Maintaining Private Branches Of SQLite

1.0 Introduction

SQLite is designed to meet most developer's needs without any changes or customization. When changes are needed, they can normally be accomplished using start-time (1) or runtime (2) (3) (4) configuration methods or via compile-time options. It is very rare that an application developer will need to edit the SQLite source code in order to incorporate SQLite into a product.

We call custom modifications to the SQLite source code that are held for the use of a single application a "private branch". When a private branch becomes necessary, the application developer must take on the task of keeping the private branch in synchronization with the public SQLite sources. This is tedious. It can also be tricky, since while the SQLite file format and published interfaces are very stable, the internal implementation of SQLite changes quite rapidly. Hundreds or thousands of lines of code might change for any given SQLite point release.

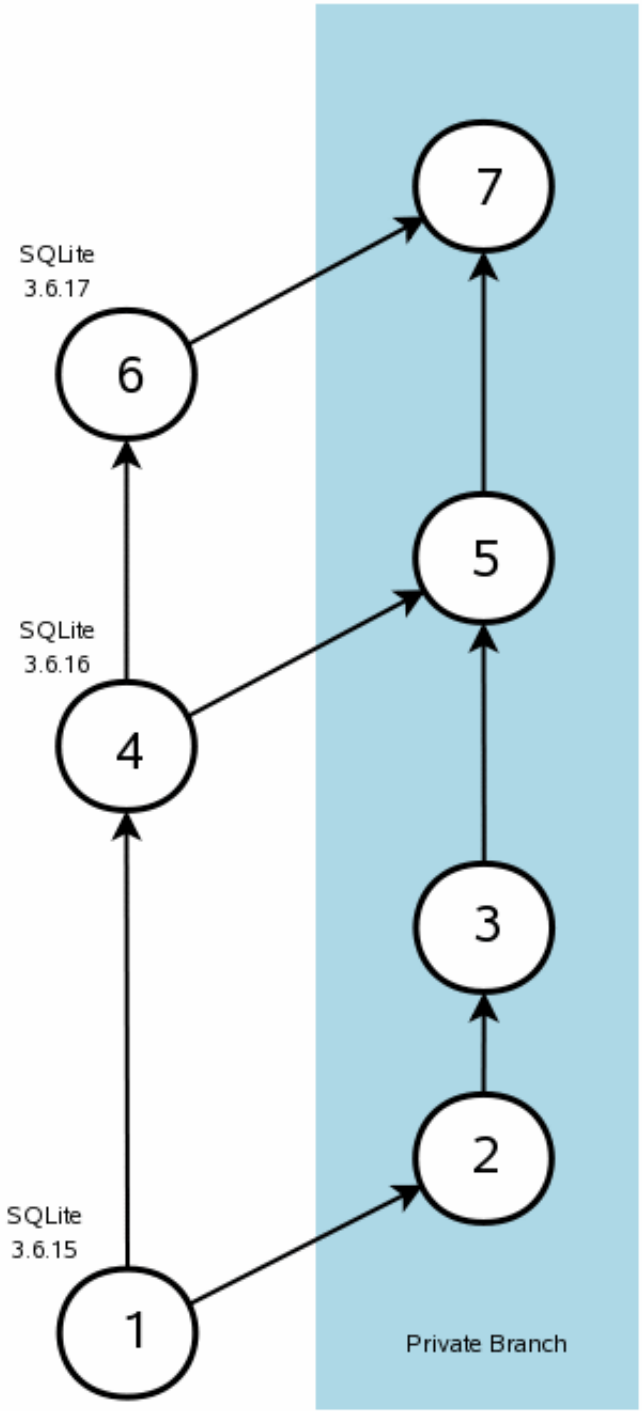
This article outlines one possible method for keeping a private branch of SQLite in sync with the public SQLite source code. There are many ways of maintaining a private branch, of course. Nobody is compelled to use the method describe here. This article is not trying to impose a particular procedure on maintainers of private branches. The point of this article is to offer an example of one process for maintaining a private branch which can be used as a template for designing processes best suited for the circumstances of each individual project.

2.0 The Basic Idea

We propose to use the fossil software configuration management system to set up two branches. One branch (the "public branch" or "trunk") contains the published SQLite sources and the other branch is the private branch which contains the code that is customized for the project. Whenever a new public release of SQLite is made, that release is added to the public branch and then the changes are merged

into the private branch.

This document proposes to use fossil, but any other distributed software configuration management system such as monotone or mercurial (a.k.a. "hg"), or git could serve just as well. The concept will be the same, though the specifics of the procedure will vary.



The diagram at the right illustrates the concept. One begins with a standard SQLite release. For the sake of example, suppose that one intends to create a private branch off of SQLite version 3.6.15. In the diagram this is version (1). The maintainer makes an exact copy of the baseline SQLite into the branch space, shown as version (2). Note that (1) and (2) are exactly the same. Then the maintainer applies the private changes to version (2) resulting in version (3). In other words, version (3) is SQLite version 3.6.15 plus edits.

Later, SQLite version 3.6.16 is released, as shown by circle (4) in the diagram. At the point, the private branch maintainer does a merge which takes all of the changes going from (1) to (4) and applies those changes to (3). The result is version (5), which is SQLite 3.6.16 plus edits.

There might be merge conflicts. In other words, it might be that the changes from (2) to (3) are incompatible with the changes from (1) to (4). In that case, the maintainer will have to manually resolve the conflicts. Hopefully conflicts will not come up that often. Conflicts are less likely to occur when the private edits are kept to a minimum.

The cycle above can be repeated many times. The diagram shows a third SQLite release, 3.6.17 in circle (6). The private branch maintainer can do another merge in order to incorporate the changes moving from (4) to (6) into the private branch, resulting in version (7).

3.0 The Procedure

The remainder of this document will guide the reader through the steps needed to maintain a private branch. The general idea is the same as outlined above. This section merely provides more detail.

We emphasize again that these steps are not intended to be the only acceptable method for maintaining private branch. This approach is one of many. Use this document as a baseline for preparing project-specific procedures. Do not be afraid to experiment.

3.1 Obtain The Software

Fossil is a computer program that must be installed on your machine before you use it. Fortunately, installing fossil is very easy. Fossil is a single "*.exe" file that you simply download and run. To uninstall fossil, simply delete the exe file. Detailed instructions for installing and getting started with fossil are available on the fossil website.

3.2 Create A Project Repository

Create a fossil repository to host the private branch using the following command:

```
fossil new private-project.fossil
```

You can call your project anything you like. The "`.fossil`" suffix is optional. For this document, we will continue to call the project "`private-project.fossil`". Note that `private-project.fossil` is an ordinary disk file (actually an SQLite database) that will contain your complete project history. You can make a backup of the project simply by making a copy of that one file.

If you want to configure the new project, type:

```
fossil ui private-project.fossil
```

The "ui" command will cause fossil to run a miniature built-in webserver and to launch your web-browser pointing at that webserver. You can use your web-browser to configure your project in various ways. See the instructions on the fossil website for additional information.

Once the project repository is created, create an open checkout of the project by moving to the directory where you want to keep all of the project source code and typing:

```
fossil open private-project.fossil
```

You can have multiple checkouts of the same project if you want. And you can "clone" the repository to different machines so that multiple developers can use it. See the fossil website for further information.

3.3 Installing The SQLite Baseline In Fossil

The repository created in the previous step is initially empty. The next step is to load the baseline SQLite release - circle (1) in the diagram above.

Begin by obtaining a copy of SQLite in whatever form you use it. The public SQLite you obtain should be as close to your private edited copy as possible. If your project uses the SQLite amalgamation, then get a copy of the amalgamation. If you use the preprocessed separate source files, get those instead. Put all the source files in the checkout directory created in the previous step.

The source code in public SQLite releases uses unix line endings (ASCII code 10: "newline" only, NL) and spaces instead of tabs. If you will be changing the line ending to windows-style line endings (ASCII codes 13, 10: "carriage-return" and "newline"; CR-NL) or if you will be changing space indents into tab indents, **make that change now** before you check in the baseline. The merging process will only work well if the differences between the public and the private branches are minimal. If every single line of the source file is changed in the private branch because you changed from NL to CR-NL line endings, then the merge steps will not work correctly.

Let us assume that you are using the amalgamation source code. Add the baseline to your project as follows:

```
fossil add sqlite3.c sqlite3.h
```

If you are using separate source files, name all of the source files instead of just the two amalgamation source files. Once this is done, commit your changes as follows:

```
fossil commit
```

You will be prompted for a check-in comment. Say whatever you like. After the commit completes, your baseline will be part of the repository. The following command, if you like, to see this on the "timeline":

```
fossil ui
```

That last command is the same "ui" command that we ran before. It starts a mini-webserver running and points your web browser at it. But this time we didn't have to specify the repository file because we are located inside a checkout and so fossil can figure out the repository for itself. If you want to type in the repository filename as the second argument, you can. But it is optional.

If you do not want to use your web browser to view the new check-in, you can get some information from the command-line using commands like these:

```
fossil timeline
fossil info
fossil status
```

3.4 Creating The Private Branch

The previous step created circle (1) in the diagram above. This step will create circle (2). Run the following command:

```
fossil branch new private trunk -bgcolor "#add8e8"
```

This command will create a new branch named "private" (you can use a different name if you like) and assign it a background color of light blue ("#add8e8"). You can omit the background color if you want, though having a distinct background does make it easier to tell the branch from the "trunk" (the public branch) on timeline displays. You can change the background color of the private branch or of the public branch (the "trunk") using the web interface if you like.

The command above created the new branch. But your checkout is still on the trunk - a fact you can see by running the command:

```
fossil info
```

To change your check-out to the private branch, type:

```
fossil update private
```

You can run the "info" command again to verify that you are on the private branch. To go back to the public branch, type:

```
fossil update trunk
```

Normally, fossil will modify all the files in your checkout when switching between the private and the public branches. But at this point, the files are identical in both branches so not modifications need to be made.

3.5 Adding Customizations To The Code In The Private Branch

Now it is time to make the private, custom modifications to SQLite which are the whole point of this exercise. Switch to the private branch (if you are not already there) using the "`fossil update private`" command, then bring up the source files in your text editor and make whatever changes you want to make. Once you have finished making changes, commit those changes using this command:

```
fossil commit
```

You will be prompted once again to enter a commit describing your changes. Then the commit will occur. The commit creates a new checkin in the repository that corresponds to circle (3) in the diagram above.

Now that the public and private branches are different, you can run the "`fossil update trunk`" and "`fossil update private`" commands and see that fossil really does change the files in the checkout as you switch back and forth between branches.

Note that in the diagram above, we showed the private edits as a single commit. This was for clarity of presentation only. There is nothing to stop you from doing dozens or hundreds of separate tiny changes and committing each separately. In fact, making many small changes is the preferred way to work. The only reason for doing all the changes in a single commit is that it makes the diagram easier to draw.

3.6 Incorporating New Public SQLite Releases

Suppose that after a while (about a month, usually) a new version of SQLite is released: 3.6.16. You will want to incorporate this new public version of SQLite into your repository in the public branch (the trunk). To do this, first change your repository over to the trunk:

```
fossil update trunk
```

Then download the new version of the SQLite sources and overwrite the files that are in the checkout.

If you made NL to CR-NL line ending changes or space to tab indentation changes in the original baseline, make the same changes to the new source file.

Once everything is ready, run the "`fossil commit`" command to check in the changes. This creates circle (4) in the diagram above.

3.7 Merging Public SQLite Updates Into The Private Branch

The next step is to move the changes in the public branch over into the private branch. In other words, we want to create circle (5) in the diagram above. Begin by changing to the private branch using "`fossil update private`". Then type this command:

```
fossil merge trunk
```

The "merge" command attempts to apply all the changes between circles (1) and (4) to the files in the local checkout. Note that circle (5) has not been created yet. You will need to run the "commit" to create circle (5).

It might be that there are conflicts in the merge. Conflicts occur when the same line of code was changed in different ways between circles (1) and (4) versus circles (2) and (3). The merge command will announce any conflicts and will include both versions of the conflicting lines in the output. You will need to bring up the files that contain conflicts and manually resolve the conflicts.

After resolving conflicts, many users like to compile and test the new version before committing it to the repository. Or you can commit first and test later. Either way, run the "`fossil commit`" command to check-in the circle (5) version.

3.8 Further Updates

As new versions of SQLite are released, repeat steps 3.6 and 3.7 to add changes in the new release to the private branch. Additional private changes can be made on the private branch in between releases if desired.

4.0 Variations

Since this document was first written, the canonical SQLite source code has been moved from the venerable CVS system into a Fossil repository at http://www.sqlite.org/src. This means that if you are working with canonical SQLite source code (as opposed to the amalgamation source code files, sqlite3.c and sqlite3.h) then you can create a private repository simply by cloning the official repository:

```
fossil clone http://www.sqlite.org/src private-project.fossil
```

This command both creates the new repository and populates it with all the latest SQLite could. You can the create a private branch as described in section 3.4.

When the private repository is created by cloning, incorporating new public SQLite releases becomes much easier too. To pull in all of the latest changes from the public SQLite repository, simply move into the open check-out and do:

```
fossil update
```

Then continue to merge the changes in "trunk" with your "private" changes as described in section 3.7.