

# Re: TIP#308 Published: Tcl Database Connectivity (TDBC)

---

*Source:* <http://coding.derkeiler.com/Archive/Tcl/comp.lang.tcl/2007-11/msg00624.html>

---

- *From:* Twylite <[twylite.crypt@xxxxxxxxxx](mailto:twylite.crypt@xxxxxxxxxx)>
  - *Date:* Fri, 16 Nov 2007 07:42:44 -0800 (PST)
- 

Kudos on putting together this spec :)

I'm going to jump right in and start making some comments, because I've recently developed a DBI for in-house use and there are a couple of things about this spec that concern me.

(Aside: Is there somewhere on the Wiki to maintain an ongoing discussion/comments?)

I'm approaching this from the view of a developer using Tcl for desktop applications that manage their data in a database, and server applications that allow storage and retrieval of data by clients but apply some set of business rules. In my experience this covers a significant amount (possibly the majority) of small to medium sized "database-driven applications".

I am not considering the requirements of enterprise applications here (where things like optimising your access to the DB become increasingly important), so I am putting more focus on ease-of-use and elegance in "simple" scenarios, with a view that you should do extra work if you want to develop an enterprise-capable application.

I am also representing a `_commercial_` (not theoretical) software engineering viewpoint: something that facilitates fast development and quality assurance is good.

## Re: TIP#308 Published: Tcl Database Connectivity (TDBC)

To summarise: simple app needs simple DBI, complex app needs complex DBI. If there is a single DBI then it must be simple to use for simple cases.

SO ...

A common pattern in database-driven applications is to display to a user some filtered set of data in tabular form, whether a TkTable on screen or an HTML table report.

Let's assume I have a table of products (name, description, price) and I want to display/report all products under a given price. Here are some example procs:

```
# Get all products under a given price using PostgreSQL.
# Returns the table of products as a list of tuples (lists).
# headingsVar will
# be set to contain an ordered list of columns that indicates the
# contents of
# the tuples.
package require pgintcl
proc pg_show_cheap_products {maxprice headingsVar} {

# Connect to the database
set opts "host = localhost port = 5432 dbname = shop user = app
password = mypass"
set dbhandle [pg_connect -conninfo $opts]

set finally_script
set failed [catch {
# Get the table of products as a list of lists
set stmt {SELECT name, description, price FROM t_products
WHERE price <= $0}
set result [pg_exec $dbhandle $stmt $maxprice]
set finally_script [list pg_result $result -clear]

if { [pg_result $result -status] ne "PGRES_TUPLES_OK" } {
error "sql query failed: [pg_result $result -error]"
}

set resulttable [pg_result $result -lister]

# Get the headings / column names
upvar $headingsVar headings
set headings [pg_result $result -attributes]
} e errorOpts]
```

```
catch $finally_script
pg_disconnect $dbhhandle
if { $failed } {
return -options $errorOpts "couldn't get product list: $e"
}

return $resulttable
}
```

Some things to note about this:

- 1) In the absence of a Tcl try/finally I have used a workaround to reduce if/then blocks but ensure clean-up.
- 2) I have to check the status of the result. It is likely that if Tcl had a try/catch/finally syntax that I could avoid this extra check.
- 3) The pg interface uses ordered parameters (\$0, \$1, ...) to the SQL statement, which requires some care from the developer.
- 4) pgintcl allows me to retrieve the resultset as a list of lists – exceptionally convenient for this sort of query.

```
# Get all products under a given price using SQLite
# Returns the table of products as a list of tuples (lists).
headingsVar will
# be set to contain an ordered list of columns that indicates the
contents of
# the tuples.
package require sqlite3
proc sqlite_show_cheap_products {maxprice headingsVar} {

# Connect to the database
set dbfile "shop.db"
sqlite3 dbhandle $dbFile

set failed [catch {
# Get the table of products as a list of lists
set stmt {SELECT name, description, price FROM t_products
WHERE price <= $maxprice}
set resulttable {}
dbhandle eval $stmt dbrow {
set row {}
set _headings $dbrow(*)
foreach col $dbrow(*) {
lappend row $dbrow($col)
}
lappend resulttable $row
}
}
```

```
# Get the headings / column names
upvar $headingsVar headings
set headings $_headings
} e errorOpts]

dbhandle close
if { $failed } {
return -options $errorOpts "couldn't get product list: $e"
}

return $resulttable
}
```

Some things to note about this:

1) The structure of the sqlite interface doesn't require a finally handler.

This is convenient, but not necessarily a good thing.

2) The interface returns the resultset in one of several ways:

2.a) in rows, as used here, where I can retrieve the ordered list of columns

that are returned by the SELECT and build a list-of-lists using a nested loop.

This is rather inefficient (in terms of performance and source code elegance)

as the DB engine puts everything into an array and I must then pull it out

again, and must also check or set the ordered list of headings from within

a loop.

2.b) as a flat list (not used here) where I must know the ordered list of

columns to be returned by SELECT as I cannot query it.

```
# Get all products under a given price using TDBC
# Returns the table of products as a list of tuples (lists).
headingsVar will
# be set to contain an ordered list of columns that indicates the
contents of
# the tuples.
package require MyTdbcCompliantDatabase
proc tdbc_show_cheap_products {maxprice headingsVar} {

# Connect to the database
set opts [dict create -host localhost -port 5432 -dbname shop -user
app \
-password mypass]
MyTdbcCompliantDatabase dbhandle {*} $opts

set finally_scripts [list dbhandle close]
```

```
set failed [catch {
# Get the table of products as a list of lists

# First we prepare the statement
set stmt {SELECT name, description, price FROM t_products
WHERE price <= :maxprice}
set stmthandle [dbhandle prepare $stmt]
lappend finally_scripts [list $stmthandle close]

# Then we execute the statement (using this stack frame for
variable
# substitution)
set resultset [$stmthandle execute]
lappend finally_scripts [list $resultset close]

# Then we need to get the ordered list of column names
set colresultset [$resultset columns]
lappend finally_scripts [list $colresultset close]

set _headings {}
while { [$colresultset nextrow row] != 0 } {
lappend headings [dict get $row "name"]
}

# Iterate over the rows (of the main resultset) and build into a
correctly
# ordered list of lists
set resulttable {}
while { [$resultset nextrow row] != 0 } {
set outrow {}
foreach col $_headings {
if { [dict exists $row $col] } {
lappend outrow [dict get $row $col]
} else {
lappend outrow {} ;# NULL value substitute – I don't care
about them
}
}
lappend resulttable $outrow
}

# Get the headings / column names
upvar $headingsVar headings
set headings $_headings
} e errorOpts]

foreach script $finally_scripts {
catch $script
}
if { $failed } {
return –options $errorOpts "couldn't get product list: $e"
}
```

```
}  
  
return $resulttable  
}
```

Some things to note about this:

- 1) It's a lot longer and more complex to do a simple query. This is IMHO Not Good.
- 2) I have had to use 4 finally scripts, the equivalent of 4 nested try/catch/finally statements in order to handle error conditions.
- 3) As with SQLite, the statement execution does something non-obvious when it substitutes variables in my stack frame into the SQL statement. This is likely to be a common source of errors.
- 4) By returning the columns as a resultset I have to do extra work to get them into a usable list. All other DBIs I have encountered make an ordered list of columns in the resultset straightforwardly available.
- 5) By using dicts to return the rows I have to:
  - 5.a) Do a bunch of extra work to build an ordered list of tuples (lists). Since most DBMS return row data as an array (in the C sense, or a list in the Tcl sense) this means that the TDBC component is doing a bunch of work to put the row into a dict, which I must then do a bunch of work to undo. That's a massive performance hit PER ROW.
  - 5.b) Handle NULLs. This will be another common source of errors, as developers assume that "SELECT a, b, c FROM t\_xyz" will return a dict with keys a, b and c .... but it won't. So at the first [dict get \$resultrow ...] they may have an error if they encountered a NULL. This is seriously non-obvious.

#### DISCUSSION ...

You may be wondering why I'm doing the extra work to figure out the ordered list of columns returned by the SQL query (after all the SQL statement is right there). It's because the majority of each of those functions is a reusable bit of logic that we could call "extract\_report":

```
proc db_specific_extract_report {stmt headingsVar vars} {
...
}

proc show_cheap_products {maxprice headingsVar} {
set stmt {SELECT name, description, price FROM t_products
WHERE price <= :maxprice}
uplevel 1 [list db_specific_extract_report $stmt $headingsVar \
[dict create maxprice $maxprice] \
]
}
```

Now I could just return a list of dicts and leave it to the calling code to sort out, but I have this rather convenient function "populate\_table" that fills up a TkTable or ttk::treeview from a list of headings and a list of tuples; and another rather convenient function "create\_html\_report" that, well, creates an HTML table in pretty much the same way.

In fact I can create an enormously flexible reporting facility by extending this just a little:

```
proc define_report {title sqlstmt args} {
# each arg is a tuple of entry-title, entry-type and entry-variable
# which describes what a report UI must solicit from the user
# entry-type is a regexp
...
}

define_report "Cheap products" \
{SELECT name, description, price FROM t_products WHERE price
<= :maxprice} \
{"Maximum product price" {\d+} maxprice}
```

# the report UI and execution is left as an exercise to the reader

If I leave column ordering to the calling code, then it must know what columns the query can return, and in what order they should be displayed or reported. So now I need to keep my column names and column order in multiple places in my code and write extra logic to re-order columns.

In the reporting example I'm doing to have to add a list of column names and some reordering code (extra overhead and extra code to maintain) to

give me  
absolutely zero extra functionality – I could have just reordered the  
fields  
in the SQL statement ...

As a \_commercial\_ software engineer having interfaces that make coding  
easy  
(time efficient, obvious, easy to maintain, etc) is really important  
to me.

I'm not assuming that queries are always this simple. In fact we have  
an  
application where the user can select up to three mutually independent  
filters,  
and the user's view of the world is so different from the data  
structure  
(unless we want to manage a heavy amount of redundancy) that two  
different  
options of the same filter can require a reordering of the SQL  
statement  
(rather than just a different value in a WHERE clause). Our option  
was to  
build the SQL statement dynamically, or to implement about 8 separate  
functions, plus one dispatcher to figure out which combination of  
filter options leads to which function. But when it comes to running  
that SQL  
query we hand it to something very similar to `extract_report` (above)  
and  
everything happens magically from there.

#### MY CONCERNS ...

So here is a list of specific concerns that arise from this exercise:

(1) The DBI is complex. The comparative lengths of my example  
functions  
illustrate that clearly.  
If `::tcl::db::execute` had a mechanism to report the ordered list of  
columns  
returned this would make "simple" development a lot easier (comparable  
to  
SQLite). This change alone would also void my comments about error  
handling  
(at least to some degree).

(2) The amount of error handling this DBI requires is significant – 4  
nested  
error handlers. Other DBIs require 1 to 3.  
This level of error handling detracts from my task of writing in  
functionality

makes quality assurance more difficult.

(3) Taking variables from the current stack frame is a recipe for trouble, and makes writing logic like `extract_report` more complex and slower (best case: `[dict with ...]`; worst case: `foreach {name value} { set ... }`)  
I would prefer to see `execute` take a dict; then it is at least clear what variables you are (and are not) providing. Taking variables from the stack without clearly indicating them `_in Tcl code_` (as opposed to another language like SQL) is a little to much of a DWIM for my liking.

(4) Returning the columns as a resultset is a real pain. Rather make the columns available as  
4.a) a list of tuples; or  
4.b) a list of column names, plus a function to get a dict of information about the column (given its name)

(5) The primitive function to retrieve a row needs to retrieve it as a list, IM(NS)HO. This is the natural structure to represent ordered data as retrieved from a DBMS, and avoids a bunch of overhead involved in creating and then parsing out a dict.

Every DBI I have used that uses an array instead of a list (like SQLite) just causes me extra work. It is easy for me to lassign out of a list, hard to lassign multiple values out of a dict (especially if you have to have error handling for NULL cases), and hard to build an ordered list from an array. It is of course hard to build a dict from pairwise ordered lists, so clearly there is a trade-off between the list/dict options.  
Sidenote: I realise that "easy" and "hard" here amount to a difference of 2 or 3 lines of code, but it is (i) extra code to maintain, (ii) extra code obscuring what is actually going on, (iii) a loop that causes a performance hit which – over a 1000 row report – can add up to a user-perceptible difference (and should certainly be a concern for enterprise use!).

I understand the concerns around NULL, but returning this data as a dict is bad for at least two reasons:  
5.a) The performance hit; AND  
5.b) The implementation dangers you face by not having keys in the dict that the developer sanely expects to find there.

Furthermore, in a typical/simple database-driven application environment you either (i) don't expect to deal with NULL or (ii) expect NULL and an empty string to be equivalent, so having to deal with this problem explicitly (as I did in my example) is just added and unnecessary complexity.

Even further to that, the very concept of NULL is not Tclish and Tcl developers don't have to think about "dereferencing NULL"; so introducing a landmine like that is a Bad Idea.

I find SQLite's solution to this problem (nullvalue, see <http://www.sqlite.org/tclsqlite.html#nullvalue>) quite interesting as it covers "the typical cases" but perhaps it isn't a "good enough" solution.

Perhaps an acceptable solution may be a combination of nullvalue and isnull. The returned list has \$nullvalue substituted for null; and implementation that needs to distinguish NULL from not-NULL can check each fields that matches \$nullvalue by calling a "\$resultset isnull \$colidx" function.

A better solution may be to have nullvalue plus nextrow\_dict and nextrow\_list methods. Hmm.

(6) I am concerned by the number of ensembles that a simple query must create (and clean up). This smells like a performance concern, especially for a pure-Tcl implementation for a specific database.

I am also worried that the "set handle [create\_x] ; \$handle" pattern can be a source of errors, is impossible to statically analyze, and is a pain to debug (think 'invalid command name ""' versus 'invalid result handle ... while

executing pg\_result') (and it's about the closest Tclers can get to a NULL dereference ;) ).

I suppose this is really a question of whether the API should be modelled in an OO or procedural (data-passing) style. There are benefits to each.

(7) A SELECT operation is fundamentally different from other commands that can be executed. This is clear for example when you read the spec for "resultset rows", which does not provide a meaningful value in the case of SELECT. How you query a resultset is thus dependant on the operation that was requested, which can make creating generic DB manipulation functions rather fun. Other DBIs have also struggled with this problem.

May I propose a distinction between a "query" function and an "execute" or "alter" function? We have made this distinction in our in-house DBI and it works well for us. I'd be interested to hear opinions from others.

Okay, that's my 2c and 400 lines.  
Regards,  
Twylite