



MultiSQLite

C# Edition



User's Guide

Version 1.1.0.1

HAUFE.Group

uwe.stahlschmidt@haufe-lexware.com



Table of Contents

Introduction	3
Installation	4
Step 1: Open GitHub-Project Site	4
Step 2: Download the project archive	5
Step 3: Extracting the Archive	6
Starting the Application	7
User Interface.....	9
Starting and stopping threads	9
Continuous Monitoring.....	10
Version Label and Alive Signal	11
User's manual	11
The prompt: Direct access to the database	11
The Analyzation Tree	12
The Main Menu	13
The menu "File"	13
The menu "Actions"	14
The menu "Help"	14
Working with MultiSQLite	15
Simulating multiple threads within the same application	16
Simulating multiple instances of the same C# Application	17
Simulating mixed C++ / C# Environments	19
Answers & Conclusions	21
How well does SQLite support Multithreading in C#?	21
How is the workload balanced across multiple threads?	24
Does SQLite work with multiple instances of the same application in C#?	24
How is the workload balanced across multiple C# applications?	25
How is the workload balanced across multiple C# application?	26
Does the performance of the database decrease with time or with a growing database?	27



Introduction

Haufe MultiSQLite is an external tool that can be used independent of the on-premises applications of the Haufe Group in order to evaluate the feasibility of using SQLite in the different situations arising from the environments surrounding the products. This is done by simulating the existing cases which are dominant throughout the development teams:

- Accessing the same SQLite database from different threads within the same application
- Using one SQLite database from different applications that are developed in the same language (C# or C++)
- Using SQLite for joint products that are using both C++ and C# with different connectors for each
- Having different applications that are developed with different development tools access that same database.

MultiSQLite is available as an open source GitHub project, so that developers can look up how the access to the database is done and hence can expect similar results if the connection is done in a similar way in the corresponding applications.

As of this moment, the tool is available in two different editions:



In addition to this edition, which is implemented in C#, a very similar version is also available for C++. The different editions are designed to work along with each other, so that analysis can also be undertaken in mixed environments or for products, that are developed using a combination of C# and C++. For that purpose, the different editions can be started separately. Since it is insured that the database is located in a predefined directory, analysis in mixed C++ / C# environments if possible without any further ado, however it must be ensured that compatible editions of MultiSQLite are used in a mixed setup.

Please note that the editions are developed based on the needs at hands and hence are not necessarily in sync. For that reason, the functionality and user interface of the different editions may vary depending the current state of development.

Installation

As was already mentioned above, Haufe MultiSQLite is maintained as a GitHub-Project in order to allow further development by different developers.

The GitHub-Project is hosted under the following link:

<https://github.com/ushaufe/Sqlite4CS.git>

The executable version of the product as well as the source-files are available in the subfolders of this file structure.

Note:

Currently (Version 1.0.0.0) of MultiSQLite is not bundled with an installer. For that reason the installation must be done by downloading and extracting the corresponding files manually as explained in the following sections.

Step 1: Open GitHub-Project Site



HAUFE.Gruppe

As a first step, the MultiSQLite archive must be downloaded from the GitHub-Site. In a webbrowser of your choice navigate to the following link hosting the project: <https://github.com/usshaufe/Sqlite4CS.git>

This site lists the directory structure including all files of the project. The file structure can be browsed to have access to single files.

Step 2: Download the project archive

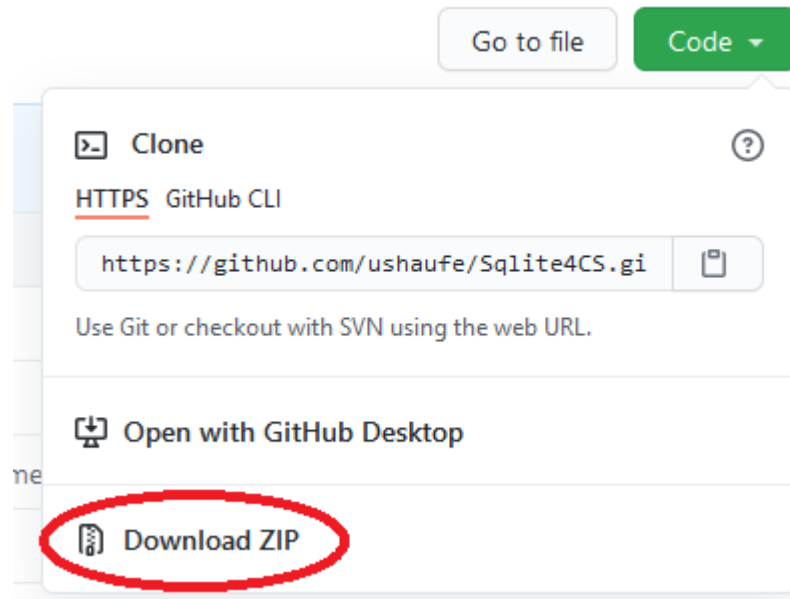
Code ▾

In order to have a locally executable version of MultiSQLite, the whole project must be downloaded as an archive.

In order to achieve that, locate the button “Code” in your GitHub-project and click on it in order to expand it.

Once the section grouped under the icon “Code” is expanded, the user is given a choice of options.

In that expanded dialog click on “Download ZIP” and save the zip folder on your local harddrive.



This file contains the source-code as well as the executable application in a compressed form.

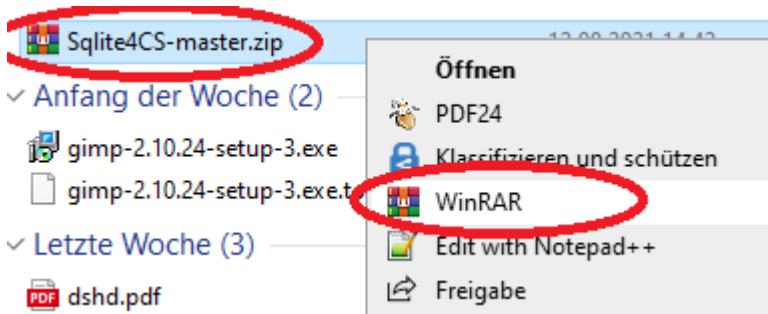
Note:

If you are a developer that is already actively using Git, you might have tools



with a graphical user interface installed (i.e. Tortoise-Git). In that case you can also use those tools to download the project. This will enable you to commit changes.

Step 3: Extracting the Archive



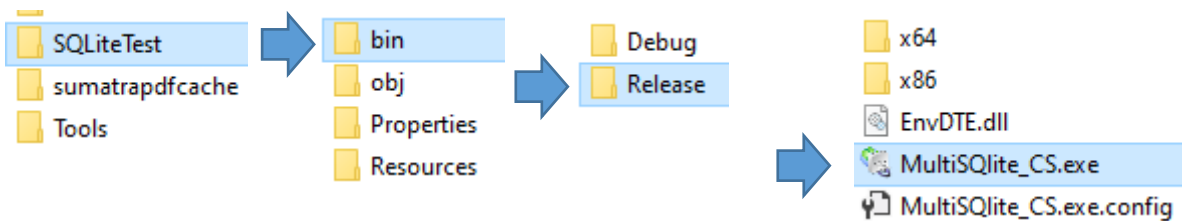
Once downloaded, extract the file “Sqlite4CS-master.zip” to a folder of your choosing.

This will create a file-structure in the underlying directory that looks like this

Name	Größe	Gepackt	Typ	Geändert	CRC32
..			Dateiordner		
.vs	6.025.892	504.925	Dateiordner	11.08.2021 18:25	
packages	32.493.968	19.140.079	Dateiordner	11.08.2021 18:25	
res	47.389	40.376	Dateiordner	11.08.2021 18:25	
SQLiteTest	22.182.230	11.778.787	Dateiordner	11.08.2021 18:25	
sumatrapdfcache	7.919	7.919	Dateiordner	11.08.2021 18:25	
Tools	6.434.995	3.711.202	Dateiordner	11.08.2021 18:25	
1920px-SQLite370.svg...	76.725	71.987	IrfanView PNG File	11.08.2021 18:25	982C7660
Connection2.jpg	170.781	170.077	IrfanView JPG File	11.08.2021 18:25	205CC02A
Connection2_Light1.jpg	397.543	392.883	IrfanView JPG File	11.08.2021 18:25	597A2FA0
Connection2_Light2.jpg	325.204	321.708	IrfanView JPG File	11.08.2021 18:25	A1C9DD29
ConnectionIcon.ico	67.646	21.321	IrfanView ICO File	11.08.2021 18:25	138A1ADE
ConnectionIcon.png	19.192	19.192	IrfanView PNG File	11.08.2021 18:25	25D1A485
Connections.ico	67.646	16.969	IrfanView ICO File	11.08.2021 18:25	961B7F45
Connections.png	15.524	15.524	IrfanView PNG File	11.08.2021 18:25	205C75D4
Connections2.jpg	376.609	358.847	IrfanView JPG File	11.08.2021 18:25	D90EFA9D
Connections2.png	2.105.669	2.105.669	IrfanView PNG File	11.08.2021 18:25	D506A41B
Connections2_Light.jpg	256.566	252.316	IrfanView JPG File	11.08.2021 18:25	DCF0585C

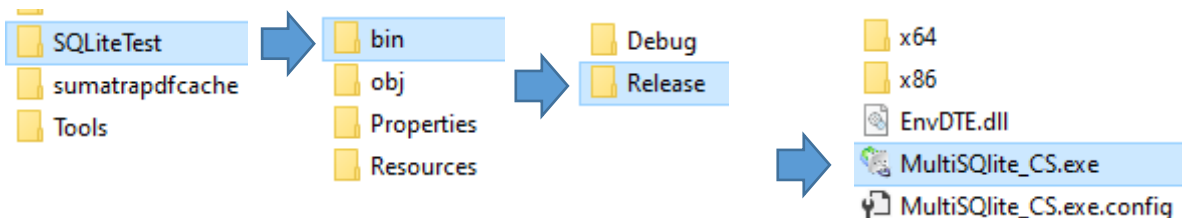
Step 4: Finding and starting the application

The file structure that was extracted contains the executable application in addition to the source files. In order to start the application, move to the following subfolder and start the following application:



Starting the Application

As mentioned in the above chapter, your application has been extracted to the following subfolder of the file structure that you have created.



The application can be started directly by launching the executable “MultiSQLite_CS.exe” in the “Release”-subfolder. If the Debug-folder also contains an executable version of that file, it is preferable to start it from the “Release”-subfolder because this should contain the version with a better performance.

Note:

The application will only launch and work correctly if in addition to the executable, all additional files have been extracted properly and are included in the “Release”-folder. The executable of the application will not launch properly without the additional files.

Once the application is started, it automatically sets up a database that is used for testing. If the application is restarted from a previous session, the same database is reused.

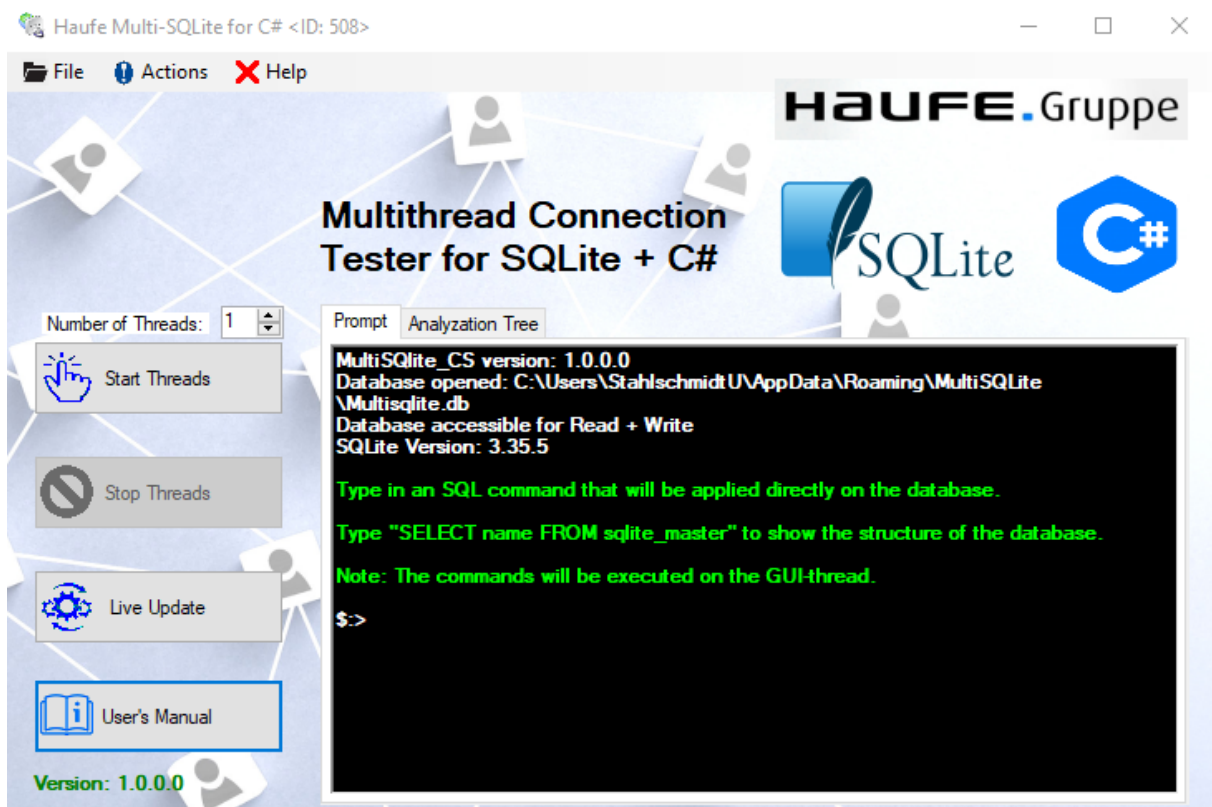
In case multiple instances of the application are started, they all use the same database and simulate the sharing of a database by several applications. No



HAUFE.Gruppe

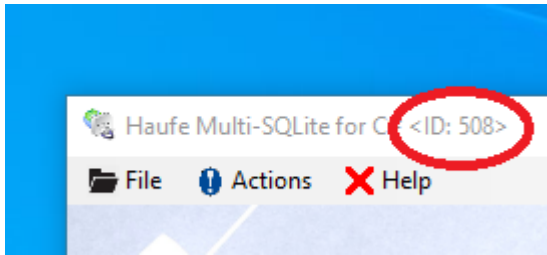
further action has to be taken by the user to indicate the database or other running instances of the application.

Once the application is started the user is presented with the following GUI:

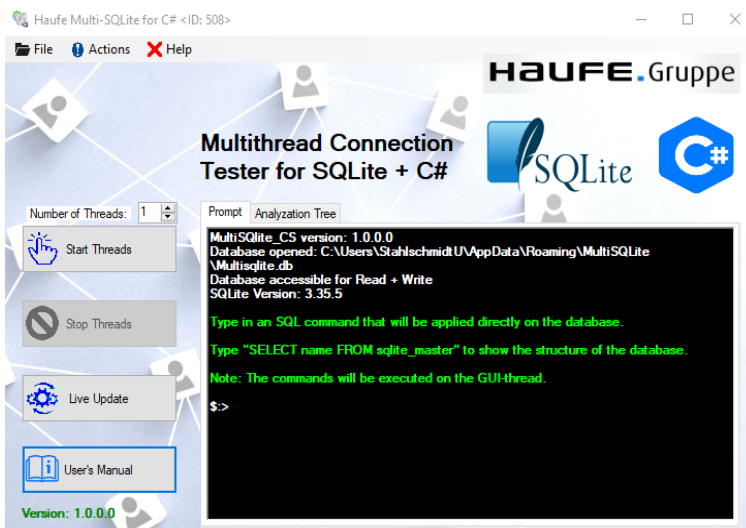


In the prompt window that is shown during startup the user can see if the connection to the database was successful. By default the application opens one connection to the database from now on referred to as the GUI-Connection.

In the title bar of the application the instance of the running application is indicated, so that several instances of the same application can be distinguished.



User Interface



The main screen of the user interface consists of two areas:

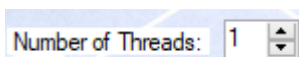
A row of buttons on the left side to perform different actions

A tabbed window on the right side allowing the user to interact with the database.

The functionality of each element will be described in the following subsections.

Starting and stopping threads

With the two buttons and the number control on the left side the user can start and stop threads that serve to continually insert datasets into the database.



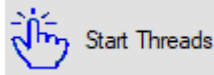
With this number control the user can set how many threads should insert data in the database simultaneously. Each thread is inserting the data as fast as it can on an individual connection.

When the threads are started it serves as an indicator showing the number of threads currently running,



HAUFE.Gruppe

when the threads are stopped it allows the user to select the number of threads that should be started.



Start Threads

Clicking this button starts the number of threads indicated in the number control. The corresponding number of threads is created, each thread owning its individual connection. Each of the threads grabs as much capacity as it can and tries to insert as many datasets as possible into the database. The bandwidth is concurrently shared among the threads. As soon as the inserting-process is started, the button is disabled, and the “Stop Threads” button is enabled. By pushing this button, the view on the right side of the window is switched to tree view.



Stop Threads

This button terminates the threads currently active and stops the inserting-process. After the running threads have been stopped, only the GUI-process remains open allowing the user to access on a step-by-step base. The button becomes disabled as soon as the threads are terminated and toggles the “Start Threads” Button to enabled.

Continuous Monitoring



Live Update

By enabling “Live Update” the tree view on the left right side is continually updated and the corresponding elements in the tree are showing live values.

The button toggles to gray when the automatic polling is activated and toggles back into the default state if pressed again, hence indicating that the automatic polling has been disabled.

As with the preceding buttons, the view on the right side of the window is automatically switched to tree view.

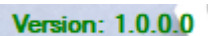


Note

This feature is switched off by default and should be used with caution. Depending on the tree elements that are currently opened on the right side, enabling this feature uses a lot of performance and puts a lot of load on the GUI-thread.

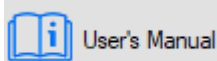
In addition, the button is flickering each time the information in the tree view is updated.

Version Label and Alive Signal



In addition to showing the current version of the application, the version label serves the purpose of giving an alive signal to the user: The database is polled at given intervals and status information about the access-levels is updated in the database. Every time this happens, the version label flickers in red.

User's manual



Pushing this button brings up this user's manual. Note that that this button will bring up the latest version of the manual, which might not necessarily correspond with the version of the application.

The prompt: Direct access to the database

The right side of the application houses a tabsheet, with two tab-views. The tab "Prompt" gives the user direct access to the database, allowing him to execute SQL-commands in plaintext form.



```
Prompt  Analyzation Tree

MultiSQLite_CS version: 1.0.0.0
Database opened: C:\Users\Stahlschmidt U\AppData\Roaming\MultiSQLite
\Multisqlite.db
Database accessible for Read + Write
SQLite Version: 3.35.5

Type in an SQL command that will be applied directly on the database.
Type "SELECT name FROM sqlite_master" to show the structure of the database.
Note: The commands will be executed on the GUI-thread.

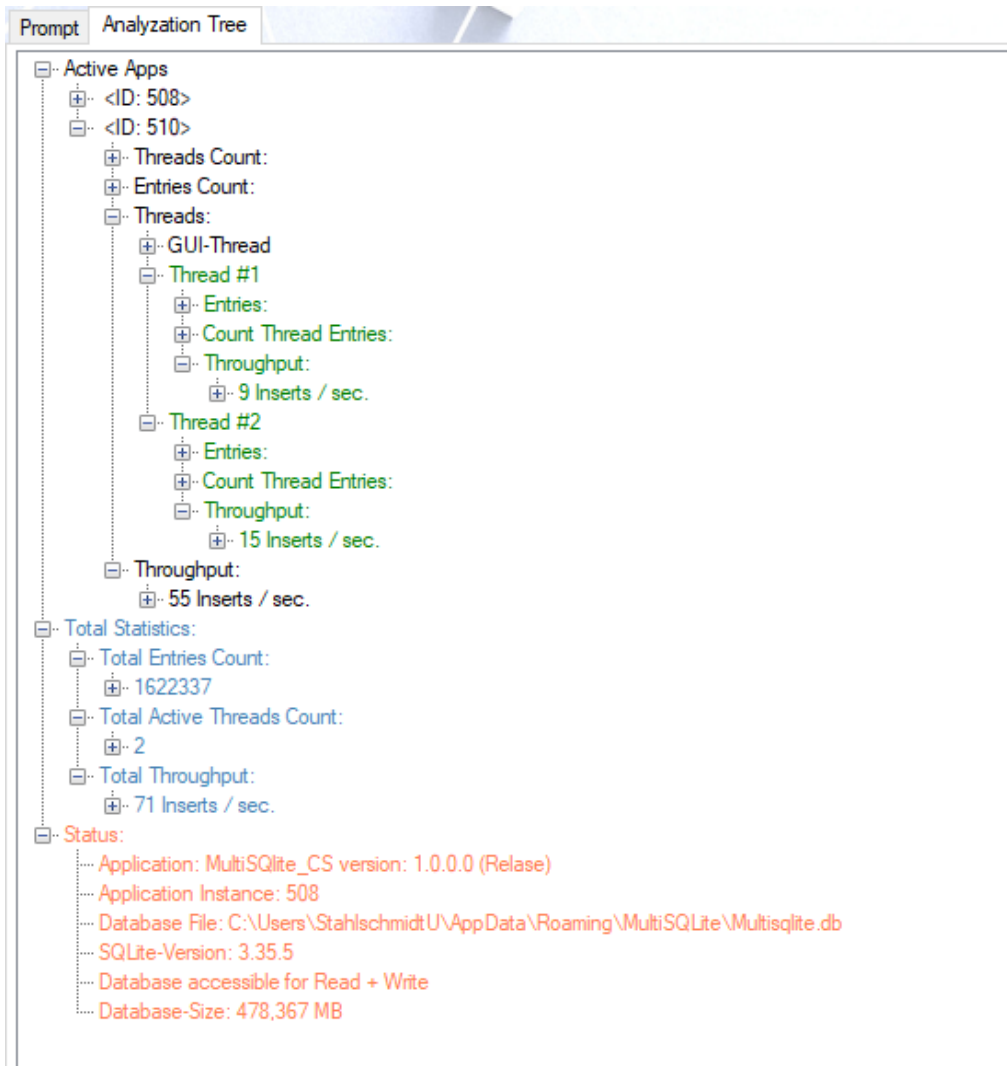
$:>
```

Note

The commands typed in this prompt are always executed on the GUI-threads, hence not affecting the processes under way in the other threads. Because of that fact, SQL-commands putting heavy load on either the database or the output-shell will freeze the GUI for the time the command is executed.

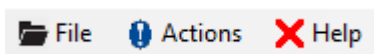
The Analyzation Tree

If the tab "Analyzation Tree" is opened, the user is shown a tree providing the different applications currently accessing the database along with their corresponding threads and measuring data such as throughput for each of the threads and applications.



The Main Menu

The same functionality that is accessible through the main GUI is redundantly also available through the main menu menu. The main menu is grouped in three sections.



The following tables show the functionalities of each of the elements.

The menu "File"

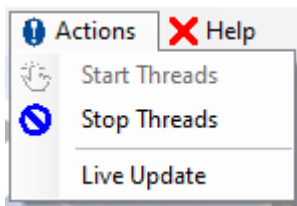




The file menu only contains the entry “Exit”, allowing the user to exit the application. At the same time all running threads will be terminated and all connections to the database will be closed.

The menu “Actions”

The actions menu gives access to all functionality, where threads are started or terminated to access the database on different levels simultaneously.



The menu contains the following entries:



Start Threads

The menu item “Start Threads” has the same functionality then the button on the GUI. It starts the number of threads set by the number selector, each of which is inserting rows into the database with highest possible priority.



Stop Threads

This menu items terminate the running threads and closes the database connections associated with it.

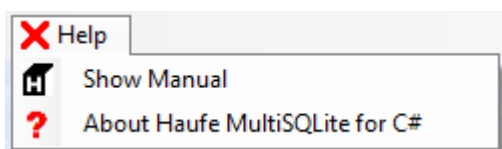


Live Update

This function continuously polls the database for changes and computes the new performance data. The data is periodically inserted in the tree.

The menu “Help”

The menu helps displays general information about the application such as an info dialog and the user’s manual.



The menu contains the following entries:



Show Manual

The menu entry “Show Manual” shows the user’s manual.



About Haufe MultiSQLite for C#

The menu entry “About” shows an info dialog with general information about the application.

Working with MultiSQLite

The following chapter describes the workflow that can be followed with MultiSQLite in real-life-scenarios.

This could be standard editions from the stack of different on-premises products of the Haufe Group, which are implemented using a mix of different programming languages (C++ and C#). An example for such a case would be the product „Buchhaltung“, where some modules are implemented in C# while others are implemented in C++. In that case the questions that should be answered by the help of this application is:

“How well does SQLite handle the case, that it is accessed from two different connectors provided for separate programming environments simultaneously?”

The next frequent scenario that occurs is the simultaneous access from multiple threads from within the same application.

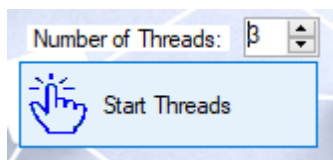
As a most sophisticated condition, several different applications could be using the same SQLite-database, where each could have multiple threads.

The working patterns described here give provide ways to find measurable and reproducible answers to those questions. The application will be further extended to yield answers for more specific cases.

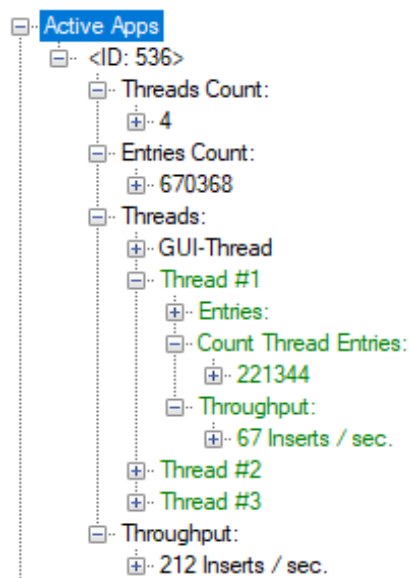


Simulating multiple threads within the same application

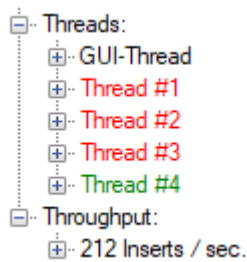
In order to find out how well mutiple threads are handled, that are simultaneously accessing the database from within the same application, start MultiSQLite and select the numbers of threads you want to be started simultaneously in the number selector and then click on „Start threads“.



These threads will each be inserting rows into the database containing random-data of varying length. As soon as the threads are started, the view on the right side is automatically changed to a tree-view, yielding measurement and performance data.



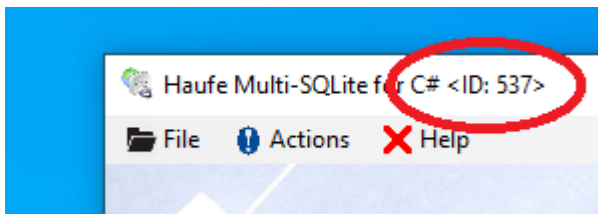
When expanded, the tree shows the application with the corresponding threads and the data and performance measurements generated by the thread. Note that the number of threads is the number of active threads plus the GUI-threads plus the stopped threads. The inactive threads are also shown to provide access to historic information.



The throughput is computed for each thread separately and for the entire applications. The sum of the throughput of all threads should be roughly equal to the total throughput of the application.

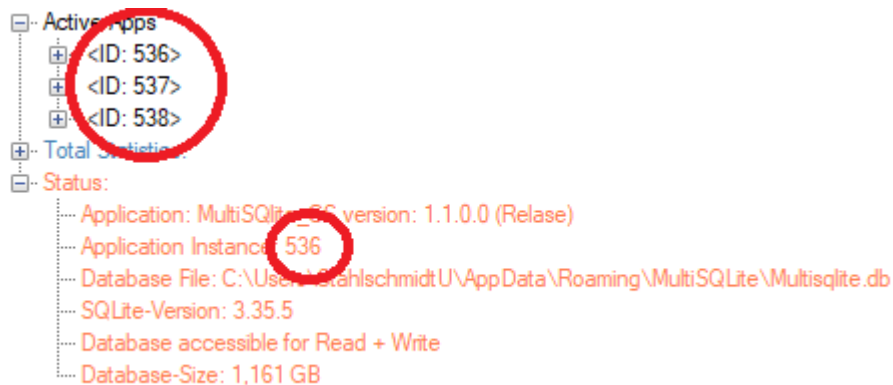
Simulating multiple instances of the same C# Application

Several instances of the MultiSQLite for C# applications can be started simultaneously. Each instance is given a unique application ID which is shown in the title bar of the application:

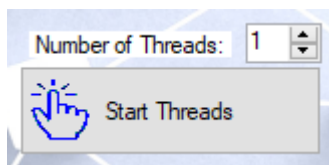


When multiple applications are started simultaneously, each application is showing in the tree view as a separate entries.

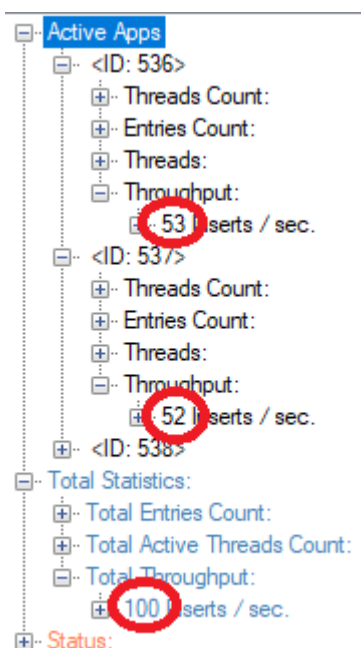
This behaviour can be observed on all running instances



Note that the instance of the particular application is also shown under the node „Status“.



As a next step, in each of the opened applications (or a couple of them) one thread should be started by selecting „1“ in the selector and clicking on start threads.



In any of the opened instances of the application (including the ones without active threads) the nodes in the tree view representing the instances of the applications can be expanded and the total throughput of the individual applications can be observed.

In addition the total load of the database can be shown by expanding the node „Throughput“ under the top-level node „Total Statistics“. This node contains statistics concerning the whole database as such, independent of the applications or threads.



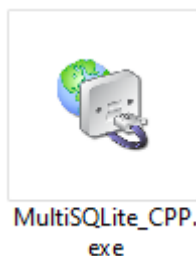
As in the case of multiple threads, the total sum of throuputs of the individual instances of the application should be roughly equal to the „Total Throughput“ on the database-level shown under „Total Statistics“.

Simulating mixed C++ / C# Environments

To simulate the behaviour in mixed C++/C#-environements there is a sister project MultiSQLite for C++, which is also available on GitHub:

<https://github.com/ushaufe/SQLite4CPP.git>

After downloading and extracting, one of several instances of the executable can be started in addition to the C# edition of MultiSQLite.

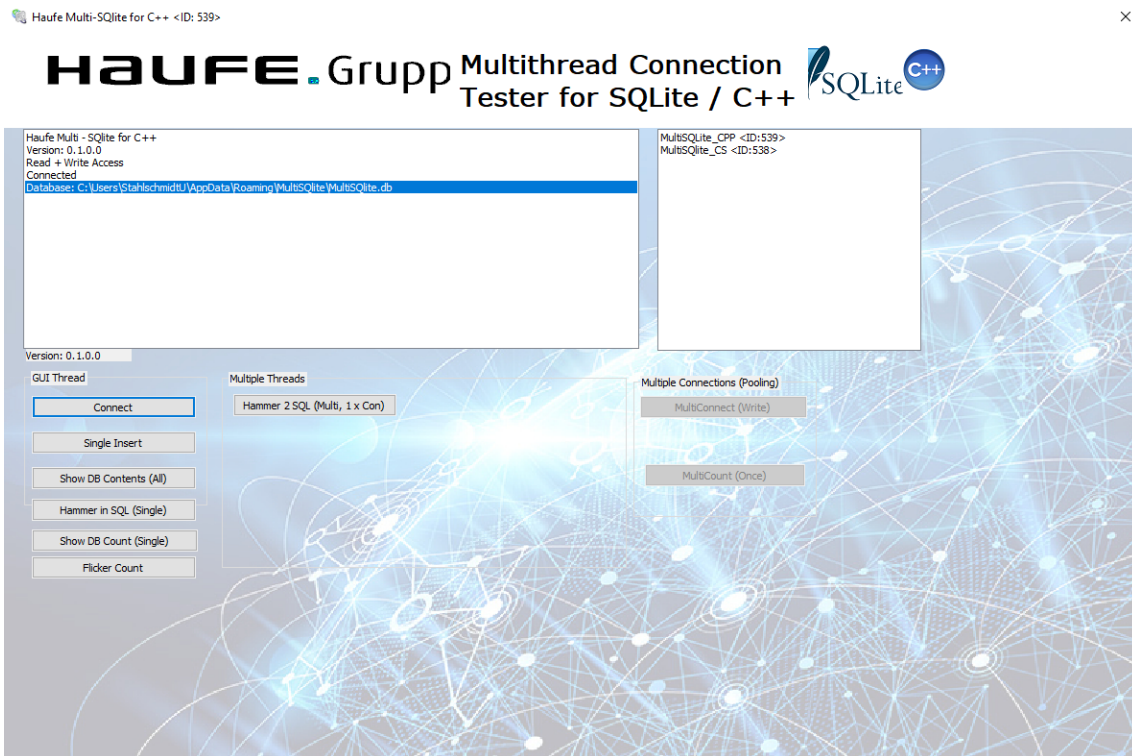


Note that depending on the edition the GUI of that application might look different, but the applications are designed to work together and access the same database so that mixed environments can be simulated and performance and statstic measurements can be shared:

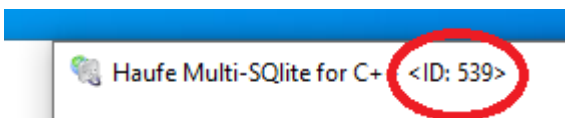


MultiSQLite C# Edition

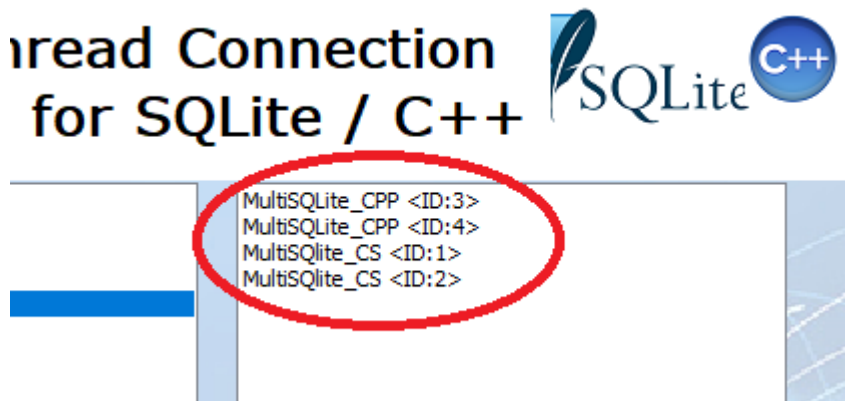
HAUFE.Gruppe



As is the case for the C# Edition, the application is automatically assigned a unique ID that is shown in the headline.



In the view showing the running applications, all running instances of both the C# and the C++ editions of MultiSQLite are shown:



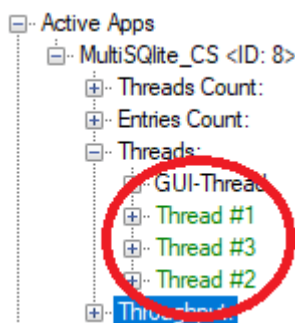
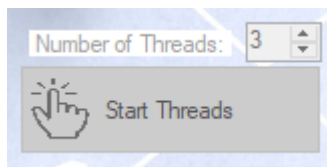
The exact workflow of the C++ edition is described in the user's manual of the corresponding edition. (As of now not provided).



Answers & Conclusions

This chapter is a summary of the conclusions that have been drawn from using SQLite in the different scenarios and answers the questions, what SQLite is capable to do in different environments.

How well does SQLite support Multithreading in C#?



The observations done with this application lead to the conclusion, that Multithreading works in a seaming less way without further ado.

Any number of threads can be started as described in the chapters above, working in the background and providing simultaneous access to the database in a straightforward way.

How does the connection have to be created in C#-Code if Multithreading should be used?

In order to achieve comparable results with this application, version 3.35 of SQLite should be used. In every thread, a connection object should be created

```
SQLiteConnection con = null;
```

and then this connection should be opened with a Connection-String that enables pooling:



```
0 Verweise
public void Connect()
{
    string connectionsString = "Data Source=" + strDatabaseFile + ";Version=3;Pooling=True;Max Pool Size=100;";
    SQLiteCommand cmd = null;

    con = new SQLiteConnection(connectionsString);
    con.Open();

    if (con.State == ConnectionState.Open)
    {
        //... Do something
        //... with the connection
    }
}
```

The pool size determines the number of connections, that this application may simultaneously establish. That's already it.

Handling the threads is also straightforward, there is nothing but normal C# code for creating threads involved:

```
// These threads are used for simultaneous writing...
List<Thread> listThreads = new List<Thread>();
```

Then a class is created for each of the threads that should create a database connection. In the constructor the connections for that thread are opened.

```
3 Verweise
public class CThreads
{
    // Two different Connection-Objets
    SQLiteConnection con1 = null;
    SQLiteConnection con2 = null;

    static int maxThreadID;

    private int appID;

    // Control-Variable if thread is running
    public Boolean running = false;

    // Constructor creates a separate connection for each thread
    1 Verweis
    public CThreads(int appID, String strDatabaseFile)
    {
        this.appID = appID;
        maxThreadID = 0;

        // First connection is opened, note Max Pool Size=100
        string cs = "Data Source=" + strDatabaseFile + ";Version=3;Pooling=True;Max Pool Size=100;";
        // First object is instantiated
        con1 = new SQLiteConnection(cs);
        con1.Open();

        // Second connection is opened, note Max Pool Size=100
        con2 = new SQLiteConnection(cs);
        con2.Open();
        SQLiteCommand cmd = null;
    }
}
```



To start the thread, a function containing an infinite loop must be started from the main application or from the thread where the thread should be started. In that function the connection that has been created in the thread can simply be used as if it was the only connection of the application.

```
1 Verweis
public void insert_thread_function()
{
    int threadID = ++maxThreadID;

    SQLiteCommand cmd = null;

    String strStartThread = String.Format("insert into threads (threadid,appid,isActive) values ({0},'1',1)", threadID, appID);
    cmd = new SQLiteCommand(strStartThread, con1);
    cmd.ExecuteNonQuery();

    // The first thread is writing to the database in an infinite loop
    // using it's own instance of the DB-connection
    // writing can be stopped when setting running = false
    while (running)
    {
        DateTime dt = DateTime.Now;
        String str = "";
        int nNow = (int)(DateTime.Now.Ticks % Int32.MaxValue);
        Random rand = new Random(nNow);
        do
        {
            str = str + (char)rand.Next(65, 65 + 32);
        } while (rand.Next(1, 255) != 100);

        String strInsert = String.Format("insert into testtable (threadid,text,appid) values ({0},'1', {2})", threadID, str, appID);
        cmd = new SQLiteCommand(strInsert, con1);
        cmd.ExecuteNonQuery();
    }

    String strStopThread = String.Format("update threads set isActive=0 where threadID={0} and appID={1} ", threadID, appID);
    cmd = new SQLiteCommand(strStopThread, con1);
    cmd.ExecuteNonQuery();
}
```

The created threads can be started, by creating instance object of each Thread-class and supplying the name of the function containing the infinite-loop as an argument to the constructor of the thread object. The created object can be cast to the thread class.

Each created thread can then simply be started by calling the Start() method of the class containing the thread.

```
for (int i = 0; i < numThreads.Value; i++)
{
    listThreads.Add(new Thread(threads.insert_thread_function));
}
threads.running = true;

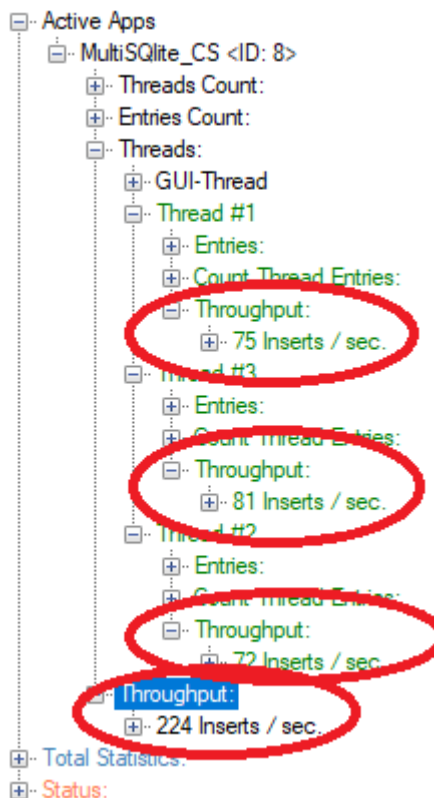
foreach (Thread thread in listThreads)
{
    thread.Start();
}
```



No further precautions have to be taken to synchronize threads or prevent locking the database. The connections are automatically closed when the threads are terminated, since they are objects of the thread class and hence run out of scope when the threads are terminated and their classes are removed from memory.

How is the workload balanced across multiple threads?

Here three threads are started within the same instance of MultiSQLite for C#. Each thread shows in a separate node of the tree with it's own set of data.



The default behaviour is that each thread and each connection that is associated with it is getting the same share of performance.

Balancing between the different threads is hence automatically taken care of by the framework.

As seen in the example on the left side, three threads have been started in addition to the GUI-thread. Each of the threads tries to insert as many of randomly created rows. The result is that all three threads get about 75 inserts / secs, resulting in a total of 225 insert / secs. The GUI-thread is not considered here, since it is IDLE most of the time, but when active, it would behave in the same way.

Does SQLite work with multiple instances of the same application in C#?

Yes, working with multiple applications is also very straightforward. If not defined otherwise in the connection-string, the default behaviour of current



SQLite-Versions is that every application has read-write-access to the SQLite file, and the synchronization to prevent a file lock is handled by SQLite in the background with the temporary creation of journaling-files that are stored and removed automatically in the same directory as the database:

Multisqlite.db	13.08.2021 18:52	Data Base File	1.429.116 KB
Multisqlite.db-journal	13.08.2021 18:52	DB-JOURNAL-Datei	0 KB

This happens in the background, and nothing has to be done by the user in order to handle simultaneous access to the file.

How is the workload balanced across multiple C# applications?

For this example, three different instances of MultiSQLite are started. In each application a simple thread is started.

Active Apps	
MultiSQLite_CS <ID: 10>	
Threads Count:	
Entries Count:	
Threads:	
Throughput:	
	63 Inserts / sec.
MultiSQLite_CS <ID: 11>	
Threads Count:	
Entries Count:	
Threads:	
Throughput:	
	66 Inserts / sec.
MultiSQLite_CS <ID: 12>	
Threads Count:	
Entries Count:	
Threads:	
Throughput:	
	62 Inserts / sec.
Total Statistics:	
Total Entries Count:	
Total Active Threads Count:	
Total Throughput:	
	174 Inserts / sec.
Status:	

As was the case with multithreading, the workload is balanced about equally among the different instances of the C# application.

Each application grabs as much of the bandwidth as is available, and the bandwidth is shared equally.

The only difference is that there seems to be a slightly (but hardly relevant) decrease in overall database performance when used with several application as opposed to several threads.



What about accessing applications writing in different programming languages?

In order to simulate the behaviour of mixed C++ / C# code and accessing the same database with applications written in different languages, several instances of MultiSQLite for C# have been started along with several instances of MultiSQLite for C++. Which can be found here:

<https://github.com/ushaufe/SQLite4CPP.git>

The result is, that this is principally working, too. Unlike the above example with different instances of the C# edition, work-balancing doesn't seem to be as efficient, as the C++ side appears to grab less bandwidth as compared to the C# edition. Also reliability seems to be more of an issue, since the C++ side sometimes goes into a file-lock if too much load is put on the database from too many different sides.

It is not clear yet if this behavior is systematic or if it can be avoided by using alternative ways of using the connection-object on the C++ side. Evaluation is still under way and further details will follow in this chapter along with the development of the C++ Edition of MultiSQLite.

How is the workload balanced across multiple C# application?

On this system (Intel core i7-8650U@1,9GHZ, 16 GB RAM and SSD) a total of about 250 inserts / seconds are consistently reached for the database, more or less independently from the amount of instances or threads accessing the SQLite-database.

The inserts produce random text data of random length (between 0-255 bytes per field).



Does the performance of the database decrease with time or with a growing database?

That doesn't seem to be the case. The following tests have been repeated with a database with about 1.4 GB in size and compared to a brand new database – the results are about the same.

