# Dynamically loading SpatiaLite as an *extension module*

## Theoretical foundations

Technically speaking, SpatiaLite simply is an extension module adding Spatial processing capabilities to SQLite.
Consequently some appropriate mechanism allowing to load both binary libraries (*libsqlite3* and *libspatialite*) then properly registering within SQLite3 the many ***extended Spatial SQL functions*** implemented by SpatiaLite is a strictly required task.
A further complication should be taken in proper debt; *libspatialite* isn't a simple stand-alone self-contained library and depends on several further libraries (*libgeos*, *libproj*, *libxml2* and so on).

There are several possible alternative paths leading to this final goal:

- Using ***static linkage***: a straightforward solution avoiding many subsequent headaches related to installation and run-time configuration.
  Following this approach a single monolithic executable internally containing all required binary code will be created at build time; this monolithic executable will consequently have no external dependencies at all, thus radically simplifying any subsequent installation task.
  Such strategy nicely works in the case of stand-alone self-contained apps (e.g. both **spatialite.exe** and **spatialite_gui.exe** Windows binaries are usually built this way).
  Unhappily this elementary simple solution doesn't fit well in Linux and Unix-like environments; and even worst often directly conflicts with the internal architecture supported by many language connectors or by other complex frameworks (as e.g. is **QGIS**).
- Using ***shared libraries*** (aka ***DLLs*** on Windows).
  This second approach carefully avoids linking once for all a monolithic executable; at the opposite, each single code component will be packaged as an autonomous individual member, and only at run-time all components will be finally bound together.
  The good news: following this approach each library/component will be (hopefully) installed only once on the target system, and every executable depending on that library will load exactly the same binary code thus granting a robust overall consistency.
  The bad new: if a single required component couldn't be loaded at run-time (for any possible reason) the executable as a whole will fail to run.
  To make things even worst, this schema works nicely on Linux, Unix and Mac OS X; but on Windows it's plagued by many serious issues (the infamous ***DLL hell***).
- Using ***dynamic loading*** aka ***late binding*** (which hasn't to be confused with simply using shared libraries).
  Following this third approach an extension module could be selectively loaded or not and the actual action will be activated only at run-time depending on some request event. If for any reason the main executable fails to load an eventual extension module, it will still continue to work by simply ignoring every optional feature requiring the failing module. As a direct consequence, an executable built this way can theoretically growth to the infinite in a highly dynamic fashion; this corresponds to a so called ***plug-in*** architecture.
  Happily enough SQLite3 fully supports this more advanced mechanism; and this option

is widely supported by many language connectors (*Python*, *Java/JDBC*, *PHP*, *C# .NET* and others).

Silly oddities (to be carefully avoided):

- Nothing forbids to create hybrid configurations, e.g. adopting a mix of both static and dynamic linkage.
- Such a solution could easily appeal naive packagers, because it apparently allows to simplify installation and configuration task.
- This is often a dangerous pitfall, simply leading to potentially unstable and self-conflicting configurations: it's the so called ***cobra effect*** (i.e. when an attempted solution to a problem actually makes the problem worse).
- **Golden rule**: never attempt to introduce statically linked libraries in a configuration widely based on shared libraries. This one isn't a clever solution, and will simply introduce many further unexpected complications.

**Dynamic loading: how it works**

*libsqlite3* has the capability to dynamically load extension modules. This feature is available as a C-language API, but is available as a *special* SQL function as well:

```
SELECT load_extension('mod_spatialite');
```

This SQL function make elementary simple (and absolutely language independent) the problem of loading any possible extension; you are simply expected to execute a rather trivial SQL statement in order to load an external extension only when it's actually required.

Loadable extension anatomy:

- Any SQLite's extension module is expected to declare a well known conventional interface. Except for this, any SQLite's own extension simply is an ordinary shared library (or DLL) as any other.
- A real extension module will never explicitly depend on SQLite3, because the following actions will be automatically taken by SQLite3 when loading:
- All SQL functions defined/supported by the extension will be added to the standard list; and eventually an extension could ever overload any pre-defined SQL function directly defined by SQLite3 itself.
- That's not all; SQLite3 will implicitly pass to the extension a *mirror copy* of its own APIs: this will ensure that exactly the same code will support both the SQLite's own core and the extension, and will effectively avoid any possible conflict between them.
- Starting since SQLite version 3.7.17 (released on 2013-05-20) the loadable extension mechanism was significantly enhanced:
- Any extension module is now expected to declare its own individual entry point directly related to the name of the extension; this make simpler loading more modules at the same time.
- Just the ***base name*** of the module is expected to be specified; any eventual ***suffix*** (as e.g. *.so* or *.dll*) will be automatically added by SQLite itself accordingly to the specific target system expectations: this allows for an easier cross-platform portability.
- Any extension module adopting the more recent conventions could be eventually be loaded even when using some obsolete version (< 3.7.17), but in this case same special actions are required. Please consult the SQLite's own documentation for more details.

Common failure causes:

- If the expected conventional interface isn't correctly declared by the shared library being loaded, then SQLite3 will simply assume that it's not a valid extension module.
  And consequently will refuse to load the module without any other consequence (except may be printing some appropriate error message).
- A more obvious failure cause could be the one to attempt loading a not existing module (or possibly, a module whom actual address couldn't be resolved by applying the standard system-dependent searching rules).

- Finally, the module could effectively exists and could correctly declare a valid conventional interface. Anyway, if the corresponding shared library has further dependencies some broken link could possibly forbid a successful loading.
- The last hill-fated combination: SQLite3 is a strongly configurable component. If for any highly questionable choice your system packager had decided to disable at all the extension module mechanism you'll then never be able to load any extension. Full stop. (anyway you'll probably be able to switch to some more reasonable / less paranoid distribution).
- **Useful hint**: many language connectors support a very minimalistic and rather useless diagnostic. If you are experiencing some trouble in successfully loading SpatiaLite as an extension, any attempt to directly debug your code will probably be a frustrating and inconclusive experience.
  Anyway you can always apply an alternative approach:
  - Start the **sqlite3** Command Line front-end.
  - Then execute your problematic **SELECT load_extension** statement.
  - If some failure forbids to successfully load the extension module then **sqlite3** will always print many useful error messages clearly explaining where is the offending cause. In a reasonable configuration both the sqlite3 tool and your language connector should invoke the same identical copy of ***libsqlite3***, so you'll effectively test the same run-time configuration adopted by the language connector but taking profit from richer diagnostic messages (unhappily not all configurations are always net and clean as you could expect ... this rule has many exceptions, most notably on Windows).

**What's new in *mod_spatialite***

You'll probably be already accustomed to load SpatiaLite as a dynamic extension; but any previous version before **4.2** made no real distinction between loadable modules and general-purpose shared libraries.
As we painfully learned by direct on-the-field experience, this apparently simpler configuration caused lot of troubles: instabilities, sudden crashes and so on. Making a clearer distinction between a *general-purpose shared library* and a *pure loadable module* seems to be the definitive solution.
Exactly here is the radical innovation introduce starting since version **4.2**; now SpatiaLite is distributed in two alternative flavors:

- ***libspatialite*** (***.so***, ***.dll***, ***.dylib*** and so on): a genuine classic shared library.
  It will always depend on some external *libsqlite3*, and is uniquely intended to be used by stand-alone applications (such as e.g. ***spatialite*** or ***spatialite_gui***).
  You'll never be able to successfully load this ***libspatialite*** shared library as a dynamic extension via the **SELECT load_extension** mechanism, simply because it doesn't declare the required conventional interface.
- ***mod_spatialite*** (***.so***, ***.dll***, ***.dylib*** and so on): this is simply intended as pure loadable module lacking any explicit SQLite3 dependency.
  You'll never be able to directly link this ***mod_spatialite*** shared library following the classic way, because it doesn't declare any external link symbol except that a single one: i.e. the conventional interface. The unique possible way to load and activate this module is by calling a **SELECT load_extension** SQL statement.

# SpatiaLite on Windows

Chris Locke edited this page on Apr 7 SpatiaLite on Windows · sqlitebrowser/sqlitebrowser Wiki · GitHub

It's possible to use SpatiaLite on Windows by downloading the official `mod_spatialite` binaries. You can download the binaries at the bottom of the page here.
Direct download links:

- 32bit Download
- 64bit Download

Because the `mod_spatialite` library was built as multiple DLLs, you have to make sure Windows can find all of them. DB4S will load an absolute path to the `mod_spatialite.dll`, and `mod_spatialite.dll` in turn loads the rest of the shared libraries like `libgeos_c-1.dll`, `libproj-9.dll` and others. In order to find these dependent .dll's, they must be located in a directory Windows will search.
There are 2 different ways to get it to work:

1. Add the directory containing the .dll files to your `%PATH%`. Windows searches `%PATH%` for dll files so it will be able to locate all of them. This has the side-effect of any program being able to use them.
2. Copy the .dll files to the same directory containing `DB Browser for SQLite.exe`. e.g. `C:\Program Files\DB Browser for SQLite` or `C:\Program Files (x86)\DB Browser for SQLite` whichever version you have installed (32bit or 64bit).

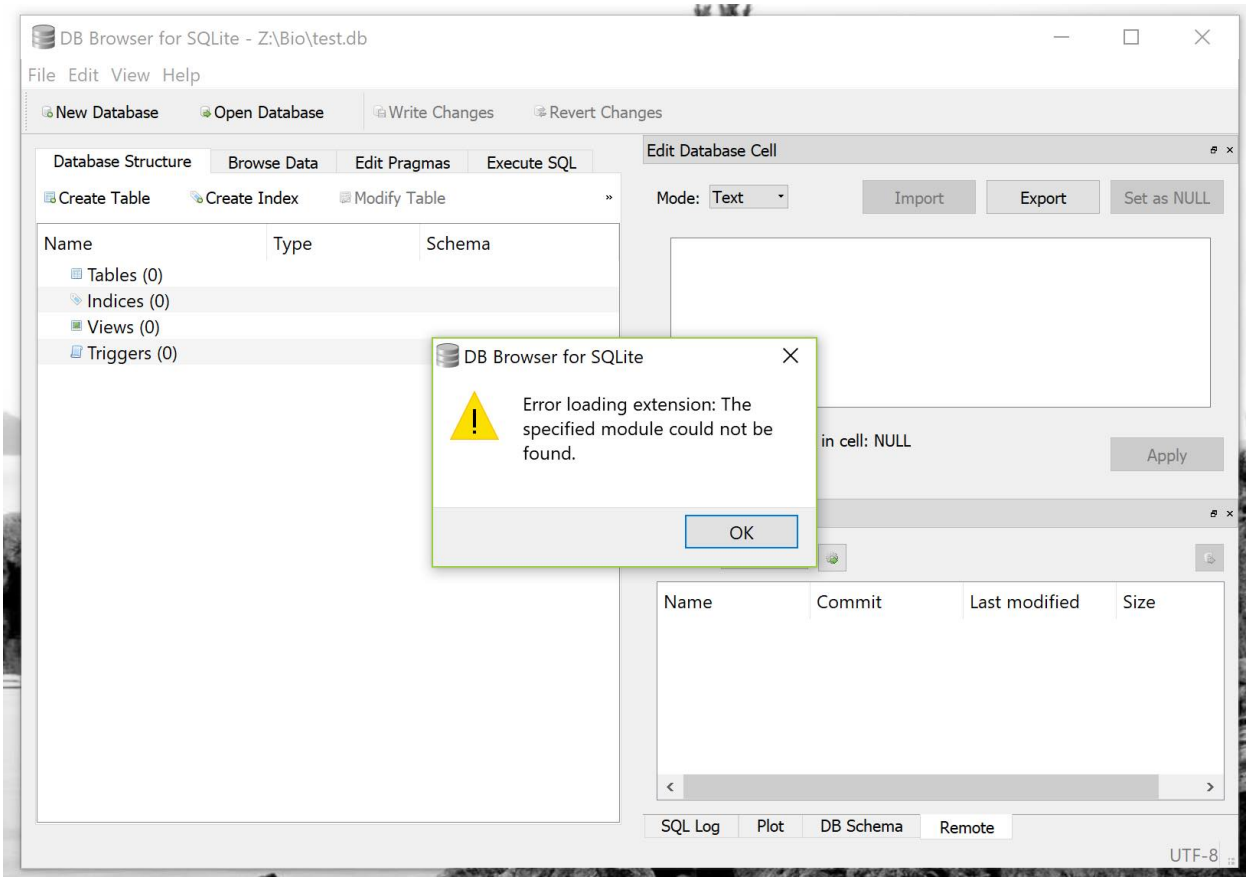
You will also want to make sure you use the correct x86 or x64 dll files that correspond to the version of DB4S you have.

## Alternative instructions, with screenshots

- Working with Geospatial Data - This guide by @bmcbride has a set of step-by-step instructions + screenshots showing how to install SpatiaLite.

## Workaround for Windows 10 failure

If you're using Windows 10 and the SpatiaLite extention fails to load, there is a workaround:




As @karim points out, Spatialite includes an older version of `libstdc++-6.dll`, which works fine in Windows 7 but not Windows 10.
This file comes from MinGW, the compiler they use to build the extension. Replacing it with a newer version works.

1. Delete `libstdc++_64-6.dll` from DB4S folder
2. Go to https://sourceforge.net/projects/mingw-w64/files/ and download one of **x86_64-win32-seh** files
    - Or use this direct link here
3. Extract it somewhere and look into the `bin` folder
4. Copy both `libstdc++-6.dll` and `libgcc_s_seh-1.dll` to DB4S folder
5. Rename `libstdc++-6.dll` to `libstdc++_64-6.dll`
6. Try to load the module again. This time it should work.