `errorState="! #dataMap.get('oracle.getObjectsByType. STATUS'). equals('VALID')"` |
| is-empty-output | **continue**/stop | If result set is empty, use this to control whether child GroupNode/DataNodes should be added anyway |

| Attribute | Value | Description |
| --- | --- | --- |
| drop-on-condition | | Read more in **drop-on-condition attribute** |
| order-before | | Specifies the order of this DataNode among a collection of nodes having the same parent node. It can either be an index starting at 0 (first) or a node type. Ex. **order-before="View"** will order this node before nodes defined by the **type="View"** attribute |
| order-after | | Specifies the order of this DataNode among a collection of nodes having the same parent node. It can either be an index starting at 0 (first) or a node type. Ex. **order-after="0"** will order this DataNode after the first node definition |

The Command definition in the example above is simple, since it doesn't use any variables in the SQL. Continue reading the next section for details about passing input data to commands.

XML ELEMENT - COMMAND

The SQL used to generate the data used by the DataNodes are defined in the **Command** element.

A command is referenced by the **idref** attribute and that **id** must already be defined in the Commands section of the profile. For most DataNode definitions input must be supplied with the command and this is done by adding **Input** elements as children to the Command.

```
<DataNode type="Login" label="${sybase-ase.getLogins.Name}" isLeaf="true">
  <Command idref="sybase-ase.getLogins">
    <Input name="name" value="sa">
    <Input name="suid" value="${sybase-ase.getProcesses.suid}">
  </Command>
</DataNode>
```

The **value** for a variable specified in an **Input** element is evaluated using the syntax outlined in the result set section.

> For detailed information about the capabilities with the Command element, check the Command section.

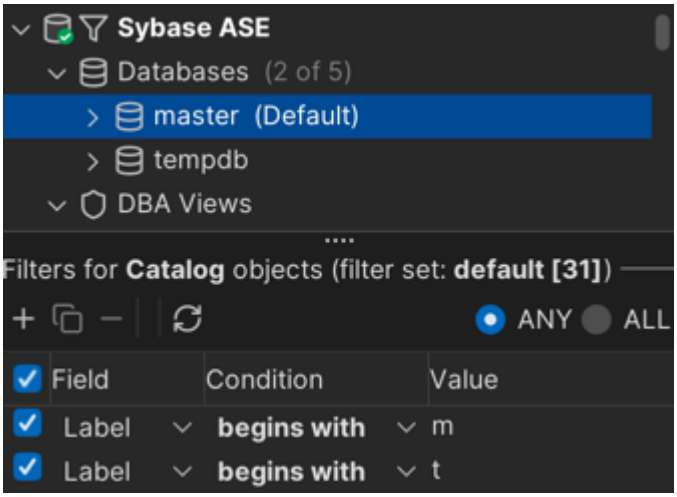
XML ELEMENT - FILTER

The Filter element is specific for Command elements that appear in the DataNode element. A filter define which data for a DataNode that can be searched in filter. This filter functionality is commonly referred as the Database Objects

Tree Filtering in DbVisualizer. The filtering setup appears below the database objects tree, and the following example shows that filtering may be specified for these object types:

- Catalog
- Table
- System Table
- View
- User
- Group
- Trigger
- Procedure

For each of the filter definitions, one or several columns can be included in the filtering criteria.




```
<DataNode type="View" label="${sybase-ase.getViews.Name}" isLeaf="true">
  <Command idref="sybase-ase.getViews">
    <Filter index="TABLE_NAME" label="View Table"/>
  </Command>
</DataNode>
```

The above filter definition specifies a filter for the Catalog (database) object type. The **index** attribute should specify a **column name** or **index** in the result set while the **label** attribute is the name how it appears in the object type drop-down list.

SHOW ONLY DEFAULT DATABASE/SCHEMA

This generic action available with any profile uses a special filter definition that must be declared for **Show only Default Database/Schema** to work:

For Catalog objects (name="Name" is the required name while index should identify the column in the result set):

```
<Filter>
   <Column index="TABLE_CAT" name="Name"/>
</Filter>
```

For Schema objects (name="Name" is the required name while index should identify the column in the result set):

```
<Filter>
   <Column index="TABLE_SCHEM" name="Name"/>
</Filter>
```

XML ELEMENT - SETVAR

The SetVar element is used in the ObjectsTreeDef for GroupNode and DataNode elements. Some object types have special meaning in DbVisualizer. Two examples are the **Catalog** and **Schema** object types. For DataNode objects, you must use SetVar elements to identify them with the name attribute set to **catalog** or **schema**, respectively.

```
Catalog:
<DataNode type="Catalog" label="${getCatalogs.TABLE_CAT}">
  <SetVar name="catalog" value="${getCatalogs.TABLE_CAT}">
</DataNode>

Schema:
<DataNode type="Schema" label="${getSchemas.TABLE_SCHEM}">
  <SetVar name="schema" value="${getSchemas.TABLE_SCHEM}">
</DataNode>
```

All DataNodes except Catalog and Schema must use SetVar to set the **objectname** variable:

```
<DataNode type="View" label="${sybase-ase.getViews.Name}" isLeaf="true">
  <SetVar name="objectname" value="${sybase-ase.getViews.Name}">
  <SetVar name="rowcount" value="true">
</DataNode>
```

The **objectname** variable is used to identify the object represented by the data node, so that it can be uniformly referenced in object views and object actions. Its value should be the identifier for the object as it is identified in the database, for example a table name or view name.

The **rowcount** variable is optional (default is false) and controls whether the object supports showing row count information when Show/Hide Table Row Count right-click menu choice is enabled for the database connection.

Another optional variable (not shown in the example above) is named **acceptInQB** (default is false). If set to true, nodes of this type can be used in the Query Builder. It should only be set to true for object types representing tabular data that can be queried with an SQL SELECT statement, such as tables, views, materialized views, etc.

Variables defined with SetVar are by default invisible in for example the node form viewer. If you want to override this behavior then add the **action** attribute and set its value to **show**. If you want to drop a variable completely from the node simply set the **action** attribute to **drop**.

Using SetVar for GroupNode's is used to set static values (since GroupNode's doesn't execute a Command). This may be used to pass a static value for later use in an Action or DataView. See vertica.xml which illustrates using SetVar for GroupNode's.

## XML element - ObjectsViewDef

The ObjectsViewDef element defines all views for the object types in the object's tree. These views are displayed in the Object View area for the selected object. What views should appear when selecting a node in the tree is based on the object type for the tree node and the corresponding object view definition.

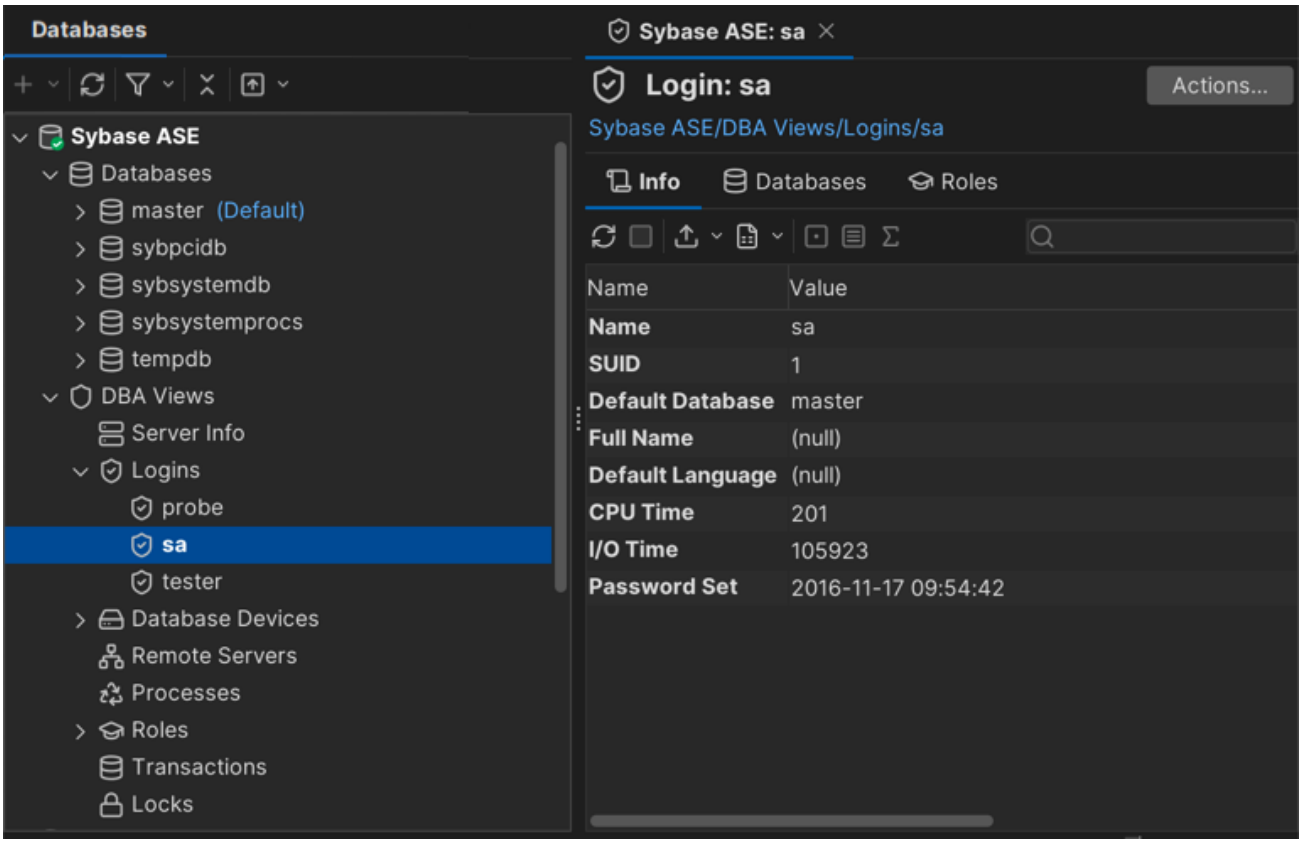
```
<ObjectsViewDef extends="true">
   <ObjectView type="xxx">
      <DataView id="yyy" label="yyy">
         ...
      </DataView>
   </ObjectView>
</ObjectsViewDef>
```

The **extends="true"** attribute specifies that this definition will extend the ObjectsViewDef definition in the database profile being extended.

> All database profiles should extend **generic** profile as the very top level profile.

When an object is opened in the database tree (**sa** in the screenshot below), a corresponding object view tab is created (right in the sample). Each of the DataView elements in the ObjectView will appear as sub tabs in the object view tab. The selected object and its information is passed to each of the data views for processing and presentation. The following example shows the Object View in DbVisualizer and its ObjectView element definition.

Representation in DbVisualizer:




XML definition:

```
<ObjectView type="Logins">
   <DataView type="Logins" label="Logins" viewer="grid">
      <Command idref="sybase-ase.getLogins"/>
   </DataView>
</ObjectView>

<ObjectView type="Login">
   <DataView type="Info" label="Info" viewer="node-form"/>
   <DataView type="Databases" label="Databases" viewer="grid">
      <Command idref="sybase-ase.getLoginDatabases"/>
   </DataView>

   <DataView type="Roles" label="Roles" viewer="grid">
      <Command idref="sybase-ase.getLoginRoles"/>
   </DataView>
</ObjectView>
```

The screenshot and the database tree show both the **Logins** node and its child nodes, **jstask**, **probe** and **sa**. These nodes are instances of the object types **Logins** (labeled **Logins** in the screenshot) and **Login** (the three sub nodes: **jstask**, **sa** and **probe**).

The ObjectView XML definitions above show the data views for these two types, **Logins** and **Login**. Opening the node labeled **Logins** in the tree will show the object view for the **<ObjectView type="Logins">** definition while opening the node labeled **jstask**, **probe** or **sa** will show the object view for the **<ObjectView type="Login">** .

The example shows **sa** being selected. Its DataView definitions displayed as tabs in the object view are (by label):

- Info
- Databases
- Roles

## XML element - ObjectView

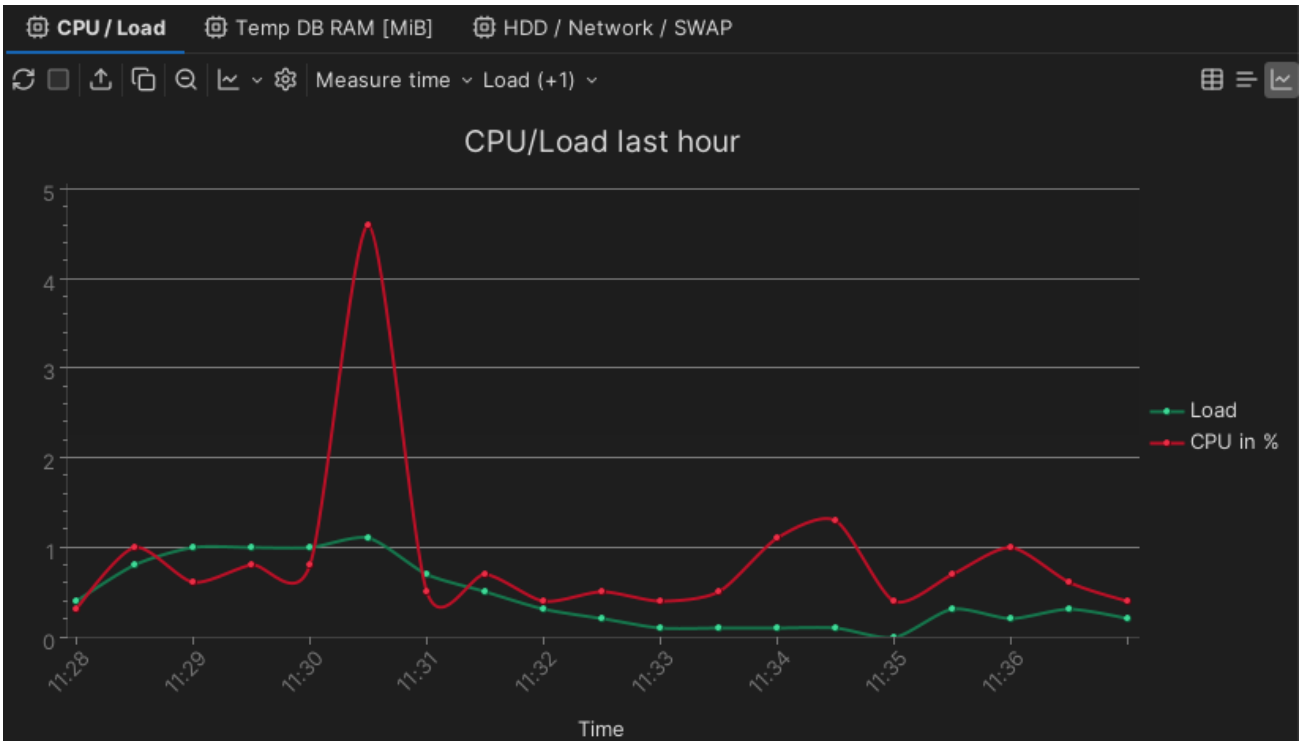
The ObjectView element is associated with an object type and groups all DataView elements that appear when the object type is selected in the database objects tree. Here follows the ObjectView definition for the Login object type.

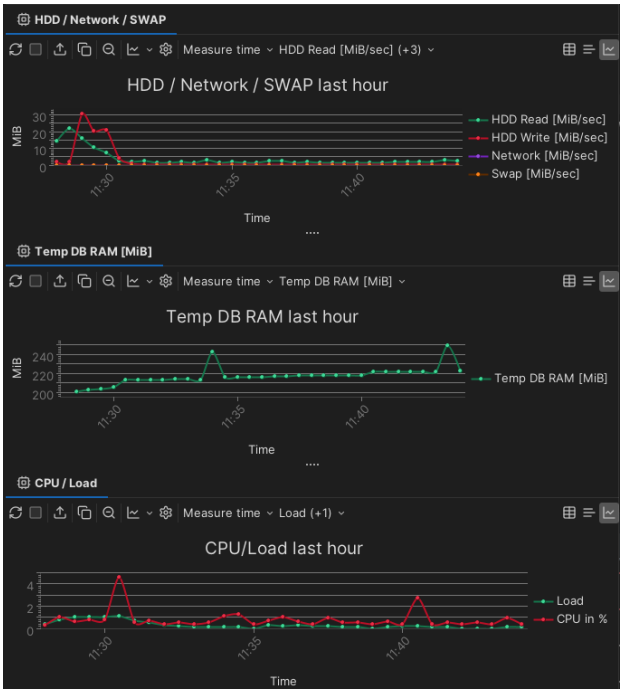
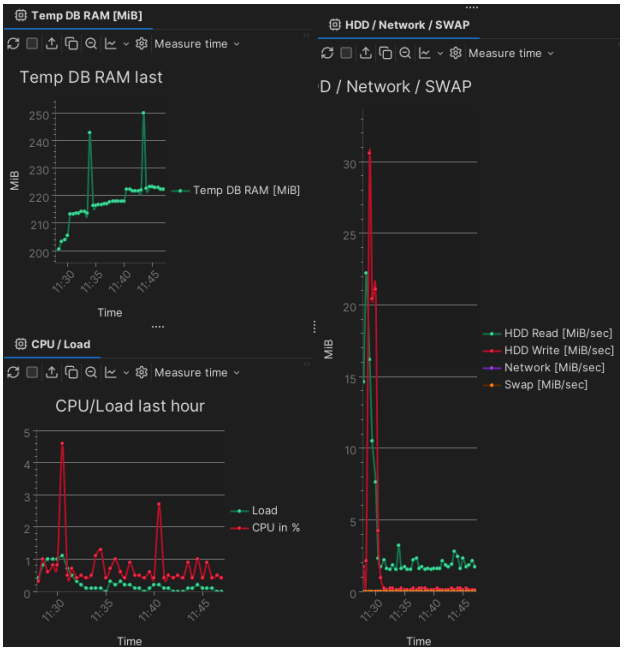
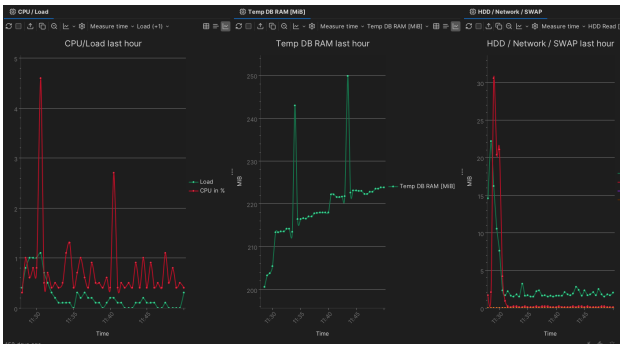
```
<ObjectView type="Login">
  ...
</ObjectView>
```

The **type** attribute value is used when a node is clicked in the database objects tree to map with the corresponding ObjectView definition. The following lists the attributes for ObjectView:

| Attribute | Description |
|---|---|
| **type** | The type of the ObjectView as declared in the GroupNode and DataNode elements in the ObjectsTreeDef section |
| layout | The layout attribute is used to tile the contained DataViews so that they are all visible att once in the object view. The default (collapse) shows all DataViews as tabs in a single tab group while tile shows them all side-by-side - **tilevertical <n>** - **tilehorizontal <n>** - **collapse** (default) The number **<n>** specifies how many DataViews should be present in the first group (vertical or horizontal) of DataViews. |
| drop-on-condition | Read more in **drop-on-condition attribute** |

The **layout** attribute is useful to tile DataViews side-by-side, either vertically or horizontally. Leaving it out will **collapse** the DataView tabs in a single tab group:


CPU / Load
Temp DB RAM [MiB]
HDD / Network / SWAP
Measure time
Load (+1)
CPU/Load last hour
5
4
3
2
1
0
11:28
11:29
11:30
11:31
11:32
11:33
11:34
11:35
11:36
Time
Load
CPU in %

| layout="tilevertical" Tiles all tabs vertically in a single column | layout="tilevertical 2" Tiles the tabs vertically with two tabs in the first column and the rest in the second column |
| --- | --- |
|  |  |
| **layout="tilehorizontal"** Tiles all tabs horizontally in a single row | **layout="tilehorizontal 2"** Tiles the tabs horizontally with two tabs in the first row and the rest in the second row 2 |
|  |  |

The previous tile examples all use the viewer='chart' for the DataView's. Please note that any supported viewer may be tiled.

## XML element - DataView

The DataView element is comparable with the DataNode element in the ObjectsTreeDef. It defines what SQL (command) should be executed, labeling, viewer type (presentation form) and other characteristics. The following is

the DataView definitions for the **Login** object type. (The ObjectView element is part of the sample just for clarification).

```
<ObjectView type="Login">
  <DataView type="sybasease-login-info" icon="Info" label="Info" viewer="node-
form"/>
  <DataView type="sybasease-login-databases" icon="Databases" label="Databases"
viewer="grid">
    <Command idref="sybase-ase.getLoginDatabases"/>
  </DataView>
  <DataView type="sybasease-login-roles" icon="Roles" label="Roles" viewer="grid">
    <Command idref="sybase-ase.getLoginRoles"/>
  </DataView>
</ObjectView>
```

All three DataView elements have a **viewer** attribute identifying how the data in the view should be presented, e.g., as a grid or a form. See the next sections for a list of viewers. The following lists all attributes for DataView:

| Attribute | Description |
|---|---|
| **id** | Every DataView element must have a unique id which is not only unique in the current profile but also with all id's in extended profiles. To make sure the id is unique use the following recommended format: **profileName-objectViewType-viewerLabel**. Ex: sybasease-login-databases (The id should not contain any empty space or special characters other than dash ("-")). |
| label | The label for the viewer as it will appear in the tab |
| icon | The icon as defined in the icons.prefs file(s) |
| viewer | One of: - **chart** - **ddl** - **form** - **grid** - **message** - **navigator** - **node-form** - **procedure-editor** - **table-data** - **table-refs** - **tables-refs** - **table-rowcount** - **text** See the viewers section in this document for more information |
| drop-label-not-equal | Drop the viewer if its label is not equal to the value of this attribute |
| class | Used to specify a custom Java class used as the viewer |
| classargs | Used to pass arguments to a custom viewer |
| drop-on-condition | Read more in **drop-on-condition attribute** |
| doclink | Relative HTML link to the related chapter in the user guide. |
| order-before | Specifies the order of this DataView among a collection of viewers having the same parent ObjectView. It can either be an index starting at 0 (first) or a node type. Ex. **order-before="sybasease-login-databases"** will order this DataView before viewers defined by the **id="sybasease-login-databases"** attribute |
| order-after | Specifies the order of this DataView among a collection of viewers having the same parent ObjectView. It can either be an index starting at 0 (first) or a node type. Ex. **order-after="sybasease-login-databases"** will order this DataView after viewers defined by the **id="sybasease-login-databases"** attribute |

The viewer attribute for a DataView defines how the data for the viewer should be presented. The following sections walk through the supported viewers:

- **chart**
- **ddl**
- **form**
- **grid**
- **message**
- **navigator**
- **node-form**
- **procedure-editor**
- **table-data**
- **table-refs**
- **tables-refs**
- **table-rowcount**
- **text**

The following sample illustrates the viewer attribute.

```
<ObjectView type="Login">
  <DataView type="Info" label="Info" viewer="node-form"/>
</ObjectView>
```

DataView definitions may be nested and the viewers are then presented with the nested DataView in the lower part of the screen.

### Viewer - chart

The chart viewer presents a result set as a chart. This is typically useful for displaying monitoring metrics, statistics, and any result set from a query from the database. Using it with the **layout** attribute for the **ObjectView** element allows creating dashboards with charts and other viewers.

Here is a simple example of defining the **DataView** using the chart viewer:

```
<DataView id="exasol-monitoringlastday-cpu" icon="Monitoring"
   label="CPU / Load" viewer="chart">
   <Command idref="exasol.getMonitorLastDay"/>

   <Input name="displayMode" value="chart"/>
   <Input name="toolbarVisible" value="false"/>

   <Input name="chart.category_column" value="Measure time"/>

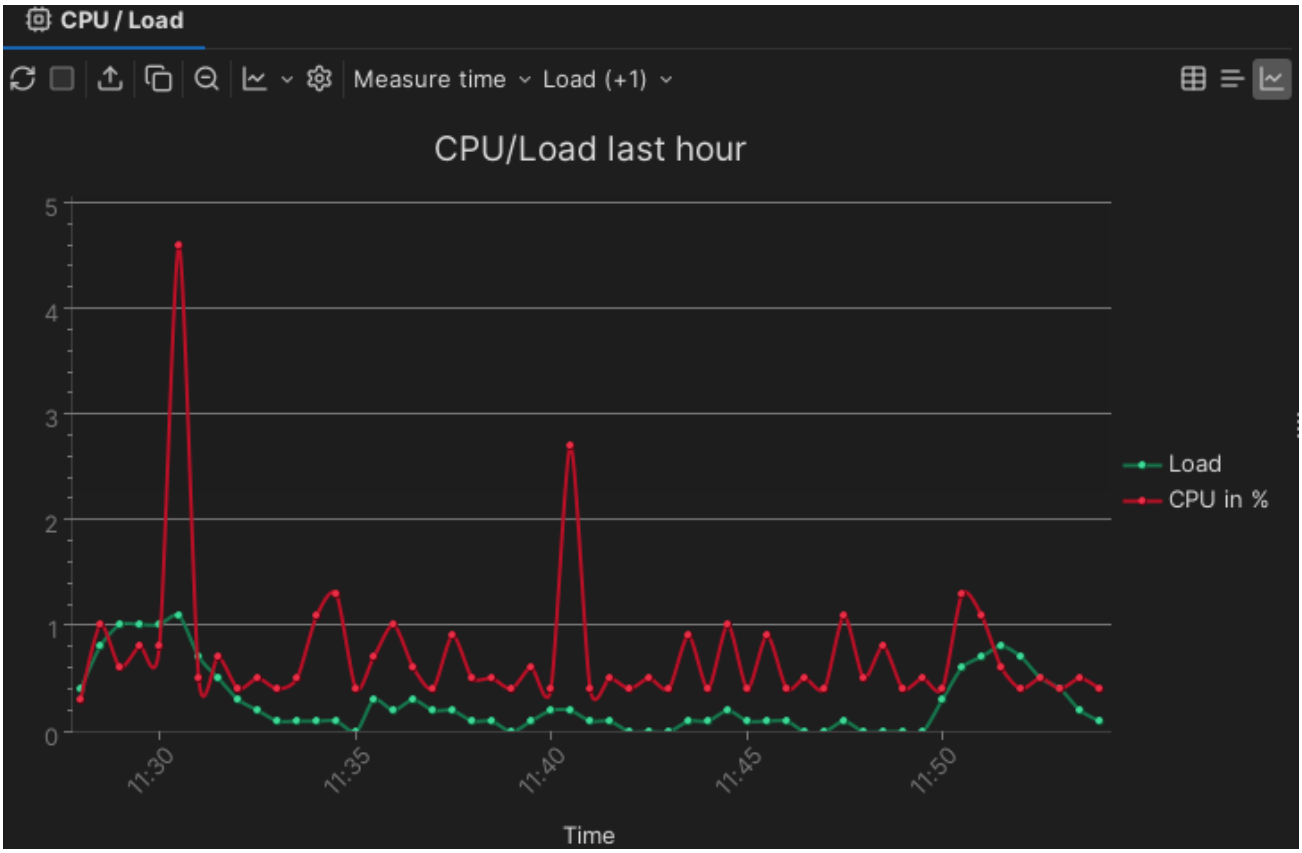
   <Input name="chart.serie.0.name" value="Load"/>
   <Input name="chart.serie.0.visible" value="true"/>
   <Input name="chart.serie.1.name" value="CPU in %"/>
   <Input name="chart.serie.1.visible" value="true"/>

   <Input name="chartTitle" value="CPU/Load last hour"/>
   <Input name="xaxisHandleTimeAsText" value="false"/>
   <Input name="xaxisTitle" value="Time"/>
   <Input name="yaxisTitle" value=""/>
</DataView>
```

Here is the **Command** used in the previous example:

```
<Command id="exasol.getMonitorLastDay">
   <SQL>
      <![CDATA[
SELECT
    MEASURE_TIME 'Measure time',
    LOAD 'Load',
    CPU 'CPU in %',
    TEMP_DB_RAM 'Temp DB RAM [MiB]',
    HDD_READ 'HDD Read [MiB/sec]',
    HDD_WRITE 'HDD Write [MiB/sec]',
    NET 'Network [MiB/sec]',
    SWAP 'Swap [MiB/sec]'
FROM
    EXA_STATISTICS.EXA_MONITOR_LAST_DAY
WHERE  MEASURE_TIME >= add_minutes(CURRENT_TIMESTAMP,-60)
ORDER BY 1
      ]]>
   </SQL>
</Command>
```

And here is how it looks having the **Measure Time** column in the result set as the category (along the X axis) and the **Load**, and **CPU in %** as series.




All the options available to configure a chart in the DbVisualizer GUI are available as parameters using this syntax:

```
<Input name="displayMode" value="chart"/>
```

These are the parameters controlling what series should be available in the chart.

For **chart.serie.<n>...** parameters below, the **<n>** should be replaced with sequential integers starting with 0 and are increased with 1 for every serie.

| Parameter | Description | Values |
|---|---|---|
| **chart.category_column** | Identifies the column in the original result set that should represent the category column. These are the labels that should be | |
| displayed along the X axis. | | |
| **chart.serie.<n>.name** | Defines the column in the result set that should appear as a serie. If you are using aliases to rename columns, use the alias | |
| name here. | | |
| **chart.serie.<n>.visible** | Defines whether the serie should be initially visible or not. It will still be selectable in the chart series drop-down. | |
| Valid values: **true** (default) or **false** | | |
| **chart.serie.<n>.label** | If the name of the column in the result set defined by **chart.serie.<n>.name** needs to be changed, this parameter | |
| can be used to set an alternative name. | | |

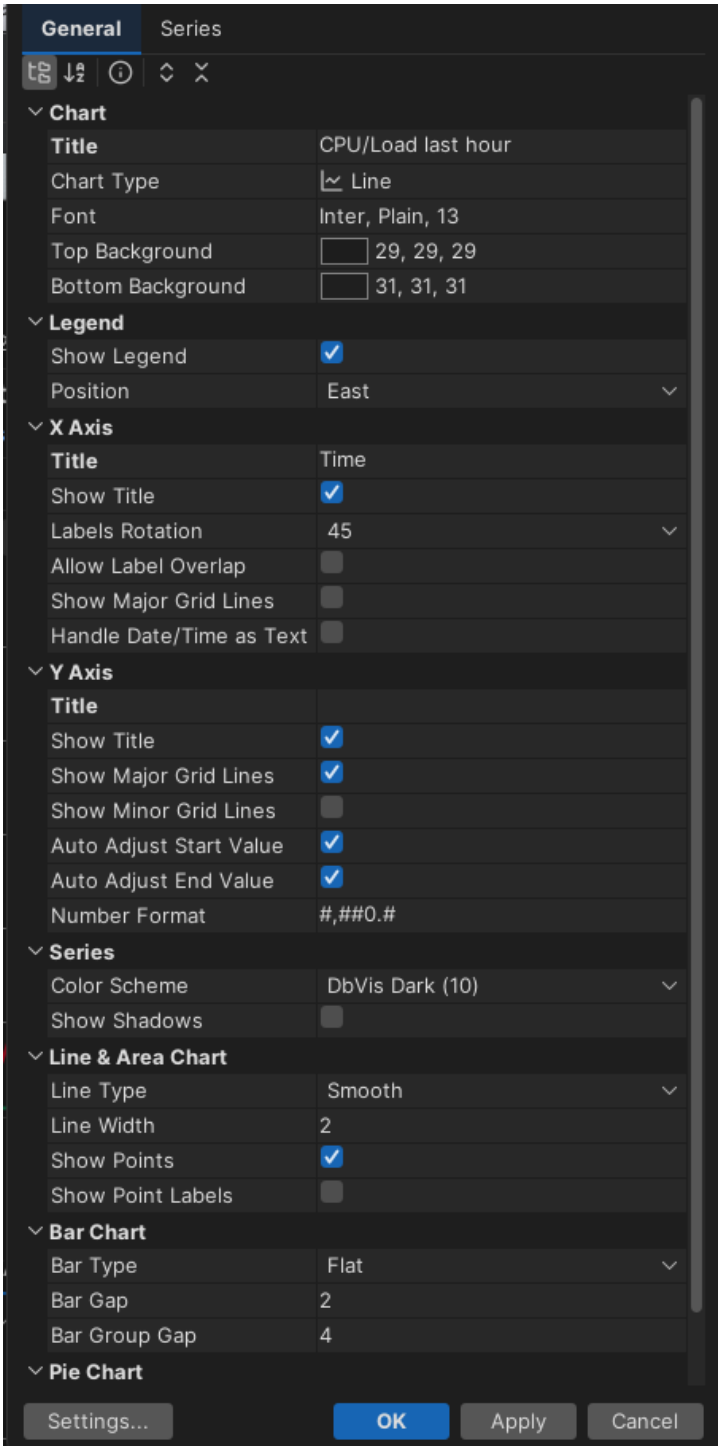
The following parameters control the appearance and content of the chart:

| Parameter | Description | Values |
| --- | --- | --- |
| **displayMode** | Defines how the result set should be presented when displayed. The mode can be changed by the user using the toolbar or right-click menu. | **chart**, **grid** or **text** |
| **toolbarVisible** | Specifies if the toolbar should be visible. | **true** (default) or **false** |
| **statusBarVisible** | Specifies if the status bar in the grid or text mode should be visible. | **true** (default) or **false** |
| Chart | | |
| **chartTitle** | The chart title displayed in big text at the top | |
| **chartType** | The type of chart | • **result-set-chartviewer-line-chart-command** (default)<br>• **result-set-chartviewer-point-chart-command**<br>• **result-set-chartviewer-area-chart-command**<br>• **result-set-chartviewer-stacked-area-chart-command**<br>• **result-set-chartviewer-bar-chart-command**<br>• **result-set-chartviewer-stacked-bar-chart-command**<br>• **result-set-chartviewer-pie-chart-command** |
| **chartFont** | The font for all text in the chart | Syntax: **<FontFamily>-<Style>-<Size>** Example: **Lucida Grande-plain-13** Default: same |
| **chartTopBackground** | The top background color. This is used as start color for a gradient background | Example: **#ffffff** Default is based on the current theme. |
| **chartBottomBackground** | The bottom background color. This is the end color for a gradient background | Example: **#ccffcc** Default is based on the current theme. |
| Legend | | |
| **legendVisible** | Check this to show legend box with labels for all series in the chart | **true** (default) or **false** |
| **legendLocation** | The position of the legend box | **North**, **West**, **South** or **East** (default) |
| X Axis | | |
| **xaxisTitle** | The X axis title displayed below the X axis | |
| **xaxisTitleVisible** | Check this to display the X axis title | **true** (default) or **false** |
| **xaxisRotation** | The rotation of the X axis labels | **Horizontal**, **45** (default) or **Vertical** |
| **xaxisLabelOverlap** | Check this to allow X axis labels to overlap each other | **true** or **false** (default) |
| **xaxisMajorGridLines** | Check this to display X axis major grid lines | **true** or **false** (default) |
| **xaxisHandleTimeAsText** | Check to handle date | **true** (default) or **false** |

| Parameter | Description | Values |
|---|---|---|
| Y Axis | | |
| **yaxisTitle** | The Y axis title displayed to the left of the Y axis | |
| **yaxisTitleVisible** | Check this to display Y axis title | **true** (default) or **false** |
| **yaxisMajorGridLines** | Check this to display of Y axis major grid lines | **true** (default) or **false** |
| **yaxisMinorGridLines** | Check this to display of Y axis minor grid lines | **true** or **false** (default) |
| **yaxisAutoStartValue** | Check to adjust the Y-axis start value based on the data | **true** (default) or **false** |
| **yaxisStartValue** | The start value for the Y-axis | 0.0 |
| **yaxisAutoEndValue** | Check to adjust the Y-axis end value based on the data | **true** (default) or **false** |
| **yaxisEndValue** | The end value for the Y-axis | 0.0 |
| **yaxisNumberFormat** | The number format for value labels | #,##0.# Format documentation. |
| Series | | |
| **chartColorScheme** | Choose color scheme for the chart series | These are the available color schemes. Make sure the name is specified as presented below.<br>• **Qualitative Set1 (9)**<br>• **Qualitative Set3 (10)**<br>• **Diverging Spectral (10)**<br>• **HTML Color Names (140)**<br>• **New Tau (10)**<br>• **Tau (20)** (default)<br>• **Tau Color Blind (10)**<br>• **Divirgent Color Scale (13)** |
| **chartShadow** | Check this to display a shadow border for series | **true** (default) or **false** |
| Line & Area Chart | | |
| **chartLineType** | Render straight or smooth lines for line charts | **Straight** (default) or **Smooth** |
| **chartLineWidth** | The line width in pixels | **2** |
| **chartShowPoints** | Check this to display points for each value | **true** (default) or **false** |
| **chartShowPointLabels** | Check this to display point labels | **true** or **false** (default) |
| Bar Chart | | |
| **chartBarType** | Render bars as raised or flat | **Flat** (default), **Raised**, **Cylinder or 3D** |
| **chartBarGap** | The gap in pixels between bars | **2** |
| **chartBarGroupGap** | The gap in pixels between groups of bars | **4** |
| Pie Chart | | |

| Parameter | Description | Values |
| --- | --- | --- |
| **chartPieType** | Render pie as raised | **Flat** (default), **Raised** or **3D** |
| **chartPieLabelType** | Use this setting to specify how slice labels will be displayed | **Line Labels** (default), **Simple Labels** or **No Labels** |

As a reference to the chart configuration in the DbVisualizer GUI, here are the same parameters (and order of appearance) as the list of parameters explained above:



Viewer - ddl

The **ddl** viewer is special as it requires support in the DbVisualizer codebase to work properly. When supported it will show the DDL based on the actual object's **catalog**, **schema**, and **objectname** variables.

```
<DataView id="oracle-table-ddl" icon="Source" label="DDL" viewer="ddl">
   <Input name="formatSQL" value="true"/>
   <Input name="objectType" value="Table"/>
</DataView>
```

The **formatSQL** parameter specifies if the generated DDL should be formatted (false by default). The **objectType** parameter is used by some DDL generators to distinguish what object type is being processed.

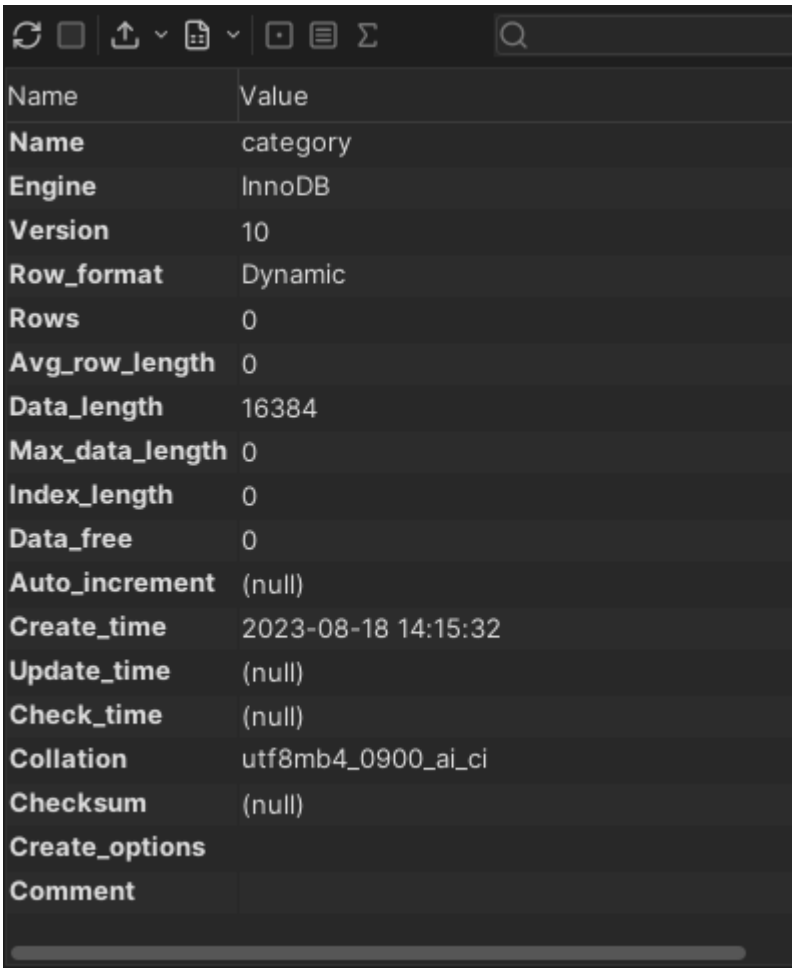
Viewer - form

The **form** viewer displays row(s) from a result set in a form. If several rows are in the result, they are presented in a list. Selecting one row from the list presents all columns and data for that row in a form.

Here is a sample of the XML for the form viewer:

```
<DataView id="mysqlbase-table-info" icon="Info" label="Info" viewer="form" order-
before="0">
   <Command idref="mysqlbase.getTable">
      <Input name="catalog" value="${catalog}"/>
      <Input name="table" value="${objectname}"/>
   </Command>
</DataView>
```

And here is a screenshot of the Info tab based on the previous definition.



| Name | Value |
|---|---|
| Name | category |
| Engine | InnoDB |
| Version | 10 |
| Row_format | Dynamic |
| Rows | 0 |
| Avg_row_length | 0 |
| Data_length | 16384 |
| Max_data_length | 0 |
| Index_length | 0 |
| Data_free | 0 |
| Auto_increment | (null) |
| Create_time | 2023-08-18 14:15:32 |
| Update_time | (null) |
| Check_time | (null) |
| Collation | utf8mb4_0900_ai_ci |
| Checksum | (null) |
| Create_options | |
| Comment | |

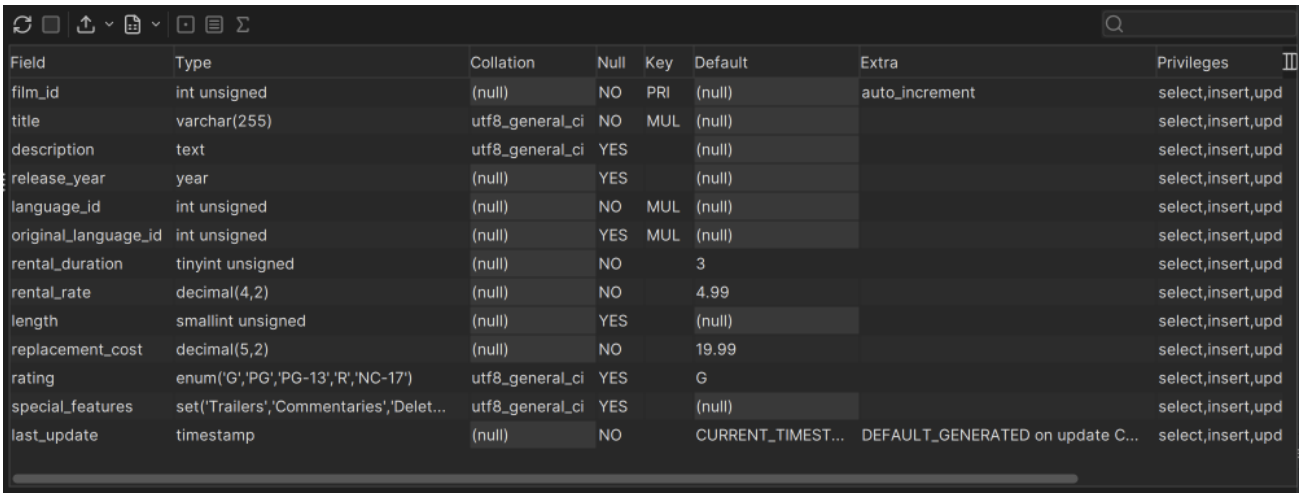
Viewer - grid

The **grid** viewer presents a result set in a grid with standard grid features such as search, copy, fit columns, export, and so on. The result set is presented exactly as it is produced by the associated **Command** and any optional **Output** processing.

Here is a sample of the XML for the grid viewer:

```
<DataView id="mysqlbase-table-columns" icon="Columns" label="Columns"
viewer="grid" order-after="mysqlbase-table-info">
   <Command idref="mysqlbase.getColumns">
      <Input name="catalog" value="${catalog}"/>
      <Input name="table" value="${objectname}"/>
   </Command>
</DataView>
```

And here is a screenshot of the standard grid viewer created from the above definition.



| Field | Type | Collation | Null | Key | Default | Extra | Privileges |
|---|---|---|---|---|---|---|---|
| film_id | int unsigned | (null) | NO | PRI | (null) | auto_increment | select,insert,upd |
| title | varchar(255) | utf8_general_ci | NO | MUL | (null) | | select,insert,upd |
| description | text | utf8_general_ci | YES | | (null) | | select,insert,upd |
| release_year | year | (null) | YES | | (null) | | select,insert,upd |
| language_id | int unsigned | (null) | NO | MUL | (null) | | select,insert,upd |
| original_language_id | int unsigned | (null) | YES | MUL | (null) | | select,insert,upd |
| rental_duration | tinyint unsigned | (null) | NO | | 3 | | select,insert,upd |
| rental_rate | decimal(4,2) | (null) | NO | | 4.99 | | select,insert,upd |
| length | smallint unsigned | (null) | YES | | (null) | | select,insert,upd |
| replacement_cost | decimal(5,2) | (null) | NO | | 19.99 | | select,insert,upd |
| rating | enum('G','PG','PG-13','R','NC-17') | utf8_general_ci | YES | | G | | select,insert,upd |
| special_features | set('Trailers','Commentaries','Delet... | utf8_general_ci | YES | | (null) | | select,insert,upd |
| last_update | timestamp | (null) | NO | | CURRENT_TIMEST... | DEFAULT_GENERATED on update C... | select,insert,upd |

Drill-down view

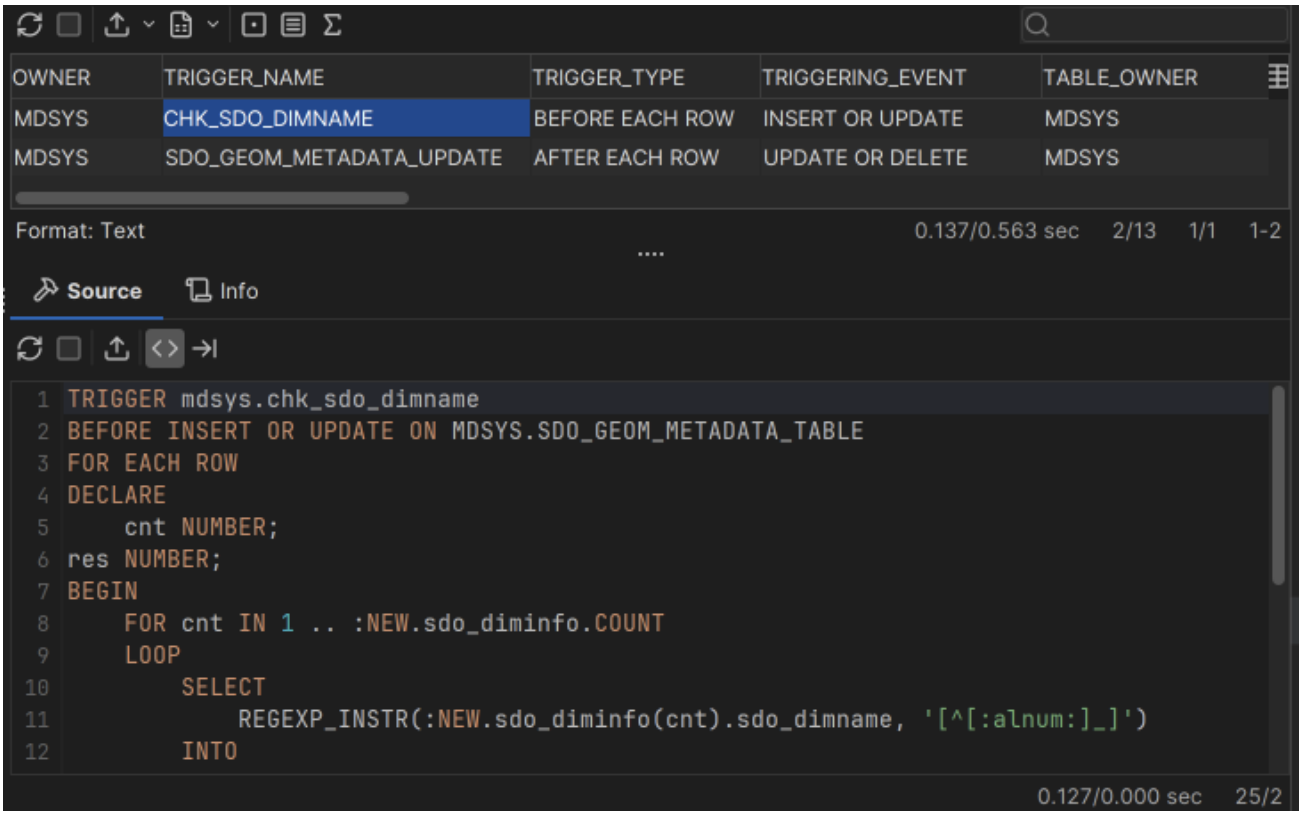
The nesting capability for grid viewers is really powerful, as it can be used to create a **drill-down** view of the data. Consider the scenario with a grid viewer showing all Trigger objects. Wouldn't it be nice to offer the user the

capability to display the trigger source when selecting a row in the list? This is easily accomplished with the following definition:

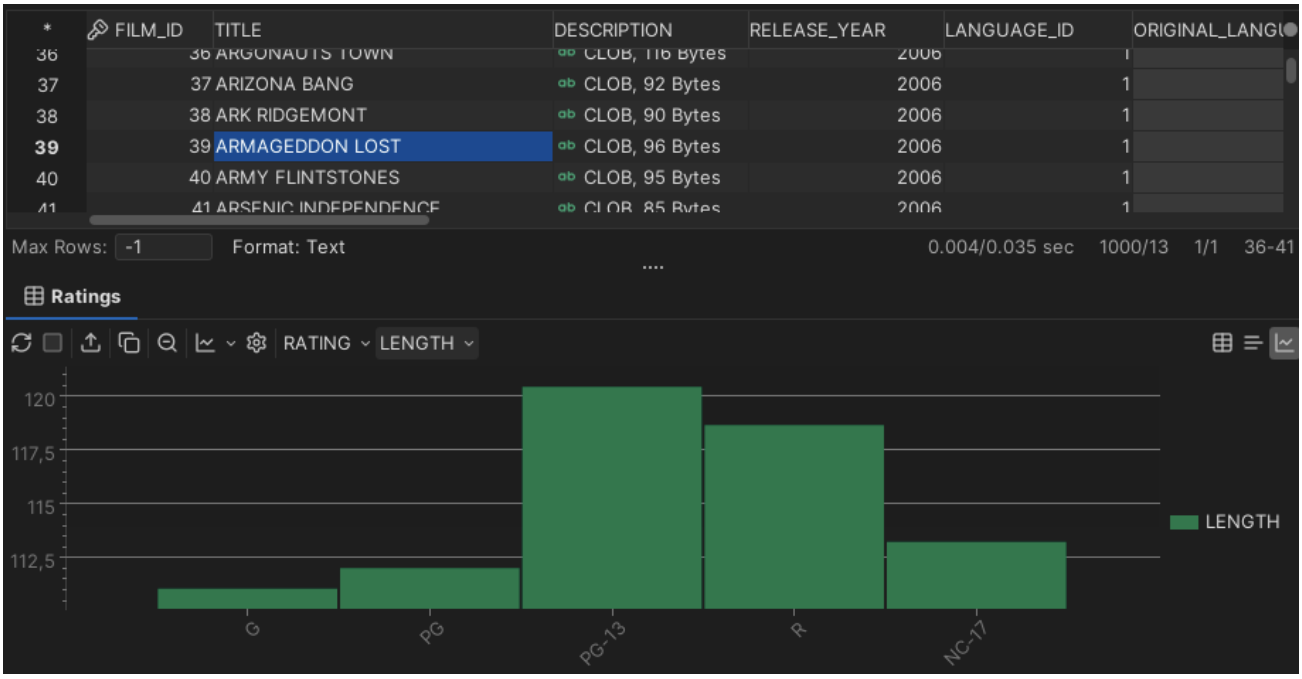
```
<DataView id="oracle-table-triggers" icon="Trigger" label="Triggers"
viewer="grid">
   <Command idref="oracle.getTriggers">
      <Input name="owner" value="${schema}"/>
      <Input name="condition" value="${triggersCondition}"/>
   </Command>
   <DataView id="oracle-table-triggers-source" icon="Source" label="Source"
viewer="text">
      <Input name="dataColumn" value="text"/>
      <Input name="formatSQL" value="true"/>
      <Command idref="oracle.getTriggerSource">
         <Input name="owner" value="${OWNER}"/>
         <Input name="name" value="${TRIGGER_NAME}"/>
      </Command>
   </DataView>
   <DataView id="oracle-table-triggers-info" icon="Info" label="Info"
viewer="node-form"/>
</DataView>
```

- The first DataView element defines the top grid viewer labeled **Triggers** and the command to get the result set for it
- The next DataView is the **nested text viewer** labeled **Source**, specifying various input parameters for the viewer along with the command to get the source for the trigger. The difference here is that the input parameters for this command reference column names in the top grid. Since this viewer is nested, it will automatically be notified whenever an entry in the top grid is selected
- The third DataView labeled **Info** is presented as a tab next to the **Source** viewer, and presents additional information about the selected trigger

The following screenshot illustrates the above sample:




A drill-down type of setup can also be done with a chart as drill-down viewer showing data in the chart based on the selection in the top grid:




Adding custom menu items in the grid

The grid right-click menu contains a lot of standard actions. Custom commands can be defined in the DataView element, and these will appear last in the menu.

```
<Input name="menuItem" value="Open in New Tab...">
   <Input name="action" value="open-object-in-new-tab-command ${schema||OWNER}$
{object||TABLE_NAME}"/>
</Input>
<Input name="menuItem" value="Open in Floating Tab...">
   <Input name="action" value="open-object-in-floating-tab-command ${schema||
OWNER}${object||TABLE_NAME}"/>
</Input>
   <Input name="menuItem" value="Script: SELECT ALL">
   <Input name="command" value="select * from ${schema||OWNER}${object||
TABLE_NAME}"/>
</Input>
   <Input name="menuItem" value="Script: DROP TABLE">
   <Input name="command" value="drop table ${schema||OWNER}${object||TABLE_NAME}"/
>
</Input>
```

The **<Input name="menuItem">** element defines a menu item entry that should appear in the grid right-click menu. The **value** for the menuItem is the * label for the item as it will appear in the menu while the child Input element with name="command" is the SQL command that should be produced for all selected rows when the menu item is selected. Invoking a custom menu item will not execute the produced SQL directly but rather copy the statements to a SQL Editor*. In the SQL Editor you will then need to manually execute the script and track the result.

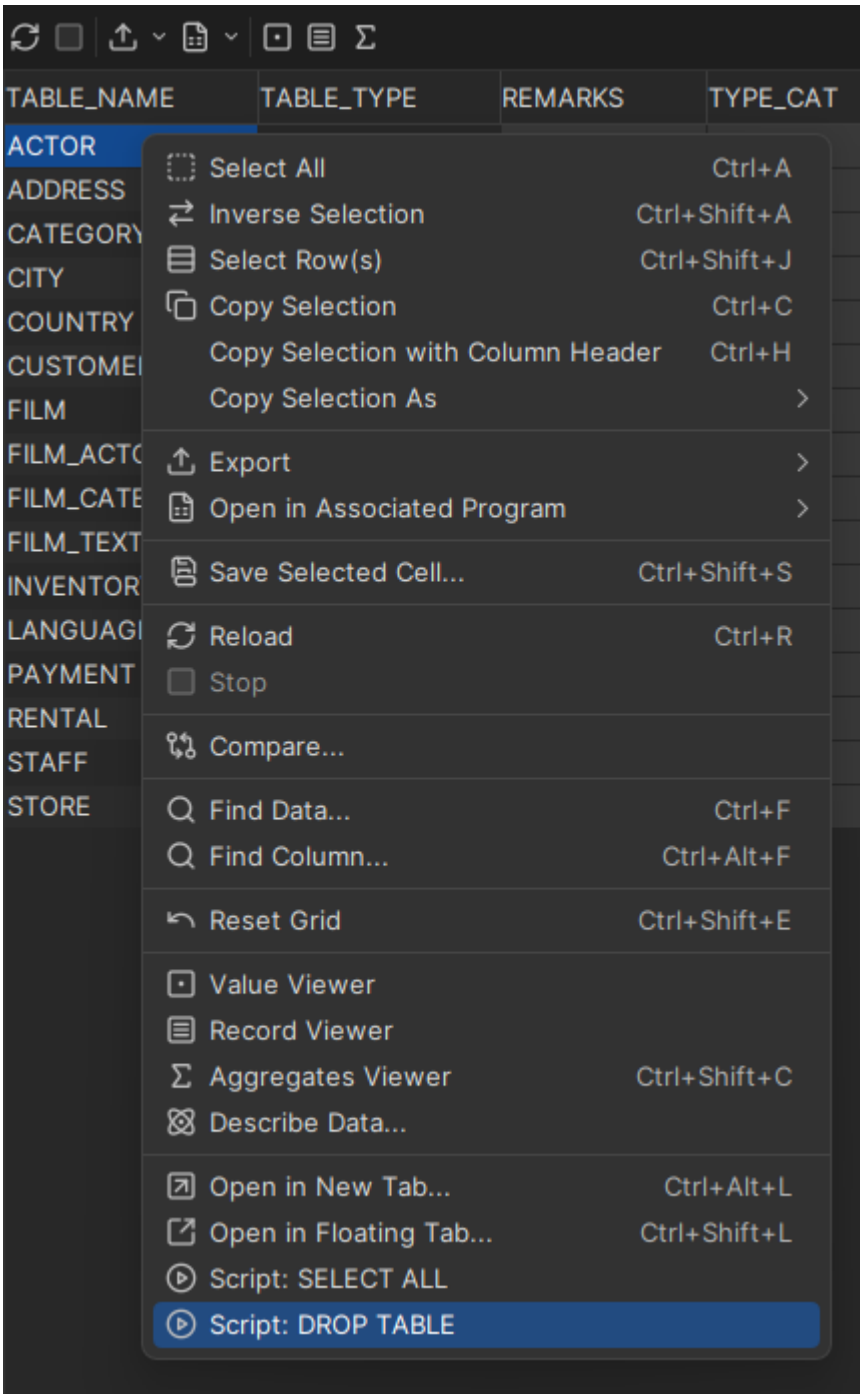
The **name="action"** attribute declares that the value is a pre-defined action. Valid actions are:

- **open-object-in-new-tab-command**
- **open-object-in-floating-tab-command**

Any variables in the SQL statement should identify column names in the result set. The user may select any cells in the grid and choose a custom menu item. It is only the actual rows that are picked from the selection as the columns are predefined by the menuItem declaration.

The variables specified in these examples start with **${schema=...}** and **${object=...}**. These define that the first variable represents a schema variable while the second defines an object. This is needed for DbVisualizer to determine whether delimited identifiers should be used and if identifiers should be qualified, as defined in the connection properties for the database.

Here is a sample:




Setting initial max column width

Some result sets may contain wide columns. The following parameter sets an initial maximum width for all columns in the grid.

```
<Input name="columnWidth" value=""/>
```

Viewer - message

The message viewer is really simple as it just shows a message in the DataView.

```
<DataView id="mimer-auto-properties" icon="properties" label="Mimer SQL Auto"
viewer="message">
   <Message>
      <![CDATA[
<html>
Mimer SQL Auto database server type has no capabilities to list the contents of
the database schema. <br/>
Please enter the precompiled statement names in the command window to perform
operations
towards the database.
</html>
      ]]>
   </Message>
</DataView>
```

It may also be useful to define viewers based on conditions such as in this example:

```
<If test="#ROLE_IS_SECURITY_ADMIN eq 1">
   <DataView id="snowflake-users-users" icon="Users" label="Users" viewer="grid">
      <Command idref="snowflake.getUsers">
         <Input name="catalog" value="${catalog}"/>
      </Command>
   </DataView>
</If>
<Else>
   <DataView id="snowflake-users-users" icon="Users" label="Users"
viewer="message">
      <Message>
         <![CDATA[
<html>
Insufficient privileges. SECURITYADMIN Role is required to view Users.
</html>
         ]]>
      </Message>
   </DataView>
</Else>
```

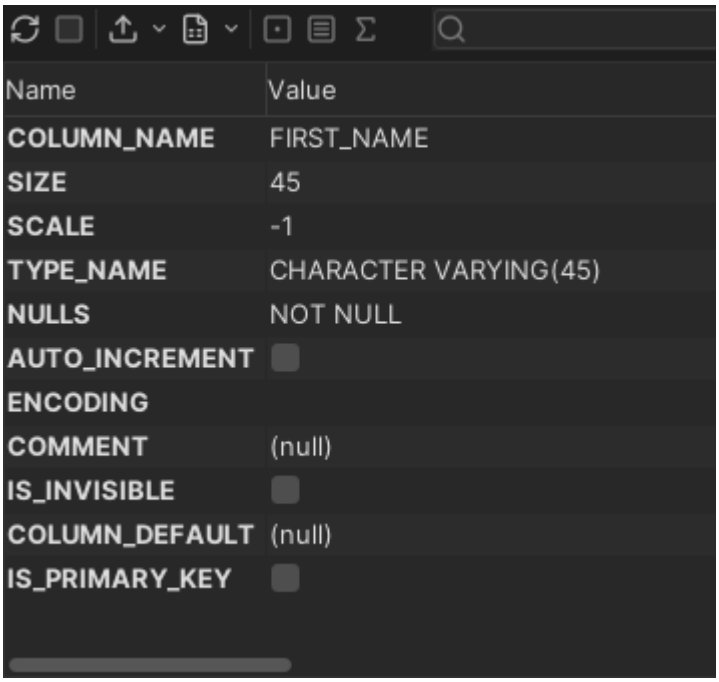
Viewer - navigator

The **navigator** viewer defines the Navigator tab for table objects in the DbVisualizer UI.

```
<DataView id="generic-table-navigator"
          icon="Navigator" label="Navigator" iewer="navigator"/>
```

Viewer - node-form

The **node-form** viewer presents all data associated with the selected DataNode (variables). Here is a sample of the XML for the node-form viewer:




Hiding columns

There may be data associated with the object that you don't want to present in the node form. The **hidecolumn** input parameter control what data for the object that should be invisible, and you may repeat this option as many times you like to handle multiple variables that shouldn't be displayed.

```
<Input name="hidecolumn"
       value="oracle.getKeys.TABLE_OWNER"/>
```

Viewer - node-text

The **node-text** viewer presents data from **a single cell i**n a DataNode in a text browser (read-only).

This viewer is typically used to present large chunks of data, such as source code, SQL statements, etc. where the data is already loaded and no queries need to be executed.

```
<DataView id="presto-base-queries-queries-query"
    icon="Query" label="Query" viewer="node-text">
   <Input name="dataColumn" value="text"/>
   <Input name="formatSQL" value="true"/>
</DataView>
```

The node text viewer requires the name of the column to be specified.

```
<Input name="dataColumn" value="query"/>
```

Enable SQL formatting of the data

The node text viewer has the **SQL Formatting** function, which when invoked formats the SQL buffer in the viewer. The **formatSQL** input parameter is used to control whether formatting should be made automatically when the data first displayed. If formatSQL is not specified, no initial formatting is made.

```
<Input name="formatSQL" value=""/>
```

Viewer - procedure-editor

The **procedure-editor** is special as it requires support in the DbVisualizer codebase to work properly. When supported it will open the procedure editor for the selected code object such as procedure, function, package, package body, etc.

```
<DataView id="mimer-function-functioneditor" classargs="FUNCTION"
icon="SourceEditor" label="Function Editor" viewer="procedure-editor"/>
```

Viewer - table-data

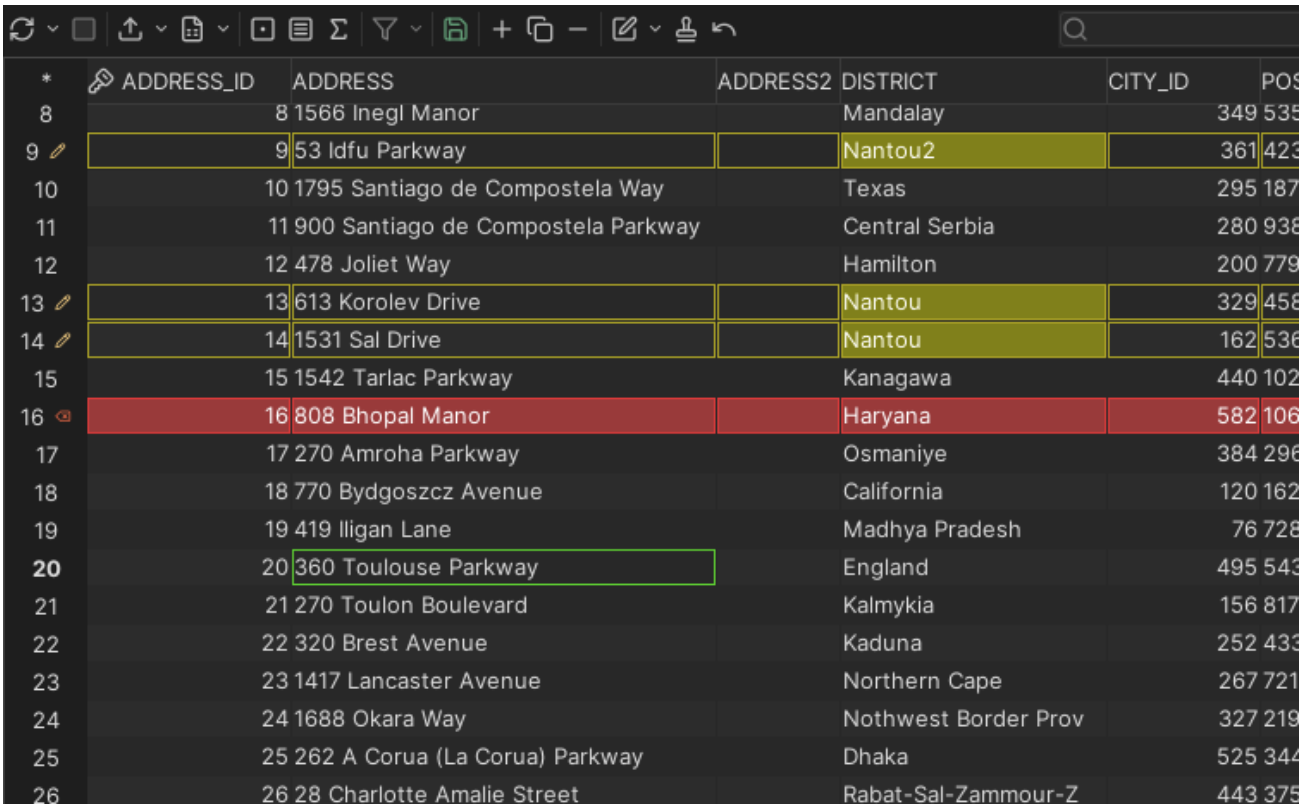
The **table-data** viewer shows the data for a table in a grid with various features such as filtering and editing (if licensed) functionality. This viewer is referred as Data tab in the DbVisualizer UI.

> Information presented in the grid is obtained automatically by the viewer via a standard **SELECT * FROM table** statement, i.e., the object type having this viewer defined must be able to support getting a result set via this SQL statement.

It is important that the **Database Type** in the connection setup is properly set to match the database being accessed. The reason is that the identifiers ( schema, database, table) are delimited automatically. Delimiters are database-specific and if having the wrong database type set it may result in an error getting the result. Here is a sample of the XML for the table-data viewer:

```
<DataView id="h2-view-data" icon="Data" label="Data" viewer="table-data"/>
```

And here is a screenshot of the **Data tab** based on the previous definition.



| * | ADDRESS_ID | ADDRESS | ADDRESS2 | DISTRICT | CITY_ID | POS |
|---|---|---|---|---|---|---|
| 8 | 8 | 1566 Inegl Manor | | Mandalay | 349 | 535 |
| 9 | 9 | 53 Idfu Parkway | | Nantou2 | 361 | 423 |
| 10 | 10 | 1795 Santiago de Compostela Way | | Texas | 295 | 187 |
| 11 | 11 | 900 Santiago de Compostela Parkway | | Central Serbia | 280 | 938 |
| 12 | 12 | 478 Joliet Way | | Hamilton | 200 | 779 |
| 13 | 13 | 613 Korolev Drive | | Nantou | 329 | 458 |
| 14 | 14 | 1531 Sal Drive | | Nantou | 162 | 536 |
| 15 | 15 | 1542 Tarlac Parkway | | Kanagawa | 440 | 102 |
| 16 | 16 | 808 Bhopal Manor | | Haryana | 582 | 106 |
| 17 | 17 | 270 Amroha Parkway | | Osmaniye | 384 | 296 |
| 18 | 18 | 770 Bydgoszcz Avenue | | California | 120 | 162 |
| 19 | 19 | 419 Iligan Lane | | Madhya Pradesh | 76 | 728 |
| 20 | 20 | 360 Toulouse Parkway | | England | 495 | 543 |
| 21 | 21 | 270 Toulon Boulevard | | Kalmykia | 156 | 817 |
| 22 | 22 | 320 Brest Avenue | | Kaduna | 252 | 433 |
| 23 | 23 | 1417 Lancaster Avenue | | Northern Cape | 267 | 721 |
| 24 | 24 | 1688 Okara Way | | Nothwest Border Prov | 327 | 219 |
| 25 | 25 | 262 A Corua (La Corua) Parkway | | Dhaka | 525 | 344 |
| 26 | 26 | 28 Charlotte Amalie Street | | Rabat-Sal-Zammour-Z | 443 | 375 |

Disable data editing

The default strategy for the **table-data** viewer is to automatically check whether the data can be edited or not. If editing is allowed a few related buttons will appear in the toolbar. However, sometimes you may want to disable editing completely for the **table-data** viewer. Do this with the following input element:

```
<DataView ...>
   <Input name="editDisabled" value="true"/>
</DataView>
```

Trim cell values

Data is presented as-is as retrieved from the database. Some databases return whitespace padded data which may not be desired. Use the following input element to trim (remove leading and trailing whitespaces) the data in every cell:

```
<DataView ...>
   <Input name="trimCellValues" value="true"/>
</DataView>
```

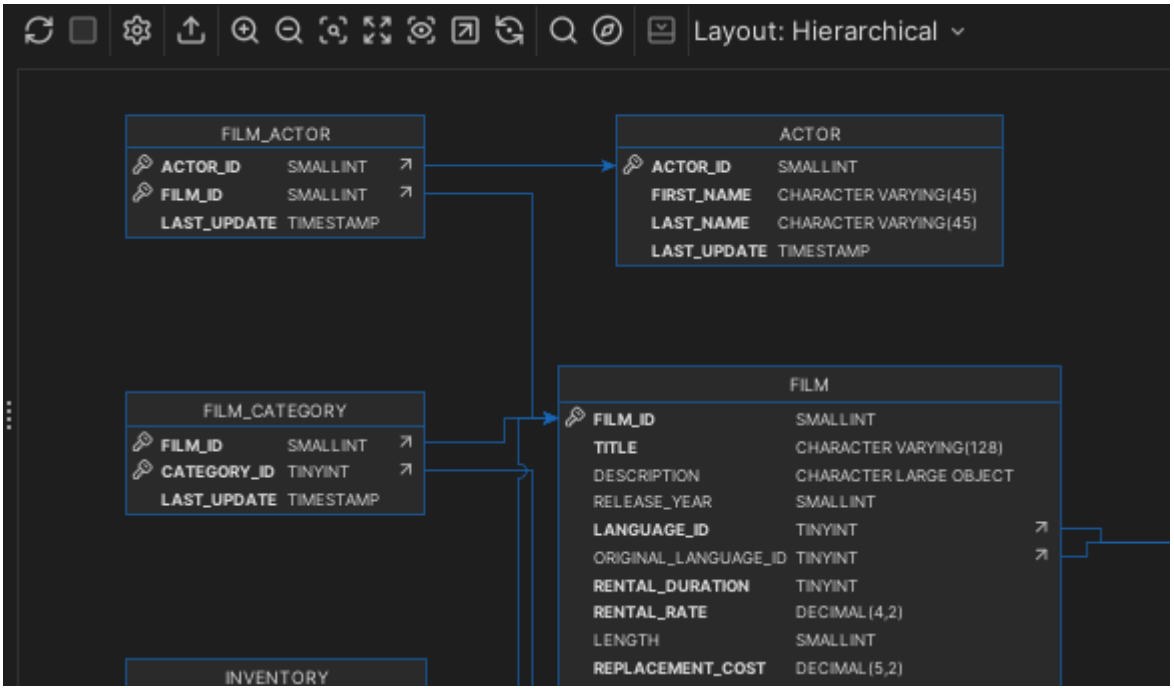
Viewer - table-refs

The **table-refs** viewer shows the reference graph for the current object (this must be an object supporting referential integrity constraints, such as a Table).

Here is a sample of the XML for the table-refs viewer:

```
<DataView id="generic-table-references" icon="References" label="References"
viewer="table-refs"/>
```

And here is a screenshot of the References tab based on the previous definition.

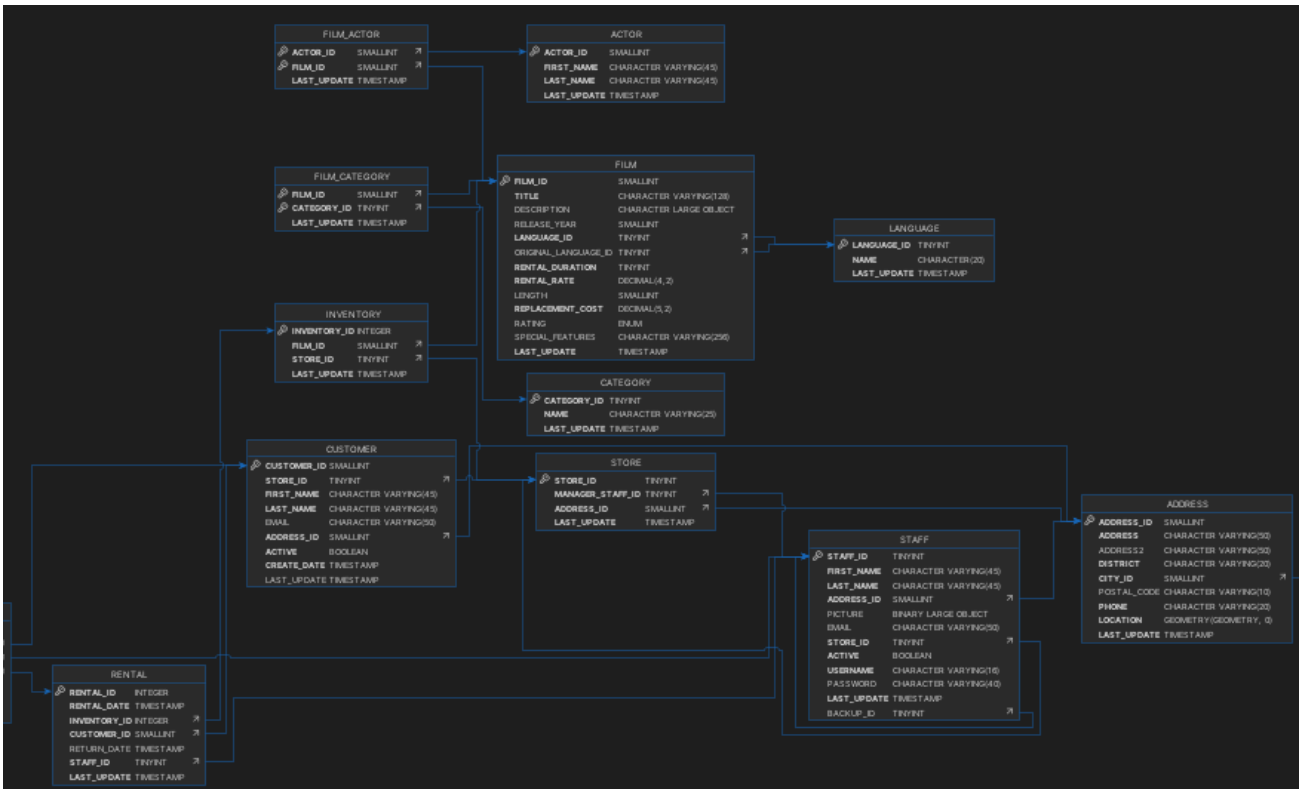



Viewer - tables-refs

The **tables-refs** viewer shows the references graph for several tables in the result set (the result set must contain objects supporting referential integrity constraints, such as a Table). Here is a sample of the XML for the tables-refs viewer:

```
<DataView id="oracle-tables-references" icon="References" label="References"
viewer="tables-refs">
   <Command idref="oracle.getTables">
      <Input name="owner" value="${schema}"/>
      <ProcessDataSet action="renamecolumn" index="OWNER" name="TABLE_SCHEM"/>
      <ProcessDataSet action="renamecolumn" index="TABLE_NAME" name="TABLE_NAME"/>
   </Command>
</DataView>
```

And here is a screenshot of the References tab based on the previous definition.




Viewer - table-rowcount

The table-rowcount viewer shows the row count for a (table) object.

> The row count is obtained automatically by the viewer via a traditional **SELECT COUNT(*) FROM table** statement, i.e., the object type having this viewer defined must be able to support getting a result set via this SQL statement.

It is that the **Database Type** in the connection setup is properly set to match the database being accessed. The reason is that the identifiers (schema, database, table) are delimited automatically. Delimiters are database-specific, and if having the wrong database type set it may result in an error getting the result. Here is a sample of the XML for the table-rowcount viewer:

```
<DataView id="generic-table-rowcount" icon="RowCount" label="Row Count"
viewer="table-rowcount"/>
```

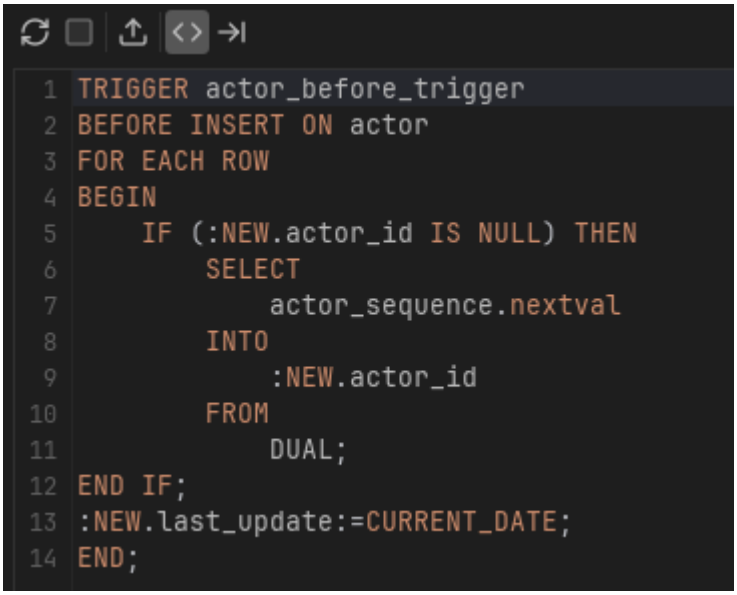
Viewer - text

The **text** viewer presents data from **one column** in a result set in a text browser (read only). This viewer is typically used to present large chunks of data, such as source code, SQL statements, etc. If the result set contain several rows, the text viewer reads the data in the column for each row and present the combined data.

Here is a sample of the XML for the text viewer:

```
<DataView id="oracle-table-triggers-source" icon="Source" label="Source"
viewer="text">
   <Input name="dataColumn" value="text"/>
   <Input name="formatSQL" value="true"/>
   <Input name="newline" value""/>
   <Command idref="oracle.getTriggerSource">
      <Input name="owner" value="${OWNER}"/>
      <Input name="name" value="${TRIGGER_NAME}"/>
   </Command>
</DataView>
```

And here is a screenshot of the Source tab based on the previous definition.



```
1  TRIGGER actor_before_trigger
2  BEFORE INSERT ON actor
3  FOR EACH ROW
4  BEGIN
5      IF (:NEW.actor_id IS NULL) THEN
6          SELECT
7              actor_sequence.nextval
8          INTO
9              :NEW.actor_id
10         FROM
11             DUAL;
12 END IF;
13 :NEW.last_update:=CURRENT_DATE;
14 END;
```

Specify what column to browse

By default, the text viewer uses the data in the first column. This behavior can be controlled by using the dataColumn input parameter. Specify the name of the column in the result set or its index (starting at 1 from the left).

```
<Input name="dataColumn" value=""/>
```

Enable SQL formatting of the data

The text viewer has the **SQL Formatting** function, which when invoked formats the SQL buffer in the viewer. The **formatSQL** input parameter is used to control whether formatting should be made automatically when the data first displayed. If formatSQL is not specified, no initial formatting is made. Valid values are: true, false or disable. "disable" will disable the formatting option in the user interface.

```
<Input name="formatSQL" value=""/>
```

Adding newline to each row

For a result set containing multiple rows and all rows should be displayed in a text viewer, the **newline** parameter define the character(s) that should separate the rows in the viewer. A **\n** somewhere in the value will be converted to a platform-dependent newline sequence in the viewer. By default, there is no newline sequence between multiple rows.

```
<Input name="newline" value="\n"/>
<Input name="newline" value="as-sql"/>
```

The **as-sql** value will append an SQL statement delimiter and a platform-specific newline sequence at the end of every row.

XML ELEMENT - COMMAND

Check the commands section for more information.

XML ELEMENT - INPUT

The **Input** element is supported for some viewers. Check the viewer sections for more information.

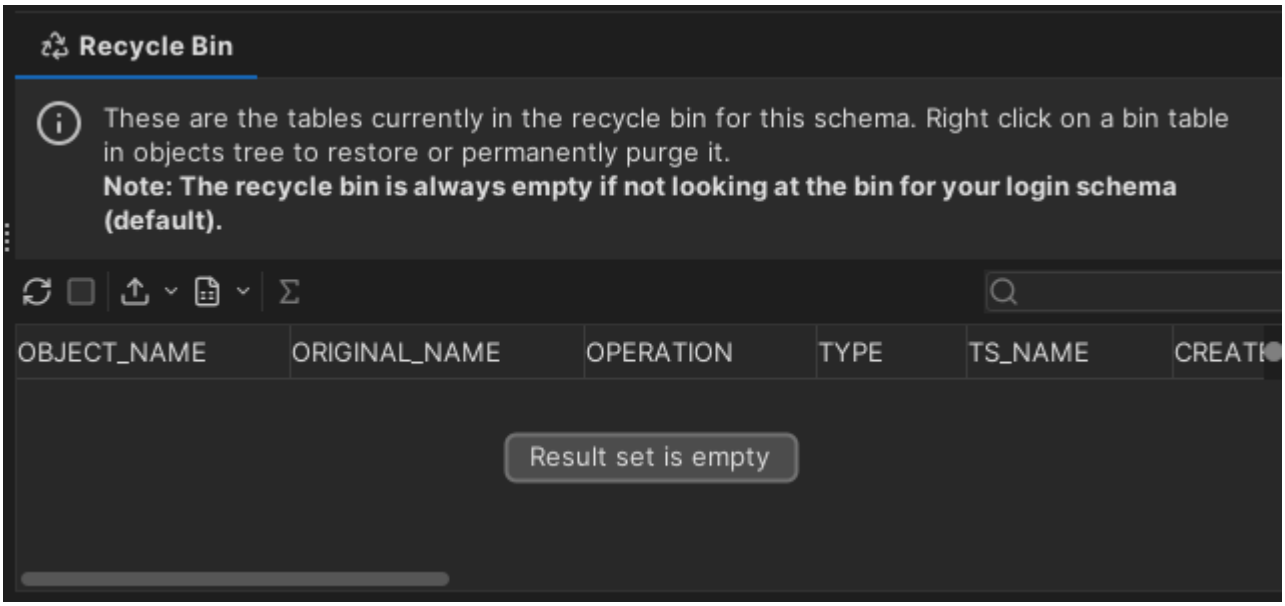
XML ELEMENT - MESSAGE

The **Message** element is basic as it just defines a message that should appear at the top of the viewer. The text in the message may contain HTML tags such as <b> (bold), <i> (italic), <br> (line break), etc.

Here is a sample of the XML for using the message element in a grid viewer:

```
<ObjectView type="RecycleBin">
   <DataView id="oracle-recyclebin-recyclebin" icon="RecycleBin" label="Recycle
Bin" viewer="grid">
      <Command idref="oracle.getRecycleBin">
         <Input name="schema" value="${schema}"/>
         <Input name="login_schema" value="${dbvis-defaultCatalogOrSchema}"/>
      </Command>
      <Message>
         <![CDATA[
<html>
These are the tables currently in the recycle bin for this schema. Right click on
a bin
table in objects tree to restore or permanently purge it.
<b>Note: The recycle bin is always empty if not looking at the bin for your
login schema (default).</b>
</html>
         ]]>
      </Message>
   </DataView>
</ObjectView>
```

And here is a screenshot of the **Recycle Bin tab** based on the previous definition.


Recycle Bin
These are the tables currently in the recycle bin for this schema. Right click on a bin table in objects tree to restore or permanently purge it.
Note: The recycle bin is always empty if not looking at the bin for your login schema (default).
OBJECT_NAME
ORIGINAL_NAME
OPERATION
TYPE
TS_NAME
CREATI
Result set is empty

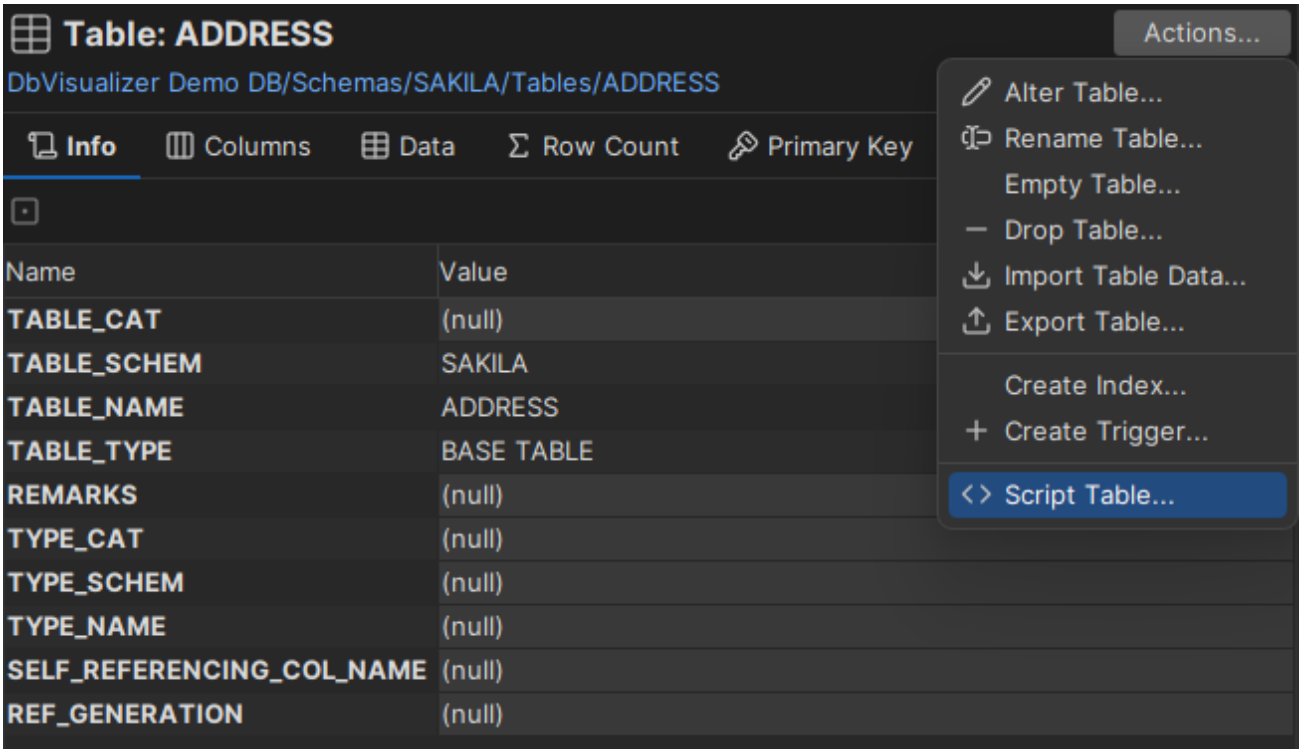
# XML element - ObjectsActionDef

## Introduction

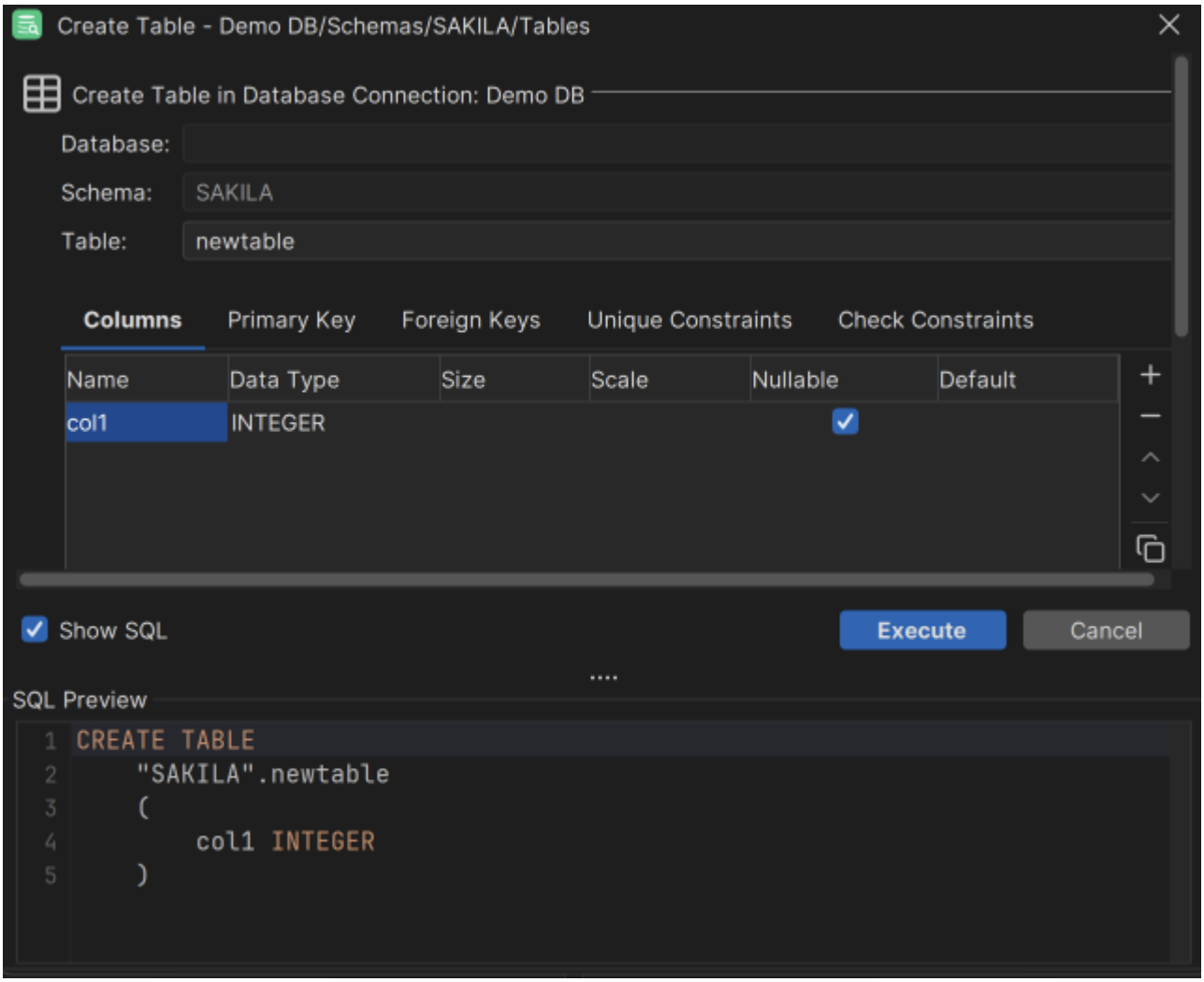
Objects actions (ObjectsActionDef) define what operations are available for the object types defined in the **ObjectsTreeDef**. Object actions are powerful, as they offer an extensive number of features to define actions for almost any type of object operation.

In DbVisualizer, the object actions menu is accessed via the right-click menu in the **Databases** tab or via the **Actions** button in the object view.

> The right-click menu in the Databases tab is context sensitive meaning that the listed actions in the menu depends on the selected object type.




All of the operations for the current **Table** object in the figure above are expressed in the ObjectsActionDef section in the database profile. The implementation for these actions are either declared entirely in XML via standard definitions, or via custom definitions. (The Java API for action handlers is not yet documented). The following screenshot shows the dialog appearing when executing an action via a standard XML definition:




The first field in the dialog, **Database Connection**, is always present and shows the alias of the database connection the current object is associated with. At the bottom, there is a **Show SQL** control that, when checked, displays the final SQL for the action. The bottom right buttons are used to run the action ( the label of the button may be **Execute** or **Script** based on the action mode), or to **Cancel** the action completely.

## Variables

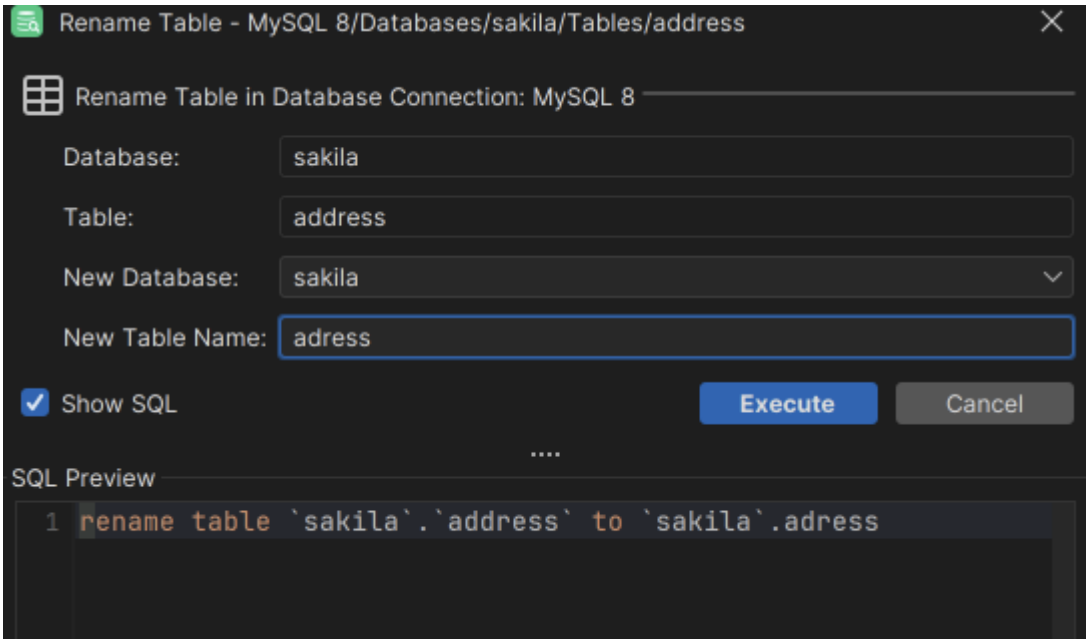
Variables are used to reference data for the object for which the action was launched, and the data for all its parent objects in the objects tree. Variables are also used to reference input data specified by the user in the actions dialog. Variables are typically used in the **Command**, **Confirm**, **Result** and **SetVar** elements.

Variables are specified in the following format:

**${variableName}**

The following is an example for a **Rename Table** action. It first shows the name of the database connection (which is always present) with information about the table being renamed. The last two input fields should be entered by the user and identify the new name of the table. The **New Database** component is a list from which the user should select the name of the new database. The new table name should be entered in the **New Table Name** field.

If the **Show SQL** control is checked, you will see any edits in the dialog being reflected immediately in the final **SQL Preview**.



The complete action definition for the previous **Rename Table** action is as follows:

```
<Action id="mysql-table-rename" label="Rename Table" reload="true" icon="rename">
  <Input label="Database" style="text" editable="false">
    <Default>${catalog}</Default>
  </Input>

  <Input label="Table" style="text" editable="false">
    <Default>${objectname}</Default>
  </Input>

  <Input label="New Database" name="newCatalog" style="list">
    <Values>
      <Command><SQL><![CDATA[show databases]]></SQL></Command>
    </Values>
    <Default>${catalog}</Default>
  </Input>

  <Input label="New Table Name" name="newTable" style="text"/>

  <Command>
    <SQL>
      <![CDATA[
rename table `${catalog}`.`${objectname}` to `${newCatalog}`.`${newTable}`
      ]]>
    </SQL>
  </Command>

  <Confirm>
    <![CDATA[
Confirm rename of ${catalog}.${objectname} to ${newCatalog}.${newTable}?
    ]]>
  </Confirm>

  <Result>
    <![CDATA[
Table ${catalog}.${objectname} renamed to ${newCatalog}.${newTable}!
    ]]>
  </Result>
</Action>
```

First, there is the **Action** element with some attributes specifying the label of the action, icon and whether the objects tree (and the current object view) should be reloaded after the action has been executed.

The next block of elements are **Input** fields defining the data for the action. As you can see in the example, there is a * ${catalog} variable in the Default element for the Database input and an ${objectname} variable in the Default element for the Table input. The values for these variables are fetched from the current object in the objects tree (GroupNode or DataNode). Variables are evaluated by first checking if the variable is in the scope of the action dialog (i.e., another input field), then if the variable is defined for the object for which the action was launched, and then if it is defined for any of the parent objects until the root object in the tree (Connections node) is reached. If a variable is not found, its value is set to (null)*.

In the XML sample, the value of the **${catalog}** variable is the name of the database in which the table object is stored. The **${objectname}** is the current name of the table (these variables are described in the ObjectsTreeDef section).

The **New Database** input field is a list component showing a list of databases based on the result set of the specified **SQL** command. The **Default** setting for the database will be the database in which the table is currently stored based on the **${catalog}** variable.

The **New Table Name** input field is a simple text field in which the user may enter any text (the new table name).

Both the **New Database** and **New Table Name** fields are editable and should be specified by the user. This data is then available via the variables specified in the name attribute, i.e., **newCatalog** and **newTable**.

The **Command** element declares the **SQL** statement that should be executed by the action. In this example, the SQL combines static text with variables.

## XML element - ActionGroup

The **ActionGroup** element is a container and groups a collection of **ActionGroup**, **Action** and **Separator** elements. It is used to define what actions should be present for a particular object type. It also define in what order the actions should appear in the menu and where any separators should be located. ActionGroup elements can be nested and these will be displayed as sub menus in DbVisualizer.

```
<ActionGroup type="Table">
   <Action id="xxx">
      ...
   </Action>
</ActionGroup>
```

The attributes for an ActionGroup are:

| Attribute | Value | Description |
|---|---|---|
| type | | This defines what object type the ActionGroup is mapped to. This attribute is required and valid only for top level ActionGroup elements (not nested ActionGroup elements). An example is the **Table** object type, the corresponding **<ActionGroup type="Table">** will only be displayed when the current object is a Table |
| label | | This attribute is required for nested ActionGroup elements and is the label displayed in the sub menu. (This attribute have no effect on top level ActionGroup elements) |
| drop-on-condition | | Read more in **drop-on-condition attribute** |
| order-before | | Specifies the order of this ActionGroup among a collection of ActionGroup elements located at the same level. It can either be an index starting at 0 (first) or a node type. Ex. **order-before="Views"** will order this ActionGroup before ActionGroup elements defined by the **type="Views"** attribute |
| order-after | | Specifies the order of this ActionGroup among a collection of ActionGroup elements located at the same level. It can either be an index starting at 0 (first) or a node type. Ex. **order-after="Views"** will order this ActionGroup after ActionGroup elements defined by the **type="Views"** attribute |

## XML element - Action

The **Action** element defines the characteristics of the action. The following show the complete definition of the **Drop Table** action in Oracle.

```
<Action id="oracle-table-drop" label="Drop Table" reload="true" icon="remove">
   <Input label="Schema" style="text" editable="false">
      <Default>${schema}</Default>
   </Input>
   <Input label="Table" style="text" editable="false">
      <Default>${objectname}</Default>
   </Input>
   <Input label="Drop Referential Integrity Constraints" name="cascade"
style="check"
      tip="Enable this to drop all referential integrity constraints
               that refer to primary and unique keys in the dropped table">
      <Values>cascade constraints</Values>
   </Input>
   <If test="#util.isDatabaseVersionGTE(10)">
      <Input label="Purge Space" name="purge" style="check"
         tip="Enable this if you want to drop the table and
              release the space associated with it in a single step">
         <Values>purge</Values>
      </Input>
   </If>
   <Else>
      <SetVar name="purge" value="''"/>
   </Else>
   <Command>
      <SQL>
         <![CDATA[drop table "${schema}"."${objectname}" ${cascade} ${purge}]]>
      </SQL>
   </Command>
   <Confirm>
      Really drop table ${schema}.${objectname}?
   </Confirm>
   <Result>
      Table ${schema}.${objectname} has been dropped!
   </Result>
</Action>
```

The available attributes for the **Action** element:

| Attribute | Value (bold = default) | Description |
| --- | --- | --- |
| **id** | | Every Action element must have a unique id which is not only unique in the current profile but also with all id's in extended profiles.<br>The recommended format is **profileName-actionGroupType-action**. Ex: **oracle-table-drop** |
| | | |
| icon | | The name of the icon that should be displayed next to the label in the actions menu |
| label | | The label for the action as it should be displayed in the list of actions and in the actions dialog |
| reload | true/**false** | Specifies if the parent node (in the objects tree) should be reloaded after successful execution. This is recommended for actions that change the visual appearance of the object, such as remove, add or name change |
| mode | • script<br>show the action dialog, process user input and send the final SQL to the SQL Commander without executing the command<br>• script-immediate<br>will not show the action dialog but instead pass the final SQL directly to the SQL Commander without executing the command | Specifies how the action will be prepared and displayed |
| resultaction | • mergeasscript<br>• mergeastext | • **mergeastext**<br>will merge multiple result sets to to a single result tab<br>• **mergeasscript**<br>will merge multiple result sets to a single result tab, each row will be terminated with a semi-colon (";") Default is that result sets are displayed in individual tabs |
| resulttarget | editor | Only applicable if the **resultaction** attribute is specified. With the value **editor** the merged results will be opened in a new SQL Commander tab |
| hideif | | There may be situations when an action should be dropped due to a condition. The hideif attribute is used to express a condition which is evaluated when the list of actions is created. Example: |

| Attribute | Value (bold = default) | Description |
| --- | --- | --- |
| | | **hideif="#dataMap.get('actionlevel') neq 'toplevel'"** |
| resetcatalogs | true/**false** | Setting this attribute to true will reset any cached databases for the actual database connection. Useful when for example the action create, rename or delete a database |
| resetschemas | true/**false** | Setting this attribute to true will reset any cached schemas for the actual database connection. Useful when for example the action create, rename or delete a schema |
| supportsmultipleobjects | **true**/false | An action support processing multiple objects if the style attribute for all input elements is one of: - check - check-list - list - radio - separator - node - label - read-only text The **supportsmultipleobjects="true"** attribute is used to disable multi object processing even if the previous criteria is satisfied |
| class | | Used to specify a custom Java class used as the action |
| classargs | | Used to pass arguments to a custom action |
| doclink | | Relative HTML link to the related chapter in the user guide |
| drop-on-condition | | Read more in **drop-on-condition attribute** |
| order-before | | Specifies the order of this Action among a collection of Action elements located at the same level. It can either be an index starting at 0 (first) or a node type. Ex. **order-before="View"** will order this Action before Action elements defined by the **type="View"** attribute |
| order-after | | Specifies the order of this Action among a collection of Action elements located at the same level. It can either be an index starting at 0 (first) or a node type. Ex. **order-after="View"** will order this Action after Action elements defined by the **type="View"** attribute |

XML ELEMENT - INPUT

An Input element specifies the characteristics of a field component in the actions dialog. The **label** attribute is recommended and is presented to the left of input field. If a label is not specified, the input field will occupy the

complete width of the action dialog. All input fields are editable by default. The **name** attribute is required for editable fields and should specify the name of the variable in which the user input is stored.

| Attribute | Value (bold = default) | Description |
|---|---|---|
| label | | The label for the input component |
| name | | For editable input this should be the name of the variable holding the value specified by the user |
| tip | | Message displayed when hovering over the component |
| editable | **true**/false | Enables or disables editing of the component |
| linebreak | true/**false** | If set to true, no line break will be made after the input component. This is useful when for example having multiple **<Input style="check">** elements in a single row |
| style | • list<br>• radio<br>• **text**<br>• check<br>• check-list<br>• password<br>• number<br>• text-editor<br>• wrapped-text-editor<br>• grid<br>• separator<br>• label<br>• note | The style of the input element. See following sections for more details |
| hideif | | There may be situations when an **Input** element should be dropped due to a condition. The hideif attribute is used to express a condition which is evaluated when the action is initialized. Example: **hideif="#dataMap.get('actionlevel') neq 'toplevel'"** |
| runsetdefaultwhenvaluechanged | | The **runsetdefaultwhenvaluechanged** attribute defines what other inputs default command should be triggered when the value for the input is changed |

This is a minimal definition of an input field. It will show a **read-only text** field control labeled Size.

```
<Input label="Size" editable="false"/>
```

If the input field is changed to be **editable**, the **name** attribute must be used to specify the identifier for the **variable name**.

```
<Input label="Size" editable="true" name="theSize"/>
```

Any input element may contain the **tip** attribute. It is used to briefly document the purpose of the input field and is displayed as a tooltip when the user hover the mouse pointer over it.

```
<Input label="Size" editable="true" name="theSize" tip="Please enter the size of
the new xxx"/>
```

The **hideif** attribute is useful to limit what input fields should appear for an action. The condition specified in the hideif attribute have the same syntax as described in the <SetVar> section. Example:

```
<Input label="Unit" hideif="#dataMap.get('actionlevel') neq 'toplevel'">
```

Input fields can be aligned on a single row with the **linebreak** attribute. The default behavior is that every input field is displayed on a single row. Use the **linebreak="false"** attribute to define that the next input field will be arranged on the same line. To re-start the automatic line breaking feature you must use the **linebreak="true"** attribute.

```
<Input name="size" label="Size" style="number" linebreak="false">
  <Default>10</Default>
</Input>
<Input name="unit" label="Unit" style="list" linebreak="true">
  <Labels>KB|MB</Labels>
  <Values>K|M</Values>
  <Default>M</Default>
</Input>
```

The previous example show the use of the linebreak attribute. The **Size** number field and the **Unit** list will appear in the same row.

Specifying the **default value** as a result from an SQL statement is a trivial task:

```
<Input label="Size" editable="true" name="theSize">
   <Default>
      <Command>
         <SQL>
select size from systables where tablename = '${objectname}'
         </SQL>
      </Command>
   </Default>
</Input>
```

The **Default** definition above will execute a **SQL** statement, it will automatically pick the value in the first row's first column and present it as the default value for the input component. SQL may be specified in the Default element for all styles while SQL in **Values** and the **Labels** elements are * valid only for list, radio, and check styles. In some rare situations it may not be possible to express a SQL statement that will return a single column that should be displayed

for Values, Labels and Default. An example is when data is collected via a stored procedure. To solve this problem specify the column attribute. Its value must be a column name or column index* in the result set.

**column index** (**column="2"** attribute):

```
<Input label= Size" editable="true" name="theSize">
   <Default column="2">
      <Command idref="getSize">
         <Input name"objectname" value="${objectname}"/>
      </Command>
   </Default>
</Input>
```

or by **column name** (**column="THE_SIZE"** attribute):

```
<Input label= Size" editable="true" name="theSize">
   <Default column="THE_SIZE>
      <Command idref="getSize">
         <Input name"objectname" value="${objectname}"/>
      </Command>
   </Default>
</Input>
```

An alternative to embedding the SQL in the element body, as in one of the previous examples, is to refer to a command via the standard **idref** attribute:

```
<Input label= Size" editable="true" name="theSize">
   <Default>
      <Command idref="getSize">
         <Input name"objectname" value="${objectname}"/>
      </Command>
   </Default>
</Input>
```

Instead of having duplicated SQLs in multiple actions, consider using **<Command idref="xxx">** elements instead.

> Referring commands in actions via the idref attribute is recommended when the same SQL is used in several actions. Use Input elements for the Command to pass parameters to the command.

The following sections presents the **supported styles** that can be used in the Input element.

Style - check (true/false, on/off, selected/unselected)

The **check** style is suitable for **yes/no**, **true/false**, **here/there** types of input. Its checked state indicates that the **Value** for the input will be set in the final variable. If the check box is unchecked, the variable value is blank.

```
<Input label="Cascade Constraints" name="cascade" style="check">
   <Values>compact</Values>
</Input>
```

- This will create a check component with the label **Cascade Constraints**
- Checking the check box will set the value of the variable identified by name (cascade) to the value of **Value**, which is **compact**
- If the check box is unchecked, the variable value will be blank

Style - check-list (large number non exclusive choices)

The **check-list** style is much like the **list** style except that each choice have a check box in front allowing multiple values to be selected

```
<Input label="Weekdays" name="weekdays" style="check-list">
   <Arg name="output" value="jschours[${value}] = true"/>
   <Arg name="newline" value=", "/>
   <Labels>Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday</Labels>
   <Values>00|11|22|33|44|55|66</Values>
   <Default>22|44</Default>
</Input>
```

To produce a desirable output from the check-list it is possible to define how each choice should be generated in the final result. The **output** argument and its **value** attribute is used to define how each value should be generated. In the example each choice will be formatted as: **jschours[22] = true**. The * newline* argument defines any delimiter between each choice. If the user selects both Monday and Friday in the action window the following will be generated:

**jschours[00] = true, jschours[44] = true**

Style - grid (configurable multi row/columns input)

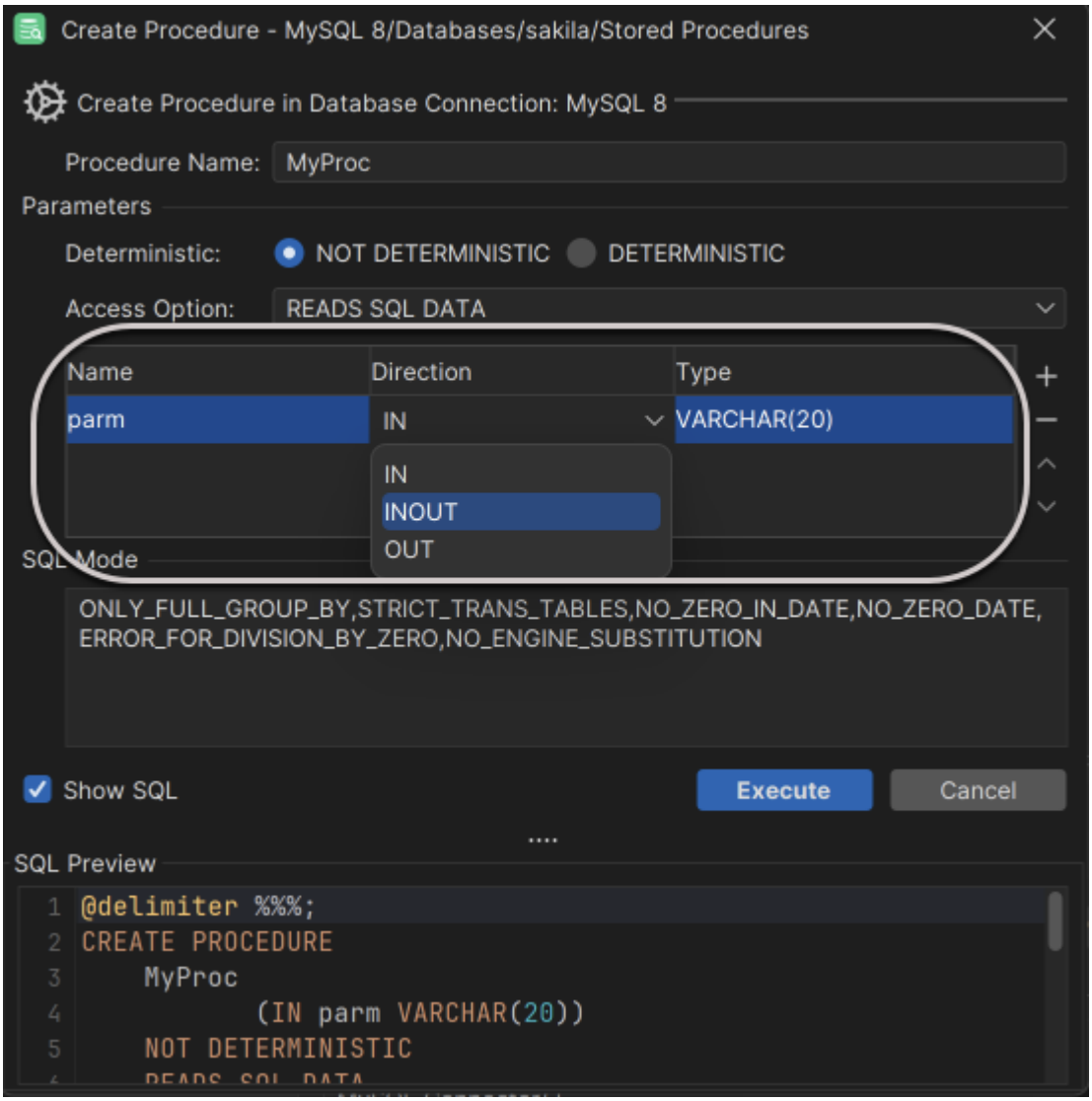
The **grid** input style is presented as a grid with user controls to **add**, **remove** and **move** rows. The columns that should appear in the grid are defined by using any of the primitive styles: **text**, **number**, **password**, **check**, **list** and **radio**. The grid style is useful for data that allows the user to define multiple entries. Examples are, defining columns that should appear in a table index, setup data files for a tablespace or databank.

This example show a grid style definition that will ask the user for parameters that will be part of a create procedure action.

```
<Input name="parameters" style="grid">
   <Arg name="output" value="${direction} ${name} ${type}${_default}"/>
   <Arg name="newline" value=", "/>

   <Input name="name" label="Name" style="text">
      <Default>p1</Default>
   </Input>
   <Input name="direction" label="Direction" style="list">
      <Values>IN|INOUT|OUT</Values>
      <Default>IN</Default>
  </Input>
  <Input name="type" label="Type" style="text">
      <Default>VARCHAR(20)</Default>
  </Input>
</Input>
```

Here is the result:




The sub elements for the grid style is different from the other input styles as it accepts nested **Input** elements. These input styles define what columns should appear in the grid and the first input style will appear to the leftmost and the last in the rightmost column.

This example doesn't specify the label attribute as we want the grid to extend the full width of the actions dialog. The grid style use the **Arg** elements to customize the appearance and function of the field. The following arguments are handled by the grid style:

- **output** Defines the output format for each row in the grid. The value may contain variables and static text. To create conditional output check the **SetVar** element below
- **newline** Defines the static text that should separate every row in the grid. A "\n" somewhere in the value will be converted to a newline sequence in the final output
- **rowprefix** Specifies any prefix for every row in the grid
- **rowsuffix** Specifies any suffix for every row in the grid

The resulting parameter list is created automatically by the control and is available in the variable name specified in the example to be parameters.

The **SetVar** element in the context of a grid style is used to process the data that will appear as defined by the **<Arg name="output">** element. It is used to process the data for every row in the grid. Let's say that the output must contain the word **" default "** if the value in a column named **Default ** is entered.** SetVar** is used to handle this:

```
<SetVar name="_default" value='#default.equals("") ? "" : " default " + #default'/
>
```

The **#default** input value is here evaluated and if it is not empty the **" default "** text s prefixed to the value of the #default value. The result is stored in the **_default** variable which is also refered in the output argument above.

Style - label

The **label** style is useful when the standard left aligned label should not be displayed. The following example will show seven check boxes each labeled individually and all displayed in the same row.

```
<Input style="label" linebreak="false"/>
<Input label="Mon" name="mon" style="check">
   <Values>mon</Values>
</Input>
<Input label="Tue" name="tue" style="check">
   <Values>tue</Values>
</Input>
<Input label="Wed" name="wed" style="check">
   <Values>wed</Values>
</Input>
<Input label="Thu" name="thu" style="check">
   <Values>thu</Values>
</Input>
<Input label="Fri" name="fri" style="check">
   <Values>fri</Values>
</Input>
<Input label="Sat" name="sat" style="check">
   <Values>sat</Values>
</Input>
<Input label="Sun" name="sun" style="check">
   <Values>sun</Values>
</Input>
```

Style - list (large number of exclusive choices)

The **list** style displays a list of choices in a drop-down component. Only one choice can be selected. The list can be editable, meaning that the field showing the selection may be editable by the user. Here is a sample XML for the list style.

```
<Input label="Select index type" name="type" style="list">
   <Values>Pizza|Pasta|Burger</Values>
   <Default>Pasta</Default>
</Input>
```

The **Values** element should, for static entries, list all choices separated by a **vertical bar (|)** character. A Default value can either list the name of the default choice or the index number (first choice starts at 0). In the example above, setting Default to **{2}** would set Burger to the default selection.

It is also possible to use the **Labels** element. If present, this should list all choices as they will appear in the actions dialog. Consider the following example and the labels shown to the user, while **Values** in this case should list the choices that will go into the final SQL via the variable.

```
<Input label="Select index type" name="type" style="list">
   <Values>Pizza|Pasta|Burger</Values>
   <Labels>Pizza the French style|Pasta Bolognese|Texas Burger</Labels>
   <Default>Pasta</Default>
</Input>
```

If the users selects **Texas Burger** then the value for variable type will be **Burger**.

The following show how to use SQL to feed the list of values:

```
<Input label="New Database" name="newCatalog" style="list">
   <Values>
      <Command>
         <SQL>
            <![CDATA[
show databases
            ]]>
         </SQL>
      </Command>
   </Values>
   <Default>${catalog}</Default>
</Input>
```

Here a **Command** element is specified as a sub element to **Values**. The result of the **show databases** SQL will be presented in the list component.

To make the list editable, specify the attribute **editable="true"**.

Style - note

The **note** style is just s simple label that can be put after for example input fields. Often used in combination with the **linebreak** attribute.

```
<Input label="Sleep Between Each Statement" name="sleep" style="number"
linebreak="false">
   <Default>1000</Default>
</Input>
<Input label="millis" style="note" linebreak="true"/>
```

Style - number

A **number** style is the same as text except that it only accept number values.

```
<Input label="Size" name="size" style="number" editable="true"/>
```

Style - password

A **password** field is the same as text except that it masks the value as ***.

```
<Input label="Password" name="pw" style="password" editable="true"/>
```

Style - radio (limited number of choices)

The **radio** style display a list of choices organized as button components. The only difference between the radio and list styles are:

- All choices for a radio style are displayed on the screen (better overview of choices but suitable only for a limited number of choices)
- The **<Arg name="direction" value="vertical"/>** element can be specified for radio style to present the radio choices vertically (default is horizontally)

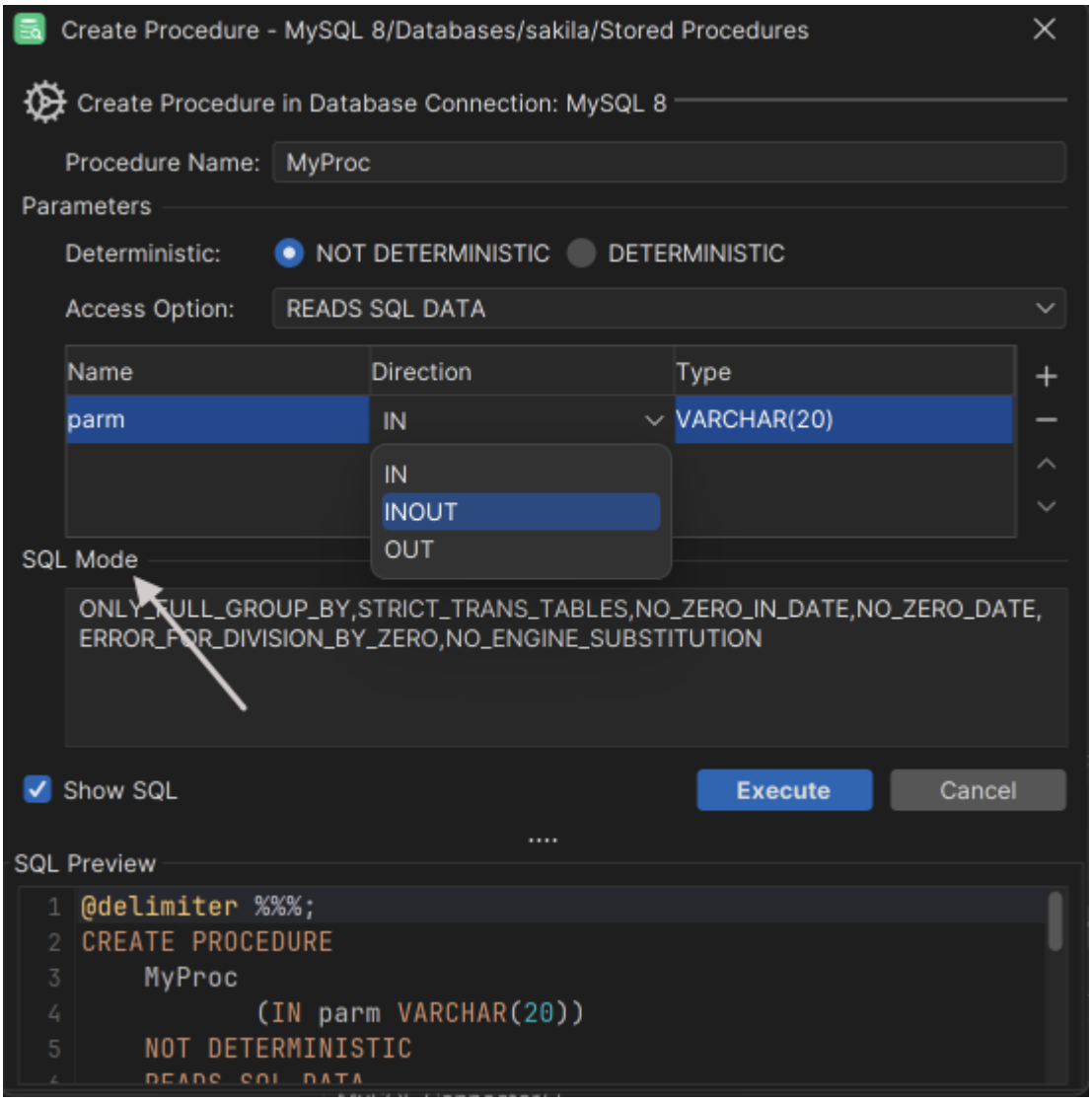
See the **list** style for complete capabilities of the radio style.

Style - separator (visual divider between input controls)

The **separator** style is not really an input element but is used to visually divide input components in the in the action dialog. If the **label** attribute is specified, it will be presented to the left of the separator line. If no label is specified, only the separator is displayed.

```
<Input label="SQL Mode" style="separator"/>
```

The separator is a useful substitute for the standard label presented to the left of every input field. Here is a sample:




The figure shows the use of separators and two fields that extend to the full width of the action dialog. The separators for **Parameters** and **SQL Mode** are here used as alternatives to labels for the fields below them.

**Style - note**

The **note** style is especially useful when a description or text should be anchored directly after another input.

```
<Input label="Sleep Between Each Statement" name="sleep" style="number"
linebreak="false">
   <Default>1000</Default>
</Input>
<Input label="millis" style="note" linebreak="true"/>
```

The previous sample looks like this:




Style - text (single line)

The **text** style is used to present single-line data in a text field.

```
<Input label="Enter your userid" name="userid" style="text">
   <Default>agneta</Default>
</Input>
```

- The optional **Default** element is used to define a default value for the field. Variables, static text and Command elements can be used to define the default value.
- A text input is editable by default. To make it read only specify **editable="false"**

Style - text-editor (multi line)

A **text-editor** field is the same as the text style except that it presents a multi-line field.

```
<Input label="Description" name="desc" style="text-editor" editable="true">
   <Arg name="height" value="30"/>
</Input>
```

The **Arg** element defines the height (in DLU) for the text-editor. The default height is 80 DLU's.

Style - wrapped-text-editor (multi line)

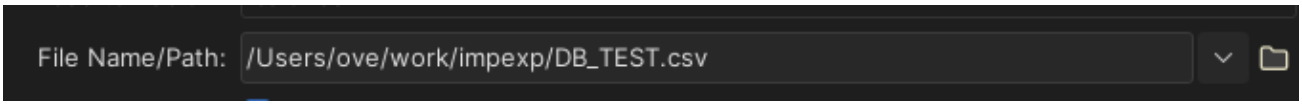
This is exactly the same as text-editor except that the wrapped-text-editor wraps long lines.

Style - file-chooser

A **file-chooser** field displays an editable text field with tools for file selection and recent files.

```
<Input label="File Name/Path" name="file" style="file-chooser">
    <Default></Default>
</Input>
```

The previous sample looks like this:




XML ELEMENT - SETVAR

The **SetVar** element is used to do conditional processing and create new variables based on the content of other variables or static text.

Consider an SQL statement for creating new users in the database:

```
create user 'user' identified by 'password'
```

In this case it is quite easy to map the user field to an **Input** element for the action since it is a required field. The question arise for password which is optional. The **identified by** clause should only be part of the final SQL if the password is entered by the user. The solution for this scenario is to use the **SetVar** element. Here is the complete action definition:

```
<Action id="mydb-user-create" label="Create User" reload="true" icon="add">
   <Input label="Userid" name="userid" style="text"/>
   <Input label="Password" name="password" style="password"/>

   <SetVar name="_password"
value='#password.equals("") ? "" : " identified by \"" + #password + "\""'/>

   <Command>
      <SQL>
         <![CDATA[
create user ${userid} ${_password}
         ]]>
      </SQL>
   </Command>
</Action>
```

The SetVar element accepts three attributes:

| Attribute | Description |
|---|---|
| name | The name of the new variable |
| value | Should contain the expression that will be evaluated. The expression is based on the OGNL toolkit. This is an expression library that mimics most of what is being supported by Java. Variables are referenced as **#variableName** |
| action | This attribute is optional and can have any of the following values: - **show** - default and indicates that the variable (and its value will appear in node-form viewers) - **hide** - the variable will not display in node-form viewers - **drop** - the SetVar element is not evaluated when actions is being processed - **runwheninit** - runs the SetVar only during initialization of the action (before window is displayed) |

The expression in the example above checks whether the password variable is empty. If it is empty, a blank value is being assigned to the **_password** variable. If it is not empty, the value for **_password** will be set to **identified by theEnteredPassword**.

The SQL in the Command element now refer the new **${_password}** variable instead of the original **${password}**.

It is recommended that variables produced via `SetVar` elements are prefixed with an underline (_) to highlight were they come from.

A SetVar having 'password' in its name attribute will be displayed as '***' in the SQL Preview pane.

XML ELEMENT - CONFIRM

The **Confirm** element is displayed to the user when a request to Execute the action is made. If there are only read-only input fields in the action, this message is displayed in the body of the action dialog. Otherwise the message is displayed in a confirmation dialog.

```
<Confirm>Really drop table ${table}?</Confirm>
```

Note that the message text can be composed of HTML tags such as <b>, <i>, <br>, etc.

XML ELEMENT - RESULT

The **Result** element is optional and if specified, it is shown in a dialog after successful execution.

Result elements are currently not displayed in DbVisualizer. It is, however, recommended that you specify these as they will most likely appear in some way or another in a future version. If you want to test the appearance of Result elements, then open the DBVIS-HOME/resources/dbvis-custom.xml file in a text editor and make sure **dbvis.showactionresult** is set to true.

```
<Result>Table ${table} has been dropped!</Result>
```

- The Result message will be displayed in a dialog after successful execution.
- If the execution fails, a generic error dialog is displayed and the Result is not displayed.

XML ELEMENT - COMMAND

The **Command** element specifies the SQL code that is executed by the action.

```
<Command>
  <SQL>
    <![CDATA[
drop table ${table} mode ${mode} including constraints ${includeconstraints}
    ]]>
  </SQL>
</Command>
```

For more information about the Command element check the XML element - Commands section.

XML ELEMENT - MESSAGE

The **Message** element can be used to specify a highlight message that will appear at the top of the action window.

```
<Message>
    <![CDATA[<html>
This action will be <b>deprecated</b> in a future version as it use database calls
that has been declared by the database vendor as <b>extremely bad performing</
b>.</html>
    ]]>
</Message>
```

You may use simple HTML tags in the message content.

ACTION SHOWING JUST A MESSAGE

There may be situations when an action should show a message in a simple dialog with just an OK button. One scenario when this is useful is when a condition is evaluated for an action requiring certain DB privileges to run it. If proper authorization is missing a message should be displayed.

This is accomplished by having a single **Confirm** element for the **Action** element. The following illustrates an example:

```
<If test="#SUPERUSER">
   <Action id="vertica-table-analyze-workload" label="Analyze Workload For Table"
resultaction="show">
      <Input label="Schema" style="text" editable="false">
         <Default>${schema}</Default>
      </Input>
      <Input label="Table" style="text" editable="false">
         <Default>${objectname}</Default>
      </Input>
      <Command>
         <SQL>
            <![CDATA[SELECT analyze_workload('${schema}.${objectname}')]]></SQL>
      </Command>
      <Confirm>
         Really Analyze Workload on ${schema}.${objectname}?
      </Confirm>
   </Action>
</If>
<Else>
   <Action id="vertica-table-analyze-workload-INFO" label="Analyze Workload For
Table">
      <Confirm>
         <![CDATA[
            This feature requires the <b>super user</b> authorization.
         ]]>
      </Confirm>
   </Action>
</Else>
```

# Icons

## Introduction

Icons related to functionality defined in a database profile are displayed in the database objects tree, actions, and object viewers. Icons are declared by mapping a logical name with the name for the icon.

These elements supports setting the `icon="xxx"` attribute:

- `GroupNode`
- `DataNode`
- `DataView`
- `Action`

The `type="xxx"` attribute for **GroupNode** and **DataNode** elements is the key attribute as it is the association with the other segments (commands, actions, and viewers) in the database profile. For **GroupNode** and **DataNode** it is also used as the icon name in case the `icon="xxx"` attribute is not explicitly specified.

Icons are normally of minor interest until you decide to build your own database profile or extend an existing one.

The following shows the use of the **icon** attribute for a **DataNode** element. A condition controls what icon to use based on the value of **getCatalogs.TABLE_CAT**.

```
<DataNode type="Catalog" label="${getCatalogs.TABLE_CAT}"
   icon="#dataMap.get('getCatalogs.TABLE_CAT').equals('sales') ? 'salesIcon' :
           (#dataMap.get('getCatalogs.TABLE_CAT').equals('support') ?
'supportIcon' : null)">
</DataNode>
```

The following sections explain how icons are handled and what choices you have to add your own icons.

## icons.prefs file

Icons are defined in a simple text file with each row in the format: **name=iconName**. In DbVisualizer there is the DBVIS-HOME/resources/profiles/icons-db.prefs file provided with the installation. It maps all icons used in the

supplied database profiles stored in the same folder as the icons-db.prefs file. Here is a sample of the icons-db.prefs file provided with DbVisualizer:

```
#
# Name                              Lucide (SVG)
# -------------------------------- ------------------------------------------------
collection.size.small             =16x16
collection.size.large             =24x24
collection.size.small.folder      =/images/lucide/lucide;/images/lucide/lucide-fix
collection.size.large.folder      =/images/lucide/lucide;/images/lucide/lucide-fix
collection.size.overlay.scale     =0.7
collection.disabled.icons.folder  =
collection.base.color.default     =
collection.badge.color.default    =ASIS

icon.db.Alert                     =megaphone.svg
icon.db.ASM                       =database.svg
icon.db.ASMDisk                   =hard-drive.svg
...
```

The **collection.x.y** entries defines the characteristics for all icons defined in the file, i.e. the search path, scaling, etc., while all **icon.db.<name>=<icon name>.svg** entries maps a logical name with the actual icon name. DbVisualizer uses the Lucide Icon collection where all icons are in the SVG format. In DbVisualizer we've added support for dynamic coloring, explicit scaling, and overlay support for SVG icons.

> The object type may refer to grouping objects (**GroupNode**) such as **Tables**, **Views**, **Procedures** and specific objects (**DataNode**) such as **Table**, **View**, **Procedure**.

The general recommendation is to name the object type for a GroupNode in a plural form and in singular form for DataNode objects. The icon representing for example **Tables** and **Table** is in most cases the same, still there must be two definitions in the icons-db.prefs file."

## Icons Search Path

The database profile search path defined in **Tool Properties / General / Database Connection / Database Profile** not only defines what directories are searched for profiles but also any custom icon definitions.

These are the default folders searched:

```
${dbvis.prefsdir}/ext/profiles
${dbvis.home}/resources/profiles
```

- `${dbvis.prefsdir}` is replaced with the setting directory for DbVisualizer on the platform being used. On Windows this is C:\Users\<user>\.dbvis.
- `${dbvis.home}` is the installation directory for DbVisualizer.

Any icons-db.prefs file will be read from the same folder, if it exists.

This is an example of the ${dbvis.prefsdir}/ext/profiles/icons-db.prefs file:

```
icon.db.Alert                    =megaphone.svg
icon.db.ASM                      =database.svg
icon.db.ASMDisk                  =hard-drive.svg
```

The actual icon represented by the **icon.db.Alert** entry, i.e., the megaphone.svg Lucide Icon is provided with DbVisualizer.

# Conditional Processing

## Introduction

Conditional processing simply means that a profile can adjust its content based on certain conditions. A few examples:

- Which version of the database is being accessed
- The format of the database URL
- The client environment i.e Java version, vendor, etc.
- User properties
- Database connection properties

Conditional processing is especially useful when adapting the profile for different versions of the database (and/or JDBC driver). Another use is to replace generic error messages with more user friendly messages.

If you have some programming skills conditions are expressed using **If**, **Elseif** and **Else** statements.

There are two phases when conditions are processed:

1. **Conditional processing when database connection is established If**, **ElseIf** and **Else** elements can be specified almost everywhere in the profile
2. **Conditional processing during command execution** The **OnError** element is used to define a message that will appear in DbVisualizer if a command fails. Conditions are used to control what message should appear

DbVisualizer uses the **type** attribute to determine which **If** elements should be executed in which of the two phases. If this attribute is set to the value **runtime**, it will be processed in the second phase. If it is not specified, it will be processed in the first phase.

## Conditional processing when database connection is established

These are the call signatures for the utilities used when processing conditions:

```
boolean #util.isNull(String string)
boolean #util.isNullOrWhitespace(String string)
boolean #util.isDatabaseVersionLT(Integer major)
boolean #util.isDatabaseVersionLT(Integer major, Integer minor)
boolean #util.isDatabaseVersionLTE(Integer major)
boolean #util.isDatabaseVersionLTE(Integer major, Integer minor)
boolean #util.isDatabaseVersionEQ(Integer major)
boolean #util.isDatabaseVersionEQ(Integer major, Integer minor)
boolean #util.isDatabaseVersionGTE(Integer major)
boolean #util.isDatabaseVersionGTE(Integer major, Integer minor)
boolean #util.isDatabaseVersionGT(Integer major)
boolean #util.isDatabaseVersionGT(Integer major, Integer minor)
boolean #util.isDatabaseType(String type)
boolean #util.isNotDatabaseType(String type)
```

The following example shows the use of conditions that are processed during connect of the database connection.

```
<Command id="sybase-ase.getLogins">
   <If test="#util.isDatabaseVersionLTE(5)">
      <SQL>
         <![CDATA[
select name from master.dbo.syslogins
         ]]>
     </SQL>
   </If>
   <ElseIf test="#util.isDatabaseVersionEQ(9)">
      <SQL>
         <![CDATA[
select name, suid from master.dbo.syslogins
         ]]>
      </SQL>
   </ElseIf>
   <Else>
      <SQL>
         <![CDATA[
select name, suid, dbname from master.dbo.syslogins
        ]]>
      </SQL>
    </Else>
</Command>
```

The above means that if the major version of the database being accessed is less than or equal to 5, the first SQL is used. If the major version is equal to 9, the second SQL is used, and the last SQL is used for all other versions. The test attribute may contain conditions that are ANDed or ORed. Conditions can contain multiple evaluations, combined using parenthesis. The If, ElseIf and Else elements may be placed anywhere in the XML file.

Here is another example that controls whether certain nodes will appear in the database objects tree or not.

```
<!-- Getting Table Engines was added in MySQL 4.1 -->
<If test="#util.isDatabaseVersionGTE(4, 1)">
   <GroupNode type="TableEngines" label="Table Engines" isLeaf="true"/>

   <!-- "Errors" was added in MySQL 5 -->
   <If test="#util.isDatabaseVersionGTE(5)">
      <GroupNode type="Errors" label="Errors" isLeaf="true"/>
   </If>
</If>
<Commands>
   <OnError>
      <!-- The ORA-942 error means "the table or view doesn't exist" -->
      <!-- It is catched here since these errors typically indicates -->
      <!-- that the user don't have privileges to access the SYS and/or -->
      <!-- V$ tables. -->
      <If test="#result.getErrorCode() eq 942" context="runtime">
         <Message>
            <![CDATA[
You don't have the required privileges to view this object.
            ]]>
         </Message>
      </If>
      <ElseIf test="#result.getErrorCode() eq 17008" context="runtime">
         <Message>
            <![CDATA[
Your connection with the database server has been interrupted!
Please <a href="connect" action="connect">reconnect</a> to re-establish the
connection.
            ]]>
         </Message>
      </ElseIf>
   </OnError>
   ...
</Commands>
```

As you can see, this example contains **nested** uses of **If**.

## Conditional processing during command execution

Using conditional processing to evaluate any errors from a Command may be useful to rephrase error messages to be more user friendly.

```
<Commands>
   <OnError>
      <!-- The ORA-942 error means "the table or view doesn't exist" -->
      <!-- It is catched here since these errors typically indicates -->
      <!-- that the user don't have privileges to access the SYS and/or -->
      <!-- V$ tables. -->
      <If test="#result.getErrorCode() eq 942" context="runtime">
         <Message>
            <![CDATA[
You don't have the required privileges to view this object.
            ]]>
         </Message>
      </If>
      <ElseIf test="#result.getErrorCode() eq 17008" context="runtime">
         <Message>
            <![CDATA[
Your connection with the database server has been interrupted!
Please <a href="connect" action="connect">reconnect</a> to re-establish the
connection.
            ]]>
         </Message>
      </ElseIf>
   </OnError>
   ...
</Commands>
```

The **OnError** element can be used in **Commands** and **Command** elements. If used in **Commands** element, its conditions are processed for all its commands. If it's part of a specific Command, it is processed only for that command.

## Conditions using connection properties

Connection properties are all settings made in the **Properties** tab for a connection. These can be referred in the database profile using the `#cp.` prefix. Check this for a sample from the vertica.xml profile.

```
<If test="#cp.getBool('SinglePhysicalConnection')">
   <Action id="vertica-local-temp-tables-create-table" label="Create Table"
           icon="add" reload="true"
           class="tableCreate"
           classargs="create"
           doclink="/doclink/create-table"/>
</If>
<Else>
   <Action id="vertica-local-temp-tables-create-table-INFO" label="Create Table"
icon="add">
      <Confirm>
         <![CDATA[
Creating <b>Temporary Table</b> is only supported with <b>Single Shared Physical
Connection</b> enabled.
         ]]>
      </Confirm>
   </Action>
</Else>
```

The names for the properties are unfortunately not publicly available. Just send us an email and we will let you know.

## Conditions using JDBC "driver properties"

Driver properties are all settings made in the **Properties** tab and in the **Driver Properties** category. These can be referred in the database profile using the `#dp.` prefix.

```
<If test="#dp.useSSL eq true">
   ...
</If>
```

## drop-on-condition attribute

The drop-on-condition attribute is processed during profile load and may have the constant value **"always"** or a boolean statement that is evaluated. This attribute is valid for the following XML elements:

- Action
- ActionGroup
- GroupNode
- DataNode
- ObjectView
- DataView

This will drop the element if the database version is less than 4.1:

```
<GroupNode type="TableEngines" label="Storage Engines"
   drop-on-condition="#util.isDatabaseVersionLT(4,1)">
```

This will drop the element unconditionally which is useful in a sub profile needing to for example drop a parent DataView defined in the same ObjectView:

```
<DataView id="generic-catalog-tables" drop-on-condition="always"/>
```

If you are looking to drop for example the parent ObjectView and all its DataView elements, and then add a replacement for the ObjectView in the sub profile, you need to drop each of the DataView elements in the sub profile rather than first doing a drop of the parent ObjectView and then re-define it in the sub profile. Here is an example that will not work resulting that both ObjectView="Procedures" definitions will be removed:

```
<ObjectView type="Procedures" drop-on-condition="always"/>

<ObjectView type="Procedures">
   <DataView id="redshift-procedures-Procedures" icon="Procedures"
label="Procedures" viewer="grid">
      ...
   </DataView>
</ObjectView>
```

Instead you need to override the ObjectView and in it, drop the DataView:

```
<ObjectView type="Procedures">
   <DataView id="generic-procedures-procedures" drop-on-condition="always"/>

   <DataView id="redshift-procedures-Procedures" icon="Procedures"
label="Procedures" viewer="grid">
      ...
   </DataView>
</ObjectView>
```

The same apply for all elements supporting the drop-on-condition attribute.

# Database Profile Utilities

In the **Database Profiles** list in **Connection Properties** there is a right-click menu with various commands to process the database profile and list of profiles. These

commands are useful while developing a database profile. Read the following sections for more details about each command.

## Analyze Database Profile

This command verify the loaded database profile and perform various consistency checks and may report:

| Check |
| --- |
| GroupNode and DataNode elements with no matching ObjectView (by "type" attribute) |
| ObjectView elements with no matching GroupNode or DataNode (by "type" attribute) |
| ActionGroup elements with no matching GroupNode or DataNode (by "type" attribute) |
| DataView elements located in the same ObjectView having same "label" attribute |
| No matching icon for these elements (by the "type" or "icon" attributes) |

## Show All Type and Icon Attributes

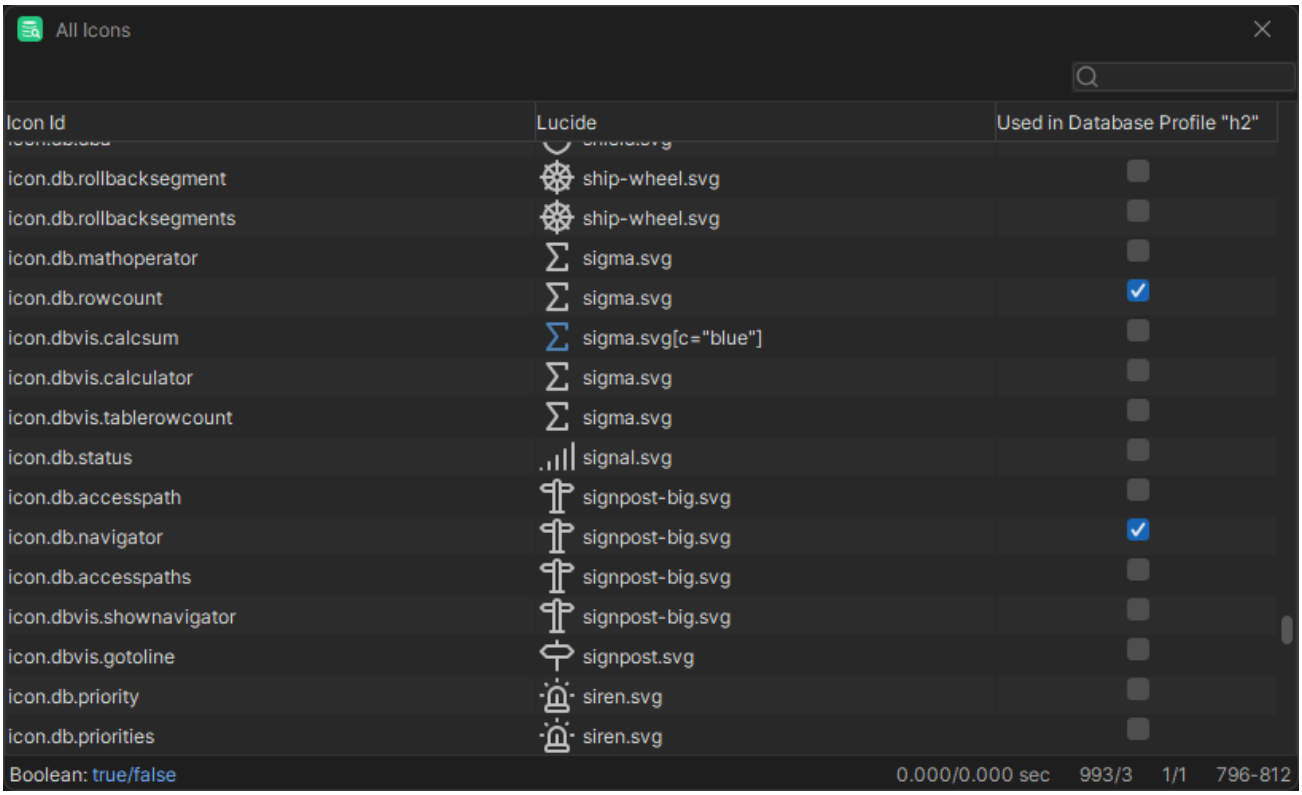
This command examine the loaded profile and report all used icons (logical name) as referenced by the object type and specific icon attributes. The following is an example doing running this command with the MySQL profile loaded.

```
These are all "type" and "icon" references in the profile:
add
catalog
catalogs
charset
collation
column
columns
data
databases
dba
edit
errors
event
events
export
function
functions
import
index
indexes
info
login
navigator
primarykey
privileges
procedure
procedures
process
processes
references
remove
rename
role
rowcount
rowid
schema
schemas
scriptobject
slavestatus
source
sourceeditor
stateerror
status
table
tableengine
tableengines
tableprivileges
tables
trigger
triggers
users
```

```
variables
view
views
warning
warnings
```

## Show Available Icons

This will show a window with all available icons, their name, icon file name and an indicator if the icon is used in the loaded database profile.




## Export Merged Profile

This will export the currently loaded profile. If the profile has been merged using the extend attribute the exported profile file is the complete and processed profile.

## Configure Search Path

This will open **Tool Properties / General / Database Connection / Database Profile** pane used to configure the search path for database profile loading.

## Reload Database Profiles List

This will reload the list of database profiles in case a new profile has been added (or renamed) since last launch of DbVisualizer.

# Database Profile Changes in 25.2

This document collects changes in the database profile framework in DbVisualizer 25.2.

## New attributes for the ProcessDataSet sub-element for Command

The ProcessDataSet element now supports a new action:

- **splitline**

Read more in: XML element - Commands

# Database Profile Changes in 24.3

This document collects changes in the database profile framework in DbVisualizer 24.3.

## New attributes for the Input sub-element for Action

The Input element now supports a new style.

- **style="file-chooser"**

Read more in: Input type - file-chooser

## New attributes for the ProcessDataSet sub-element for Command

The ProcessDataSet element now supports new actions

- **copycolumn**
- **expanddataset**
- **removerowsifnotequalto**

Read more in: XML element - Commands

# Database Profile Changes in 23.3

This document collects changes in the database profile framework in DbVisualizer 23.3.

## New attributes for the ProcessDataSet sub element for Command

The ProcessDataSet element now supports a new action

- **mapvalues**
- **reduceDataSet**

Read more in: XML element - Commands

# Database Profile Changes in 14.0

This document collects changes in the database profile framework in DbVisualizer 14.0.

## New "usedataset" action for ProcessDataSet

The ProcessDataSet element now supports a new action

- **action="usedataset"**
- Possible values **"keepfirst"**, **"keeplast"**, **"removefirst"**, **"removelast"**

Read more in: XML element - Commands

# Database Profile changes in 13.0

This document collects changes in the database profile framework in DbVisualizer 13.0.

## New "node-text" viewer for DataView elements

The DataView element now supports the new **node-text** viewer.

- **viewer="node-text"**

Read more in: Viewer - node-text

# Database Profile changes in 11.0

This document collects changes in the database profile framework in DbVisualizer 11.0.

## Common attribute changes

- The **action="drop"** attribute replaced with **drop-on-condition="..."**
- Read more in **drop-on-condition** attribute

## New attributes for the ProcessDataSet sub element for Command

The ProcessDataSet definition defines if and how a DataSet generated by a Command should be processed after retrieved from the database.

The following new actions for ProcessDataSet are new in 11.0:

- **addrow**
- **convertnullvalues**
- **convertsqlwarningtodataset**
- **movecolumn**
- **truncatedataset**

Read more in XML element - Commands

## New attributes for the ObjectView element

The DataView element now supports this new attribute:

- **layout="tilehorizontal/tilevertical/collapse"**

Read more in: XML element - ObjectView

## New "chart" viewer for DataView elements

The DataView element now supports the new **chart** viewer.

- **viewer="chart"**

Read more in: Viewer - chart

# Database Profile changes in 9.5

This document collects changes in the database profile framework in DbVisualizer 9.5.

## New/changed attributes for Command

These are the changes in the attributes for the Command element.

- exectype="**script**/asis/explain" (replaces parsesql="true/false")
- **script** will execute each individual statement in a multi statement script, **asis** will execute all of the SQL as one statement and **explain** will try generate an explain plan for the single statement
- processmarkers="**true**/withdriver"
- Defines whether markers such as :name, ? should be processed either internally by DbVisualizer or by letting the driver (if it supports it) do the parsing
- autocommit="**true**/false"
- Specifies if the command should be auto committed after execution
- whensuccess="commit/rollback/ask"
- If autocommit="false" then this attribute defines what to do after successful execution
- whenerror="commit/rollback/ask/ignore"
- If autocommit="false" then this attribute defines what to do after failed execution

## Action element improvements

- Moved processmarkers="true|false" to **Command processmarkers="true|withdriver"**
- Removed **execute** value for the **mode** attribute (since it is default)
- Removed **resulttype="resultset|dbmsoutput"** and replaced by using the **@set server output** client side command in script when needed
- Replaced **resultaction="ask|show|showtext|script"** with **resultaction="mergeastext/mergeasscript"**
- Added **resulttarget="editor"**

Notes:

- If none of **resultaction** and **resulttarget** is present and there are results generated, the results are presented as tabs (same as SQL Commander). The result set tab option to **Merge All Result Sets** tabs is available
- If **resultaction** is present all result sets will be merged into a single result tab
- If **resulttarget** is present it will copy the merged results to a new SQL Commander tab

## New runsetdefaultwhenvaluechanged attribute

The **runsetdefaultwhenvaluechanged** attribute defines what other inputs should be triggered when the value for the input is changed. Whenever this happens the other inputs Default commands are executed. This is useful if there are for example two list inputs. Changing the selected entry in the first list will then notify the second input so that its Default command is re-executed using the new value as condition.

```
<Action id="postgresql-settings-alter" label="Alter Setting" reload="true"
icon="edit">
   <Input name="setting" label="Setting"
runsetdefaultwhenvaluechanged="value,desc" style="list">
      <Values>
         <Command><SQL><![CDATA[select name from pg_settings]]></SQL></Command>
      </Values>
   </Input>
   <Input name="value" label="Value" style="text">
      <Default>
         <Command><SQL><![CDATA[select setting from pg_settings where name = '$
{setting}']]></SQL></Command>
      </Default>
   </Input>
   <Input name="desc" label="Description" style="text" editable="false">
      <Default>
         <Command><SQL><![CDATA[select short_desc from pg_settings where name = '$
{setting}']]></SQL></Command>
      </Default>
   </Input>
   <Command>
      <SQL><![CDATA[alter system set ${setting} = ${value}]]></SQL>
   </Command>
   <Confirm>
      Really alter setting ${setting}?
   </Confirm>
</Action>
```

### Check list style

The new check-list input shows a list with options allowing multiple selections.

```
<Input label="Weekdays" name="weekdays" style="check-list">
   <Arg name="output" value="jschours[${value}] = true"/>
   <Arg name="newline" value=", "/>
   <Labels>Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday</Labels>
   <Values>00|11|22|33|44|55|66</Values>
   <Default>22|44</Default>
</Input>
```

The `<Arg name="output" value="jschours[${value}] = true"/>` defines that for each selected value in the list (ex Monday, Thursday, Friday), the output in the **weekdays** variable will be:

`jschours[11] = true,` jschours[33] = true, jschours[44] = true

The **newline** argument defines the string used to separate the output entries.

### Check style

Check inputs now supports a second values argument:

```
<Input label="Check with true/false value" name="enabled" style="check">
   <Values>On|Off</Values>
   <Default>Off</Default>
</Input>
```

The first, "On" is set if the checkbox is checked. "Off" is set if the checkbox is unchecked. If the second argument is not specified an empty string is set if the checkbox is unchecked.

### Action layout

- Improved support for having input fields on the same row
- Previously when using **linebreak="false"** any label on that row was not displayed. Now if there is a label it is shown
- New: **type="label"** input attribute. Used to create just a label. Useful when **linebreak="false"** by having a general label for the row and then each component can have their individual label. Ex: "Days: Mon: X Tue: X"
- **New: type="note"** input attribute. Is the exact samt as "label" but can be used to specify some informative text that is not suffixed with a colon

## Changes for DataNode and GroupNode

- In addition to the label attribute the new label1 is now available. In the objects tree it will show optional information about the object. This is displayed in italic gray text.
- The **sort** attribute has been replaced with the **ProcessDataSet action="sortcolumn"**

## New utility class

The new #util class can be used in profiles to primarily perform conditional checks such as what target database version is used. This is useful to adjust what parts of the database profile should be visible to the user. These are the methods available:

```
boolean #util.isNull(String string)
boolean #util.isNullOrWhitespace(String string)
boolean #util.isDatabaseVersionLT(Integer major)
boolean #util.isDatabaseVersionLT(Integer major, Integer minor)
boolean #util.isDatabaseVersionLTE(Integer major)
boolean #util.isDatabaseVersionLTE(Integer major, Integer minor)
boolean #util.isDatabaseVersionEQ(Integer major)
boolean #util.isDatabaseVersionEQ(Integer major, Integer minor)
boolean #util.isDatabaseVersionGTE(Integer major)
boolean #util.isDatabaseVersionGTE(Integer major, Integer minor)
boolean #util.isDatabaseVersionGT(Integer major)
boolean #util.isDatabaseVersionGT(Integer major, Integer minor)
boolean #util.isDatabaseType(String type)
boolean #util.isNotDatabaseType(String type)
boolean #util.replaceAll(String source, String search, String replacement )
```

## Changed icons definition

Many icons are composed of a full size (16x16, 24x24) base icon and a badge icon (half the size) representing things like, new, delete, error, clock, and so on. The badge icon is typically anchored south east on the base icon.

Prior to DbVisualizer 9.5 there were no separate badge icons as these were integrated with the base icon resulting in many duplicated base icons only with different badge symbols.

In DbVisualizer 9.5 a new icon library was introduced and now base icons are separate from the badge icons. Any badge icons that should be attached are defined in the icons.prefs file and the final icon is dynamically created.

# Troubleshooting

## Troubleshooting

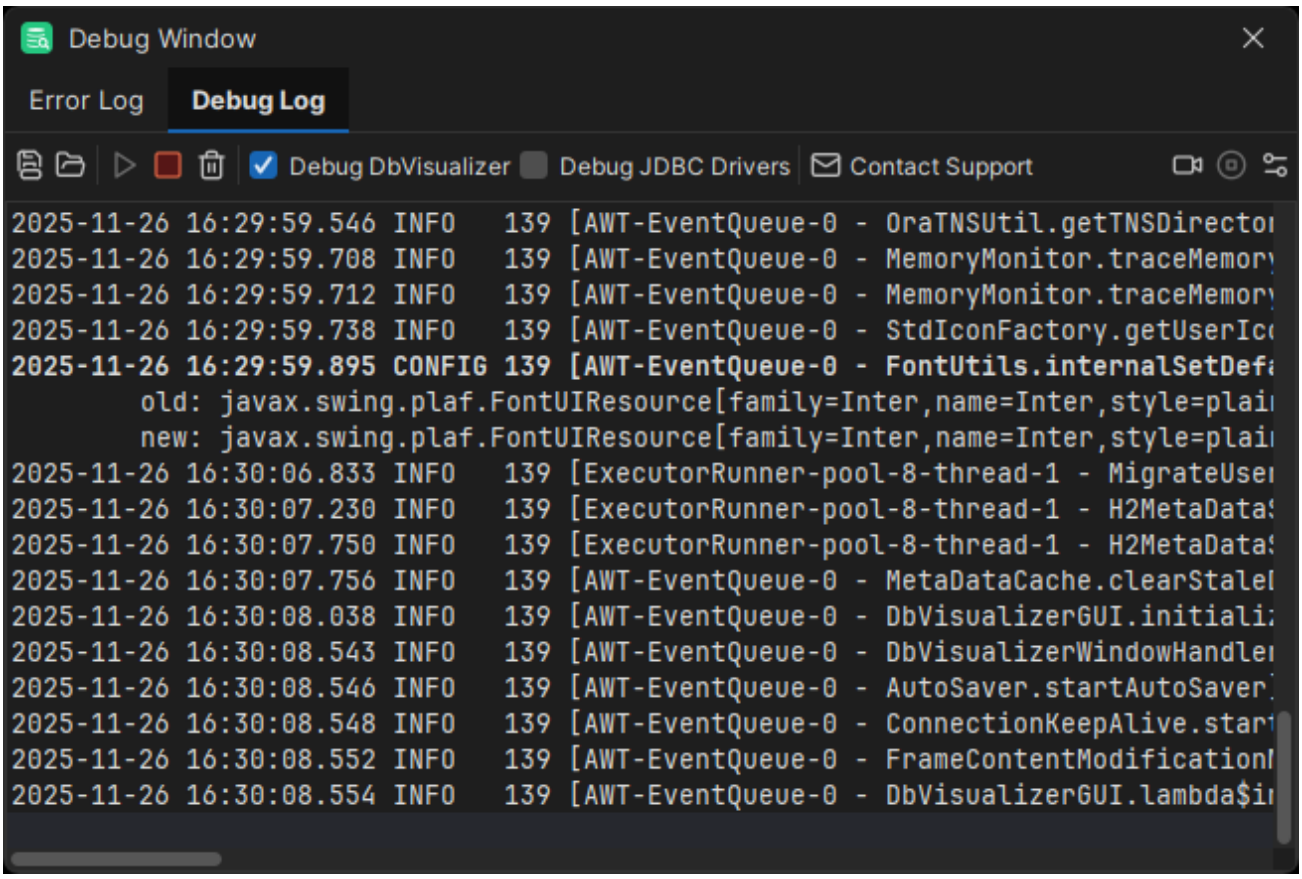
Even though we make our very best to ensure the quality of DbVisualizer, you may run into problems of different kinds. The runtime environment for DbVisualizer is rather complicated when it comes to tracking the source of a problem, since it's not only DbVisualizer that may cause the problem but also the JDBC driver, or even the database engine.

There are a few things that you can try before reporting a problem, depending on the nature of the problem:

1. Make sure you are using the latest version of Java available for your platform (Java 21 or later),
2. Make sure you are using a version of the JDBC driver that we've tested DbVisualizer with, or a later, production quality version,
3. Check the DbVisualizer knowledge base,
4. Read the DbVisualizer User Guide,
5. ... the last resort is to post a question via the problem report form or send an email to support@dbvis.com. (Note that we generally love detailed reports as well as screenshots when possible)

# Debugging DbVisualizer

The **Tools → Debug Window** is useful to see what is going on in DbVisualizer and the JDBC driver(s). The checkboxes at the top of the **Debug** tab controls what parts of DbVisualizer should be debugged. The **Debug JDBC Drivers** option will enable debug of the current JDBC driver. Note that the amount of output is determined by the JDBC driver.




The log window shows the end of the log files produced by DbVisualizer. The **Debug Log** tab shows the debug log file and the **Error Log** tab the error log file. DbVisualizer writes its log files to a log directory which may be opened using the **Open Log Directory** link at the top right. DbVisualizer maintains the content of the log directory in a way that limits the size of its content. The content of the log tabs is also automatically truncated to preserve memory.

The toolbar contains the following buttons:

- **Save** saves the content of the visible log tab to file
- **Copy** copies any selection to the clipboard
- **Clear** clears the content of the log tab
- **Start** starts monitoring of the corresponding log file. This button is of course disabled if monitoring is already started
- **Stop** stops the monitoring of the log files (if started)

The button labeled **Contact Support** serves as a shortcut to open the window for How to Report Issues where you may optionally attach the content of the DbVisualizer log files to a support ticket.

# Fixing Connection Issues

There may be many reasons why you cannot connect to a database, some of the most common are listed below in this section.

## Errors occurring when connecting

- Incorrect values for the **Database Server** or **Database Port** fields in the **Object View** tab for the connection,
- TCP/IP access is not enabled in the database server,
- A firewall between the client and the database server blocks connections to the database port,
- A syntax error in a manually entered JDBC URL,
- The user account is not authorized to connect from the client where you run DbVisualizer,
- Native libraries for a JDBC driver are not found.

The first three problems usually result in a somewhat cryptic message about I/O errors or a time-out.

If you run into issues at this stage, use the **Test Connection** button. DbVisualizer will then test step by step whether a connection can be established to the specified server and port, whether the SSH tunnel can be established (if configured), and whether the user account is authorized to connect to the database. This will give you a clearer picture of what is going wrong.

If you still need help after running this test, please ask your system or database administrator for help.

## Common error messages

JDBC syntax errors typically result in one of two error messages:

- "**The selected Driver cannot handle the specified Database URL. The most common reason for this error is that the database URL contains a syntax error preventing the driver from accepting it. The error also occurs when trying to connect to a database with the wrong driver. Correct this and try again.**"
- "**java.sql.SQLException: Io exception: Invalid number format for port number Io exception: Invalid number format for port number**"

In both cases, we recommend using the **Server Info** settings format instead of the **Database URL** format for the connection, and let DbVisualizer build a valid JDBC URL for you. If you must enter a JDBC URL manually, make sure that you replace possible placeholders enclosed with "<" and ">" in a template you have copied, such as <1521>, and look for other syntax errors. Also verify that the JDBC driver is installed correctly.

Authorization problems are usually described by more straight forward messages. Ask you database administrator to help you get it resolved.

## Native library errors

If you get a message about native libraries not being found, e.g. "**no ocijdbc11 in java.library.path**" or similar, it is because you have not installed these in a location where DbVisualizer can read them. Unless you have a very good reason for using a JDBC driver that requires native code, we recommend that you use a pure Java JDBC driver (a Type 4 driver) instead, like the "Oracle Thin" driver for Oracle. For more information about native drivers see Using drivers depending on native API (Type 2 JDBC driver)

## SSH problems

If SSH is used for a database connection, please verify that the SSH tunnel can be established. As described earlier, this is done when you click **Test Connection** in the connection view. You can also check this in **Tools → Tool Properties** and in the **SSH Configurations** category. More info in Using SSH.

## This was of no help...

If none of this helps, please contact us using the **Help → Contact Support** form. Read more in How to Report Issues.

For up-to-date articles related to connection issues please check the Database Specific section in our support portal. For security related information, check the Security section.

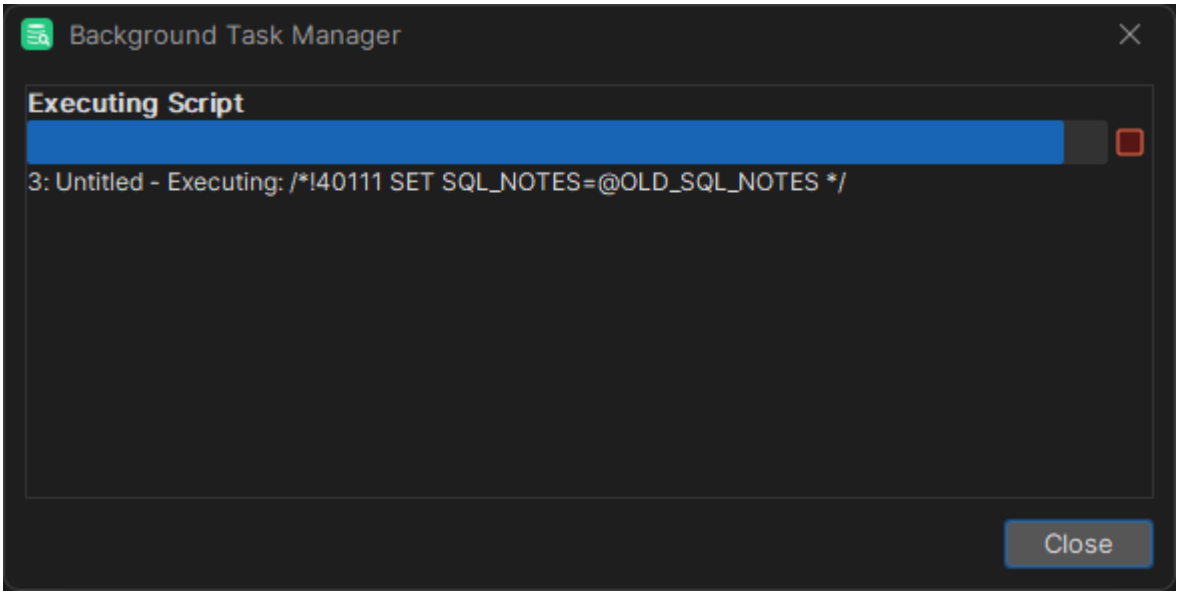
# Handling Dropped Connections

All tasks that may potentially take a bit of time to perform or that may not finish at all because a database connection has dropped are executed in the background so that you can still do other things. You can see that background tasks are running by looking at an indicator icon in the status bar.




If there are tasks running, the indicator icon is animated, showing a spinning pattern.

You can see exactly which tasks are running by clicking on the icon, or by choosing **Help → Background Task Manager** in the main menu. This opens the **Task Manager** window.




You can abort a long running task by clicking the **Stop** button next to the progress bar.

If the task did not complete because the connection dropped, you also need to reconnect to the database. If this happens frequently, you can enable Connection Keep-Alive to send a dummy SELECT statements to the database occasionally to prevent time-outs.

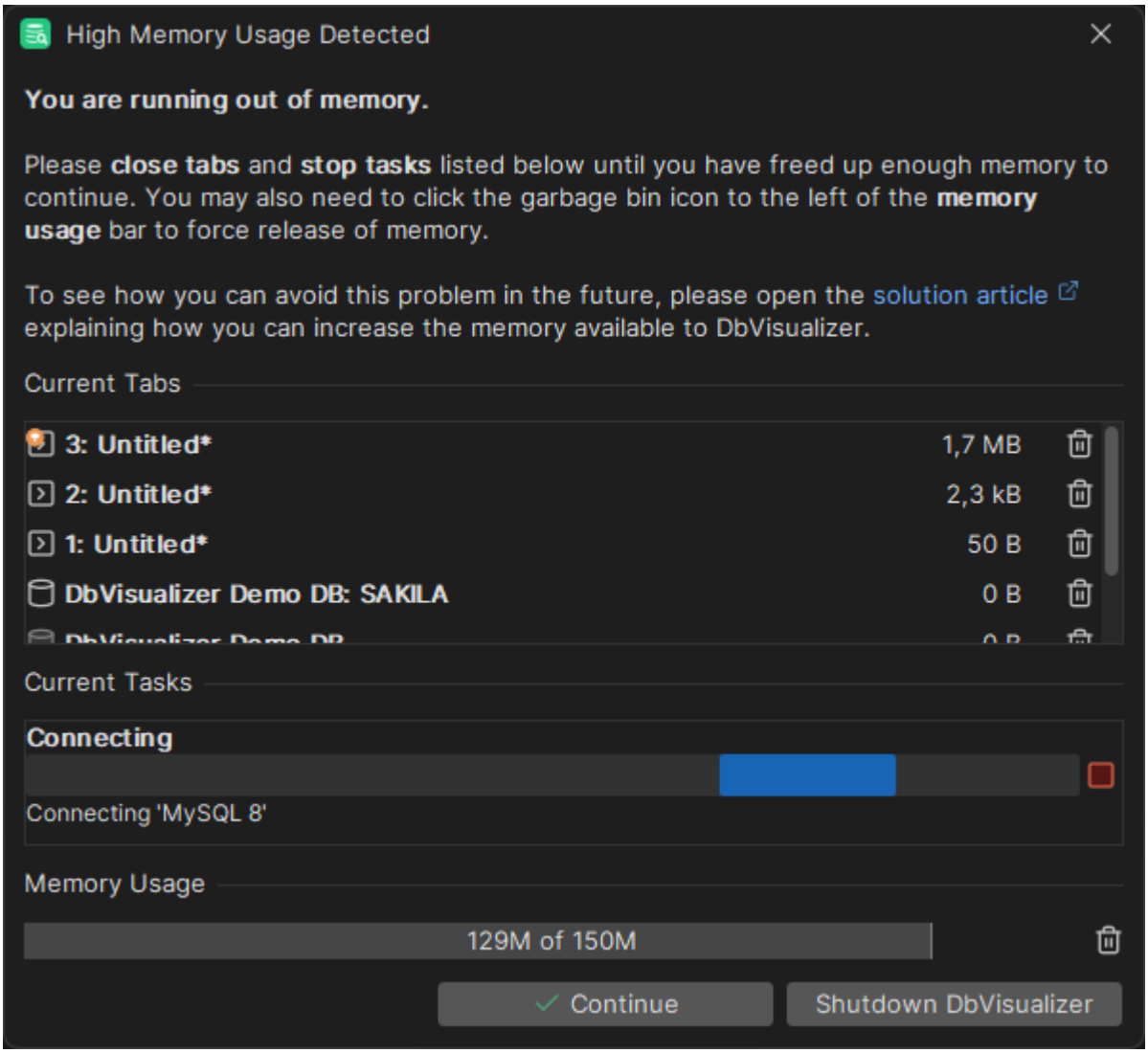
DbVisualizer will by default show a verification prompt when connect is needed. This appearance of this prompt may be controlled, find out more in Connection Permissions - Confirming connect.

# Handling Memory Constraints

DbVisualizer has a fixed amount of memory available, and things may go bad if you load so much data that you're getting close to this limit. The most common effect is that the GUI becomes very unresponsive or freezes completely.

> The number of open SQL Commander tabs and their content is in most uses the biggest consumer of memory. A general recommendation is to save the content to file and open only when needed.

To minimize the risk of this happening, DbVisualizer keeps track of the memory usage when you load tables or files and similar tasks that consume memory. If you're getting so close to the limit that problems are likely to begin to show, all memory-consuming background tasks are suspended and the High Memory Usage window pops up.




All open tabs are listed along with an estimate of how much memory each tab uses. All background tasks are also shown. You need to resolve the high memory usage problem before you can continue working with DbVisualizer.

Click on the red cross next to tabs that use lots of memory to close them and stop tasks that consume memory.

When you have released enough memory to get below the critical level, the icon in the Continue button changes to a green checkmark. Click it to close the window and continue to work.

If you often see this window, first consider using features that minimize the memory usage, such as using the `@run` command to execute your script and the `@export` command to write query results to a file. As a last resort, you can increase the amount of memory for DbVisualizer.

In rare cases, closing tabs and stopping tasks do not release enough memory to continue, possibly due to memory leaks in the DbVisualizer code. If this happens, you can click the **Create Heap Dump** button. The heap dump file is named heap.bin and stored in the preferences folder, typically the .dbvis folder in your home folder. It may help us find memory leaks and fix them, so please send it to us with a description of what you were doing when you ran into this problem. The file can be huge, so please compress it (e.g. using the zip command) before mailing it to us.

If you cannot release enough memory to continue, you can use the **Shutdown DbVisualizer** button to shut down and start fresh.

# How to Report Issues

## Contacting support

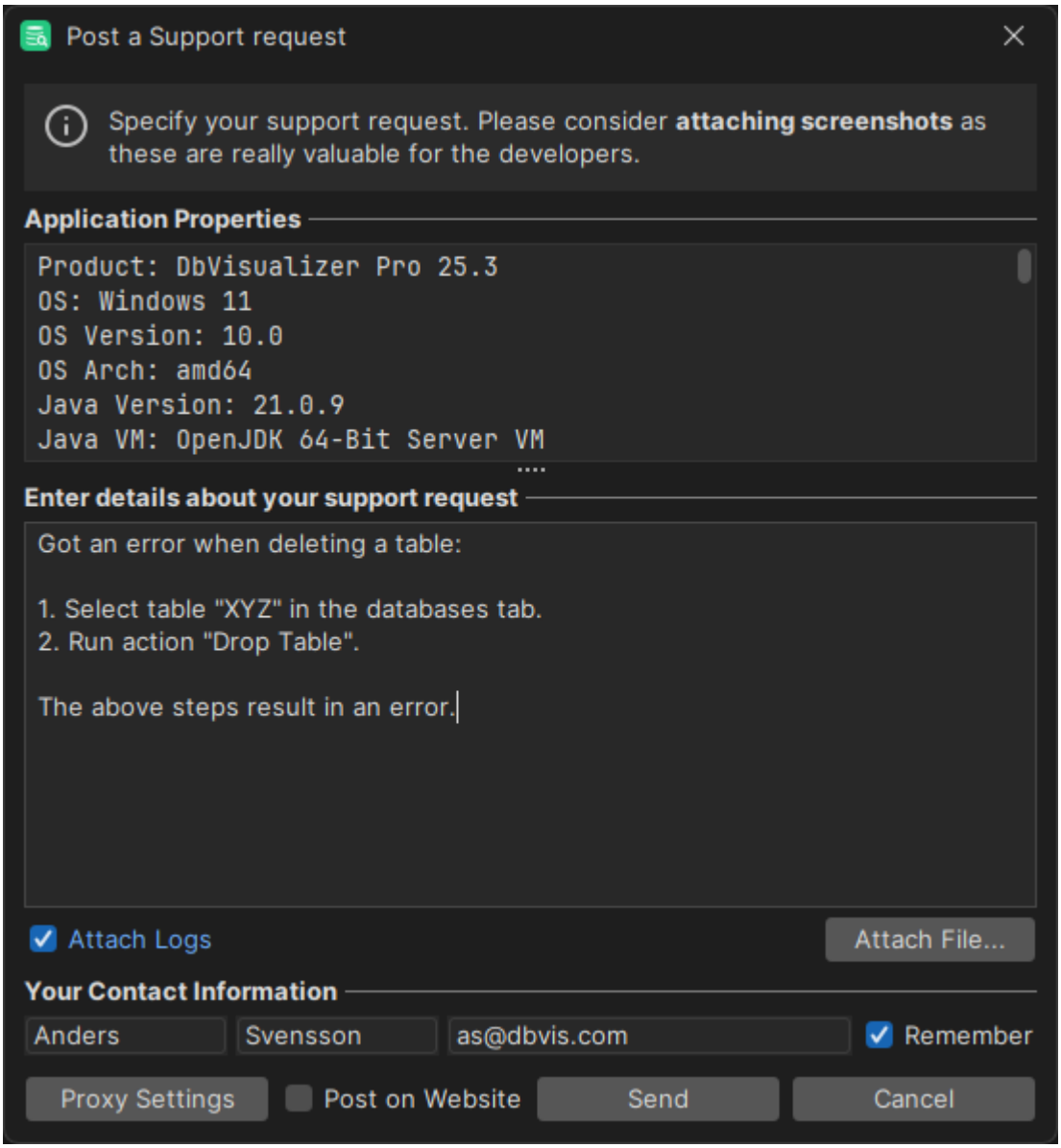
There following are the alternatives getting help with issues or support requests:

- Using **Help → Contact Support** in the DbVisualizer application (recommended)
- Using the web form at https://www.dbvis.com/company/contact/
- Using the knowledge base at https://support.dbvis.com

### Using Help → Contact Support

This is the recommended way to report an issue or a support request as it gathers information about your settings and connections that help us provide you with better analysis of any problems without having to get back to you and ask for additional information.

Please describe the problem in as much detail as possible, including the actions leading to the issue.

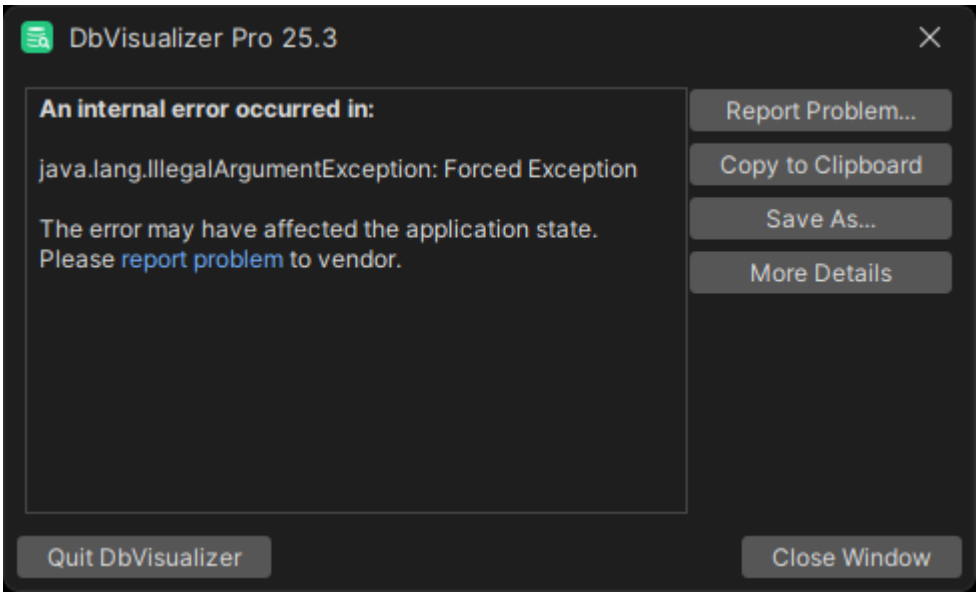



## Encountering errors

DbVisualizer sometimes detects errors that need the attention from the user. When this happens, the application will either display and error dialog or pop up a "red balloon" in the status bar of the main application window.

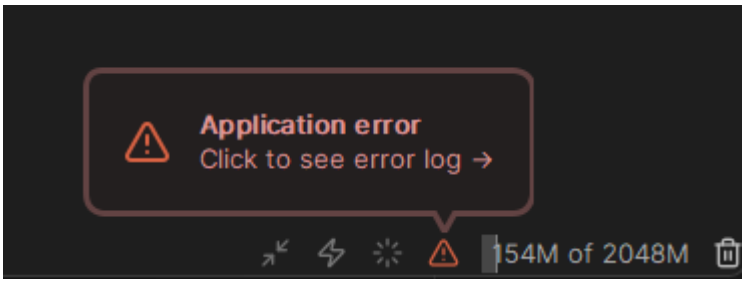
### Error dialog

If you encounter an error that causes an error alert dialog to pop up, please click the **Report Problem** button to open the contact form with the details of the error filled in.




## Red balloon

DbVisualizer monitors the log entries written to its log files. If log entries representing errors are detected, a "red balloon" pops up in the DbVisualizer status bar as indicated below.




Clicking the ballon will bring up the debug window.

# Using special characters in passwords

Passwords entered for connections etc. can optionally be saved between sessions by DbVisualizer. Passwords containing special characters such as ♥ can not be saved when the default encoding for passwords is used. More specifically DbVisualizer only supports characters in the 8859-1 character set with the default encoding; other characters are corrupted when saved.

To be able to save passwords containing any characters, you need to enter a Master Password. Please see the section Setting a Master Password for more information.

# Reference Material

## Reference Material

Here you find details about areas that are not covered elsewhere.

- GUI Command Line Arguments
- Installation Structure
- Installing a JDBC Driver
- Special Properties

# GUI Command Line Arguments

As an alternative to start DbVisualizer via the menu items and icons created by the installer, you can also start DbVisualizer from a shell (terminal) on all operating systems using the following scripts:

```
# DbVisualizer GUI on Windows:
DBVIS-HOME\dbvisgui.bat

# DbVisualizer GUI on Linux/UNIX:
DBVIS-HOME/dbvisgui.sh

# DbVisualizer GUI on macOS:
/Applications/DbVisualizer.app/Contents/Resources/app/dbvisgui.sh
```

The scripts supports a number of command line arguments. These are also listed in the **Help → About** menu choice, under the **Command Line** tab, in DbVisualizer.

```
Usage: dbvisgui [-connection <name>] [-userid <userid>] [-password <password>]
                [-encoding <encoding>]
                [-workspace <directory>]
                [-windowtitle <title>]
                [-help] [-version]
                [<filename>]
General Options:
 -connection <name>     Database connection name (created with the GUI)
 -userid <userid>       Userid to connect as
 -password <password>   Password for userid
 -encoding <encoding>   Encoding for the SQL script file
 -workspace <directory> Load preferences from an alternate workspace
 -windowtitle <title>   Additional window title
 -help                  Display this help
 -version               Show version info
 <filename>             SQL script file to load into editor
```

## JAVA_EXEC

⚠ Please note that these scripts use the first Java version that is found in the PATH. The result may be that a non supported Java version is used.

You can specify a specific version by setting the environment variable `JAVA_EXEC` to point at an executable Java (note: set this using the operating system's mechanism, not by editing the **dbvisgui** script).

### Example: Windows

(adjust paths to match your environment)

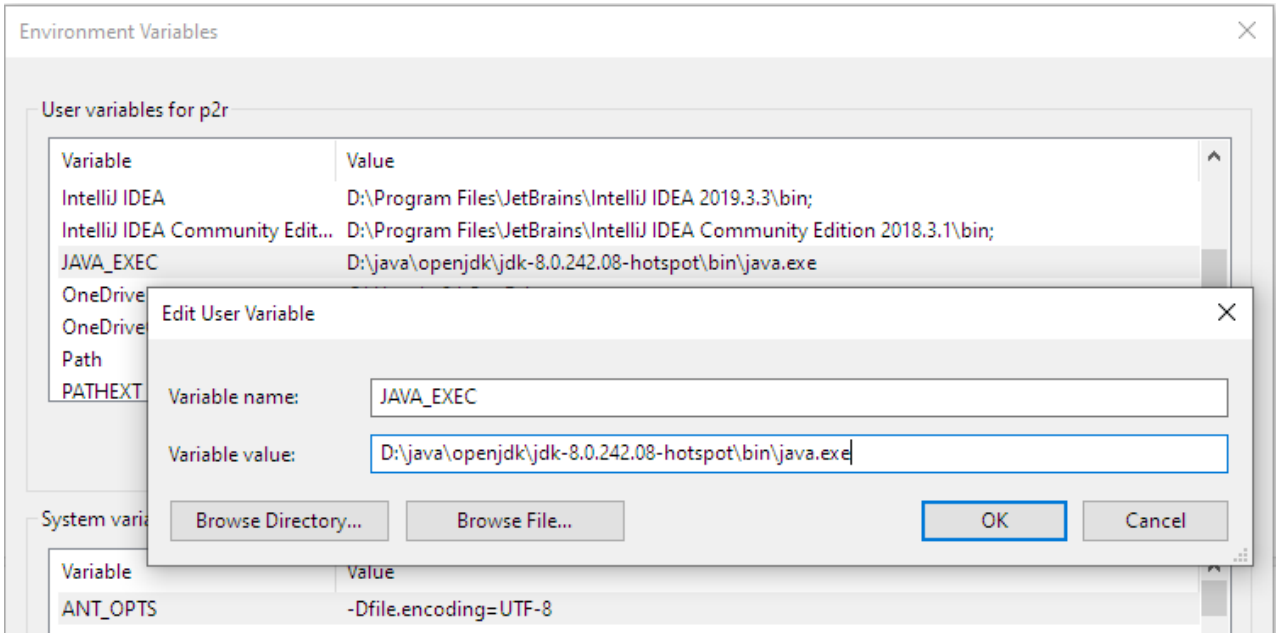
- From the command line before launching `dbvisgui.bat`:

```
set JAVA_EXEC=D:\java\openjdk\jdk-17.0.4.1+1-jre\bin\java.exe
```

- In your own script:

```
setlocal
set JAVA_EXEC=D:\java\openjdk\jdk-17.0.4.1+1-jre\bin\java.exe
call "d:\Program Files\DbVisualizer\dbvisgui.bat"
```

- Using the system control panel (type "**Windows + X**", click **System** -> **Advanced System Settings** -> **Environment Variables**):



# Installation Structure

The installer and launcher for DbVisualizer is based on the install4jTM product. The structure of the installation directory (referred as DBVIS-HOME throughout the User Guide) contains the following on Windows. (The exact content differ between platforms):

- .install4j/
- addon/
- doc/
- jre/
- lib/
- resources/
- wrapper/
- dbvis.exe
- dbvis.vmoptions
- dbviscmd.bat
- dbvisgui.bat
- java9-args
- README.txt
- resolveJRE.bat
- uninstall.exe

The dbvis.exe file is used to start DbVisualizer. The remaining files and directories are only of interest if you need to do nonstandard customization. For information on how to increase the memory for the Java process that runs DbVisualizer, and also on how to modify the Java version being used, please check the DbVisualizer support portal.

# Installing a JDBC Driver

DbVisualizer bundles JDBC drivers for most common databases, so typically you do not need to install a JDBC driver.

This page describes the way JDBC drivers are managed in DbVisualizer. If a JDBC driver for your database is bundled with DbVisualizer, see Driver Info on the Supported Databases page, you typically do not need to read this chapter.

If, however, any of the these things apply to you, keep on reading:

- want to learn how the Driver Manager in DbVisualizer works,
- need to have several versions of the same JDBC driver loaded simultaneously,
- need to add a Driver that does not exist in the list of default drivers .

## What is a JDBC Driver?

DbVisualizer is a generic tool for administration and exploration of databases. DbVisualizer does not deal directly with how to communicate with each database type. That job is done by a JDBC driver, which is a set of Java classes. All JDBC drivers conform to the JDBC specification and its standardized Java programming interfaces. This is what DbVisualizer relies on. A JDBC driver implements all details for how to communicate with a specific database and database version, and there are drivers available from the database vendors themselves as well as from third parties. To establish a connection to a database, DbVisualizer loads the driver and then gets connected to the database through the driver.

The following sections describe the steps for installing a JDBC Driver, and also how to configure DbVisualizer to use JNDI to obtain a database connection.

## Get the JDBC driver file(s)

DbVisualizer comes bundled with ready-to-use drivers in predefined templates for the most commonly used JDBC drivers that have licenses that allow for distribution with a third party product. Currently, drivers for Azure SQL Database, Db2, Greenplum, H2, JavaDB/Derby, Mimer SQL, MySQL, NuoDB, Oracle, PostgreSQL, SQLite, Vertica, Yellowbrick as well the jTDS driver for SQL Server and Sybase, are included with DbVisualizer. If you only need to connect to databases of these types, you can skip the rest of this chapter and jump straight to the Creating a Connection page, because by default, DbVisualizer creates a user configuration for these drivers automatically when you create a connection.

The Driver Manager is used to create driver configurations. If you need to connect to a database that is not supported by a bundled JDBC driver template, you must create a user configuration for a JDBC driver that works with your database type and version. This is done from a predefined template in the Driver Manager. The following web page contains an up-to-date listing of the database/driver combinations we have tested and supports via templates:

https://www.dbvis.com/features/database-drivers/

For other databases/drivers you need to use the Custom Template and do all the configuration. To find a JDBC driver for your database, go to the database vendor's website or search for the name of the database plus the word `JDBC`. Many drivers are accessible from Maven or HTTP download sites. A good place too search for drivers distributed with Maven is https://mvnrepository.com/

You can download Drivers that is accessible via Maven or HTTP directly in the Driver Manger. Note: ZIP files that are accessible via HTTP, e.g. Google BigQuery, are unpacked automatically after download.

For proprietary drivers you need to download the driver to an appropriate directory. Make sure to read the installation instructions provided with the driver. Some drivers are delivered in ZIP or JAR format. ZIP files need to be unpacked to make the driver files loadable in the Driver Manager. The Databases and JDBC Drivers web page describes where you can download some drivers and also what additional steps may be needed to install and load the driver in DbVisualizer.

> Drivers are categorized into 4 types. We're not going to explain the differences here, just give you the hint that the "type 4," aka "thin," drivers are the easiest to maintain, since they are pure Java drivers and do not depend on any external DLL's or dynamic libraries. Even though DbVisualizer works with any type of driver, we recommend that you get a type 4 driver if there is one for your database.

When you have downloaded and configured the JDBC driver in Driver Manager, you can go ahead and create a database connection, as described in the Creating a Connection page.

## Driver Manager

The **Driver Manager** in DbVisualizer is used to define and configure the drivers that will be used to communicate with the databases. DbVisualizer comes with predefined configuration templates that is used to create the user driver configurations that will be used in connections. Templates may be updated in new versions of DbVisualizer, but your user configuration will not be changed. The templates may contain:

- Ready-to-use configuration with bundled driver files
- Predefined configuration with downloadable artifacts (you will have to download driver files via the artifacts from in the Driver Manager)
- Just a configuration (you will have to add driver files or artifacts)

The Custom Template has no configuration and can be used to create a driver for a database the we have not tested yet

### Loading and Configuring Drivers Manually

You can also load and configure JDBC drivers manually using the Driver Manager.

The left part of the driver manager dialog contains a list of driver names with a status symbol indicating whether the driver has been configured (green checked icon) or not. The list columns also shows how many connections that uses the driver and the version of the driver. The list shows both the current user drivers and the templates (read only). You can search and sort the list. Initially, the driver list contains a collection of templates. The list is used to maintain the drivers and you may:

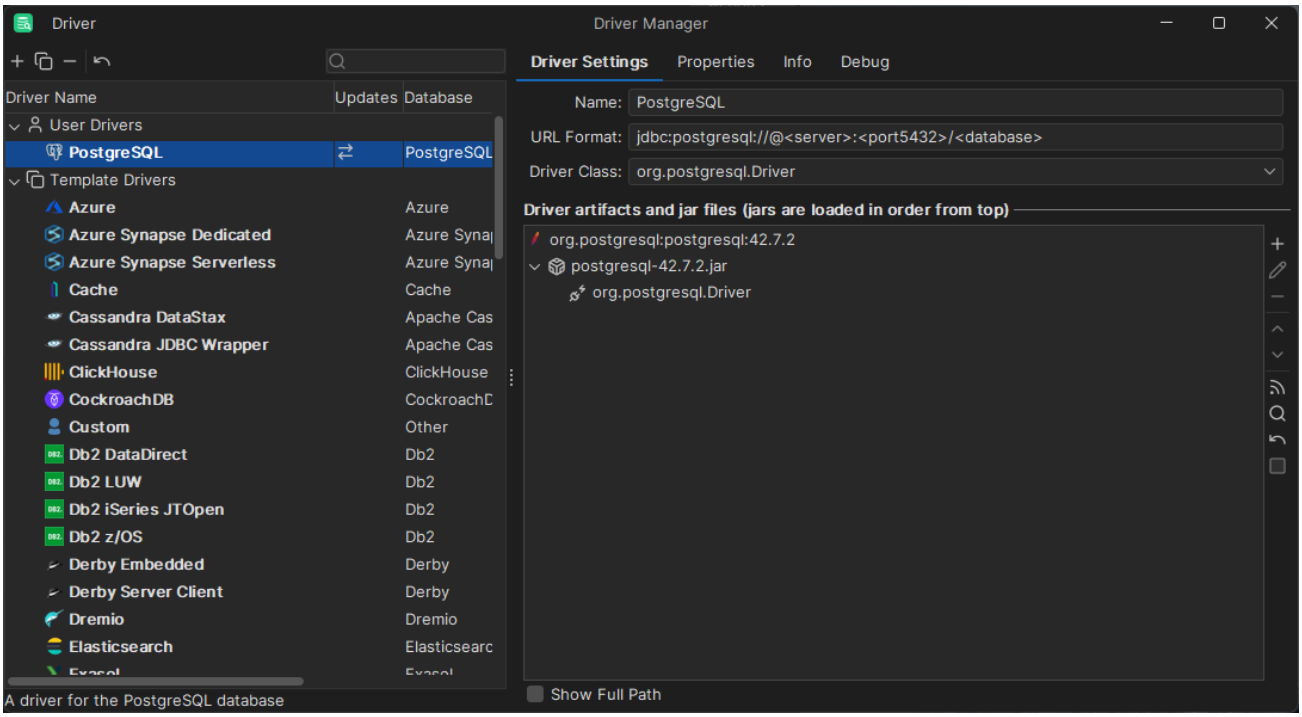
- Create a new user driver from a template (plus)
- Copy an existing user driver (double sheets)
- Delete an existing user driver (minus)
- Reset driver to it's default, i.e. the template (back arrow)

The right part displays the driver configuration for the selected driver in terms of the following:

- **Information area** Presents information about the current state if the driver and what to do next if the driver is not ready-to-use.
- **Separate tabs for**
- **Driver Settings** - described in detail below
- **Properties** - setting for driver properties that is common for all connections that uses the driver configuration (settings can still be changed on the individual connections)
- **Info**rmation - shows general information about the driver

The most important tab is the **Driver Settings** where you have:

- **Name** A driver name in the scope of DbVisualizer is a logical name for either a JDBC driver that is shown when you create a connection and in the Connection tab when selecting which driver to use for a Database Connection
- **URL Format** The URL format specifies the pattern for the JDBC URL. Its purpose is to assist the user in the Connection tab when entering URL information. See Using Variables in Connection Fields for more about how you can make it really easy to create Database Connections for this driver later on. For a custom driver the JDBC URL is used configure the connection.
- **Driver Class** Defines the main class for the JDBC driver, used for connecting to the database.
- **Driver artifacts and jar files (files tree)** Defines artifact to download driver files from Maven or HTTP or paths to search for JDBC drivers.




A driver is ready to use once a driver class has been identified, which is indicated with a green check icon in the list. Drivers that are not ready for use are shown without an icon, or with a red cross icon if an error has been detected (such as a failed download or a missing file) .

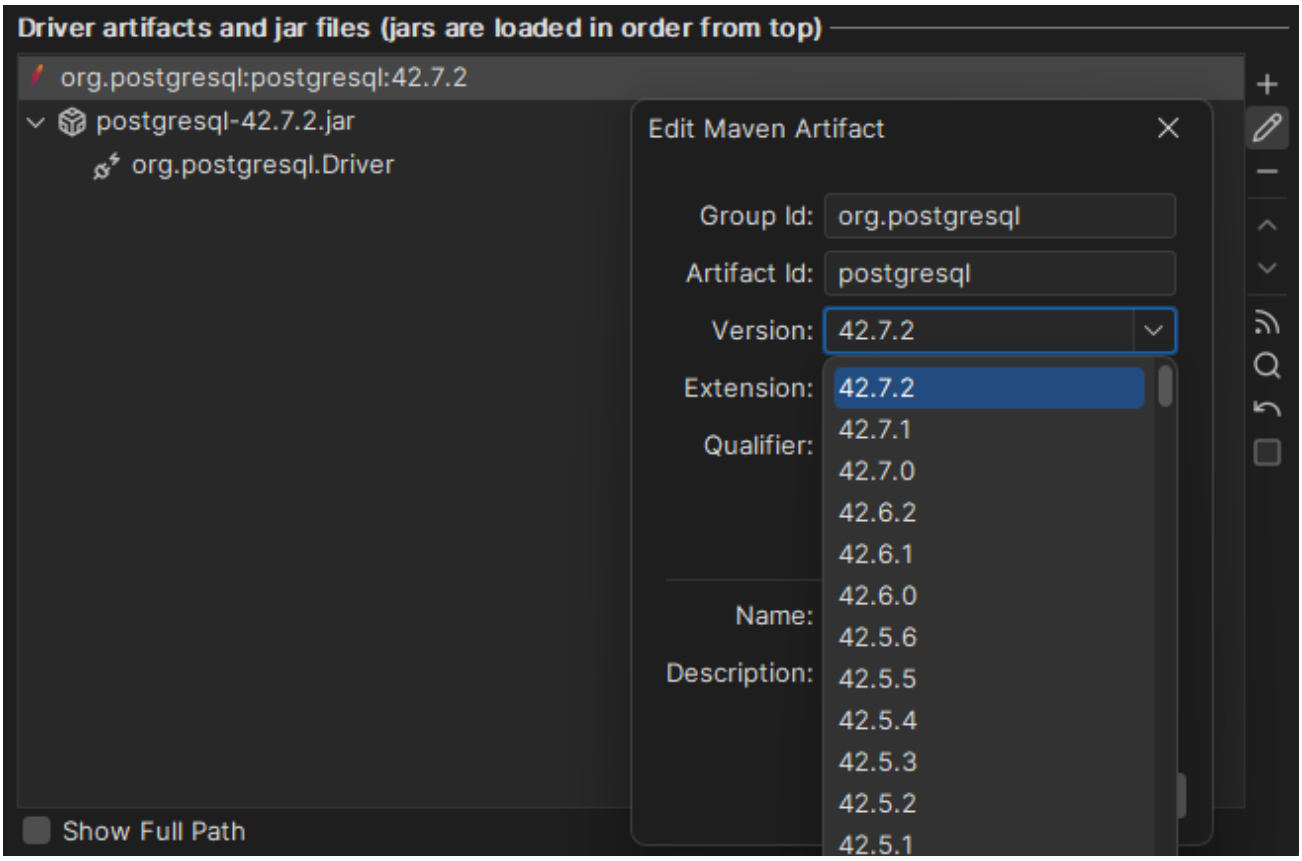
SETUP A JDBC DRIVER

The recommended way to setup a predefined driver without bundled driver files is to pick a matching driver template from the list and create a user driver and then add artifacts, files or a or directory that keeps the driver class(es). For instance, if you are going to load the JDBC driver for **Db2 DataDirect**, select the corresponding driver entry in the list and press the plus icon (or use the context menu). You can also copy an existing user driver. Artifacts (HTTP, or Maven), files and folders can be added via drag&drop, copy&paste or via the plus icon on the right side that opens editors or a file dialog). Maven artifacts are identified by the dependency that can be found in e.g. https://mvnrepository.com/ (copy&paste/drag&drop this to the left area or to the Maven editor)

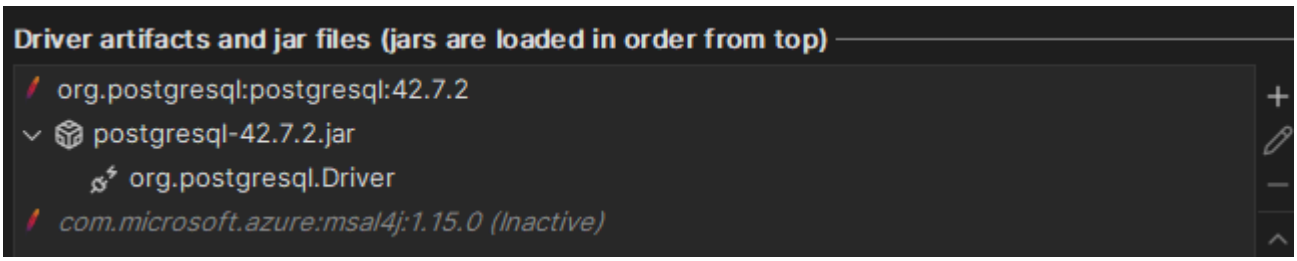



Artifacts must be downloaded via the globe icon on the right. The dowloaded or added files/folders, are scanned and if all works well a a green checkbox is displayed at the driver in the list

The preferred way to download is via Maven. The Maven editor allows you to check what versions exist. Both the Maven and the HTTP editors allows to set a name and a description.




Artifacts can be set inactive i.e. not used in download. This is mainly used in templates for artifacts that should only be used for specific connection purposes e.g. special authentication. Inactive artifacts are displayed in italic and in a disabled color:




Check the following online web page with the most current information about the tested databases and drivers:

https://www.dbvis.com/features/database-drivers/

- It lists which databases and drivers we have tested
- Download links to JDBC drivers
- Information about which files to load in the driver manager for each JDBC driver
- Information about which Driver Class to choose

A JDBC Driver implementation typically consists of several Java classes. Java classes are typically organized using a package name structure. Example:

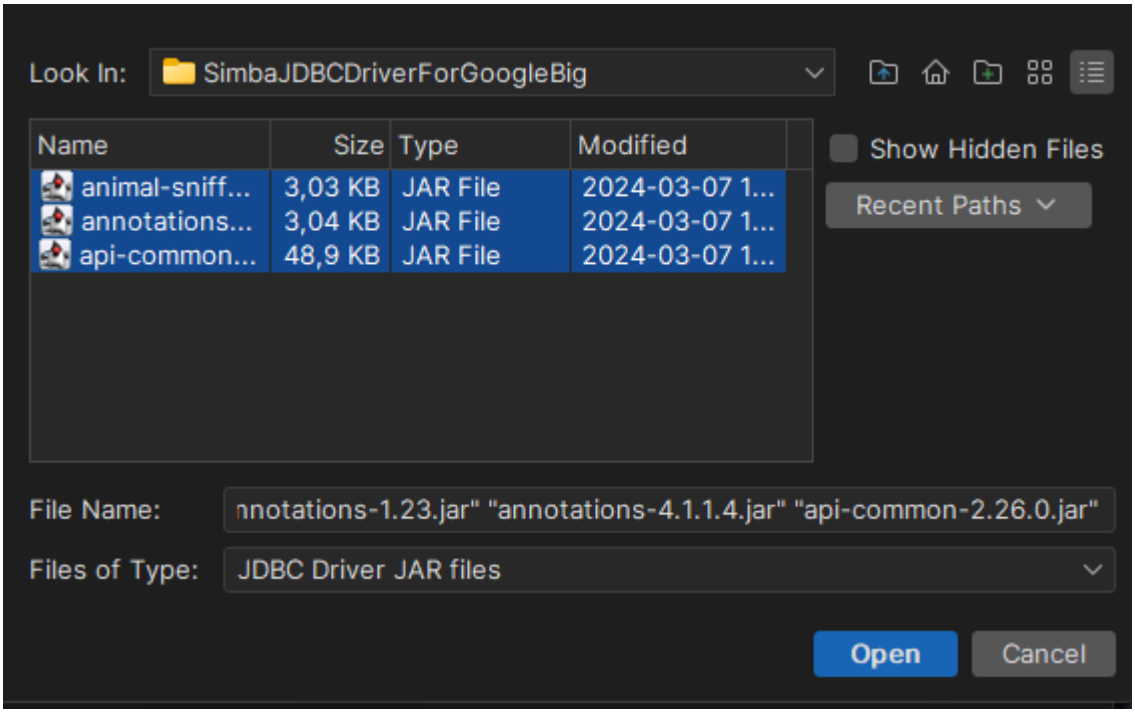
```
oracle.jdbc.driver.OracleDriver
```

When you load files in the Drivers artifacts and files tree, DbVisualizer **scans** each file to find the classes that represent main driver classes. Each such class is listed under the path where it was found in the Driver JAR Files list, and it is also added to the **Driver Class** list in the Driver Settings area above. If there is more than one class in the list, make sure you select the correct Driver Class from the list. Consult the driver documentation (or the Databases and JDBC Drivers page) for information about which class to select.

When Driver Manager scans the loaded files in Drivers artifacts and files tree, the files are searched from the top of the tree, i.e., if there are several identical classes, the topmost class will be used. Loading several paths containing different versions of the same driver in one driver definition is not recommended, even though it works (if you do this, you must move the driver you are going to use to the top of the tree). The preferred method for handling multiple versions of a driver is to create several user driver definitions.

#### JDBC DRIVERS THAT REQUIRE SEVERAL JAR FILES

Some drivers depend on several JAR files, or directories. Simply select all JARs at once and use copy&paste, drag&drop or the file dialog to add them. The Driver Manager will then automatically scan each of the loaded files and present any JDBC driver classes it finds.




#### ERRORS (WHY ARE SOME PATHS RED?)

A path in red color indicates that the path is invalid. This may happen if the path has been removed or moved after it was loaded into the driver manager. Simply remove the erroneous path and locate the correct one.

### Several versions of the same driver

The Driver Manager supports loading and using several versions of the same driver concurrently. We recommend that you create a unique user driver configuration per version of the driver and name the driver configuration properly, e.g., **Oracle 9.2.0.1**, **Oracle 10.2.1.0.1**, etc.

## Using drivers depending on native API (Type 2 JDBC driver)

The JDBC type 2 driver, also known as the Native-API driver, is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. For example: Oracle OCI driver is a type 2 driver.

These drivers often require additional components, such as DLL's or dynamic libraries, to be installed in addition to the driver JAR files. As mentioned earlier using a Type 2 driver is not recommended in DbVisualizer as pure java (Type 4) drivers often exists. An example is the Oracle Thin JDBC driver (Type 4) which is preferred over the Oracle OCI driver (Type 2).

Adding and using a Type 2 driver in DbVisualizer often involves driver dependant configuration tasks outside DbVisualizer. In some cases this is though not required as LIBRARY files can be directly configured for the driver. This is of course optional and configuration of libraries outside of DbVisualizer is always possible (using LD_LIBRARY_PATH or other environment configuration required by the driver).

Some examples follows

**SSO or Windows Authentication using JTDS**

In this case the driver template includes references to the needed LIBRARY file which is used in runtime by DbVisualizer. I.e no additional configuration is required by DbVisualizer.

**SQL Server JDBC Driver**

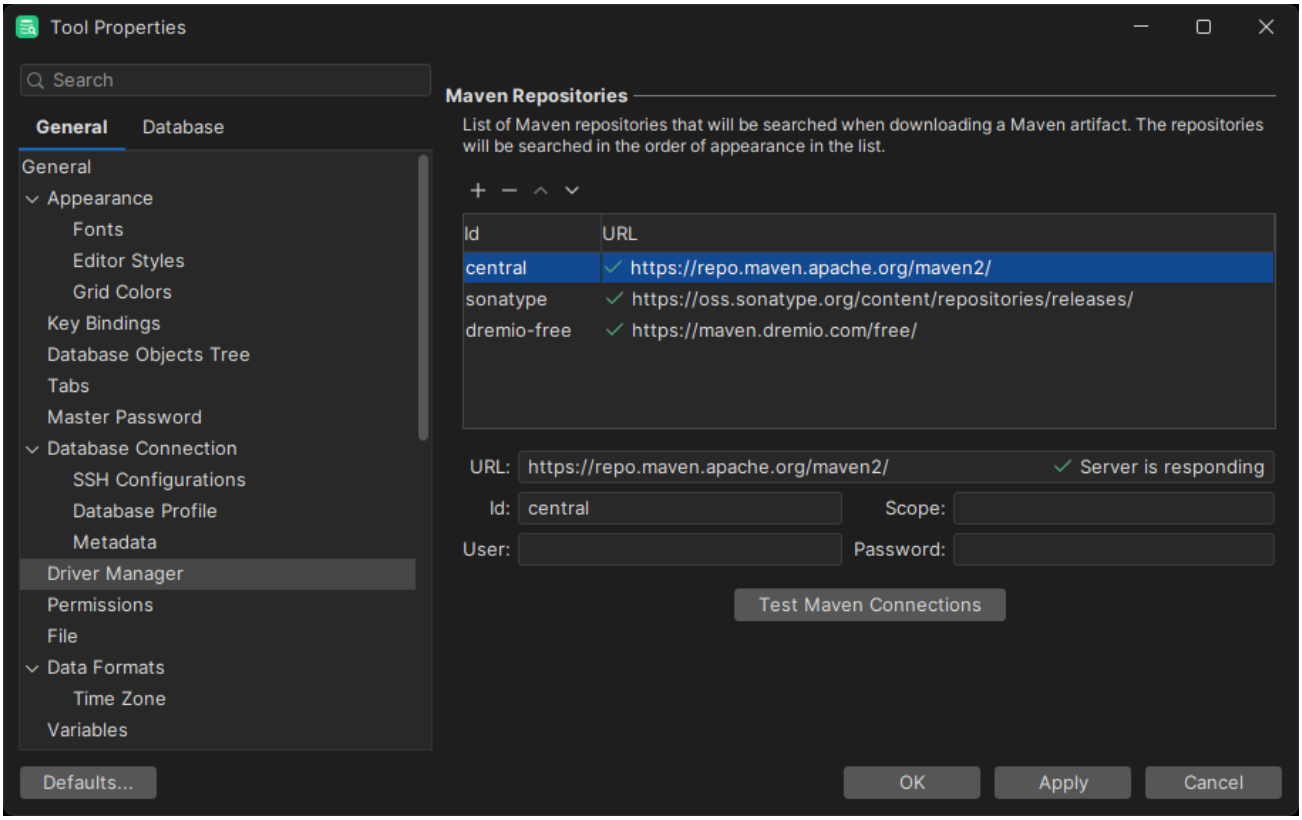
This driver definition also contains libraries (dll files) which is used by DbVisualizer without any need of additional configuration.

> Changing drivers with LIBRARY artifacts require a DbVisualizer restart in order for the change to take effect.

## Maven and Maven Repositories

Maven is a tool and framework for accessing and downloading 3rd-party libraries. Open source organisations and commercial companies provides JDBC driver libraries to on-line repositories that supports Maven for download. Many organisations also use internal Maven repositories. DbVisualizer comes with internal support for downloading with Maven in the Driver Manager. To download a driver using Maven you will need the **groupId**, **artifactId** and **version** for the driver, to be entered as a Maven artifact specification in the Driver Manager. In some cases you will also need an URL to a Maven repository including login credentials for the repository.

DbVisualizer comes with predefined templates containing Maven Artifacts in the Driver Manager and URLs for common Maven repositories. You can add Maven repositories to search for in Tools→Tool Properties→Driver Manager.




Repositories are searched from top down. Order can be changed and you can add and remove repositories. The scope is used to limit the search.

## Offline use

If you want to use Maven-based drivers without accessing public repositories, you can remove the default repositories from the list and add your own custom repository instead. If you don't have a custom repository or need to use DbVisualizer in an offline setting, you can pre-package relevant driver artifacts and distribute them to your users in the following way.

**Download relevant driver artifacts**

- Open a clean DbVisualizer workspace.
- Open Tools→Tool Properties→Driver Manager and create user drivers for the drivers that you are interested in. When doing so, make sure to download the corresponding artifacts.
- Open the workspace directory and copy the `drivers` subdirectory to a different location. This is the package you will distribute to your users.

**Distribute the driver artifacts package**

- Copy the `drivers` package to the user's workspace directory.
- Now, when the user creates a driver, the pre-packaged artifacts will be used where applicable.

# Special Properties

DbVisualizer utilizes a few special properties that you can use to modify characteristics of the application. These properties are available in the DBVIS-HOME/resources/dbvis-custom.prefs file.

| Property | Description |
| --- | --- |
| **dbvis.-AutoSaverRunInterval**=30 | The number of seconds between auto-saving open SQL editors. |
| **dbvis.disabledataedit**=false | Specifies if table data editing should be completely disabled, i.e. the form and inline editors. Note: This has an effect only when used with a licensed edition. |
| **dbvis.driver.ignore.dir**=lib:resources:.install4j | Specify directories from DBVIS-HOME that should not be listed in the Driver Manager "System Classpath" list. Directories are separated with ":". Accepted values: one or several directory names starting from DBVIS-HOME. |
| **dbvis.grid.encode**=false | Specifies if encoding of data in result set grids will be performed or not. If set to true then make sure the dbvis.grid.fromEncode and/or dbvis.grid.toEncode are also set. |
| **dbvis.grid.fromEncode**=ISO8859_1 | Encoding used when translating text data that is fetched from the database |
| **dbvis.grid.toEncode**=GBK | Encoding used when translating data that will appear in the result set grid |
| **dbvis.removepartialresultsets**=false | Defines whether the result set(s) should be removed when interrupting an ongoing execution in the SQL Commander. |
| **dbvis.savedatacolumns**=false | Column layout changes such as reordering and/or visibility are saved for all grids in the Objects Views *except* for the "Data" grid. This property can be used to also include the layout in the "Data" grid. Note: This will result in DbVisualizer saving the layout for each table that is displayed in the Data grid = huge XML file... |
| **dbvis.showactionresult**=false | This defines whether the result for all actions should be displayed or only failures (default). |
| **dbvis.sqlwarning.maxrows**=5000 | Defines the number of SQL Warning rows that should be processed before truncating. |
| **dbvis.usegetobject**=false | Specifies if the generic ResultSet.getObject() method in JDBC will be used in favor of the data type specific get methods or not. Default is false. |
| **dbvis.usestandardgridfit**=false | Enable this property and DbVisualizer will use an accurate but slow method to automatically resize grid columns. "Accurate" since it does a real calculation of the columns width. If leaving this property disabled then column widths are determined much faster but depending on what grid font is used some columns may be truncated with "...". This property has an effect only if **Grid / Auto Resize Column Widths** is enabled in **Tool Properties** |
| **dbvis.-ConnectionTestTimeout**=20 | The timeout in seconds for the "Test Connection" feature. |
| **dbvis.<database>.IgnoreMaxRowsForNonSELECT**=true | Ignore the Max Rows setting for statements other than SELECT. MS SQL Server applies Max Rows also to DELETE, INSERT and UPDATE (upto and including SQL Server 2008). |
| **dbvis.<database>.-RemoveNewLineChars**=false | Backward compatibility setting used to specify that the SQL command will be trimmed of all whitespaces, tabs and newlines just before it is executed by the DB server. |
| **locale**=en,us | Use this to specify an alternate Locale |
| **dbvis.-FileForceSync**=true | By default, all XML settings files are synced with the underlying storage device at the time when these are saved. Use this property to disable the syncing and instead rely on OS syncs. **Note**: relying on OS |

| Property | Description |
| --- | --- |
| | syncs and performing an uncontrolled shutdown of Windows may corrupt files. |
| **dbvis.-MasterPasswordRule**=.{8,} | By default, a Master Password must be at least 8 characters. This is the definition of the default implementation. Please see the dbvis-custom.prefs file for some additional examples. |
| **dbvis.-MasterPasswordRuleDescr**=\ The new password must be at least 8 characters long | Use this to specify a description of the master password rule. |

You rarely need to modify these properties, as the default values are sufficient for most usage. Also note that these properties may change in future versions of DbVisualizer. Some are also experimental and may be removed or instead introduced in the DbVisualizer GUI.