About Using SQLite With Clarion

A CHT Study Paper About SQLite

SQLite, An Improvement Over Topspeed Format?

In the past number of weeks, we've spent quite a bit of time on the HNDSCHOOL.APP and the SQLite driver. This application, based on SV's example application SCHOOL.APP, is taking shape and looks and behaves pretty well, though it would never stand up to the data-design requirements of a real college or university. Even with the extensive dictionary changes we've introduced to it, the application still only qualifies as a "demo". We make no claims of having turned a sow's ear into a silk purse. Later in this series of short essays, we will elaborate on the dictionary changes made, and our reasons for making them.

For the purposes of this study, "demo" quality is OK, though. We took on the app as a means to illustrate SQL concepts involving Clarion and CHT, at the most basic level, with a very portable SQL driver *(SQLite)*, one that any Clarion developer could readily adopt or introduce into an existing app if that app is still running on a Topspeed ISAM file system.

SQLite, Is It Any Better Over A WAN Than .TPS?

We are pretty firmly convinced at this point that the SQLite driver probably isn't any more reliable than the .TPS driver when it comes to across-the-WAN access, where we know the Topspeed file format often falls flat. The developers of SQLite have this to say on the topic:

> *"If you have many client programs accessing a common database over a network, you should consider using a client/server database engine instead of SQLite. SQLite will work over a network filesystem, but because of the latency associated with most network filesystems, performance will not be great. Also, the file locking logic of many network filesystems implementation contains bugs (on both Unix and Windows). If file locking does not work like it should, it might be possible for two or more client programs to modify the same part of the same database at the same time, resulting in database corruption. Because this problem results from bugs in the underlying filesystem implementation, there is nothing SQLite can do to prevent it."*

Nothing different than Topspeed file format there! In fact, this statement even absolves .TPS from some of it's shortcomings on *Wide Area Networks*. The problems are indemic to at least two major operating systems, not the .TPS file driver or its ISAM-based implementation.

If nothing else, the SQLite driver may be a useful intermediate step to full SQL for an overburdened .TPS-based app. Whether an "intermediate" step is needed, depends on the developer's familiarity with SQL concepts and his requirements for speed and robustness.

First off, one must recognize that all SQL systems are not created equal. At its heart, SQLite is still an ISAM *(Indexed Sequential Access Method)* data system, not a true client server data system like, MSSQL, Oracle and many others.

No doubt about one thing, though, what you end up doing to your .TPS app to take it to SQLite, dictionary design-wise, and app/form/report/process-wise, applies equally to SQLite and MSSQL with only minor variations due to several internal differences in the SQL variant. These differences range widely from one SQL system "brand" to the other.

SQLite And TopSpeed Are Non-Client Server Data Repositories

The obvious difference between SQLite and MSSQL, besides the cost of implementation, is the fact that with .SQLITE, as with .TPS , all the work of data gathering, filtering, and sortation is done by the data client, namely your app. This is a characteristic that typifies *non-client-server* apps in general. However, we say this with some qualification, because SQLite provides for the use of database *VIEWS* and *TRIGGERS*. They are something entirely foreign to TopSpeed files *(i.e we mean, VIEW and TRIGGERS located in the database, not in the app)*. And they allow a certain amount of the data gathering and filtering work to be handed off to the SQLLite database engine - *SQLite3.DLL*.

In a *client-server* SQL system, of course, the task of data gathering, filtering and sortation is handled by the data "engine" or data "server". A *client-server* application, as the name implies, consists of two separate and equally important parts, a "client" part and a "server" part with the server usually located close to the data tables and the client located somewhere out in virtual or nether space.

A *non-client-server* application, on the other hand, is comprised only of a single application doing the work of both "client" and "server". Because these applications are normally located near the data tables, they are

generally easier to manage, portable and fast enough. But when removed from the proximity of the data tables, across a *Wide Area Network*, in a shared-access environment, *non-client-server* apps tend to be slow, and prone to data table corruption, some of the reasons for which are stated above.

It only stands to reason that a "data server" sitting right next to the data files can process and manipulate information more quickly and effectively than a "client" sitting miles away across a network or across the internet.

Why Bother Using SQLite?

At this point let's begin to consider if there are any good reasons to use SQLite to replace the Topspeed file format. So far, we've discussed no rationale for adoption of SQLite in your Clarion applications other than it's value as a portable data format that could possibly replace the Topspeed format for the purpose of minimizing certain problems inherent in that format when used across the WAN and its value as a transitional step to "real" client-server SQL.

Another approach to determining a value approximation of SQLite is a quick analysis of SQLite's present use in the world community. Who is using it and for what purpose is it being used?

The following page provides a valuable overview on the topic of SQLite useage.

That investigation proves useful, and from our point of view at least, we could begin to see that spending time learning SQLite even if it offers very little or no speed and reliability benefits above and beyond Topspeed format, is still worth the effort because it widens the inter-operability of your Clarion applications with other, newer technologies, particularly technologies located on portable devices.

If you as yet, harbour any illusions that SQLite will give you a cheap, open-source replacement for MSSQL or Oracle, let's put that misconception to bed. Here is a quote from the "About Section" of **SQLITE.ORG**.

> "*SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does* **not have a separate server process**. *SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. The database file format is cross-platform - you can freely copy a database between 32-bit and 64-bit systems or between big-endian and little-endian architectures. These features make SQLite a popular choice as an Application File Format.* ***Think of SQLite not as a replacement for Oracle but as a replacement for fopen()*****"...**

In other words, SQLLite is a great "local" filing system. It's targeted exactly where TopSpeed format already exists and works effectively. This information is repeated in other words on the same site under this heading: **Appropriate Uses For SQLite**

> "*SQLite is not directly comparable to other SQL database engines such as Oracle, PostgreSQL, MySQL, or SQL Server since SQLite is trying to solve a very different problem. Other SQL database engines strive to implement a shared repository of enterprise data. They emphasize scalability, concurrency, centralization, and control. SQLite, on the other hand, strives to provide local data storage for individual applications and devices. SQLite emphasizes economy, efficiency, reliability, independence, and simplicity*"...

For Clarion developers, adopting SQLite and being able to "share" data in a seamless, way with the latest technologies from Skype to IPad is an attractive feature. While SQLite is still only going to be useful for *local* data storage, let's add *interoperability* to the recognized benefits of SQLite. *The benefits list now is, economy, efficiency, reliability, independence, simplicity and* ***interoperability with the very newest technologies.***

Situations Where SQLite Works Well

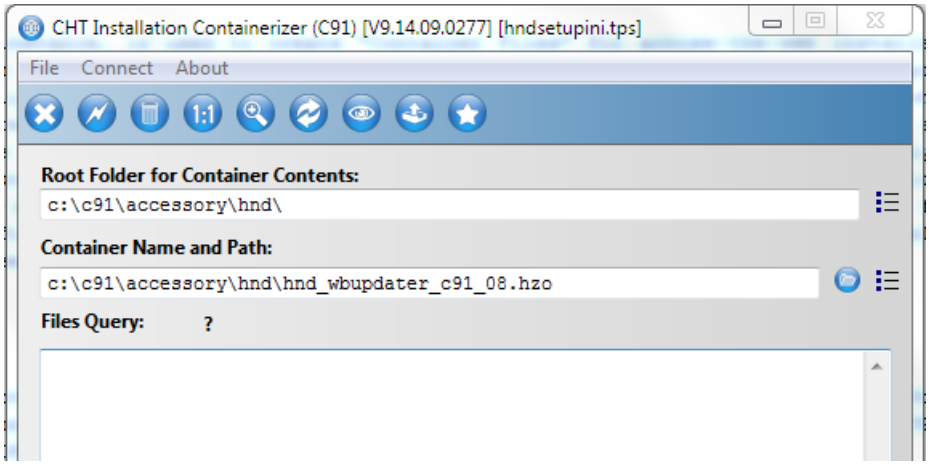
After reading **"Situations Where SQLite Works Well"** at the SQLite website, especially the part pertaining to "*Application Files*" we began thinking about some of our numerous "Utility" implementations that CHT staff have created over the years using TopSpeed file format which would immediately work at least as well, or perhaps better, using SQLite file format.

CHT Installation Containerizer

HNDSETUPCx.APP, AKA "CHT Installation Containerizer", for instance, is used to create "Container Files" for across-the-web installs. This utility application, for which you are given the source application as part of your download subscription, presently compresses, and optionally encrypts, any other file type and stores it into Topspeed file blobs from which the contents can be extracted and restored to disk by HNDSETUPCx.APP or other Clarion applications like CHT's WEBUPDATER installer/updater.

With the TopSpeed file format we're able to create additional fields and indexes that describe and randomly store and retrieve files in a way that puts .ZIP format to shame. From what we've seen of SQLite, and it's ability

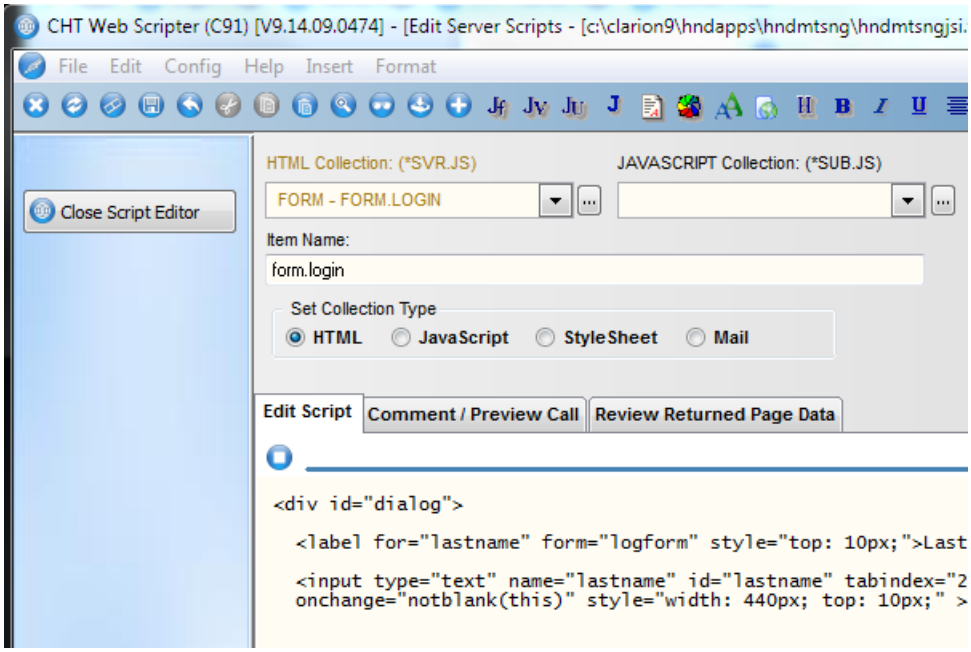
to handle BLOB entities, keys, and indexes, HNDSETUPCx.APP can easily be re-built around the SQLite file format. Wider open-source access of these files to other languages and platforms via SQLite would be of considerable benefit over the arcane, proprietary TopSpeed format, which without jumping through hoops, is only available to Clarion applications.



CHT Web Scripter

HNDSCRPT.APP, AKA "CHT Web Scripter", is another major *CHT Utility Application* that uses the TopSpeed file format to store website pages, code, javascript code and CSS definitions in a way that encapsulates an entire website or web application into a single, relational, container file.

Again, SQLite could easily be substituted for the .TPS format used by that application. A HNDSCRIPT.APP "JSItems" file does not use a BLOB entity to store the user's code but we've been thinking about changing that and will consider changing file formats to SQLite, again because of the open nature of SQLite format as well as its robustness and touted reliability. SQLite's driver also sports an "AutoBackup" API which even if not implemented via the Clarion SQLLIte file driver could easily be made to work from Clarion using a bit of C# and COM interop.

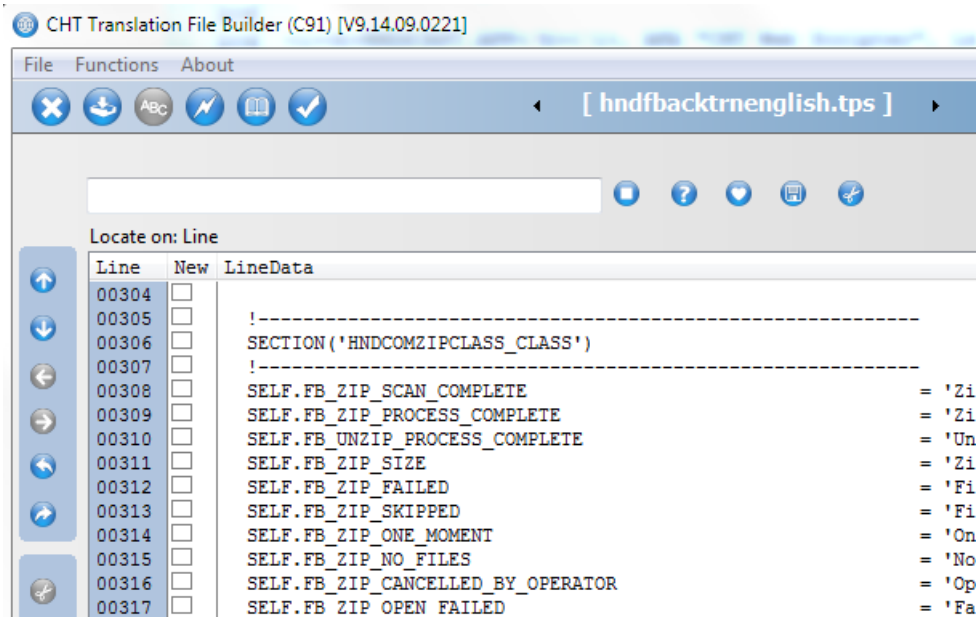


CHT Translation File Builder

HNDFBACKBUILDER.APP AKA "CHT Translation File Builder", utilizes the magic of keys and indexes in a Topspeed file to make the accurate merging of two different translation files possible. Since we make regular changes to HNDFBACK.TRN, *our built-in, English-only*, translation file for CHT classes, it is possible for developers to easily merge their localized HNDCUSTOM.TRN file with our HNDFBACK.TRN file such that anything missing from their HNDCUSTOM.TRN is inserted into it from our HNDFBACK.TRN.

This is achieved by constructing relational versions of our HNDFBACK.TRN and their HNDCUSTOM.TRN, using keys to check for duplicate or missing entries, and then reading the two files and writing out to the

HNDCUSTOM.TPS file any missing entries found in HNDFBACK.TPS. This could readily be done with an SQLLite file driver with no loss of functionality.



CHT Web Forum Server

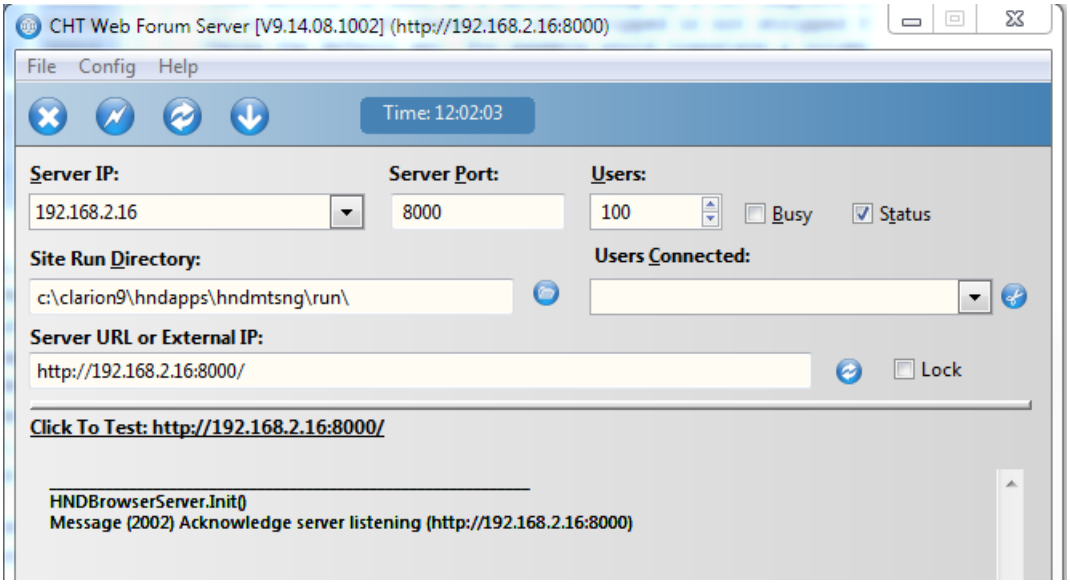
HNDMTSNG.APP, AKA "CHT Web Forum Server", serves to the Web, CHT's interactive support forum -- and like all of the above, is provided in source code to CHT subscribers.

This application currently runs on a Topspeed-format file. Our forum primary data file currently contains approximately 60,000 messages posted by CHT developers since its inception. These messages are stored in BLOBs, and there are several keyed fields to assist with message retrieval by various criteria.

There are at least three SQL-style features in SQLite, not available in Topspeed format, which temptingly draw us to consider SQLLite for this and other data-server applications in our suite. These are: *built-in autoincrement*, *data-base views* and *data-base triggers*.

And while the .TPS format has been stable and reliable for this application, a server application of this sort might be improved by using a non-proprietary file format that sets it up for easy interoperability with non-Clarion languages and systems.

CHT web servers all work on the client-server principle, serving up back-end views to remote web clients, both EXE based, and browser based, so it isn't difficult to see where our interest in SQLite may be leading.



At This Point, What Do We Think About SQLite?

Our pre-conclusion, summary, from what we've seen, read and discussed about SQLite's many features to this

point, is that there is certainly *nothing lost* in using it as a replacement for Topspeed format and that there are going to be some *significant* bonuses gained by adopting SQLite.

In the next, extensively reasearched section starting with, **SQLite: Database Views And Triggers**, we begin to explore and test some SQLite features not found - in fact, not even dreamed of - in Topspeed data file format. Some of these "extra" SQLite features are readily available to be used directly from a Clarion application, for example, *database views* and *database triggers*.

This document deals first and foremost with features readily accessed from Clarion applications and with the possible benefits of those features to your Clarion applications. This goes well above and beyond obtaining simple intercompatibility with other, newer application systems such as those found these days on tablets, and smart phones from various vendors.

SQLite: DATABASE VIEWS AND TRIGGERS

You'll see us use terms like "DB-VIEWs" or "DATABASE VIEWs" or "BACK-END VIEWs" interchangeably in this essay. Those names refer to VIEW structures located IN the SQLite database, instead of IN the Application. For clarity, let's call the VIEW structures located inside your application "CLARION VIEWs" and the VIEW structures located inside an SQLite database "DATABASE VIEWs".

If you're not sure what "CLARION VIEWs" are, please take some time to learn about them and how they're used by reading this document: **Related Table Data On Clarion Browses**.

> *(App-Reference Next Paragraph: BrowseEnrollment procedure in HNDSCHOOL.APP)*

DATABASE-VIEWS we reiterate for emphasis, are not located inside a Clarion application, they are located in the database. In the document named above, "Related Table Data On Clarion Browses" we've explained in considerable detail how to join 5 different tables on a Clarion browse and how the CLARION VIEW structure allows us to browse, sort and query those 5 tables -- *with considerable help from CHT Browse Templates* -- as if they were one table.

> *(App-Reference Next Paragraph: BrowseEnrollmentVW procedure in HNDSCHOOL.APP)*

With that out of the way, we'll next explain how with one DATABASE-VIEW, developers can do the same thing, *that is, join and access five back-end tables at once in a single browse*, with a far simpler browse design, **AND also get some bonus features**. These bonus features gained, are listed in the bullet-points which follow.

> • *Ability to header-click-sort concatenated data columns down to the last character on all the fields comprising the concatenation.*
>
> • *Ability to filter concatenated data columns on all the fields comprising the concatenation.*
>
> • *Ability to build a multi-table ABC browse as if it were accessing a single, flat-file table containing all fields chosen for the view, that are located anywhere in the multi-table set.*
>
> • *Ability to conceivably make changes to browses and forms without touching the Clarion application, simply by changing the concatenation definitions for existing fields in your view.*

SQLite Database Views In HNDSCHOOL.APP, Under The Covers

Here, next we present some pictorial evidence from inside the HNDSCHOOL.APP. Two procedures called `BrowseEnrollment` and `BrowseEnrollmentVW` are, for all intents and purposes, identical when you open them. But, technically, they're by no means identical. Let's examine these procedures both from the end-users point of view and from the developer's point of view by looking under the covers.

This first browse is based on a five-table, CLARION VIEW arrangement. The `HNDSCHOOL.APP` procedure is called, `BrowseEnrollment`.

This next browse is a based on single Clarion table called `EnrollmentView` looking at a DATABASE VIEW called `EnrollmentView` located in the HNDSCHOOL.SQLITE database.

The procedure in `HNDSCHOOl.APP` is called, `BrowseEnrollmentVW`.

Here is the table schematic for the `BrowseEnrollment` procedure, and to the right of that, in the same image, is the CLARION VIEW structure created to gather data for the browse.

Here, next, is the table schematic for the `BrowseEnrollmentVW` procedure and to the right of that, in the same image, is the CLARION VIEW structure created for the browse.

These two designs look, technically, quite different even though the browses themselves, from the end-user's perspective, look practially identical.

The difference, of course, again speaking as a developer from the technical side, is that the first, more complex browse, is accessing 5 tables directly from the application. The second, less complex browse, is accessing the same 5 tables, but is accessing them indirectly via a DATABASE-VIEW. The table joins and concatenations are taking place in the back-end, instead of in the Clarion application.

> *Keep this little gem of information we're sharing with you here, tucked away in your mind for future reference. Remember that everything you're learning here is immediately transferrable to Client Server database systems, like MSSQL or Oracle even though, SQLite is not a Client Server based SQL system. The benefits of back-end DATABASE VIEWs are transferrable to the BIG guys' SQL systems where they provide even more benefits than we're able to extract from a non-client server SQL system like SQLite.*

With Client Server systems we can move the separate `CLIENT` and `SERVER` parts thousands of miles apart **SO** we really want to minimize "chatter" happening between the client and the database server. The more "yapping" going on, the slower our data access.

It stands to reason in a *Client Server Database System* that the more work we can perform in the back-end, at the database side, the less work our application has to perform from a thousand miles away. Again, even though SQLite is not a *Client Server Database System*, it has this really beneficial feature, found also in expensive Client Server Systems, namely, DATABASE-VIEWs.

SQLite Database Views, CREATING FROM A CLARION APPLICATION

How is a DATABASE VIEW created?

It's done with an SQL command called CREATE VIEW. Next is a screen-snap from HNDSCHOOL.APP taken inside the `CreateDB()` procedure. Note that we don't need to run this procedure every time our app starts. Once the VIEW is created in the database it stays there until you remove it with SQL command DROP VIEW.

EnrollmentView can, alternatively, be inserted into the database from the SQLBrowser application called *SQLITEBROWSER.EXE*. A copy of this utility can be located in your CHT installation area. Look under `/accessory/hnd/sqlite/`. We'll discuss the use of that utility executable, separately in another one of these essays.

Here, then, is the code from our app that injects the view structure called `EnrollmentView`, into the HNDSCHOOL.SQLITE database:

Be aware that all SQL brands are not identical. They are, generally, fairly close in syntax, but certainly not identical. Consequently, when you try this with a different SQL brand, do your homework and practice the syntax with their SQLBrowser utility before you begin coding. That will save you a lot of time, since you can hack away at the syntax from the SQLBrowser tool until the `CREATE VIEW` command works for you. Then copy that SQL code into your app the way we've done in the above image. And for heaven's sake RTFM, beforehand!

Note in the next image that the `EnrollmentView` VIEW structure does not appear inside the SQLBrowser utility looking anything like a table. It is separately defined as a VIEW not a TABLE.

When we expand the VIEW structure for `EnrollmentView`, it starts to look more like a table. But it is just a structural mechanism that brings an existing group of tables and fields together on behalf or our application. Here is that expanded VIEW structure.

The really useful aspect of this is that Clarion sees and codes for this DATABASE VIEW as if it were just another file in the database. In the next image is a snapshot of the Clarion definition used to traverse this view structure in HNDSCHOOL.DCT.

Here is another snapshot from the HNDSCHOOL.DCT which furthers the impression that `EnrollmentView` is just another table in the database.

And here in the next image, below, is yet another snapshot from our HNDSCHOOL.DCT showing that it *IS* possible to join a view-based table in the dictionary, to another, standard database table. *Note that this specific join is not necessary for our `BrowseEnrollmentVW()` browse to work as it does.*

We've added this join to the `Enrollment` table so that Clarion does an automatic fetch on `Enrollment` table when users enter the ABC update form, implemented in the `UpdateEnrollment()` procedure. But that's another story, which we will explain further in a later release of this essay.

SQLite Database Views, A Brief Review

At this point, lets review briefly just how different the Clarion-side design actually is in these two procedures. First the `BrowseEnrollment` procedure accessing five, joined table directly:

Next, we see the `BrowseEnrollmentView` procedure, accessing five tables as if they were one, big, flat file. *Ignore, for now, the join we're showing to `Enrollment`.* That join is irrelevant to the browse at this point and we'll explain later how that join is used in the update form attached to the view-based browse procedure we've called, `BrowseEnrollmentView`.

SQLite Database Views, Your Turn To Test

Finally it's your turn to test these two procedures. Can you spot the behavioural differences that we've bullet-pointed above?

The latest HNDSCHOOL.APP, HNDSCHOOL.DCT and HNDSCHOOL.SQLITE database are included in your CHT Toolkit, and may be found in the /hndapps/ directory, the location of which you determined by configuration on the CHT Webupdater Interface. This is the same .APP and .DCT from which the images in the DATABASE VIEWs portion of this essay originate.
