SQLiteStudio

User manual

Version 2.1.0

http://sqlitestudio.one.pl

Contents

1) Installation and running

1.1) Supported platforms

Operating system	Platform	Status	Binaries provided
Linux	ix86 (32bit)	🟩	✓
Linux	ix86_64 (64bit)	🟩	✓
Windows 9x/Me	ix86 (32bit)	🟦 [1]	✗
Windows NT/XP/Vista/7	ix86 (32bit)	🟩	✓
Windows NT/XP/Vista/7	ix86_64 (64bit)	🟩	✓[2]
Solaris	ix86 (32bit)	🟩	✓
Solaris	SPARC (32bit)	🟦	✗
Solaris	SPARC (64bit)	🟦	✗
FreeBSD	ix86 (32bit)	🟩	✗[3]
FreeBSD	ix86_64 (64bit)	🟩	✗[3]
OpenBSD	all	🟦	✗
NetBSD	all	🟦	✗
MacOS X	ix86_64 (64bit)	🟩 [4]	✓
MacOS X	PPC	🟦	✗
MacOS	PPC	🟥	✗
AIX	IBM POWER	🟨	✗
QNX	all	🟨	✗
HP-UX	all	🟨	✗
AmigaOS	PPC	🟥	✗

[1] *That OS is not supported, but some older versions (1.0.x) of SQLiteStudio might work.*

[2] *Binaries are 32bit, but Windows 64bit supports 32bit binaries just fine.*

[3] *Binaries for FreeBSD and Solarix ix86 used to be maintained, but they took too much effort to maintain and they were rarely used. FreeBSD is still fully supported though and SQLiteStudio can be run from sources there.*

[4] *MacOS X port is known to be quiet buggy, but it works. The less buggy version will be released in some future. It requires lots of work.*

🟩	Application has been tested to work on this platform.
🟦	Application wasn't tested on this platform, but it should work.
🟨	Application wasn't tested on this platform and it's hard to say if it works.

🟥	This platform is unsupported and application won't work on it.

If you have some operating system/platform not mentioned in the table above, you can always try to run SQLiteStudio from sources. All you need to collect some dependencies mentioned in chapter 1.2.

Be aware, that it might not support your platform even running from sources.

1.2) Installing from sources

Installation from sources is much harder than from binaries (see 1.3).

SQLiteStudio 2.1.0 runs only with Tcl 8.5.11 or higher and depends on several binary Tcl extensions. This is a list of extensions required to run application:

- Itcl 3.4
- Itk 3.4
- SQLite 3.7.11
- SQLite 2.8.17 (optional)
- treectrl 2.3.2
- tkpng 0.7 or Img 1.3 (the png part of it)
- tkdnd 2.6

You have to collect and install proper Tcl and all extensions for your platform (later versions of extensions are usually acceptable).
Easiest way to do this is to download and install the ActiveState ActiveTcl distribution, which includes Tcl and all necessary extensions.

When they're ready then you can download SQLiteStudio sources package, unpack it, then enter sqlitestudio directory and execute `main.tcl` file using `tclsh` application.

To uninstall application, just delete whole sqlitestudio directory which was extracted from sources package.

Configuration directory (the “CFG_DIR”):
If you want to also delete configuration files, you have to do following:

On Windows:
Delete "sqlitestudio" directory:

- For Windows 2000/XP/2003/Vista it's placed in directory pointed by %APPDATA% (usually `C:\Documents and Settings\<PROFILE_NAME>\Application Data\`).
- For Windows 98/Me it's placed in directory pointed by %HOME% environment variable.

On Linux/Unix/MacOSX:
Delete "`.sqlitestudio`" directory in home directory of each user that run application at least once.

1.3) Installing from binaries

Installation from binaries is as simple as possible.

In few words: **Just download it and run!** No installators, no zip or rar files. You don't have to install anything else.

It really is that simple!

If you're curious about where is the configuration kept, or you still have problems running it – read below.

Permissions

On all operating systems you need to take care about permissions of your user to directory where you put sqlitestudio.exe binary file. Your user has to be able to create directories and files in that directory, otherwise it's possible that SQLiteStudio will raise critical error during startup, or at runtime.

Directory name

Binary distribution is sensitive to strange characters in it's path. It's known that some specific unicode characters cause problems with running SQLiteStudio. An example would be (under Windows):

```
C:\Users\some_user\Рабочий стол\sqlitestudio-2.0.5.exe
```

Running binary from that place will fail. The problem is these Russian characters here. Please avoid such situations and you should have no problems.

Windows

If you want to uninstall application - just delete this file. If you want to also delete configuration files, you have to delete "sqlitestudio" directory:

- For Windows 2000/XP/2003/Vista it's placed in directory pointed by %APPDATA% (usually `C:\Documents and Settings\<PROFILE_NAME>\Application Data\`).
- For Windows 98/Me it's placed in directory pointed by %HOME% environment variable.

Linux/Unix/MacOSX

Installation for Unix systems is pretty much like for Windows systems. The only difference is that you might need to set execution permission for downloaded file with command:

```
chmod +x sqlitestudio-<VERSION>.bin
```

Change "<VERSION>" to fit the file name you've just downloaded, of course.

For Unix-like systems it is good idea to rename file to "sqlitestudio" and move it to directory included in environment variable PATH, so it can be run from console by writting simply: `sqlitestudio`

If you want to uninstall application - just delete the file. That's all.

If you want to also delete configuration files, you have to delete "`.sqlitestudio`" directory in home directory of each user that run application at least once.

2) Basics

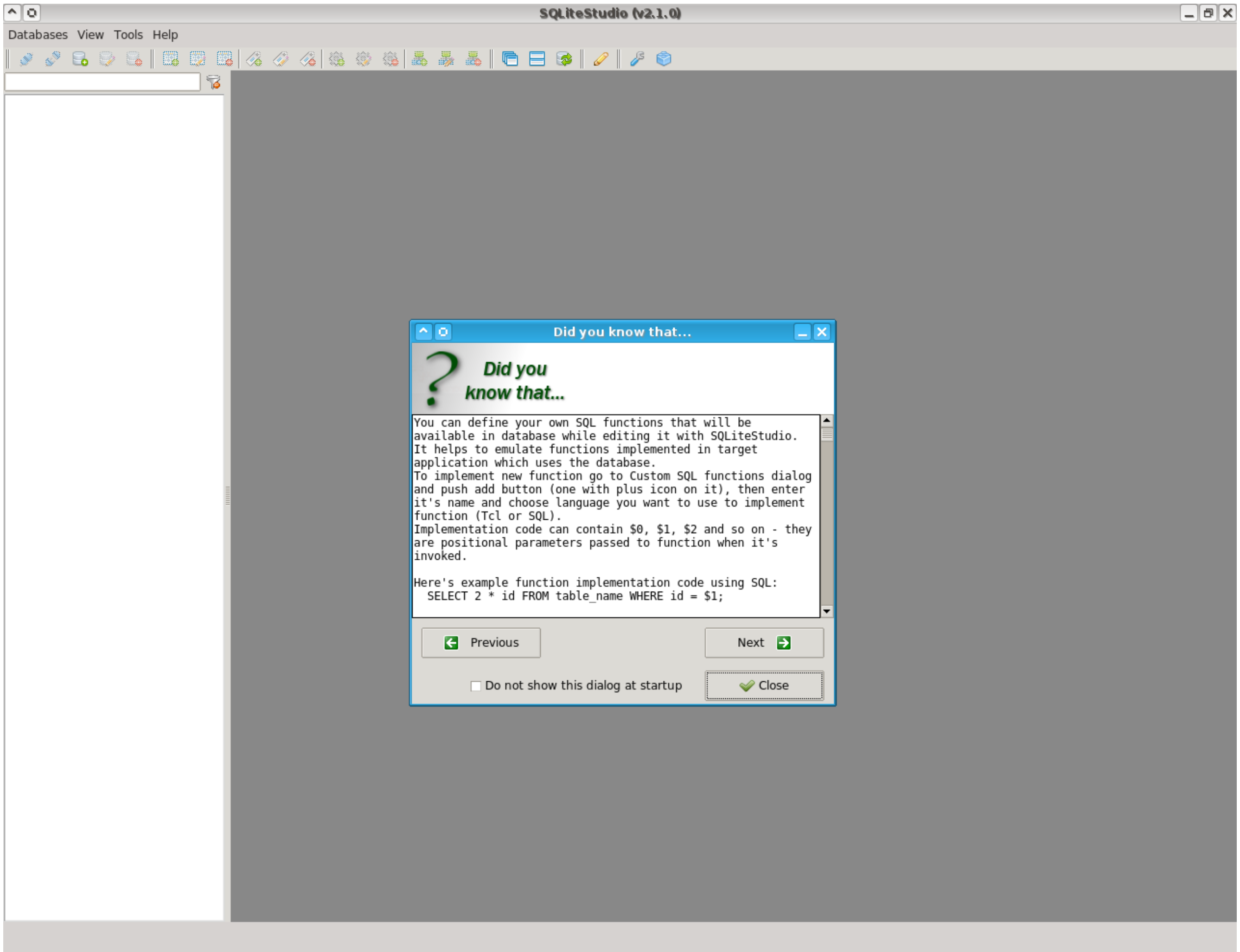
Here's few informations for complete newbies:

- SQLiteStudio is NOT the same as SQLite. SQLite is database, SQLiteStudio is application to manage such databases.
- SQLite is relational database with SQL support. If you don't know how to use/manage SQLite databases, read about relational databases and SQL first.
- SQLite database has no client-server model. Here databases are files, so you cannot connect to some remote host on some strange port, log in and manage database. You need to select local file which is (or is going to be) database.

2.1) Interface introduction

2.1.a) First start

At first start of application you will see main window and dialog known as "tips&tricks". It is good idea to read tips, since they can make your work much easier in the feature.



```
You can define your own SQL functions that will be
available in database while editing it with SQLiteStudio.
It helps to emulate functions implemented in target
application which uses the database.
To implement new function go to Custom SQL functions dialog
and push add button (one with plus icon on it), then enter
it's name and choose language you want to use to implement
function (Tcl or SQL).
Implementation code can contain $0, $1, $2 and so on - they
are positional parameters passed to function when it's
invoked.

Here's example function implementation code using SQL:
  SELECT 2 * id FROM table_name WHERE id = $1;
```

□ Do not show this dialog at startup

It will also check if new version is available. This feature can be disabled from Configuration window. No information is sneaked by it. The request that SQLiteStudio sends to server for new version information contains absolutely no information from local computer. This can be checked in source code for those, who don't trust these words.

It's good idea to go to Configuration window and check for configuration options, so you can adjust application look and behavior for your needs.

Many interface elements have their context help – just hold mouse over the element for a second and help balloon should appear. It especially helps with configuration options mentioned above, since they can be hard to understand at the very first look.

Windows only – file associations

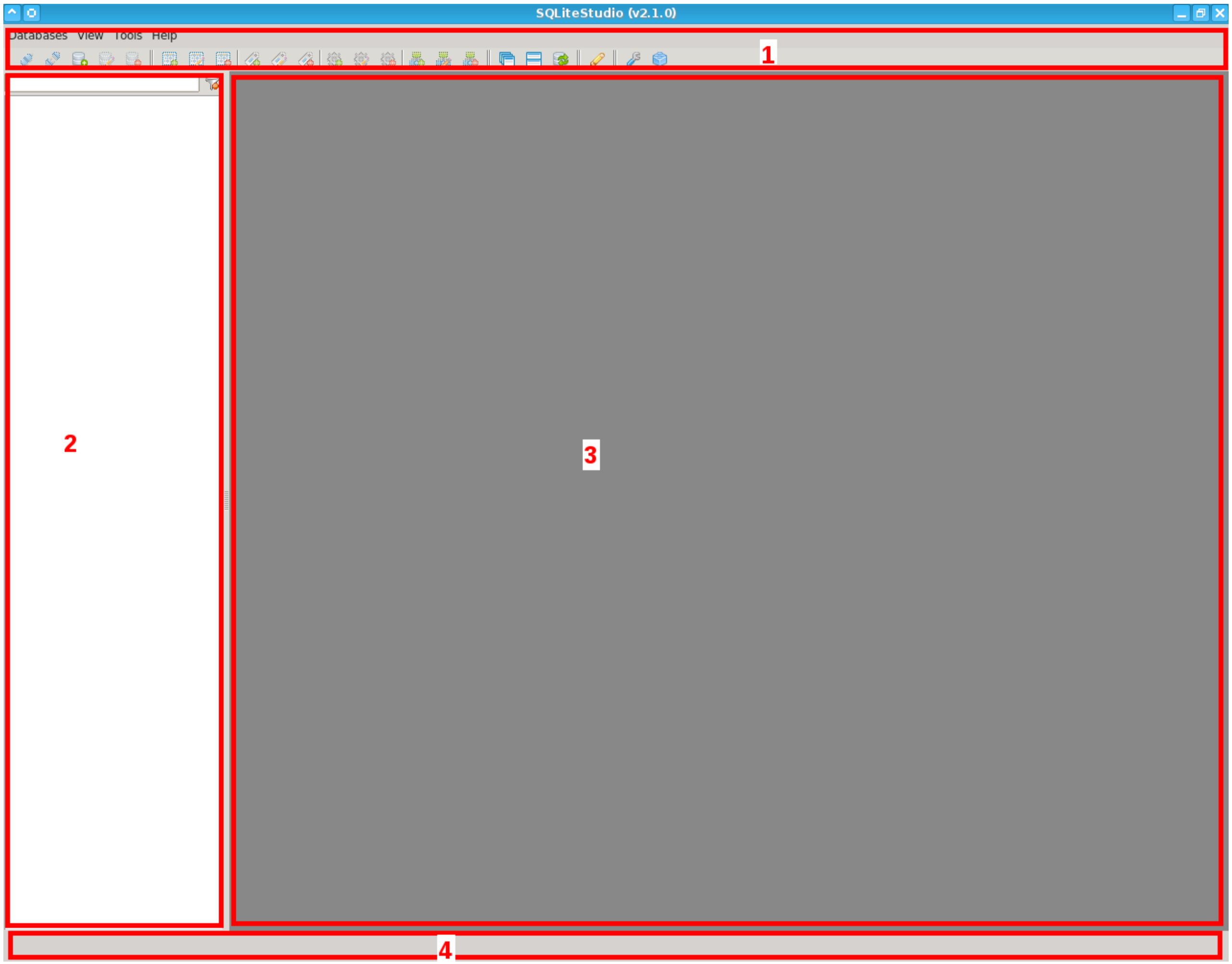
SQLiteStudio checks at startup what file associations were defined for previous SQLiteStudio version (if there was any) and might propose to redefine them for new version.

It works differently on certain versions of Windows. It should work automatically on Windows up to XP. With Vista being released, it introduced UAC and things got tricky. SQLiteStudio will try to fix file associations for you, but it might fail. Therefore you will have to do it manually by choosing SQLiteStudio as default application for certain file types you open (the databases).

2.1.b) Main window

The main window contains 4 parts:

1. Toolbar and menu on the top,
2. Databases tree on the left,
3. MDI area at the center,
4. Taskbar on the bottom.



Tasks order on Taskbar can be arranged manually by dragging and dropping tasks. Active tasks are saved on application exit and restored after next start.

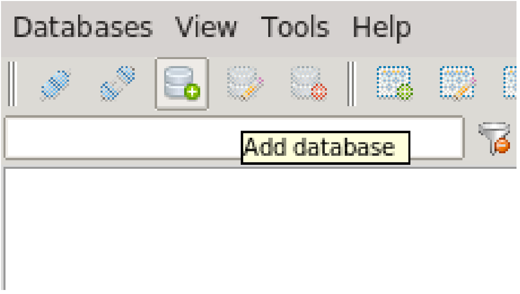
There are two main types of tasks, which are represented as MDI windows:

- Table window,
- SQL editor window.

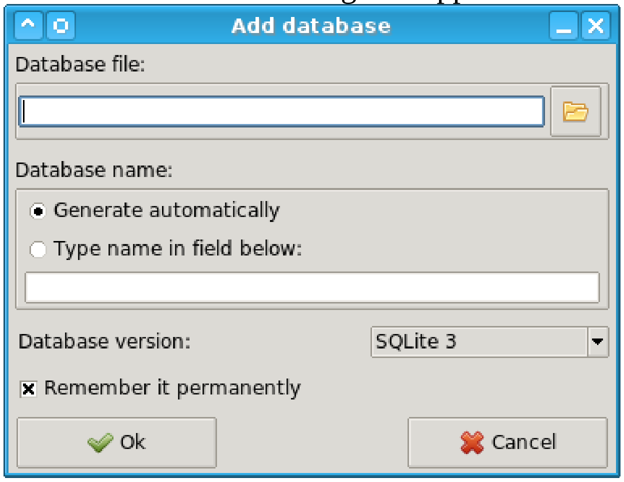
2.1.c) The daily routine - step by step

At daily basis user probably would want to add or create databases, create or modify tables and edit the data in tables. So here's how it's done.

- **Create database**
    1. Click on "Add database" button in toolbar.



2. The database edition dialog will appear:



Database file:

Database name:

● Generate automatically

○ Type name in field below:

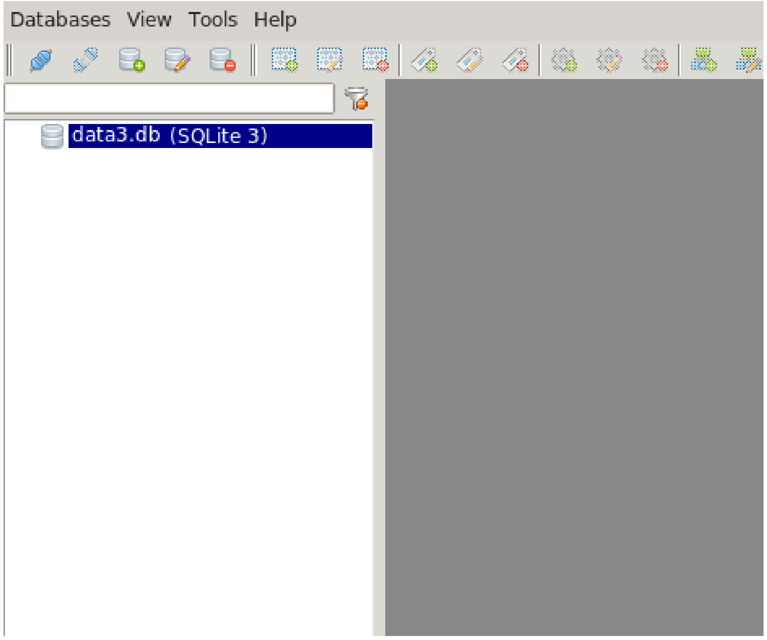
Database version: SQLite 3

☒ Remember it permanently

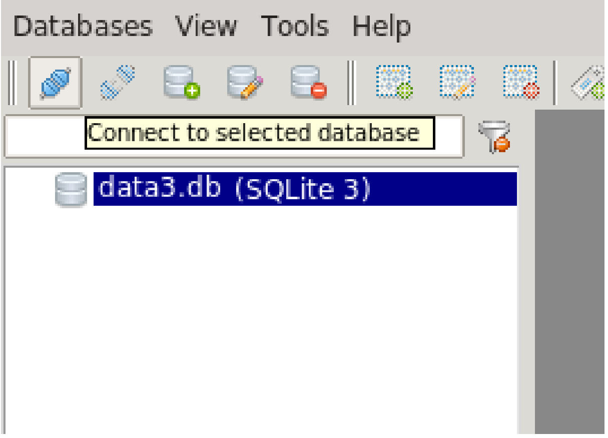
3. Now Pick a file placement and a name for your new database. Also decide if you want SQLite3 or SQLite2 database.

   The "*Remember permanently*" checkbox at the bottom is checked by default. If it's enabled, then the database will be stored in the configuration and will be available every time you run SQLiteStudio. If you disable the checkbox, then the database will be available in SQLiteStudio only for this one session and will disappear after restart.

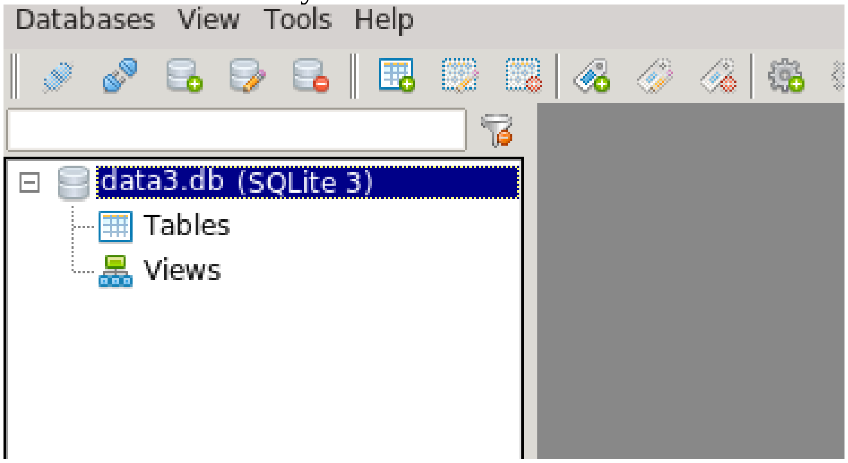
4. Now click "Ok". The database will be added on the left side of the main window:



5. Double-click on the database, or click on "Connect to database" button:



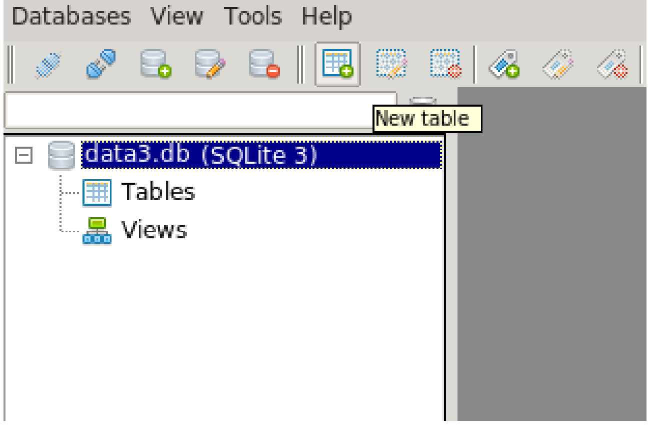
6. You should have database connected now. You can tell it by having expandable tree under the database entry:



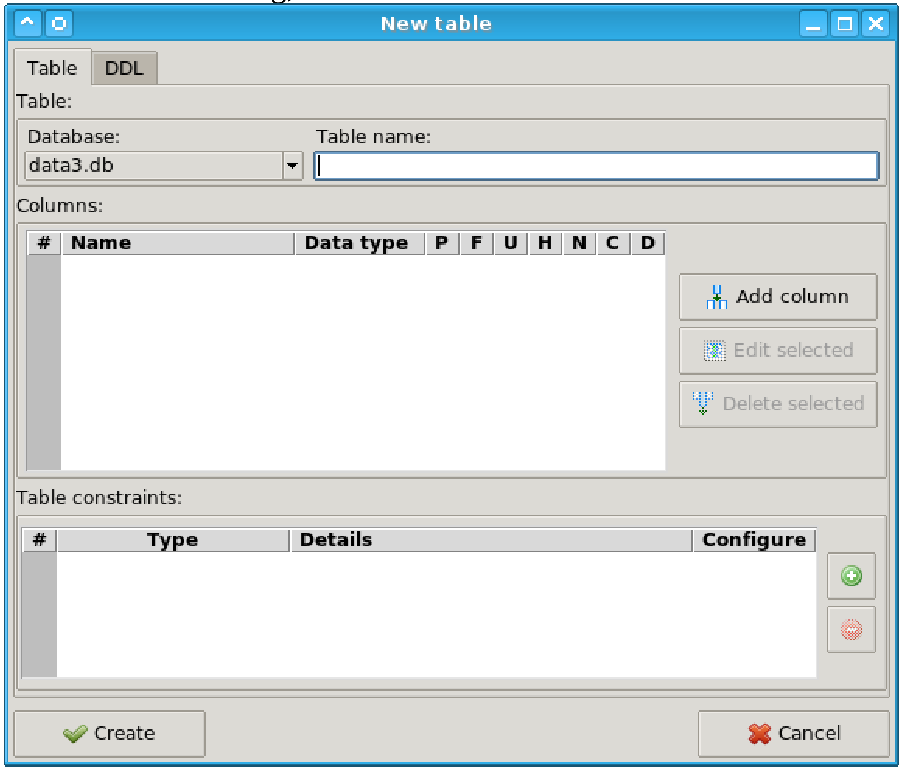
- **Edit existing databases**
  You need to take exactly the same steps as for **Create database**, except this time you won't make up the database file and name, but you'll point to existing database file name.

- **Creating new table**

  1. Click on "New table" button in toolbar.



  2. You'll see table dialog, more or less like this:



  3. Type a table name:

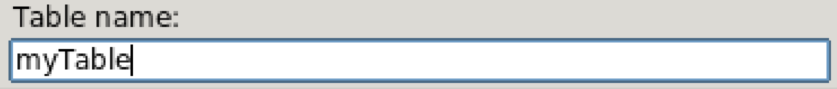
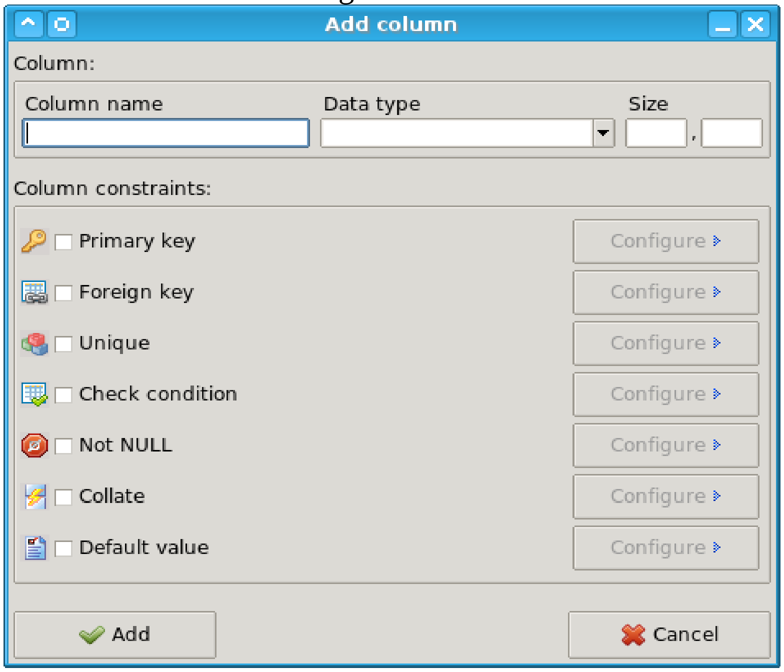


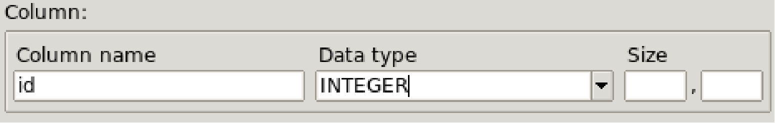
Table name: myTable

  4. Press "Add column" button:

5. You will see column dialog:



6. Type in the column name and it's type (the latter one is option):


Column:

Column name: id

Data type: INTEGER

7. Press "Add" button at the bottom:

8. Repeat column adding to add one more column (just for needs of this example).

9. Now you should have table dialog somehow similar to this:

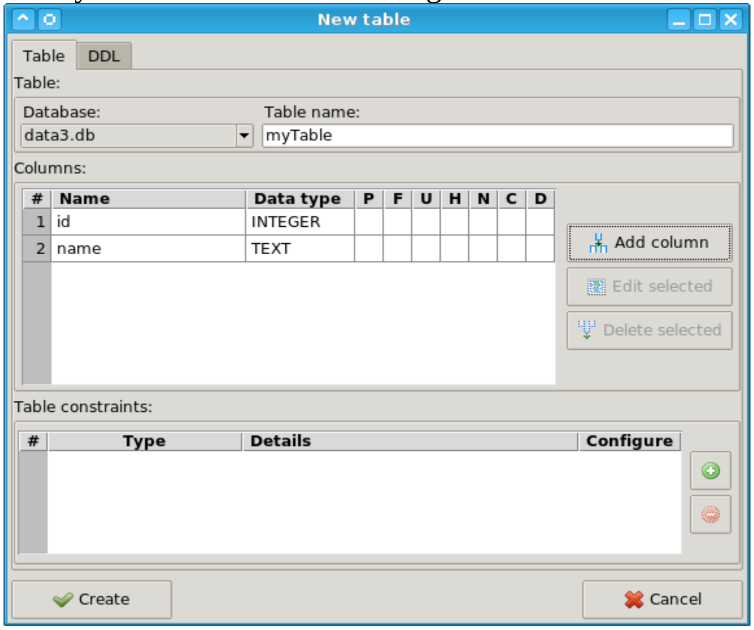


Table:

Database: data3.db

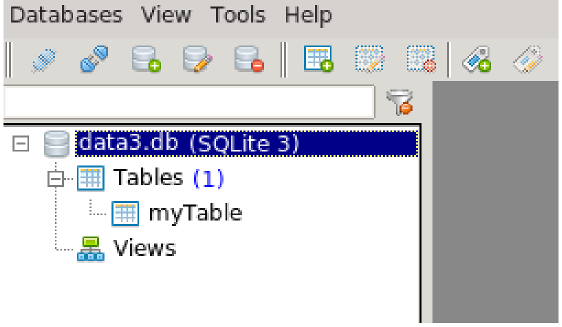
Table name: myTable

Columns:

#	Name	Data type	P	F	U	H	N	C	D
1	id	INTEGER							
2	name	TEXT							

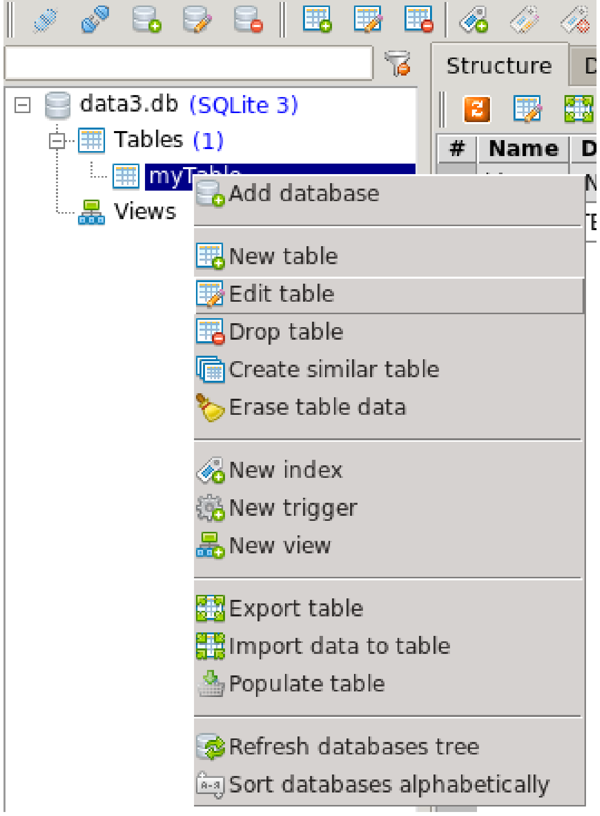
10. Click "Create" to finalize the operation and create the table:

11. Now you should see your table on the left side:



- **Editing existing table**

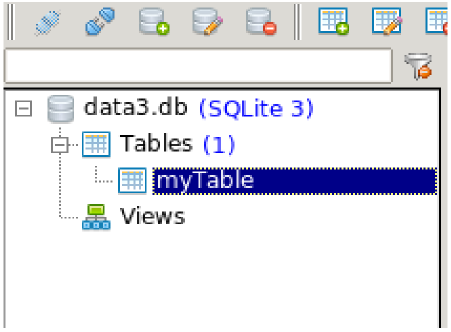
1. Right-click on the table entry on the left side of the main window and pick "Edit table":



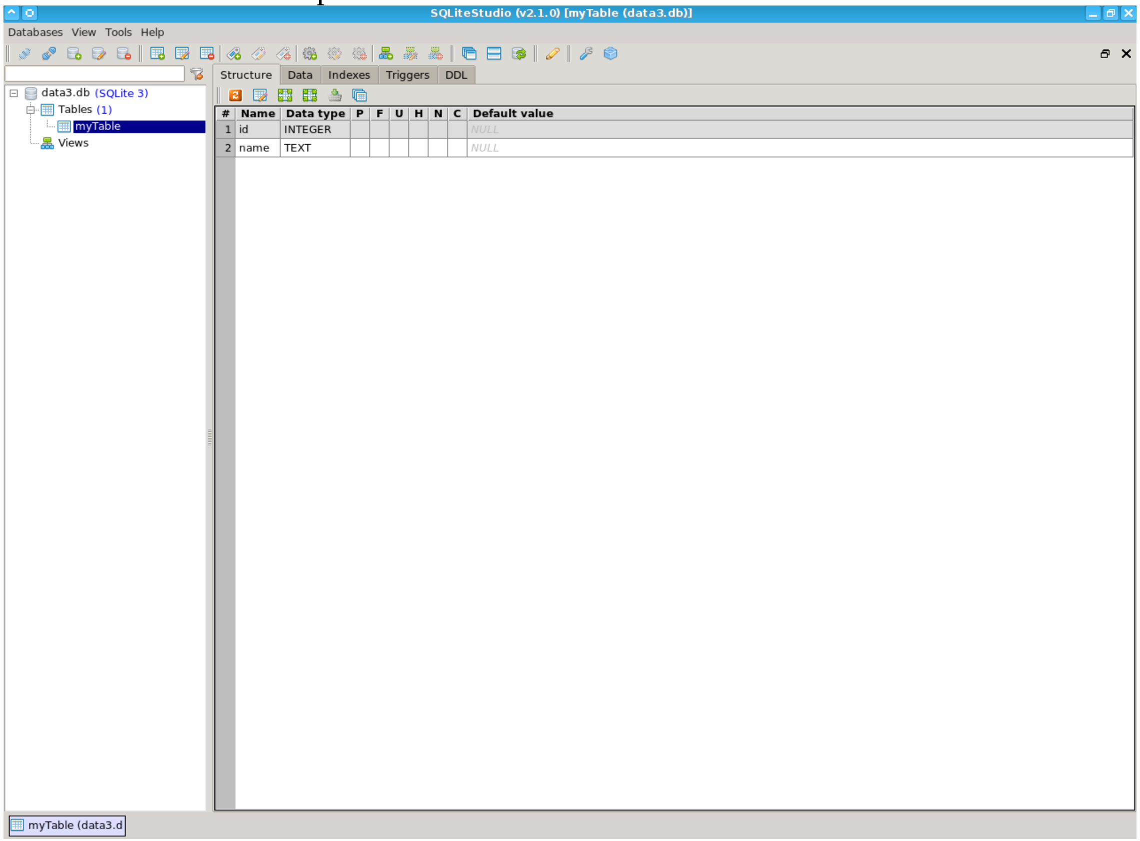
You will see the same table dialog as for creating new table, except it will already have some values filled in. You can change those values (change name, add/modify/delete columns, modify constraints) and then press "Change" to finalize the operation.

- **Modifying table data**

1. Double-click on the table entry on the left side of the main window:

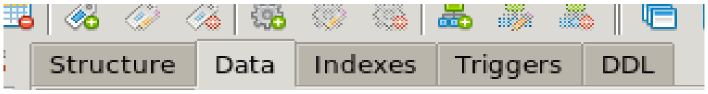


2. The table window will open:

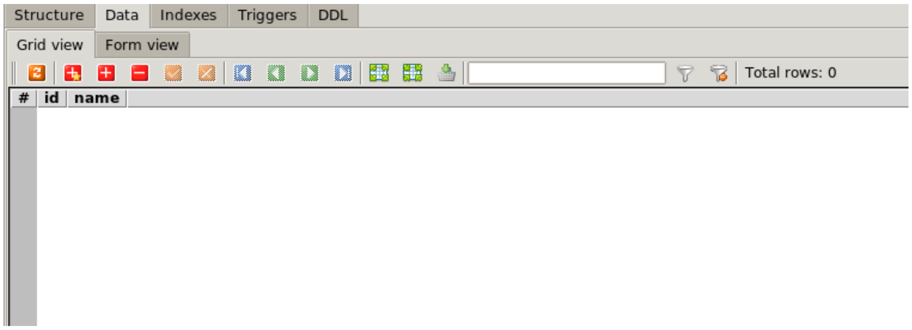


#	Name	Data type	P	F	U	H	N	C	Default value
1	id	INTEGER							NULL
2	name	TEXT							NULL

3. Click on "Data" tab:



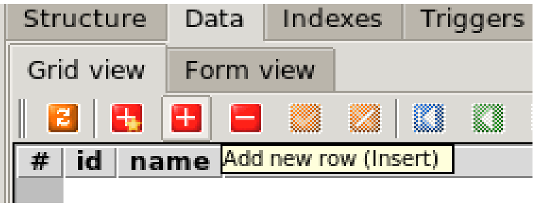
4. You will see the data view of the table. Since this is new table there will be no data:



Total rows: 0

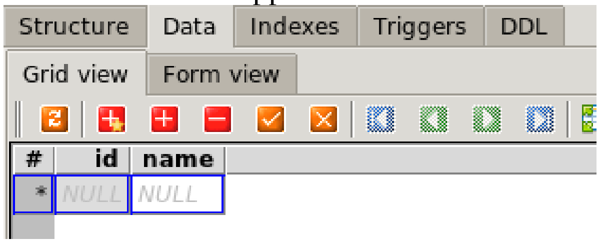
#	id	name

5. Now lets add some data. Click "*Add new row*" button:



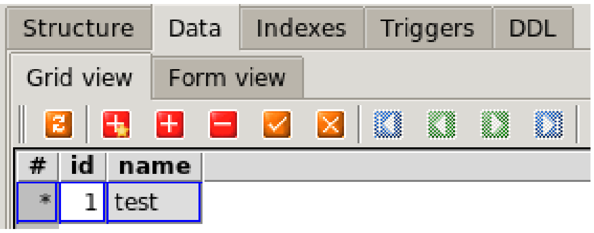
#	id	name

6. The new row will appear. It will contain NULL values by default:



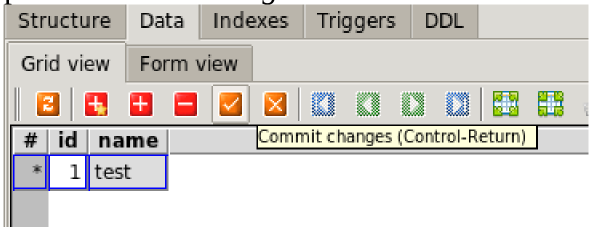
#	id	name
*	NULL	NULL

7. The row cells are also surrounded with blue border. It means that the row is still being edited and has not been committed to the database yet. Lets put some data to the cells:



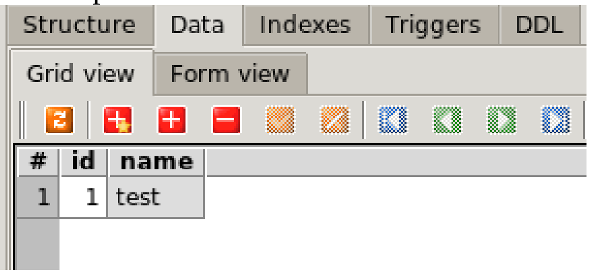
#	id	name
*	1	test

8. Good. To make the data persistent in the database you have to commit the row. To do so press "*Commit changes*" button:



#	id	name
*	1	test

9. The row should now be no longer surrounded with blue border. This means that the data is now persistent:



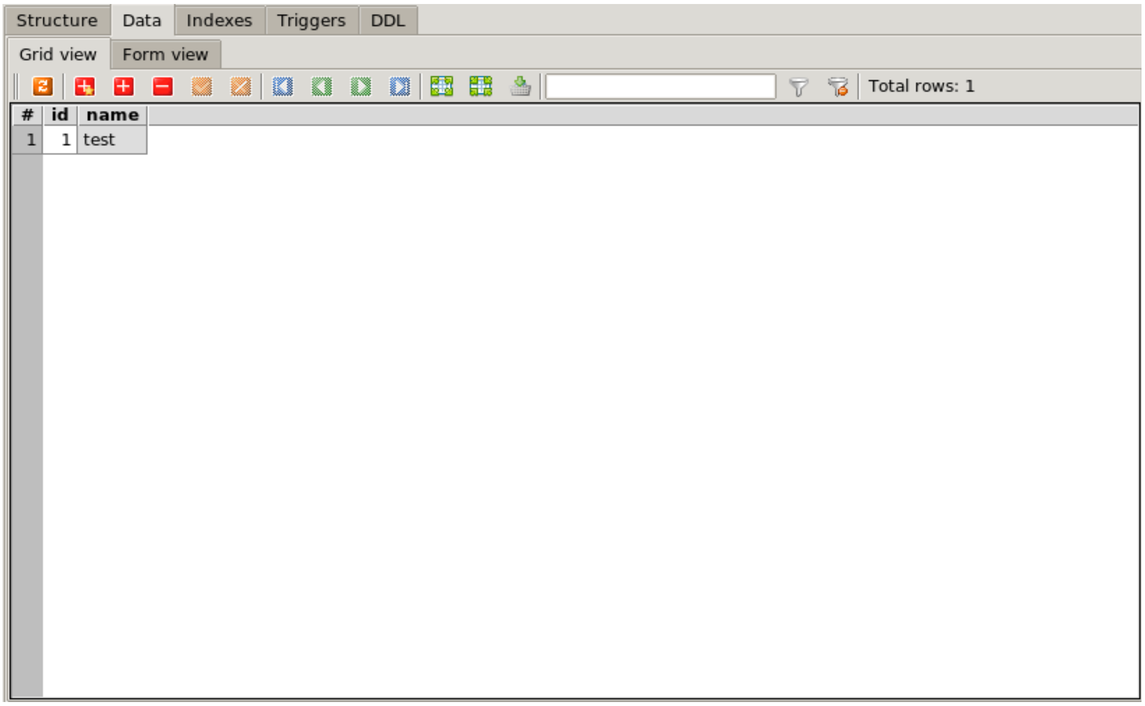
#	id	name
1	1	test

10. To edit any table cell, just double click on it to enter edition mode. You can also simply start typing new value for the cell while the selection is set to that cell.

2.1.d) Editing data

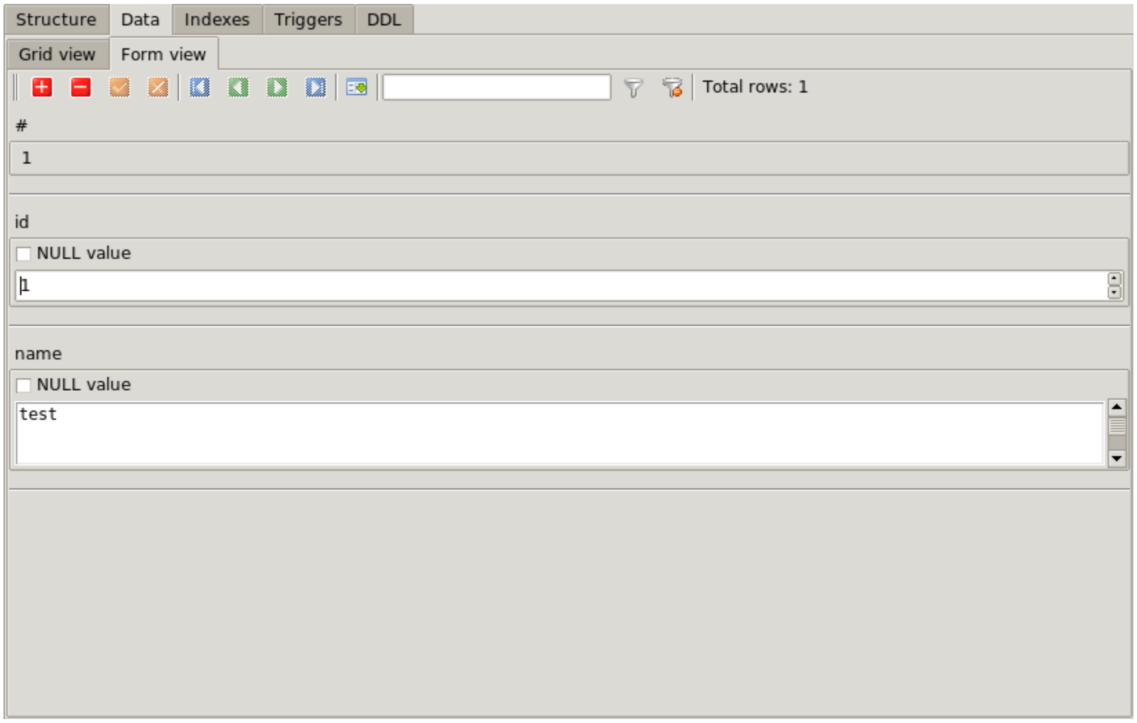
The data can be edited in either Table Window, or in SQL editor window.

Table Window provides data editing with Grid View or Form View, while SQL editor allows same ways as Table Window and additionally the generic way – by SQL queries.



#	id	name
1	1	test

Grid View



#

1

id

□ NULL value

1

name

□ NULL value

test

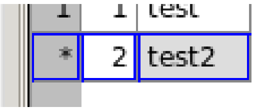
Form View

General rules are:

- **To insert new row** in Grid View you can use “Insert new row” from toolbar, or just “Insert” key from keyboard.
- **To delete selected row** in Grid View you can use “Delete selected row” from toolbar, or “Delete” key from keyboard.
- **To modify selected cell** in Grid View you can double click on the cell, or press “Enter” key from keyboard, or just start typing new value while having the cell selected.
- **To duplicate selected row** in Grid View you need to right-click and select “Duplicate row” from context menu.
- **To set NULL value for selected cell(s)** in Grid View, you can press Backspace key, or right-click and select “Set NULL value” from context menu.

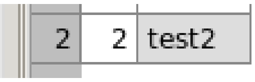
Some more details:

1. Table data view has more capabilities for editing data than query results view in SQL editor. You cannot insert new row, duplicate row, or delete row in SQL editor results. These actions are permitted only in table data view.
2. The Grid View acts very much like application to edit spreadsheet. It is actually possible to select cells in Grid View and copy them, to paste to MS Excel, OpenOffice Calc, or similar application. Same thing is possible in other way - from spreadsheet to SQLiteStudio.
3. You can hold *Control* key and click on the grid to select entire row, instead of single cell.
4. Form View is useful to edit long values of cells and BLOB type cells as hex, row by row. Hold mouse over rows navigation buttons (at top of Form View) to see keyboard shortcuts for them. They can be very handy.
5. Form View always displays full values of cells, while Grid View can truncate very long values to display them, because longer form would fit into the cell anyway. You will still work with full length values when editing cell in Grid View, they just won't be displayed whole in the cell.
6. Form View has resizable fields. It's very useful when user wants to work with big values, or large amount of small values in several columns - just adjust them.
7. Empty cell in Grid View doesn't mean its value is NULL. If cell has NULL value, you should see "NULL" label inside of that cell. You can always set NULL value to selected cells, you can use Backspace key, or "Set NULL value" from Grids context menu (right mouse click). In Form View, the NULL value is determined by "NULL value" checkbox at the top of each field.
8. To edit any cell (even empty one) with blob editor, you can right-click on the cell and pick "*Edit cell value in BLOB editor*".
9. No matter if you edit cells in Grid View or Form View, **new values are not pushed immediately into the database**. They are in **pending state** and wait for "**Commit**" or "**Rollback**" to be pressed in a toolbar above. You can tell if there are any pending commits by state of "Commit"/"Rollback" buttons, or by checking if there are any cells marked with blue border in Grid View.



1	1	test
*	2	test2

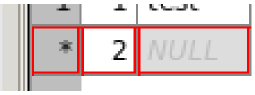
Uncommited cells



2	2	test2

Commited cells

If "Commit" fails because of constraint violation or some similar reason, then the proper message is displayed and cells/rows that could not be committed are marker with red border.



1	1	test
*	2	NULL

Tried to commit cells with values violating a constraint

The "Commit"/"Rollback" buttons in Form View commits or rolls back only single, current row, while same buttons in Grid View applies to all pending operations (deletes, updates or inserts). You can always commit/rollback only selected cells from Grid View - use adequate entry from context menu of Grid View (right mouse click).

10. Table window data view provides also filter entry in toolbar. You can type any token (a word, a sentence, a number, etc) and press Enter key or "Apply data filter" toolbar button to filter table data view to display only rows containing given token.
11. To disable data filtering use "Clear data filter" or manually delete contents of filter entry and apply it.

Notes on editing SQL results:

After executing query in SQL editor there might be some results, which will be represented in its Grid View and Form View. **Editing results is possible only if result column represents some real table column.** Multiple columns selected from multiple tables joined with “JOIN” operator can be edited as well.

To be more clear, lets walk through an example:

```
SELECT (numeric_value + 2) AS val,
       length(some_column) AS length,
       explicit_column,
       5
  FROM some_table;
```

First result column is sum of cell values and 2. It cannot be edited in results Grid View or Form View.

Second column is result of length() SQL function applied to column “some_column”, so it also cannot be edited.

Third column is table column selected explicitly, so it will be editable.

Fourth column is just a constant value with no relation to any table. It won't be editable.

Known issue:

Editing SQL query results from multiple “SELECT” statements joined with compound operator (see SQLite manual) won't work correctly. This is because of limited support of handling results in SQLite extension for Tcl and at this moment SQLite maintainers don't plan to fix it anytime soon.

2.2) Managing databases

Each database in SQLiteStudio is identified by its symbolic name. It's displayed in databases tree on the left side. You can specify it in Database Dialog window, while adding new database or editing existing one.

The file you're specifying in Database Dialog can be either:

- not existing – the file will be created and used for new database,
- existing – the file will considered as SQLite database and SQLiteStudio will try to check it to detect SQLite version valid for the file. If none of SQLite engines supported by SQLiteStudio can handle the file, then error dialog will be raised. Otherwise proper SQLite version will be set in drop-down list at the bottom right corner of Database Dialog.

Since SQLiteStudio supports SQLite in version 2 and 3, you can pick (while creating new database, using not existing file) which version of database you'd like to create. If you see only SQLite3 in drop-down list, then it's probably because your binary distribution doesn't support SQLite2 (see 1.1), or you're runnign source distribution and you don't have “sqlite” extension for Tcl in version 2.

If SQLiteStudio supports both versions of SQLite on your computer, then you can convert database from one version to another. Use “Convert database” from databases tree context menu (right mouse click), but it has its consequences (see 3.3).

2.3) Editing tables structure

You can open table structure dialog in several ways:

- by pressing “New table” from toolbar,
- by right-click on database in databases tree and selecting “New table”,
- by right-click on table in databases tree and selecting “Edit table”,
- by pressing “Edit table” from Table Window toolbar,
- by pressing “t” key from keyboard in Table Window,
- by double-click on column in Structure tab of Table Window – that way SQLiteStudio will also open Column Edit dialog on top of Table Structure dialog immediately.

While being in Table Structure dialog, you can add/edit/delete columns by buttons on the right, or by pressing keyboard keys “Insert”/”Space”/”Delete”. Note, that for editing or deleting column you have to select which column you mean (if you prefer using keyboard, use tab key and arrows to navigate through the dialog). You can also open Column Edition dialog by double-clicking on it.

Columns order can be reorganized by drag&drop.

Constraints are displayed in column list on their right side by icons. To learn the name of the constraint hold mouse over the icon for a moment and the name will be displayed.

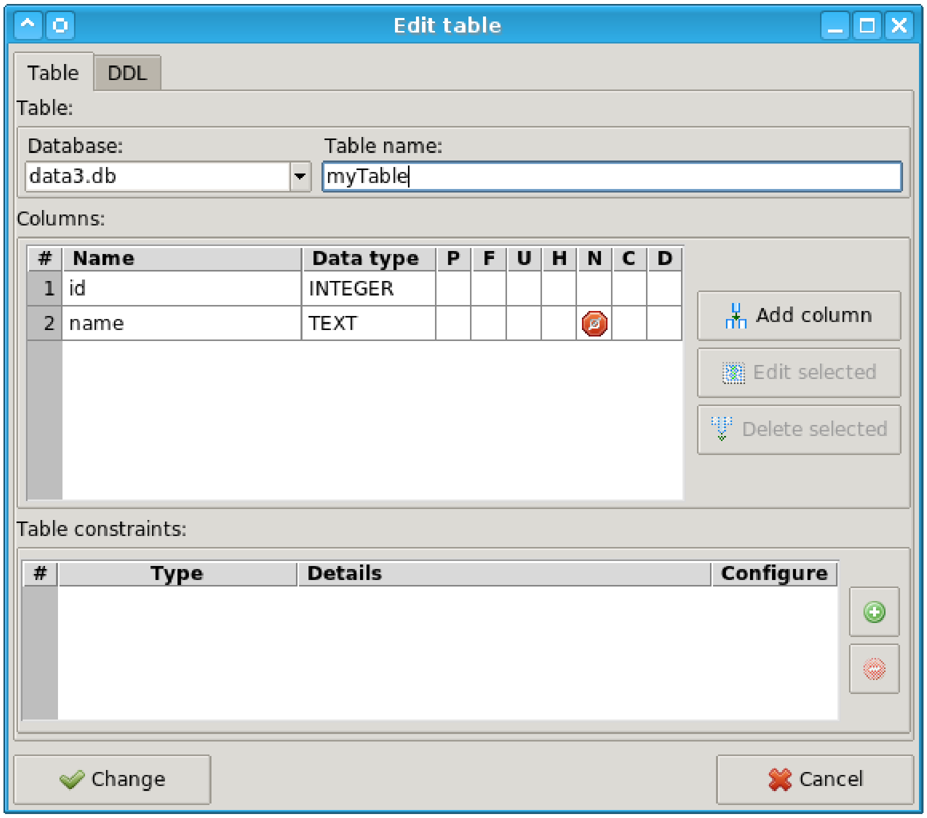


Table:

Database: data3.db

Table name: myTable

Columns:

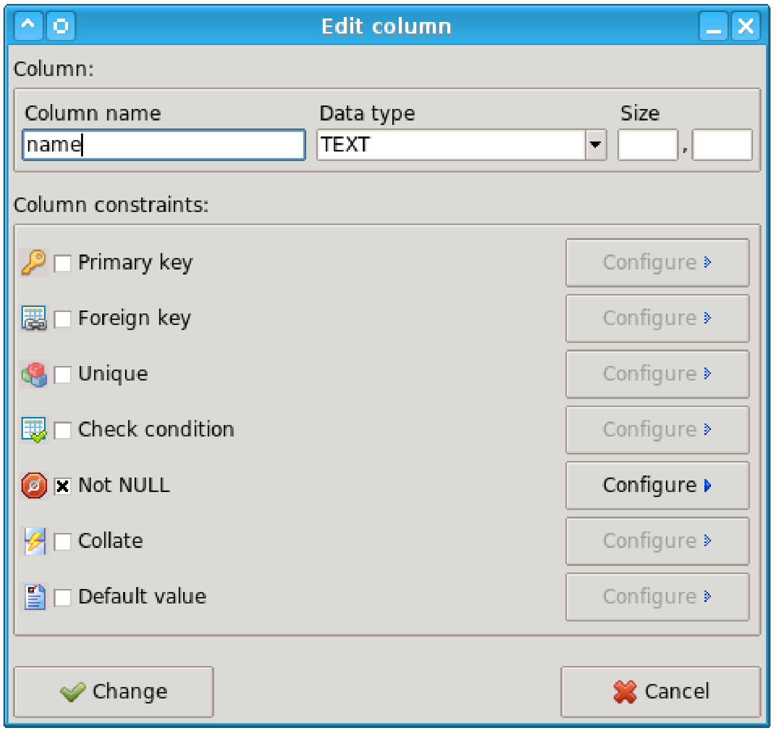
#	Name	Data type	P	F	U	H	N	C	D
1	id	INTEGER							
2	name	TEXT					●		

Table constraints:

#	Type	Details	Configure

Table Edit dialog. The "name" column has NOT NULL constraint.

SQLite supports both table-oriented constraints, as well as column-oriented constraints. Table constraints are managed directly in the Table Edit dialog, while column constraints are managed in Column Edit dialog:



Column name: name

Data type: TEXT

Size: ,

□ Primary key

□ Foreign key

□ Unique

□ Check condition

☒ Not NULL

□ Collate

□ Default value

Column Edit dialog

2.4) Exporting data

SQLiteStudio provides possibility to export entire database, single table, or SQL query results.

To export database, right-click on database in databases tree and pick “Export database”.

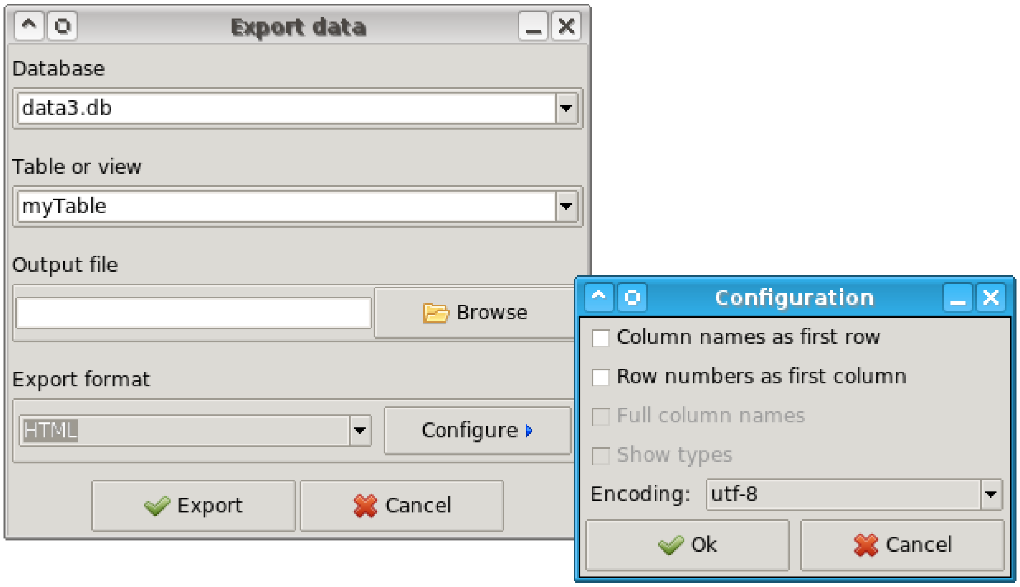
To export table, right-click on table in databases tree and pick “Export table”, or press “Export table” in toolbar of Table window.

To export SQL query results, execute the query firsts, then press “Export results” from SQL Editor window toolbar. Note, that exporting results is possible only if there actually are any.

Exporting dialog contains output file entry and can also contain drop-down list with databases and/or drop-down with tables existing in selected database for adequate export types.

The last drop-down list in the dialog contains available exporting plugins. They decide about output data format. They are described in details in paragraph 5.3.a. Each plugin has its own options for exporting and they can be configured directly from this dialog, right after user selected plugin.

Some of plugins might not support exporting of entire database. For example CSV format doesn't support schema of the database, just table data.



Export data

Database: data3.db

Table or view: myTable

Output file:

Export format: HTML

Configuration

□ Column names as first row

□ Row numbers as first column

□ Full column names

□ Show types

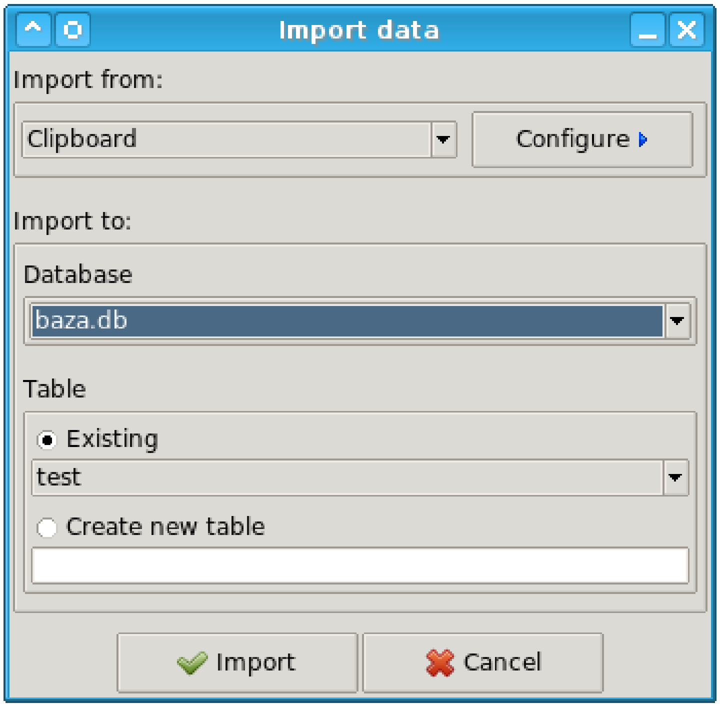
Encoding: utf-8

Exporting table data

2.5) Importing data

SQLiteStudio supports importing data from various formats (plugin based) into the table.

To import the data open a table window and press “Import data to table” button from toolbar on the top of the window. An importing dialog will appear:



Import from:

Clipboard

Import to:

Database

baza.db

Table

● Existing

test

○ Create new table

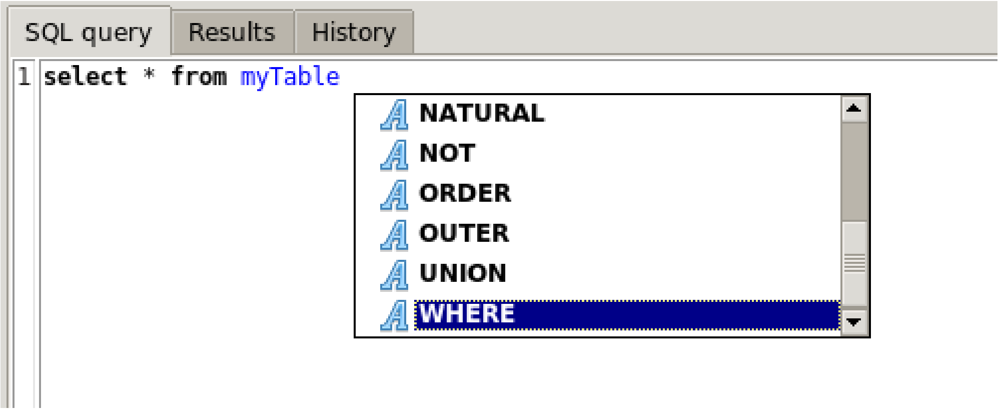
Importing data to table

On the top you choose importing plugin, which also determinates what is the input format. Plugins are described in details in chapter 5.4. Each plugin has it's own configuration available.

Then you pick the database and the table that data will be imported to. The table can be one of already existing in the database, or you can decide to create new one, basing on information from input data.

2.6) SQL editor

SQL Editor window is the MDI window that has SQL editor widget embedded. On top of SQL Editor window is placed toolbar. There is a drop-down list in the toolbar, that contains list of databases that are currently open. **Database selected in that list is used to execute typed query.** That list is updated each time a database is open or closed. You can switch between databases using Control-Up/Down shortcut (it can be changed in Configuration window).



```
select * from myTable
```

SQL editor with smart code completion

In SQL editor window you can select (with mouse, or by holding shift and using arrow keys) just a part of the SQL query and click "Execute" (or press F9) to execute only selected part of the code.

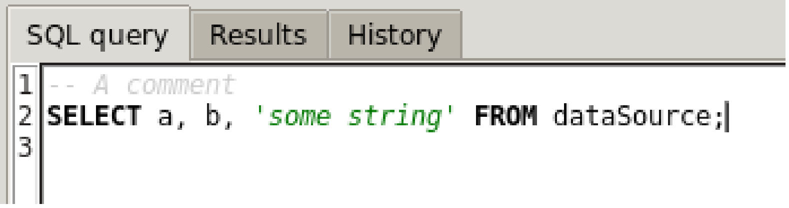
2.6.a) Writing and executing SQL query

SQL editor widget is used in SQL Editor window, but also in View Dialog and Trigger Dialog.

SQL editor provides following features:

- Syntax highlighting,
- Correct table names highlighting,
- Error checking on the fly,
- Syntax completion,
- Transparent database attaching,
- Code formatting (aka “pretty print”).

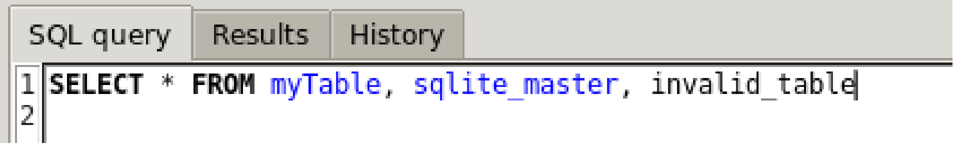
Syntax highlighting works using colors and fonts defined in Configuration window.



```
-- A comment
SELECT a, b, 'some string' FROM dataSource;

```

Table names in query that actually exists in currently chosen database are marked with individual color. You can quickly open such tables (Table window) by pressing Control key (the names will go underline) and clicking on the name.

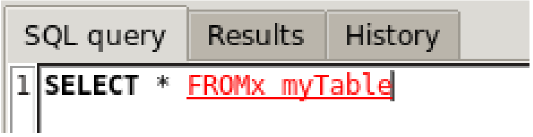


```
SELECT * FROM myTable, sqlite_master, invalid_table

```

Error checking is invoked each time after users stops typing for a short while. If any error is detected, then it's marked with color defined in Configuration window. Even error mark starts at some position and ends at the end of statement, the actual error occurs at the beginning of mark. The rest is just not understandable by SQLiteStudio, until the error is fixed.

Note, that error checking is based on syntax definition included in SQLiteStudio sources, so it might contain some wrong definitions, or can go out of date. In that case SQLiteStudio might mark error even query is valid. In that case you can ignore it and execute query anyway.



```
SELECT * FROMx myTable
```

Syntax completion is a small helper window that pops up on demand (Control-Space shortcut, or other configured), or after user typed dot character and waited a short while.

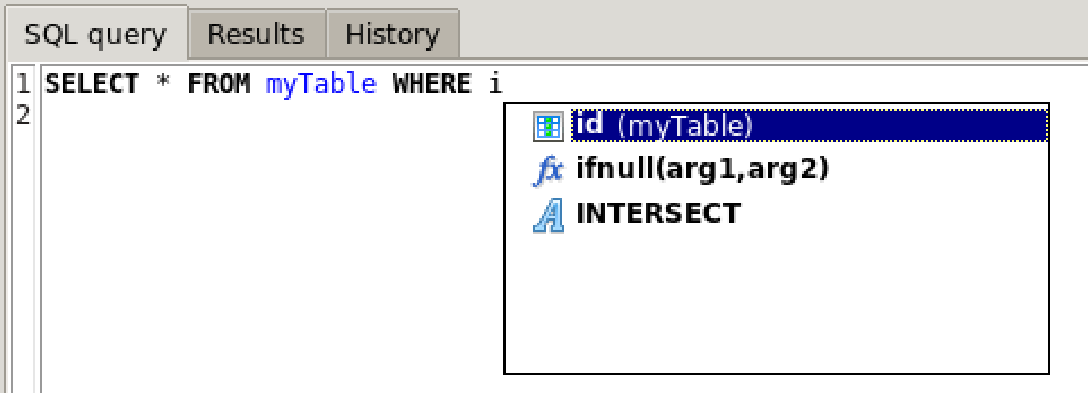
The window contains list of proposed values that would fit at current caret position. They are sorted in order, that values that user might want the most are placed at the beginning.

Each entry in list contains icon on the left to identify type of proposed value:

- column - ▦
- table - ▦
- index - 🏷
- trigger - ⚙
- view - 🖧
- database - 🛢
- SQL function - *fx*

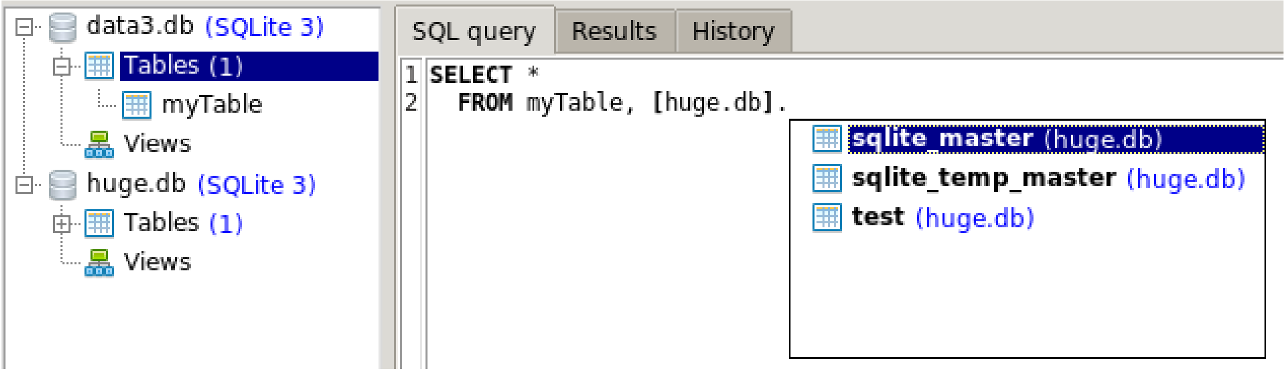
- keyword - A

Syntax completion is based on syntax definition included in SQLiteStudio, just like error checking does, so it might get out of date. This is why you should always update SQLiteStudio to newest version.



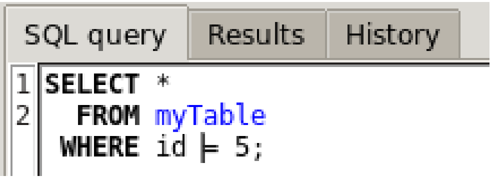
```
SELECT * FROM myTable WHERE i
```

Transparent database attaching lets user to type queries that uses multiple databases without worrying about attaching any of them – they will be attached automatically and transparently before query is executed and will be detached right after query execution has finished. To reference to other database in query, use its name as displayed in databases tree on the left.



```
SELECT *
  FROM myTable, [huge.db].
```

Code formatting organizes code so it's more human-readable. It works using plugins engine ([see 5.2.a](#)), so the results of this operation depends on chosen formatting plugin (in Configuration) and options defined for the plugin.



```
SELECT *
  FROM myTable
 WHERE id = 5;
```

2.6.b) Executing SQL directly from file

If you have some huge SQL file with lots of SQL statements, it might slowdown SQL Editor window significantly.

In that case use “Executing from file” feature.

There are two ways to execute SQL from file:

- Open SQL Editor window and click on “Execute SQL from file” from toolbar,
- Right-click on open database in databases tree and pick “Execute SQL from file”.

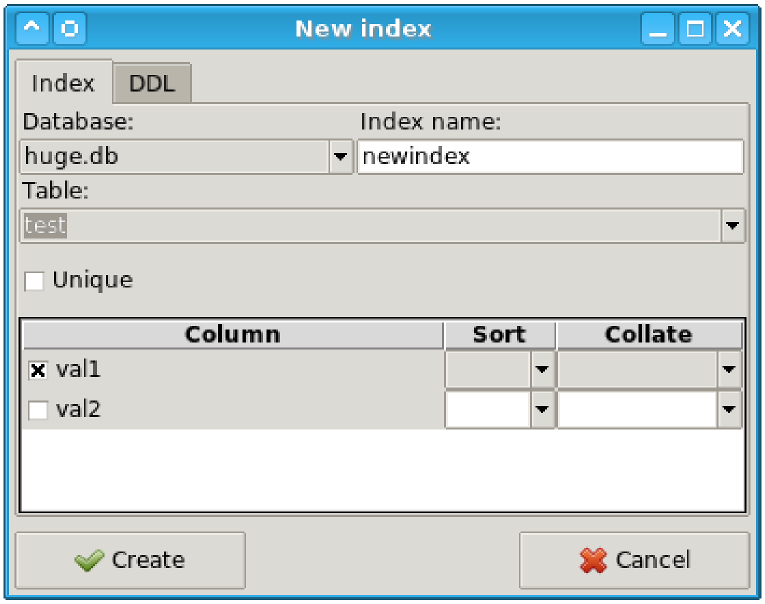
If SQL was executed sucessfly you'll be informed about it. If there was an error, the error message will appear with detailed information.

2.7) Indexes, triggers, views

Creating, modifying or deleting any of indexes, triggers or views can be done from toolbar, or from databases tree context menu, or from Table window corresponding tabs.

Dialogs for each of these objects contain two tabs – first contains configuration of the object and second current DDL of the object (as defined on first tab). If DDL cannot be created (definition is incomplete), them corresponding error will be written to DDL field.

Index dialog requires table to be selected to show available columns. It's quiet obvious, but not for everyone. If there is already selected table and user checks some columns for index, then he's able to select sorting order and collation mode (both of them optionally and supported only by SQLite3).



Database: huge.db

Index name: newindex

Table: test

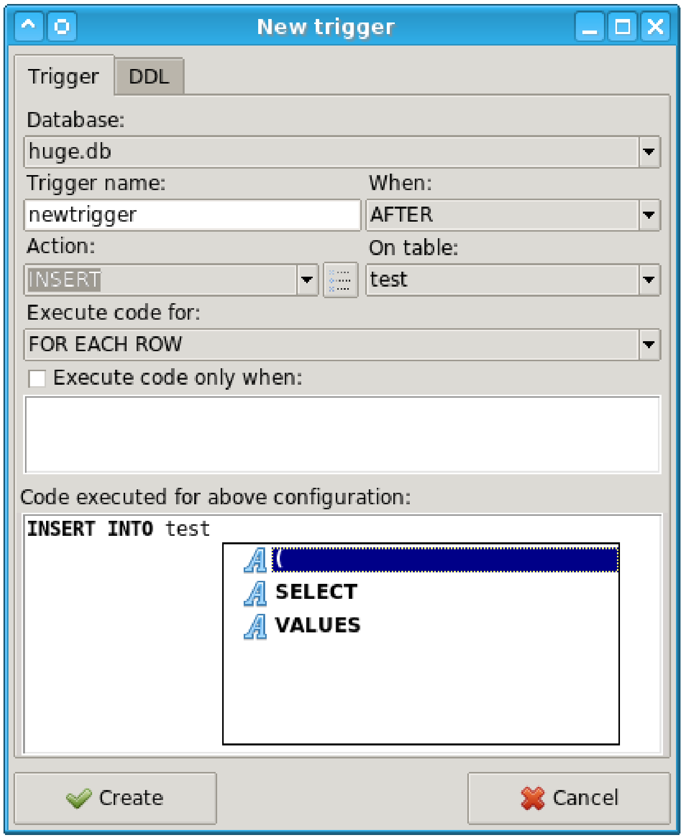
□ Unique

Column	Sort	Collate
☒ val1		
□ val2		

Index dialog

Trigger dialog has numerous fields to define. Here's their description:

- *database* – database to create trigger in,
- *trigger name* – name of new trigger to be created, or new name for edited trigger,
- *when* – AFTER, BEFORE, or INSTEAD OF – when trigger should be invoked,
- *on action* – INSERT, DELETE, or UPDATE - on what kind of actions should it be invoked,
- *on table* – on which table above actions have to take place, to invoke the trigger,
- *execute code only when* - if checked, then field below is enabled and it's a condition when trigger should or should be invoked. It supports SQL code completion feature. See SQLite documentation for details,
- *code executed for above configuration* - a body of the trigger. It supports SQL code completition feature. See SQLite documentation for details.



Database: huge.db

Trigger name: newtrigger

When: AFTER

Action: INSERT

On table: test

Execute code for: FOR EACH ROW

□ Execute code only when:

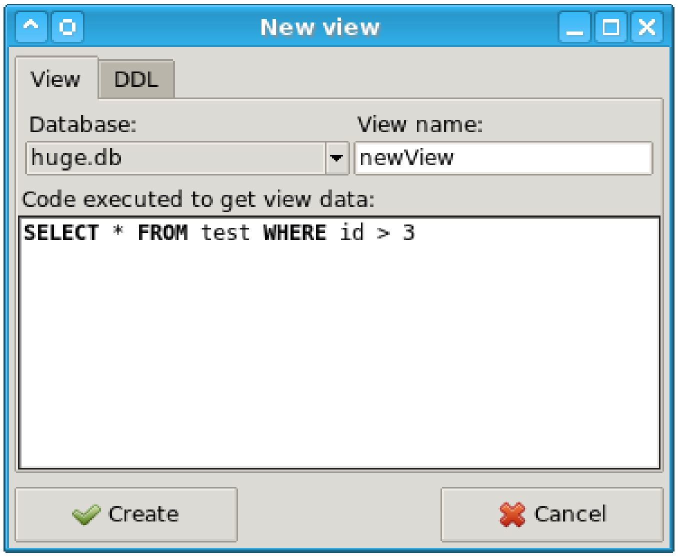
Code executed for above configuration:

```
INSERT INTO test
```

```
(
SELECT
VALUES
```

Trigger dialog. Note, that SQL editor has most of it's features available here as well.

View dialog contains only database to create it in and body of the view. The body field supports SQL code completion feature.



Database: huge.db

View name: newView

Code executed to get view data:

```
SELECT * FROM test WHERE id > 3
```

View dialog

2.8) Configuration dialog

Configuration dialog contains tabs to define colors, fonts, theme, plugins and some other, miscellaneous options. After any changes are made, the “Apply” is possible. Pressing “Ok” is equal to pressing “Apply” and closing the window.

Applying changes makes them work immediately and also saves them in application configuration file.

Few things that user should be aware of:

- Changing application theme might take a while (1-4 seconds), especially when there is a lot of MDI windows opened.
- It's good idea to use *vista* theme under Windows 7 and Vista, *xpnative* theme under other Windows NT family, *winnative* theme under Windows 9x family, *aqua* theme under MacOSX and *clam* under Unix-like systems (it's light and fast theme, which looks just ok).
  Themes that integrates with KDE (*tileqt*) or GTK (*tilegtk*) are at early stage of development and are not as stable or fast as *clam* is. They are also available only for few platforms currently.
- **Additional themes** might be installed in sub-directory `RUN_DIR/lib/`, where RUN_DIR is directory where SQLiteStudio binary file is placed (or main.tcl file for source distribution). Since version 2.0.0 it's also possible to place themes in `CFG_DIR/lib/`, where CFG_DIR is platform depended configuration directory. For Unix-like systems it's "`$HOME/.sqlitestudio/`", for Windows sytems it's "`%APPDATA%\sqlitestudio\`" for Windows NT family and "`%HOME%\sqlitestudio\`" for Windows 9x family.
- If you pressed "Change" button on "**Shortcuts**" tab and you want to **discard shortcut edition** - press "Escape". The "Escape" key cannot be used as custom shortcut in SQLiteStudio (mainly because it lets user to discard this edition).

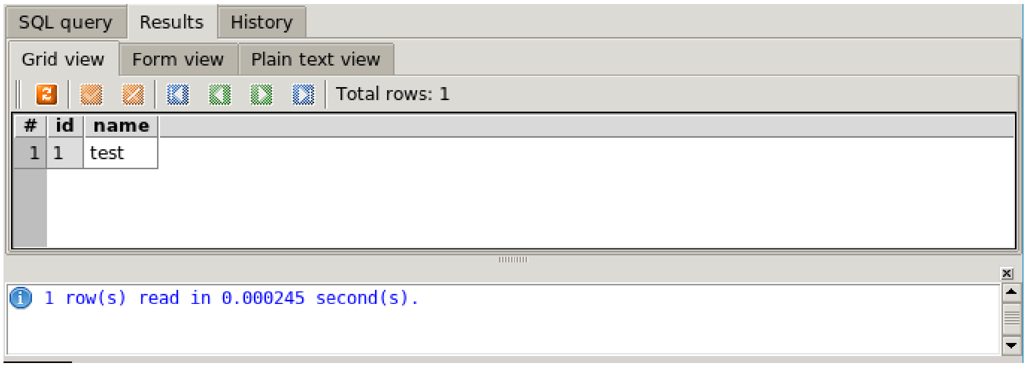
2.8.a) Miscellaneous tab

Following options on Miscellaneous tab might need explanation:

- *Restore session after next start* – recreates tasks (and their MDI windows) on next start that was open at last application exit.
- *Show line numbers in SQL editor* – if disabled, then line number/ROWID column in Grid View will be hidden.
- *Show SQLite system tables and indexes in databases tree* – if enabled, then system tables (such as sqlite_master) and indexes (such as autoincrement index in SQLite2) will be displayed in database tree on the left side of window.
- *Sort custom SQL functions list by name* – enables or disables sorting by name for list of SQL functions in Custom SQL functions dialog (see 3.1.a).
- *Ask if clipboard contents should be cut* – this defines behavior of application when user tries to paste some contents to Grid View, especially contents copied from other Grid View or spreadsheet application (such as MS Excel or OpenOffice Calc), since they will most likely

contain data for multiple rows and/or columns. If this option is disabled, then pasting 2-columns/2-rows data into last column in last row of Grid View will paste just first value from clipboard data, ignoring rest of data. If the option is enabled, then SQLiteStudio will ask user first, if he want's to skip rest of data – if not, then no data will be pasted at all.

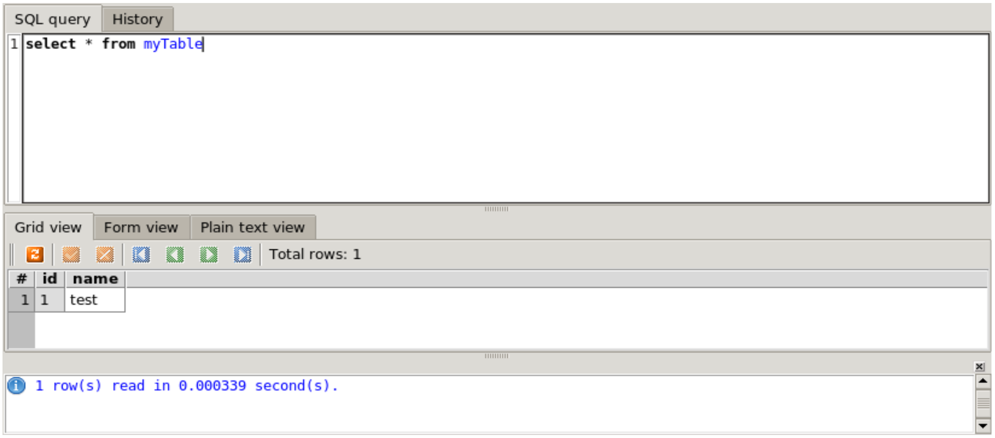
- *Results per page in Table Window and Editor Results* – defines number of rows at single page of Grid View. The less, the faster Grid View will behave, but more pages to switch across for huge amount of results.
- *History entries limit* – defines how many queries will be remembered in history, so user can go back to them and re-execute. The less, the faster new SQL Editor windows will open, but user loses amount of stored queries.
- *Web browser* – this option is available only for Unix platforms, excluding MacOSX (Windows handles default web browser by itself, same as MacOSX). It defines which web browser should be used. It's exact command to be executed. Target URLs will be passed as single argument to the command.
- *Tab in table window on open* - by default SQLiteStudio opens table window with the table schema as a selected tab. You might prefer to see the table data immediately after the table window is open. This option lets you to choose that.
- *Editor window results layout* - here you can choose if you want the SQL query results to be displayed in separate tab, or just below the query:



#	id	name
1	1	test

```
1 row(s) read in 0.000245 second(s).
```

Results in next tab



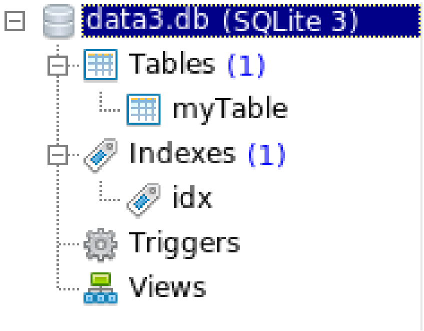
```
select * from myTable
```

#	id	name
1	1	test

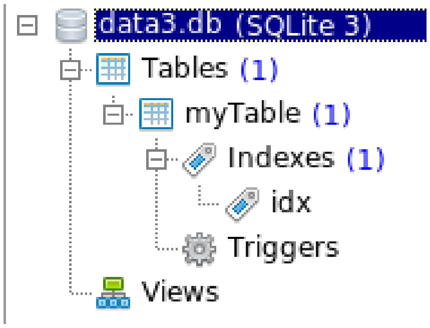
```
1 row(s) read in 0.000339 second(s).
```

Results below query

- *Database tree display mode* - determines whether indexes and triggers are in their own nodes in the database tree (on the left side), or they are linked under the table that they are related to:

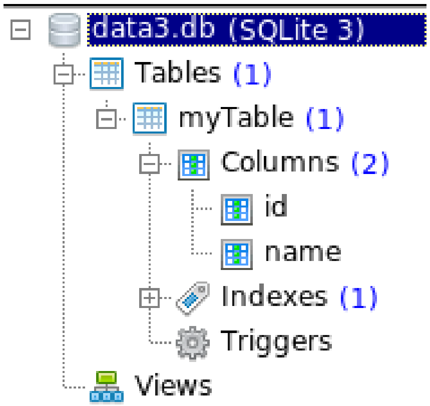


Objects in their own groups.
The SQLiteStudio 2.0.x style.



Objects under the related table.
The new SQLiteStudio 2.1.x style.

- *Show column names linked under the table* - when enabled displays columns under their table:



Columns under their table

3) Advanced topics

3.1) Built-in and custom SQL functions

SQL functions are the one used in SQL query expression, like:

```
SELECT max(id) FROM table1;
```

Here “`max()`” is the SQL function.

There are many SQL functions built-in SQLite. Additionally SQLiteStudio implements some functions by itself and finally, provides dialog window to write custom SQL functions defined by user.

3.1.a) Built-in SQLite functions

There is a wide list of SQL functions built-in SQLite. Full list with description is available in SQLite documentation – here, here and here.

3.1.b) Built-in SQLiteStudio functions

SQLiteStudio provides additional functions that can be used in SQL queries, but they are available only when managing database from SQLiteStudio, they won't be available in other application that connects to the database, unless the other application also implements such functions.

There are following additional functions:

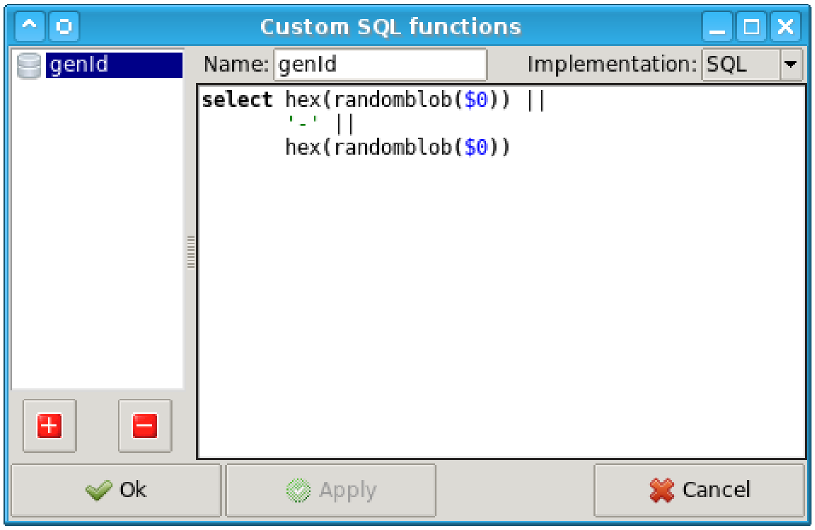
- `tcl('Tcl code')` - evaluates given Tcl code in separated Tcl interpreter and returns result of evaluation.
- `sqlfile('file')` - reads contents of given file and executes it as sequence of 0 or more SQL statements on current database connection. It returns 1 on success and 0 on error.
- `base64_encode(arg)` – encodes given argument with Base64 algorithm and returns it.
- `base64_decode(arg)` – decodes argument from Base64 algoritytym and returns it.
- `md5(arg)` – calculates MD5 digest from given argument and returns it.
- `md4(arg)` – calculates MD4 digest from given argument and returns it.
- `sha1(arg)` – calculates SHA1 digest from given argument and returns it.
- `sha256(arg)` – calculates SHA256 digest from given argument and returns it.
- `crc16(arg)` – calculates checksum value using CRC-16 algorithm and returns it.
- `crc32(arg)` – calculates checksum value using CRC-32 algorithm and returns it.
- `uuencode_encode(arg)` – encodes given argument with Uuencoding algorithm and

returns it.

- `uuencode_decode(arg)` – decodes given argument from Uuencoding algorithm and returns it.
- `yencode_encode(arg)` – encodes given argument with yEnc algorithm and returns it.
- `yencode_decode(arg)` – decodes given argument from yEnc algorithm and returns it.
- `file('file')` – reads contents of given file and returns them. If file doesn't exists or cannot be read, then empty string is returned.

3.1.c) Custom SQL functions dialog

SQLiteStudio provides support for writing user custom SQL functions on the fly. They work very much like built-in functions, except their body is defined by user in Custom SQL Functions dialog (available from toolbar).



Name: genId

Implementation: SQL

```
select hex(randomblob($0)) ||
       '-' ||
       hex(randomblob($0))
```

Custom SQL functions dialog

Remember, that custom functions won't work in other applications connecting to the database, unless they (applications) implement same functions by them-self.

Body and arguments:

Currently the body can be implemented either in SQL or in Tcl. All parameters passed to function call will replace any occurrences of placeholders `$0`, `$1`, `$2`, and so on.

Results:

Results are always returned as single cell (single column, single row).

In case of SQL function, multiple results will be concatenated into single value using whitespace as separator. NULL result will be transformed into empty string.

Examples:

Example function implemented in SQL, lets call it `getProductId` and assume that it takes one argument, a product name. The body is:

```
SELECT productId
  FROM products
 WHERE productName = '$0';
```

Other example. This time function is implemented in Tcl. Lets call it `random` and takes two arguments – minimum and maximum values expected. The body is:

```
set min $0
set max $1
expr { int((rand() * ($max - $min)) + $min) }
```

3.2) Populating tables

Populating table is a process of inserting generated data into the table.

SQLiteStudio provides dialog window to configure which columns should be populated and how should it be done.

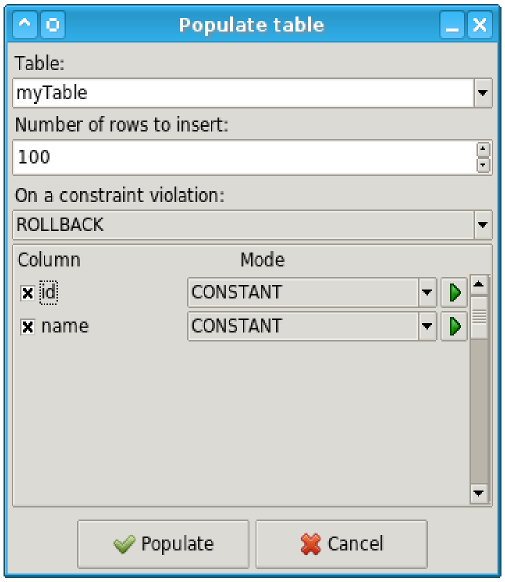


Table: myTable

Number of rows to insert: 100

On a constraint violation: ROLLBACK

Column	Mode
☒ id	CONSTANT
☒ name	CONSTANT

Populate dialog

Data generators are provided by populating plugins (see 5.1.a).

To populate table, open Table window and click on “Populate table” button in toolbar. The Populate dialog will appear. You need to define how many rows you want to insert, select which columns you want to fill and how do you want to fill them by selecting plugin for each column separately. You can configure plugin-specifig options for each column.

Populating process might take a while if you chose to insert lot of rows (like 100 000 and more). You can break the process and keep already inserted rows.

If any constraint is broken by generated data, then the action selected in “*On a constraint violation*” field is executed. Default is to rollback any changes.

3.3) Converting SQLite database version

You need SQLiteStudio installation with SQLite2 support for this feature (see 1.1).

It's possible to convert SQLite from version 3 to 2 and vice versa. However this is limited to objects with DDL supported by both versions. If any object cannot be converted to other version, then it won't be included in converted database.

Converting database **does not** overwrite the database your converting from – it stays untouched. Instead new database is created.

To convert database connect to it, then righ-click on it and pick “Convert database”. If your SQLiteStudio installation **doesn't support two versions of SQLite**, then you'll see empty value in drop-down list at bottom-right corner, thus **you won't be able to convert database**.

Next step is to fill in the name of new database – this will be used to display in databases tree on the left.

Then select new (not existing) file for converted database to be created.

Finally push “Convert” and have a hope, that DDLs are supported by target database version.

If SQLiteStudio have problems with converting, it will display message about the problem and skips object that problem was related to.

3.4) Schema importing

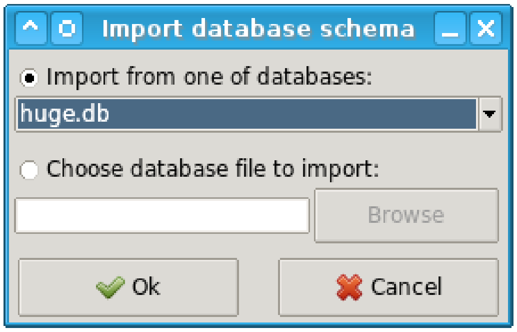
Importing schema is somehow similar to converting databases (see 3.3), but it's an utility to work from another perspective.

It lets user to get copy SQLite database schema (DDLs of objects) from target database to local database.

Differences from converting database:

- Direction – from target database, to local database,
- Data is not copied, just a schema,
- No need to register database, that user imports schema from,
- Database that schema is copied into must be already registered in SQLiteStudio,
- Database that schema is copied into can already contain some objects and importing process will try to add new objects next to old ones,
- Any error during import process will rollback all objects already copied.

Just like in case of converting database, the importing requires SQLite2 support in your SQLiteStudio installation if you want to import from SQLite2 database.



● Import from one of databases: huge.db

○ Choose database file to import:

Schema importing dialog

4) Useful utilities and features

4.1) Create similar table

Sometimes it's very handy to create new table, that is pretty much like some table that already exists in database. SQLiteStudio comes with help here.

To create similar table, right-click on table in databases tree on the left side and pick “Create similar table” from menu. Other way is to open Table window and use “Create similar table” button from its toolbar.

Dialog for “similar table” is just regular Table schema dialog, except it already has filled columns and constraints same as source table, so all you need to do is fill a name for new table and optionally do some modifications that you need.

4.2) Select duplicated rows for new unique index

When you're creating **unique** index for table, then it's possible that error will raise, that data in selected columns is not unique. Then SQLiteStudio will ask you if you want to see duplicated entries. If you say “yes”, than new SQL Editor window will be created and proper SQL query will be filled in automatically, then it will be executed and you'll see duplicates immediately.

As for regular SQL Editor window, you can go back to query tab, edit query for your needs and re-execute it. It's not any different than regular editor.

4.3) User-friendly bugs reporting

Old SQLiteStudio versions (1.x) required user to go to SQLiteStudio website and report bugs there. It's no longer necessary. If you select “report bug” from main window menu, or “report feature request”, or even “report bug” from critical error dialog, then in all these cases the new Bug Reporting dialog will appear.

The Bur Report dialog fills up all information that it can automatically (operating system, used Tcl extensions, stacktrace – if any) and needs only some context information from user about the problem (or request). Then user clicks “send bug” or “send request” and doesn't care about anything. The bug/request is logged on application forum.

User can always see how the discussion his report goes by choosing “**Bug Reports History**” from main window menu. He will see list of bugs/requests that he sent, together with link to open web browser at forum thread related to the report.

4.4) Create view from query

If you wrote some complicated query and the results fits your needs for new View, then you can press “Create view from query” from SQL Editor toolbar.

SQLiteStudio will try to find “SELECT” statement in your query. If there are more than one (separated by semicolons), it will tell you about it and will ask you to mark with selection which query you're interested in.

If there is no suitable “SELECT” statement in query, then no View will be created.

4.5) Good to know

4.5.a) Modifying table schema with lots of data in it

User has to be aware, that **modifying tables that contain huge amount of data might take long time**. The more data is in table, the slower table modification will be. This is because of SQLite limitations.

If you want explanation why it works like this, read below.

SQLite itself has very limited support for modifying existing schema. SQLite3 supports changing table name and adding new columns. That's all. It's even worse with SQLite2, which doesn't support any schema modifications at all.

This is why SQLiteStudio emulates modification layer for tables, indexes, triggers and views. What it really does, is a little complex.

For tables, it renames old table to some unique, not existing name, then creates new table with chosen name, copies all data from old table to new table (for common columns) and then drops old table.

SQLite2 it's slower, because there's no “table renaming” feature, so SQLiteStudio creates unique table, to copy data there, then drops current table, recreates it with new schema and then copies data back to new schema.

Indexes, triggers and views doesn't contain data, so modifying any of them is very quick.

4.5.b) Triggers support for SQLite2 and SQLite3 older than 3.6.19

SQLiteStudio 1.1.x emulated foreign keys using triggers. This has been removed in 2.0, because of 2 reasons:

1. It caused many problems. Didn't work like it was expected to.
2. SQLite 3.6.19 introduced full support for foreign keys and it becomes standard now.

If you would like to implement foreign key for SQLite2, then you can do it manually with triggers. SQLiteStudio won't delete them during modifications on tables – it always recreates any triggers for table that was modified.

5) Plugins

New plugins (a single tcl file with plugin class implementation) can be placed in `CFG_DIR/plugins/` directory. The `CFG_DIR` directory depends on operating system and it's described in installation from sources.

There's also command line option “`--plugins`” which you can use to specify additional plugins directory on demand.

Writing plugins requires Tcl programing language knowledge.

5.1) Populating plugins

Populating feature is described here.

5.1.a) Built-in plugins description

Standard populating plugins:

- CONSTANT
- DICTIONARY
- RANDOM CHARACTERS
- RANDOM NUMBER
- SEQUENCE

CONSTANT
It's very simple plugin. Column filled using it will contain same value for all populated rows. Constant value can be configured.

DICTIONARY
This plugin reads chosen file, splits contents using every whitespace (word-by-word) or linebreak (line-by-line) and uses results as list of elements to populate the column. Elements from list can be used in order (from first to last), or randomly (in this case any element can be used more than once).

This is very useful plugin if you have some dictionary in file and you want to put it into database.

RANDOM CHARACTERS
Fills column with sequence of random characters. User can define if he wants to include:

- alpha characters (a-z, A-Z),
- numeric characters (0-9),
- whitespace characters,
- binary characters (full ASCII range characters).

Also minimal and maximal lenght of generated values is configurable.

RANDOM NUMBER
Fills column with random number generated in defined range. Also constant prefix and suffix to the generated value is possible.

SEQUENCE
Fills column with numbers taken in order from sequence. User can choose value to start from. Also constant prefix and suffix to the generated value is possible.

5.1.b) Writing populating plugin

To write populating plugin the new class have to inherit from class `PopulatingPlugin`.
Methods that you need to implement are:

- `getName {}` - Has to return symbolic name of table population engine. Must be unique for all populating plugins. It's good idea for it to be uppercase name.
- `configurable {}` - Has to return boolean value defining if implemented populating engine supports configuration. If `true`, then methods `createConfigUI` and `applyConfig` implementations should not be empty.
- `createConfigUI {path}` - The *"path"* argument is Tk widget path to frame where whole configuration of plugin should be placed. Implementation should create configuration widget in frame pointed by given *"path"*. Can do nothing if `configurable` returns `false`.
- `applyConfig {path}` - Tk widget path to frame where configuration is placed. Implementation should extract necessary information from configuration widget and store it in local variables, so they can be used later, while populating by method `nextValue`. Configuration widget in path will be destroyed just after this method call is completed. Can do nothing if `configurable` returns `false`.
- `nextValue {}` - Implementation should generate next value to fill table cell and return it. The method called in a loop, for each populated table row.

Example populating plugin:

```
class ConstantPopulatePlugin {
    inherit PopulatingPlugin

    constructor {} {}

    public {
        variable checkState – these variables are used to cache configuration
        variable checkStateOrig

        proc getName {}
        proc configurable {}
        method createConfigUI {path}
        method applyConfig {path}
        method nextValue {}
    }
}

body ConstantPopulatePlugin::constructor {} {
    set checkState(const) ""
    array set checkStateOrig [array get checkState]
}

body ConstantPopulatePlugin::getName {} {
    return "CONSTANT"  - this name will appear in populating dialog, has to be unique
}
```

```
body ConstantPopulatePlugin::configurable {} {
	return true - this is how we say that the plugin is configurable
}
```

This method gets "path" and just build configuration GUI in it:

```
body ConstantPopulatePlugin::createConfigUI {path} {
	array set checkState [array get checkStateOrig]
	pack [ttk::frame $path.start] -side top -fill x -pady 2
	ttk::label $path.start.l -text [mc {Value to use:}] -justify left
	ttk::entry $path.start.e -textvariable [scope checkState](const)
	pack $path.start.l -side top -fill x -expand 1
	pack $path.start.e -side top -fill x -expand 1
	focus -force $path.start.e
}
```

Since configuration entry with constant value is bind with checkState array, we can just read from this array to use configured value:

```
body ConstantPopulatePlugin::applyConfig {path} {
	array set checkStateOrig [array get checkState]
}

body ConstantPopulatePlugin::nextValue {} {
	return $checkStateOrig(const) - this plugin always gives same, configured value
}
```

5.2) SQL formatting plugins

SQL formatting plugins are used to format SQL code in SQL Editor fields. They can be configured in Configuration window → Plugins tab.

5.2.a) Built-in plugins description

There are 2 built-in formatting plugins:

- BASIC
- ENTERPRISE

<u>BASIC</u>

This is very simple formatting plugin with only few configuration options. It supports keyword upper-casing and few other minor features. It doesn't depend on SQL parser built in SQLiteStudio, so if for some reason SQL parser fails for some of your queries, then this formatter will work anyway.

<u>ENTERPRISE</u>

It's very advanced plugin. It lets user to decide where would he like to have white-spaces, line breaks, how to indent code, and more... It has enhanced configuration window with live preview of options currently selected.

It depends on SQL parser built in SQLiteStudio – this is why it always know where and how to format code, but it's more vulnerable for SQL parser syntax incompatibilities – which actually should not happened very often.

5.2.b) Writing formatting plugin

Class to inherit to implement SQL formatting plugin is `SqlFormattingPlugin`.

Methods that you need to implement are:

- `getName {}` - Has to return symbolic name of table population engine. Must be unique for all populating plugins. It's good idea for it to be uppercase name.
- `configurable {}` - Has to return boolean value defining if implemented populating engine supports configuration. If `true`, then methods `createConfigUI` and `applyConfig` implementations should not be empty.
- `createConfigUI {path}` - The *"path"* argument is Tk widget path to frame where whole configuration of plugin should be placed. Implementation should create configuration widget in frame pointed by given *"path"*. Can do nothing if `configurable` returns `false`.
- `applyConfig {path}` - Tk widget path to frame where configuration is placed. Implementation should extract necessary informations from configuration widget and store it in local variables, so they can be used later, while populating by method `nextValue`. Configuration widget in path will be destroyed just after this method call is completed. Can do

nothing if `configurable` returns `false`.

- `formatSql {tokenizedQuery originalQuery {db ""}}` - this is the core method. It takes SQL query to format and should return formatted query.

  `OriginalQuery` is SQL query in original, untouched form.

  `TokenizedQuery` is query processed by lexer and it's a list of 4-element tokens. Tokens are described more precisely below.

  The optional argument `db` is contextual database that formatting was called for. You can check if it's not empty and apply some extra formatting option when you know what database it is.

Tokens passed to `formatSql` have 4 elements: type, value, begin index and end index. They represent query split in parts with type for each part and character index for begin and end of the token.

Possible token types are: `KEYWORD, OPERATOR, STRING, INTEGER, FLOAT, PAR_LEFT, PAR_RIGHT, OTHER, COMMENT.`

Example formatting plugin:

```
class BasicSqlFormattingPlugin {
    inherit SqlFormattingPlugin

    common keywords [string tolower $::KEYWORDS]
    common uppercaseKeywords 1

    common checkState

    public {
        method formatSql {tokenizedQuery originalQuery {db ""}}
        proc getName {}
        proc createConfigUI {path}
        proc applyConfig {path}
        proc configurable {}
        proc init {} {}
    }
}
```

```
# This is the method which does the main job – formats the SQL code.  It gets the SQL query on input
# in 2 forms: one is tokenized, so you can work with it easily and original, unchanged SQL string,
# so you can always refer to it. You may also receive the database object, which the SQL was intended
# to be formatted against. This matters if case of differences between SQLite2 and SQLite3.
# The database argument is not mandatory, so you have to deal without it..
body BasicSqlFormattingPlugin::formatSql \
                      {tokenizedQuery originalQuery {db ""}} {

     if {!$uppercaseKeywords} {
          # User configured to not change keywords, so our formatter has nothing to do.
          return $originalQuery
     }

     set out [list]
     # Now going through all tokens and if any token is a keyword, we uppercase it. Otherwise we
     # just put it to output as it is.
     foreach token $tokenizedQuery {
          lassign $token type contents

          if {[string tolower $tokenOut] in $keywords} {
               lappend out [string toupper $tokenOut]
          } else {
               lappend out $tokenOut
          }
     }
     return [join $out " "]
}

body BasicSqlFormattingPlugin::getName {} {
     return "Basic"   - this is the name of the plugin which will be visible in configuration
}
```

```
# This method is used to create plugin configuration window. It gets the “path”, which all widgets
# should be placed on. We create just one checkbox to decide if we want uppercasing the keywords.
body BasicSqlFormattingPlugin::createConfigUI {path} {
    ttk::checkbutton $path.c \
        -text [mc "Uppercase keywords."}] -variable \
        ::BasicSqlFormattingPlugin::checkState(upper)
    pack $path.c -side top -fill x

    set checkState(upper) $uppercaseKeywords
}

# This method is called when user accepts plugin confirugation. It should store anything from
# configuration widgets into plugin's local variables for later usage while formatting the SQL.
body BasicSqlFormattingPlugin::applyConfig {path} {
    set uppercaseKeywords $checkState(upper)
    CfgWin::save [list \
        ::BasicSqlFormattingPlugin:: uppercaseKeywords \
        $::BasicSqlFormattingPlugin:: uppercaseKeywords]
}

body BasicSqlFormattingPlugin::configurable {} {
    return true      - here we tell that this plugin is configurable
}
```

5.3) Exporting plugins

Exporting plugins are used by Export dialog (see 2.4). They define output format of exported data. Some of plugins might not support all kind of objects to export, since there are couple of them:

- query results,
- table data,
- indexes, triggers, views

All plugins are configurable from Export dialog level.

5.3.a) Built-in plugins description

There are 5 built-in exporting plugins:

- CSV
- PLAIN
- HTML
- XML
- SQL
- XLS
- PDF
- JSON
- dBase
- Clipboard

CSV

It supports tables data and query results only. Stores output in standard CSV format. Header row can be enabled in plugin configuration.

PLAIN

It's “plain text” output plugin. It exports data in similar way as seen in “Plain text” tab in SQL Editor window. Just like CSV plugin, it supports only tables data and query results. Width of columns can be configured.

HTML

Exports data as HTML document. Supports configuration of header, row order number and column data type.

XML

Exports data as XML document. The XSD for the output XML can be generated as configuration option.

SQL

Exports database as DDLs, table data as set of “INSERT” statements. Query results are also exported as set of “INSERT” statements.

XLS

Exports data in CSV format, almost the same as CSV plugin does, but it keeps compatibility with MS Excel understanding of the CSV format (which is slightly different). It also names the output file with “xls” extension, which makes it to be open by MS Excel by default.

PDF

Exports data as tables in PDF document. The final result is very similar to what you get from HTML plugin.

JSON

Exports data to JSON format. Data rows can be exported as JSON arrays or objects.

dBase

Exports data in dBase format, also known as DBF. The output DBF database doesn't support indexing yet, but BLOB fields are exported to special “dbt” files.

Data types in output DBF file are one of “C” or “M”, depending on the maximum value length in each column (the “C” type limits value length to 254).

Numeric types - such as “N” - are not used, because SQLiteStudio never knows if the data being about to be exported contains only number in “NUMERIC” column, or maybe some texts too. SQLiteStudio also cannot check entire data set before export, before it would take too long in some cases.

Clipboard

This is pretty much the same as CSV plugin, except it exports the output data to the clipboard, instead of actual file. Be careful with big data sets.

5.3.b) Writing exporting plugin

Class to inherit to implement exporting plugin is `ExportPlugin`.

Methods that you need to implement are:

- `getName {}` - Has to return symbolic name of exporting engine. Must be unique for all exporting plugins. It's good idea for it to be uppercase name.
- `configurable {}` - Has to return boolean value defining if implemented populating engine supports configuration. If `true`, then methods `createConfigUI` and `applyConfig` implementations should not be empty.
- `createConfigUI {path}` - The *"path"* argument is Tk widget path to frame where whole configuration of plugin should be placed. Implementation should create configuration widget in frame pointed by given *"path"*. Can do nothing if `configurable` returns `false`.
- `applyConfig {path}` - Tk widget path to frame where configuration is placed.

Implementation should extract necessary information from configuration widget and store it in local variables, so they can be used later, while exporting data. Configuration widget in path will be destroyed just after this method call is completed. Can do nothing if `configurable` returns `false`.

- `validateConfig {context}` - It's called just before any export* method is called. It allows to validate if all configured parameters (if any) are valid for exporting in given context. If there is any invalid parameter, then this method should call `error` command - it will be caught and displayed in error dialog. `Context` is `DATABASE`, `TABLE`, or `QUERY`. Last one is for exporting SQL query results.
- `exportResults {columns rows}` - This method gets SQL query results (from SQL editor window) and should convert them to format that is implemented by the plugin, then return them.

  `columns` is a list of columns in table. Each element in list is pair of column name (which includes 'table_name.' prefix) and data type (VARCHAR, DATE, INTEGER, TEXT, etc).

  `rows` is a table data as list of rows. Each row is list of values for each column and each value is pair of the value as Tcl sees it and second is boolean indicating if it's <code>null</code> (<code>true</code>) or it's not <code>null</code>.
- `exportTable {name columns rows ddl}` - This method gets table columns and data and should convert them to format that is implemented by the plugin, then returned.

  `name` is a name of the table to export.

  `columns` is a list of columns in table. Each element in list is sublist of: column name (without 'table_name.' prefix), data type (VARCHAR, DATE, INTEGER, TEXT, etc), primary key boolean (`true` if column is primary key), not null boolean (`true` if column is not null) and default value of column. Default value of column is pair of the value and `isNull` boolean.

  `rows` is a table data as list of rows. Each row is list of values for each column and each value is pair of the value as Tcl sees it and second is boolean indicating if it's `null` (`true`) or it's not `null`.

  `ddl` is a full DDL of table.
- `exportIndex {name table columns unique ddl}` - This method gets index specification and should convert it to format that is implemented by the plugin, then returned.

  `name` is a name of the index to export.

  `table` is a name of the table that index is created for.

  `columns` are columns of the table that index is created for. Each column is sublist of 3 elements: column name, collation, sorting order. Collation can be `NOCASE`, `BINARY`, some custom one, or empty. Sorting order can be `ASC`, `DESC` or empty.

  `unique` is a boolean value indicating if index is of UNIQUE type.

`ddl` is a full DDL of index.

- `exportTrigger {name table when event condition code ddl}` - This method gets trigger specification and should convert it to format that is implemented by the plugin, then returned.

  `name` is a name of the trigger to export.

  `table` is a name of the table or view that trigger is invoked by.

  `when` is `BEFORE`, `AFTER`, or `INSTEAD OF`.

  `event` `DELETE`, `INSERT`, `UPDATE`, or `UPDATE OF` *`columns-list`*.

  `condition` is a condition for `WHEN` statement.

  `code` is a body of the trigger.

  `ddl` is a full DDL of the trigger.
- `exportView {name code ddl}` - This method gets view specification and should convert it to format that is implemented by the plugin, then returned.

  `name` is a name of the view to export.

  `code` is a body of the view.

  `ddl` is a full DDL of view.

5.4) Importing plugins

Importing plugins are used by importing dialog window described in chapter 2.5. They are used to gather information about list of columns coming from input data source and then they parse the input data to feed the target table with it.

5.4.a) Built-in plugins description

There are 4 built-in importing plugins so far:

- Clipboard
- CSV
- dBase
- RegExp

Clipboard

Expects that system clipboard contains CSV compilant data. It parses the clipboard contents as CSV, just like the CSV plugin below does.

CSV

Parses input file as a CSV file.

If the data is imported into existing table, then number of columns in the input CSV data must be equal to number of columns in the target table.

If the data is imported into table that is about to be specially created for this purpose, then the table will be created with the same number of columns that the input data has. If user checked “Treat first row as column names” option, then values from first row of input data will be used as column names for the table. If the option was not checked, then column names will be generated automatically (like “Col_1”, etc).

dBase

Opens the input file as D-BASE database file.

Table below explains how dBase types are translated to SQLite types:

dBase type (input)	meaning	SQLite type (output)
N	Number	NUMERIC
C	Characters	VARCHAR
L	Logical	BOOLEAN
D	Date (without time)	DATE
M, B, G, P	Text memo, binary memo	BLOB
F	Number	FLOAT
Y	Currency	NUMERIC
I	Integer	INTEGER
+	Autoincrement integer	INTEGER PRIMARY KEY AUTOINCREMENT
@, T	Timestamp	DATETIME
V, X	Memo	(empty)
O	Number	DOUBLE
(any other type)	Any unexpected type	(empty)

RegExp

Imports data from text file using regular exporession. The regular expression is a common language for matching some expressions in strings and is well known across several programming languages (and not only in programming languages). If you don't know anything about it, you can learn about RegExp at http://en.wikipedia.org/wiki/Regular_expression.

You need to be aware of what are “Groups” in the Regular Expression terminology. You will have to use the groups to mark what parts of the expression should be treated as columns for importing the data.

Each match of the entire expression configured for the plugin will be treated as single row of data to be imported. Therefore number of groups in the expression has to be equal to number of columns in the target table.

5.4.b) Writing importing plugins

Class to inherit to implement exporting plugin is `ImportPlugin`.

Methods that you need to implement are:

- `openDataSource {}` - The method is called at the begining of importing process, so the plugin can setup its data source.
- `getColumnList {}` - This method should return list of columns that are expected from import source. Each column has to be 2-element list of column name and SQLite data type. In case of importing to existing table only the number of columns is checked to be equal to number of columns in the table. In case of importing conjucted with creating new table column names and data types are used to create the table.
- `getNextDataRow {}` - This method has to return list of values imported from data source for a single table row, therefore the number of values in list has to be equal to number of columns returned from `getColumnList`. Each value is a pair of actual value and boolean indicating whether the value is meant to be NULL. If the NULL indicator is true, then any value placed as the first in the value pair is ignored. Values should already be encoded with utf-8 encoding. It's up to the plugin to take care of it. Returning empty list means, that there is no more data and the import is about to be finished.
- `closeDataSource {}` - Closes data source. Needs to be defined in derived class. It will be called just before destructor.
- `getName {}` - Has to return symbolic name of importing engine. Must be unique for all importing plugins. It's good idea for it to be uppercase name.
- `configurable {}` - Has to return boolean value defining if implemented populating engine supports configuration. If `true`, then methods `createConfigUI` and `applyConfig` implementations should not be empty.
- `createConfigUI {path}` - The *"path"* argument is Tk widget path to frame where whole configuration of plugin should be placed. Implementation should create configuration widget in frame pointed by given *"path"*. Can do nothing if `configurable` returns `false`.
- `applyConfig {path}` - Tk widget path to frame where configuration is placed. Implementation should extract necessary information from configuration widget and store it in local variables, so they can be used later, while importing data. Configuration widget in path will be destroyed just after this method call is completed. Can do nothing if `configurable` returns `false`.
- `validateConfig {context}` - It's called just before any import action is taken. It allows to validate if all configured parameters (if any) are valid for importing. If there is any invalid parameter, then this method should call <code>error</code> command - it will be caught and displayed in error dialog.