Vive La Difference – How SQLite varies from VFP SQL

Whil Hentzen

"We need to get away from DBFs" is a refrain I hear regularly from fellow developers. Be it due to perceived instability of the file format, the need for tables larger than 2 GB, or the result of political machinations, the result is the same – a desire to move to a SQL database back-end. SQLite can be an excellent intermediate step – and possibly the final word - in the process of restructuring your application to talk to a SQL back-end.

In the last three articles, I've shown you (1) how to connect to a SQLite database from VFP, (2) how to handle a variety of errors you may run into when connecting, and then how to use SQLEXEC() to send SQL commands from VFP to SQLite, and (3) how to move lots of data in and out of SQLite.

In this article, I'll alert you to some of the differences between SQLite and VFP's SQL implementation.

There are three categories of differences between SQLite and VFP SQL. These are (1) language differences, (2) engine differences, and (3) implementation differences.

Language differences

The whole point of SQL is to have a standardized query language that works across platforms, hardware, languages. Thus, it may come as a surprise to some that "All SQLs are equal, but some are more equal than others."

There are several standard SQL commands not implemented in SQLite. Furthermore, the SQLite function set is much smaller than VFP's.

Commands - Joins

Visual FoxPro supports full outer joins. Suppose you've got a parent table and a child table, such as Person and Pet, where a Person could have one or more Pets, but a Pet may or may not belong to a specific Person. In VFP, you can write a query to find all Persons with one or more Pets, like so:

```
select * from person join pet on ;
pet.iidperson = person.iidperson
```

This pulls only records from Person and Pet where a Person has a Pet. It ignores the Persons who are petless as well as the Pets who are strays.

You can also write a query to find all Persons, whether they have Pets or not, but ignoring strays:

```
select * from person left join pet on ;
pet.iidperson = person.iidperson
```

You can write a query to find all Pets, regardless if they have an owner or if they're a stray, but ignoring Persons who don't have a Pet:

```
select * from person right join pet on ;
pet.iidperson = person.iidperson
```

And, finally, you can write a query to find all Persons and all Pets, regardless if they have a matching record in the other table, like so:

```
select * from person full join pet on ;
pet.iidperson = person.iidperson
```

You execute SELECTS like this with SQLite via SQLEXEC(). (See the March article for details and multiple examples.) For example,

```
m.lcXN = "DRIVER={SQLite3 ODBC Driver};" ;
   + "Database=f:\PersonPet;"
liH = sqlstringconnect(lcXN)
=sqlexec(lih, "select * from person ;
   join pet ;
   on pet.iidperson = person.iidperson")
```

will produce a cursor named "sqlresult" that can be browsed. This cursor contains all Persons who have one or more Pets, but no petless Persons, nor any strays. See **Figure 1**.

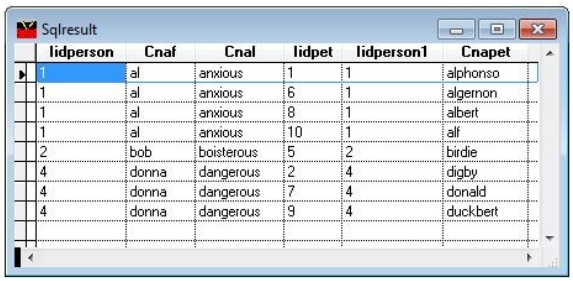

Figure 1. A join executed in SQLite.

Here's the first big difference between VFP and SQLite: SQLite only supports left joins, not right or full joins. So you can find out matches, where there is a Person and a Pet, using a regular join. You can also find Persons with or without Pets, like so:

```
select * from person left join pet on ;
pet.iidperson = person.iidperson
```

and shown in **Figure 2**.

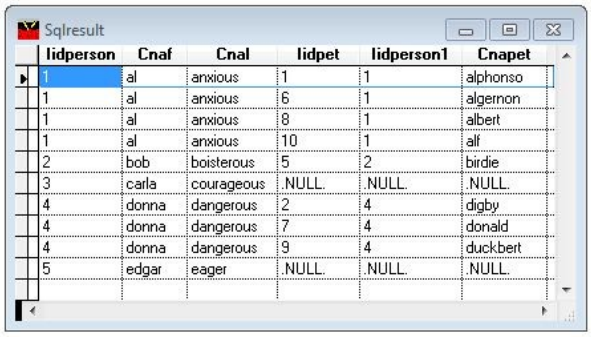

Figure 2. A left join executed in SQLite.

In order to find Pets with or without owners, you can't use a right join:

```
select * from person right join pet on ;
pet.iidperson = person.iidperson
```

If you try to execute this, you'll get no response. Specifically, no 'sqlresult' cursor will be created, nor will you see an error message. Instead, if you need a right join, you'll need to reverse the join syntax, like so:

```
select * from pet left join person on ;
person.iidperson = pet.iidperson
```

Not a big deal, but one that can bite you if you're not aware, and try to use the 'right' keyword.

Full joins aren't recognized either, but you can get the same result through a bit of trickery. Since you're looking for all possible combinations, you'll need to do a pair of left joins, reversing the syntax as shown earlier for the second, and then do a UNION to merge the results:

```
=sqlexec(lih, "select cnaf, cnal, cnapet ;
   from person left join pet ;
   on pet.iidperson = person.iidperson ;
union ;
select cnaf, cnal, cnapet ;
   from pet left join person ;
   on person.iidperson = pet.iidperson")
```

The first left join finds all Persons with or without pets; the second finds all Pets with or without owners. The UNION clause merges the two interim result sets and removes the duplicates. Unfortunately, as shown in **Figure 3**, the UNION trickery causes each field to be converted to a memo field in the cursor, so you may need to do more work, depending on your ultimate goal.

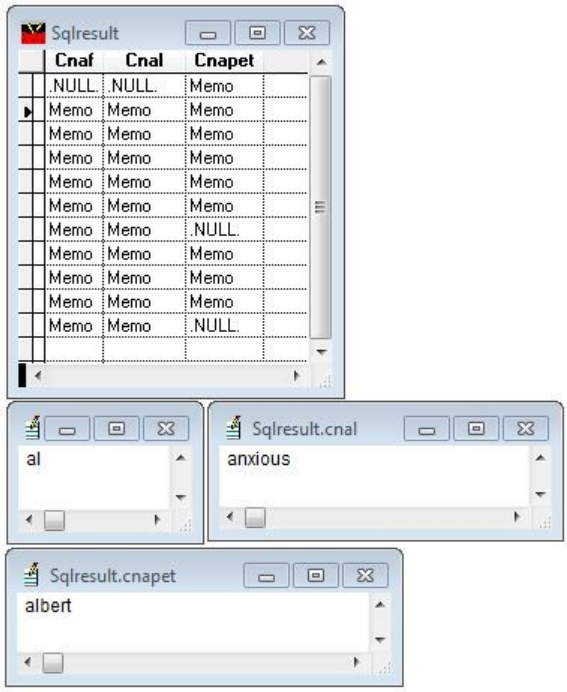

Figure 3. Fooling SQLite into doing a full join.

Commands – Alter Table

The other big difference is that VFP's Alter Table command is much more robust.

In VFP, you can:

- add a new column,
- change an existing column,
- delete (drop) a column,
- rename a column,
- add a candidate index,
- delete (drop) a candidate index,
- add a primary key,
- delete (drop) a primary key,
- add a foreign key,
- delete (drop) a foreign key,

- set a check (validation rule), and
- set an error message if a check fails.

Additionally, columns can:

- be set to be auto-incremented,
- have a default value,
- be prevented from being translated to a different code page.

By contrast, SQLite's Alter Table functionality rather limited. You can:

- rename a table
- add a column

Let's take a look. Suppose we realized we needed another table in our Person/Pet database for all of the paraphernalia associated with taking care of a critter. We add a table called 'gear' to the database. Not sure what fields we need yet, we just define a single primary key:

```
=sqlexec(lih, "create table gear (iidgear
bigint)")
```

And prove that it worked:

```
=sqlexec(lih, "insert into gear (iidgear)
values (1)")
```

Of course, as soon as we hit the Enter key, the realization that we needed a descriptor hit, so we used Alter Table to add the new column:

```
=sqlexec(lih, "alter table gear add column
cdesc char(20)")
```

And it works just fine:

```
=sqlexec(lih, "insert into gear (iidgear,
cdesc) values (2,'leash')")
```

There was already a row in the table without a value in the cdesc field, so let's take care of that as well:

```
=sqlexec(lih, "update gear set cdesc =
'sweater' where iidgear = 1")
```

Time passes, and we decide that we preferred a different name for the 'gear' table. So we'll rename it to 'equipment', like so:

```
=sqlexec(lih, "alter table gear rename to
equipment")
```

Unfortunately, THEN we discover that we can't rename columns in a SQLite table, so the primary key, iidgear, is now a misnomer, so we rename the table back:

```
=sqlexec(lih, "alter table equipment rename to
gear")
```

And all is good.

There are other differences between SQLite's implementation of SQL and the full language, but these two (Join, Alter Table) that most VFP developers need to be concerned with.

Functions

SQLite has 32 functions. Visual FoxPro has more than 32 that start with the letter 'A'. (Seriously! Check it out!)

As a result, there is a great disconcordance between SQLite and VFP's function set. A few in SQLite don't map to any in VFP, and a great many in VFP don't have correlations in SQLite.

More importantly, there are a few that do match up, albeit sometimes inexactly. It's handy to have a chart of how those do match up, and what gotchas exist. See **Table 1**.

Table 1. Mapping of VFP and SQLite functions.

VFP	SQLite	Comments
abs	abs	Returns absolute value of argument.
	coalesce ifnull	Returns first non-NULL argument.
len	length	Returns lengths of argument.
lower	lower	Returns lower case version of string argument.
ltrim	ltrim	Returns string after removing spaces from left side of string. (SQLite version has additional functionality.)
max	max	Returns largest argument. Serves as an aggregator with only a single argument.
min	min	Returns smallest argument. Serves as an aggregator with only a single argument.
	nullif	Returns first argument if the arguments are different and NULL if the arguments are the same.
rand	random	Returns a pseudo random integer. Range differs between VFP and SQLite.
strtran	replace	Returns a string formed by substituting string Z for every occurrence of string Y in string X.

round	round	Returns the first argument rounded to the # of digits defined in second argument.
rtrim	rtrim	Returns string after removing spaces from right side of string. (SQLite version has additional functionality.)
soundex	soundex	Returns the Soundex encoding of the argument. Only available in SQLite if SQLITE_SOUNDEX compile time option is turned on.
substr	substr	Returns a substring of input string X that begins with the Y-th character and which is Z characters long.
trim	trim	Returns string after removing spaces from both left and right sides of string. (SQLite version has additional functionality.)
vartype	typeof	Returns data type of argument. VFP: “C”, “N”, “D”, etc. SQLite: "null", "integer", "real", "text", or "blob".
upper	upper	Returns upper case version of string argument.

In order to practice with these functions, you can send dummy SELECT statements to SQLite, using the function as the only argument, like so:

```
=sqlexec(lih, "select abs(-4)")
```

This will return a cursor with a single row, as shown in **Figure 4**.

Figure 4. Practicing with SQLite functions.

With that technique in hand, let's look at the functions listed in Table 1 in more detail.

abs(X)
Works the same in both VFP and SQLite.

coalesce(X, Y, Z...), ifnull(X, Y)

Returns the first non-NULL argument in the argument list, or NULL if all arguments are NULL. Ifnull() is the same as coalesce() but with exactly two arguments.

len/length(X)
Difference function names for the same functionality – returning the length of the string passed as an argument. The following query:

```
=sqlexec(lih, "select *, length(cdesc) ;
   from gear")
```

produces the result set shown in **Figure 5**.

Figure 5. Results of the length() function.

lower(X)
Works the same in VFP and SQLite.

ltrim(X)
Works the same in VFP and SQLite.

max(X,Y), max(X)
Works the same in VFP and SQLite. When passed multiple arguments, returns the largest:

```
=sqlexec(lih, "select max(1,2,3,7,200,6,-4)")
```

The result is shown in **Figure 6**.

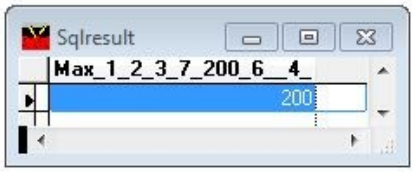

Figure 6. Passing multiple arguments to max().

However, when passed a single argument that is a numeric column in a table, the function returns the value in the row with the largest value:

```
=sqlexec(lih, "select max(iidgear) from gear")
```

The result is shown in **Figure 7**.

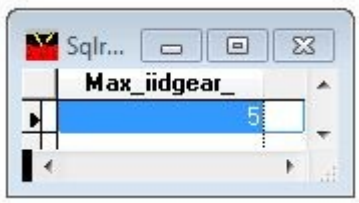

Figure 7. Passing a single argument to max().

min(X, Y), min(X)

The min() function works identically to max() except that it returns the lowest value.

nullif(X, Y)

In the same ballpark as coalesce() and ifnull(), this one returns the first argument if the arguments are different and NULL if the arguments are the same.

rand/random()

Works nearly the same in VFP and SQLite. The range of values that the random number is pulled from varies.

In VFP, the random number is a decimal between 0 and 1. It can contain up to 14 decimal places, so when multiplied by a power of ten in order to produce an integer, random numbers available range from 1 to approximately 400 quadrillion.

In SQLite, the random number ranges from approximately -9223 quadrillion to +9223 quadrillion.

Additionally, the VFP rand() function can be seeded with an argument in order to change the start of the random number sequence generated.

rtrim(X)

Works the same in VFP and SQLite.

strtran/replace(X,Y,Z)

Returns a string formed by substituting string Z for every occurrence of string Y in string X. For example,

```
=sqlexec(lih, "select *,
  replace(cdesc,'a','A') as cdescA from gear")
```

will return 'collAr' from a field that contains 'collar'. This is the same functionality as 'strtran()' in VFP.

round(X,Y)

Works the same in VFP and SQLite.

soundex(X)

Soundex is an algorithm for encoding strings as they are pronounced in English, so that words that are spelled differently but sound the same ('homophones') can be compared and sorted.

Soundex is a function contained in all major SQL databases, including Oracle, Microsoft SQL Server, MySQL, and, of course, SQLite and Visual FoxPro. As such, the implementation is the same in both VFP and SQLite. **Figure 8** shows the results of selecting the soundex value of the cDesc field from the gear table in both SQLite (top) and VFP (bottom.)

```
=sqlexec(lih, "select *,
  soundex(cdesc) from gear")
select *, soundex(cdesc) from gear
```

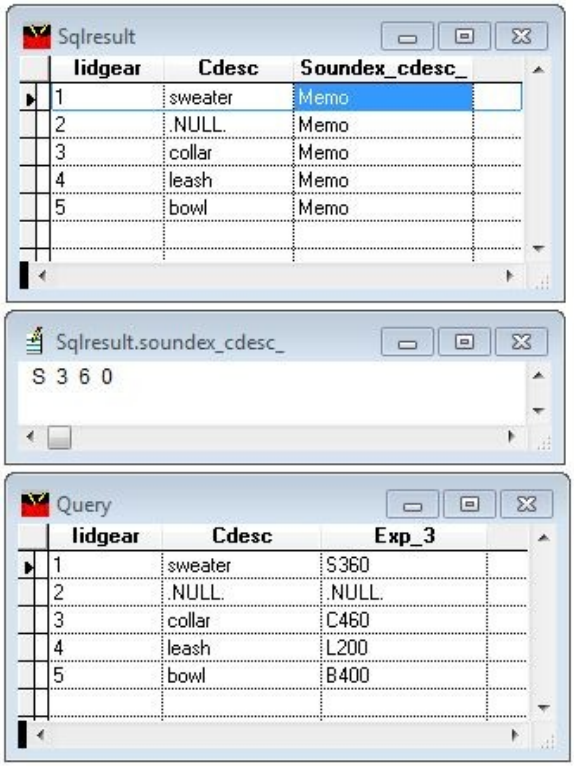

Figure 8. Soundex() results in SQLite and VFP.

Note that Soundex() may not be part of the native SQLite implmentation – it is only available if the SQLite complile-time option, SQLITE_SOUNDEX, is set when the SQLite exectutable is built. Soundex() is part of the SQLite ODBC driver.

substr(X,Y,Z)

Works the same in VFP and SQLite. Returns a string selected from the string X that begins with the Yth character and that is Z characters long. Thus,

```
substr('Visual Fox Pro', 8, 3)
```

returns 'Fox'.

trim(X)

Works the same in VFP and SQLite.

vartype/typeof(X)

Returns the type of data passed in the argument. VFP returns a single upper case character that maps to the data type (for example, "C" for Character or Memo, D for Date.) See the VFP Help file for the complete list. SQLite returns one of the following text strings: "null", "integer", "real", "text" or "blob". The SQLEXEC() command produces the top half of **Figure 9** while the SELECT command produces the bottom half of **Figure 9**.

```
=sqlexec(lih, "select *, ;
   typeof(cdesc) from gear")
select *, vartype('cdesc') from gear
```

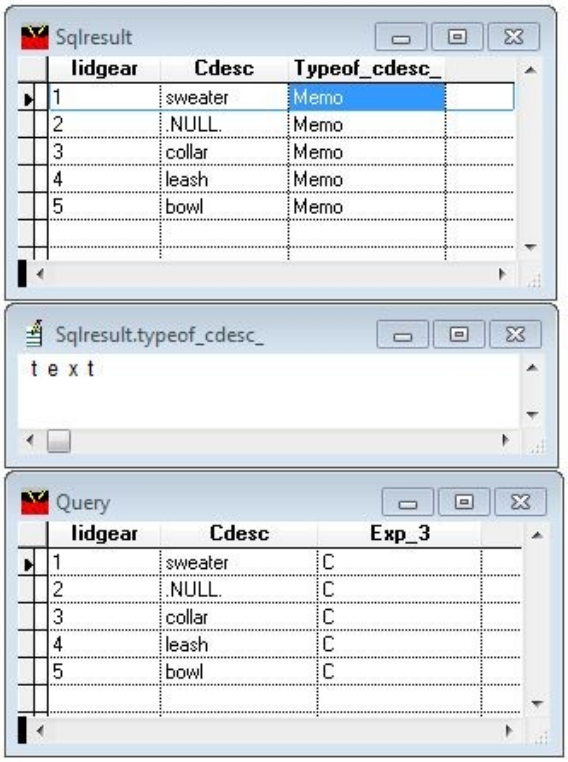

Figure 9. Comparing SQLite's typeof() and VFP's vartype() functions.

upper (X)
Works the same in VFP and SQLite.

Engine differences

Some SQLite functions don't have any direct matches with VFP. See Table 2 for a listing, plus a brief description of what they do.

Table 2. SQLite functions with no corresponding function in VFP..

SQL Function	Description
changes	Returns the number of rows in the database that were affected by the last successful INSERT, DELETE or UPDATE statement.
glob	Returns the number of rows where Y is "like" X.
hex	Returns the upper-case hexidecimal translation of the argument.
last_insert_rowid	Returns the row ID of the last row inserted into the database.
like	Returns patterns of the type Y matching the argument X.
load_extension	Strictly not a function, as it doesn't return a useful value. Simply loads a SQLite extension.
quote	Returns a string that is translated so that it can be included in a SQL statement.
randomblob	Returns a string of N random bytes.
splite_compileoption_get splite_compileoption_used	Wrappers around the get and used functions.
sqplite_source_id	Returns a string that contains the version of the source code used to build the SQLite library.
sqlite_version	Returns a string that contains the version of the SQLite library that is running.
total_changes	Returns the number of changes made by the INSERT, DELETE and UPDATE statements since the database connection was opened.
zeroblob	Returns an empty BLOB N bytes long; used to reserve space for a BLOB.

Engine differences

The key engine difference between VFP and SQLite is that SQLite data types are dynamic. In

VFP, a column is defined as a specific data type (such as character or date), and from then on, only that type of data can be put into that column.

In SQLite, the data type can change within a column; row 1 can contain a date value while the same column in row 2 can contain a numeric ("integer") or a character ("text") value.

Data types are organized into "storage classes". For example, the integer storage class has six data types – depending on the size of the integer (one byte long, two bytes long, all the way up to eight bytes.) The differences between various data types are for all practical purposes transparent to the user.

Implementation differences

The key implementation difference between VFP and SQLite is that the latter is not multi-user. Multiple users can read from the database, but only one can write. This is because when SQLite writes to the database, it locks the entire database.

As a result, unlike other SQL databases that are inherently multi-user (via locks placed on single records), only a single user can access the database at a single time.

Thus, SQLite is best used for three groups of applications:

- with only one user. This user may be an interactive user in the case of a desktop application, or the Web user in the case of a Web application.

- with multiple read-only users but only one user who will write to the database.

- with more than one regular user who is writing to the database, but where the lock of the database during writes doesn't pose a problem, because a second attempted write can be detected and handled.

Source code:

Source code for this article is contained in two files. The first contains the VFP tables for Person, Pet and Gear. The second contains the SQLite database, PersonPet, which contains all three tables.

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com