Tcl Database Connectivity

Kevin B. Kenny
GE Global Research
kennykb@research.ge.com

Abstract

Tcl Database Connectivity is a newly-approved interface in the Tcl Core as of release 8.6. It fills the need – expressed over many years – of a uniform interface to Tcl for SQL databases. TDBC is built to exploit the latest features of 8.6, including its object-oriented functionality. It is designed for close integration with Tcl; in particular, it addresses such issues as immunity to SQL injection attacks, type mismatches between Tcl and SQL, and the presence of NULL columns without requiring either special Tcl code to handle them. (It approaches everything with the attitude that it should be more difficult to write insecure code than secure code, and that SQL should have no impact on "everything is a string.") It is the intent that TDBC should be one of Tcl's foundational components, so that database applications in Tcl can have the same ubiquity as Tcl itself.

1 Introduction

Tcl has come under fire for its lack of a uniform interface to SQL databases. Other scripting languages, for instance Perl with its DBI module [Domi99], and Python with the Python Database API [Lemb08] have long had such common portable interfaces. While Tcl has had no shortage of interfaces to SQL databases, they have all been developed individually, and lack a common API. Applications that are developed against one database frequently need substantial modifications to run against another, and federated database applications can be difficult to build.

With the 8.6 release, Tcl adds a new TDBC (Tcl Database Connectivity) library that provides a starting point for addressing this lack. This paper describes the basics of TDBC's design. Section 2 presents some related work on which TDBC builds. Section 3 describes the overall structure of TDBC. Section 4 discusses some of TDBC's limitations, and gives the rationale for leaving out certain features. Section 5 discusses the work that an implementor must do to add a new database to TDBC. Finally, Section 6 presents a vision for the applications that TDBC will enable and lays out some of the future work that will be needed in TDBC.

2 Background

Several attempts have been made in the past to provide portable database interfaces in Tcl. The earliest ones tried to layer atop Microsoft's ODBC (Open Database Connectivity) [ODBC08]. The most enduring of these has been Roy Nurmi's tclodbc package [Nurm04]. This package allows for connection to a wide selection of popular databases. It is somewhat limited (it doesn't handle Unicode text effectively, installation is problematic on non-Windows systems, and ODBC performance leaves a lot to be desired). Nevertheless, it has been widely used to implement database applications.

The tclodbc package also pioneered the object-based syntax for database access, in which both connections and statements are represented as Tcl commands. To a large extent, tclodbc's object structure inspires the overall structure of TDBC.

Michael Cleverly's nstcl package [Clev04] is another design that can be said to inspire the design of TDBC. It consists of Tcl wrappers around the native API's of seven popular databases, plus ODBC, so that a program that deals with any of those interfaces faces a uniform API. It pioneered the idea of adapting database APIs with pure-Tcl drivers. Alas, it has languished in comparative obscurity, perhaps because of its association with the ill-fated Ars Digita.

3 Design

Database interfaces are one area that fit well with the ideas of object-based design. There are multiple implementations of each basic interface: "database," "statement," "result set," and so on; each implementation needs to present a uniform interface to the calling program. Having a family of base classes to represent these interfaces, with specific implementations inheriting from those base classes, is a natural approach to such a design. Three base classes: `tdbc::connection`, `tdbc::statement`, and `tdbc::resultSet`, present essentially the entire script-facing interface of TDBC.

3.1 The 'connection' class

The `tdbc::connection` class represents a connection to a database. Its most important method is `prepare`, which prepares to execute a SQL statement against the given database:

```
set s [$connection prepare {
    select surname, given_name,
           phone_number
    from directory
    where surname = :name_sought
}]
```

The idea of preparing a statement may be foreign to the users of some database interfaces. TDBC requires that all statements be prepared.[1] The reason for this requirement is that it is the easiest way to guard against SQL injection attacks. Interfaces that do not support prepared statements must sanitize database inputs to guard against such attacks, and are constantly at risk. In particular, MySQL has been vulnerable to these attacks, with its reliance on procedures like `mysql_real_escape_string` to sanitize its inputs. Exploitable vulnerabilities have been reported as recently as 2006 [MySq06].

TDBC's approach is to avoid having the Tcl code sanitize inputs. Rather, statements that accept inputs from Tcl are parametrized. A name in a statement beginning with a colon (*e.g.,* `name_sought` in the example above) is a name of a Tcl variable that is expected to appear in the calling scope at the time the statement is executed. The database driver will substitute its value in as a SQL value, safely.

The colon appears instead of the dollar sign both because names containing dollar signs are valid identifiers in many SQL systems, and because it provides a convenient approach to protecting certain fields from substitution while substituting others:

```
set s [$connection prepare "
    select $valuecolumn from $table
    where $keycolumn = :keyvalue
"]
```

The return value from the `prepare` method is an object that follows the interface of `tdbc::statement`, and allows the Tcl code to act on the database.

Next in importance to the prepare method is the `transaction` method, used to frame an atomic action. It takes the form:

```
$connection transaction {script}
```

[1] TDBC does have convenience procedures that prepare a statement and execute it immediately; all statements are still prepared behind the scenes.

The *script* is a Tcl script to execute atomically. If the script terminates normally (defined as an 'ok' return, or a 'break', 'continue' or 'return' within the script), the transaction is committed; otherwise it is rolled back and hence has no effect on the database. Code that cannot be structured with this style of transaction may call the `begintransaction`, `commit`, and `rollback` methods directly at some expense in complexity.

In addition, a connection object implements service methods that enumerate tables in the database, columns in a table, and open statements and result sets. There are also service procedures to be discussed below, to simplify the coding of certain common cases.

3.2 The 'statement' class

The `tdbc::statement` object represents a SQL statement that may accept parameters. Its fundamental API is the `execute` method:

```
set r [$statement execute]
```

or

```
set r [$statement execute $dict]
```

This method bundles the parameters of a statement (the variables that appeared as names preceded by colons), by substituting them either from Tcl variables in the caller's scope, or from values in the dictionary provided. The return value is an object implementing the interface of `tdbc::resultset`.

In addition, statements support service methods to enumerate the result sets that they have produced and the parameters that the statements expect. Finally, statements also support a `paramtype` method:

```
$statement paramtype name_sought \
    varchar 40 in
```

This method confronts an ugly reality that several databases' interfaces require that parameters in the calling program have types that match the types of columns in the database – but provide no means of introspecting what the expected types are. Code that expects to operate against such databases must therefore declare parameter types explicitly. Fortunately, most databases will accept character strings in place of parameters of any type, so in most cases the `paramtype` method is needed only for performance.

3.3 The 'result set' class

The two key methods for result sets are `nextdict` and `nextlist`, either of which stores a row in a variable provided by the caller and returns 1 if successful or 0 if no rows remain in the result set. For the `nextlist` method, the row is returned as a list of the columns' values, in the order in which they appeared in a `SELECT` statement. `NULL` values are returned as empty strings.

Using the empty string to represent a `NULL` value is problematic, because a SQL `NULL` represents an unknown or unspecified variable, not any string. (The difference between the empty string and `NULL` is analogous to the difference between the statements, "Joe Smith has no middle name," and "I don't know what Joe Smith's middle name (if he has one) is.") In many applications, for instance report generators, the distinction is irrelevant, since an empty string would be printed for Joe's middle name in either case. Some programs, nevertheless, need to be aware of the difference. Since all Tcl values are strings, there is no value that can be used to represent a `NULL`. This limitation has led to at least one proposal [Harr04] to change Tcl's semantics to allow `NULL` as a special value. Unfortunately for `NULL`'s proponents, the TCT's judgment was that the effects of the semantic change were too com-

plex and subtle to contemplate in an 8.x release.

Fortunately for users of NULL values, Tcl provides several ways to represent that a value does not exist. TDBC chooses to represent NULL's with missing keys in dictionaries. The nextdict method returns the next row of a result set in this form. The keys of the returned dictionary are column names, and the values are column values. NULL values are omitted from the dictionary.

This approach has a single remaining drawback over a direct representation of NULL, in that the columns no longer appear in the order that they appeared in the original query (simply because the NULL's are omitted; dictionaries preserve the order of keys). TDBC solves this problem by adding a columns method to the result set object that returns the list of columns in the original order. The complexity of using a second method is somewhat offset by the fact that few applications will not know what columns they expect. Most of these are applications that accept the input of *ad hoc* SQL statements from a database administrator and execute them without further analysis.

In addition to the nextdict, nextlist, and columns methods, result set objects provide a rowcount method to indicate the number of rows affected by an INSERT, DELETE or UPDATE statement (SELECT statements may not have row counts available until all results have been processed).

3.4 Putting it all together: our first application

We now have enough pieces to write a simple application that uses TDBC. Figure 1 shows such an application. It behaves as you might expect:

```
% tclsh86 phbook.tcl Flintstone
Flintstone, Fred   555-3733
Flintstone, Wilma   555-9456
```

```
# open the database and prepare a statement
package require tdbc::sqlite3
tdbc::sqlite3::connection create db \
    [file join [file dirname [info script]] phonebook.db]
set s [db prepare {
    SELECT surname, given_name, phone_number
    FROM directory
    WHERE surname = :name
}]

# apply the statement to each name on the command line and print results
foreach name $argv {
    set r [$s execute]
    while {[$r nextlist row]} {
      lassign $row surname given_name phone_number
      puts "$surname, $given_name   $phone_number"
    }
    $r close
}
# clean up resources
$s close
db close
```

Figure 1. A sample application using TDBC.

```
package require tdbc::sqlite3

tdbc::sqlite3::connection create db \
    [file join [file dirname [info script]] phonebook.db]

foreach name $argv {
    db foreach -as lists row {
      SELECT surname, given_name, phone_number
      FROM directory
      WHERE surname = :name
    } {
      lassign $row surname given_name phone_number
      puts "$surname, $given_name   $phone_number"
    }
db close
```

Figure 2. The sample application, revised to use the 'foreach' method

3.5 Convenience methods: 'allrows' and 'foreach'

There are drawbacks to the way that our first example application is structured. Chief among these is that it would be quite cumbersome to rewrite it as a procedure that would query the directory. In order to avoid leaking result set or statement objects, the code would have to be wrapped in several layers of `catch` commands, each of which would destroy an object and rethrow any error. Even having to create and destroy these objects is perhaps more code than we would like to deal with.

For this reason, each of the database, statement, and result set objects supports meth- `allrows` method returns the rows resulting from a query as a Tcl list, and the `foreach` method applies a script to the rows resulting from a query. Each method accepts additional arguments as appropriate, and properly manages the lifetime of any statement and result set objects that have to be created to fulfill the request. With the `foreach` method, the example application can be simplified as shown in Figure 2. Note that the statement and result set are both maintained implicitly.

This simplification might give rise to the question, "why isn't the simplified way the only way?" There are a couple of reasons. First, it is possible to have result sets that are large enough that it is inconvenient to hold them in memory at once. (Tables with many millions of rows are not unheard of.) Second, and more important, is that Tcl is commonly used as the glue that ties together heterogeneous systems. It is easy to foresee an application that (for example) posts updates from a lightweight local database (running, say, MySQL or SQLite) to an enterprise database running Oracle, Sybase, or DB2, and needs to join tables in both. This join would have to be done on the client side. It is easy to foresee that such a join would require external control of the iteration.[2] Finally, we may discover that using a bytecoded loop like Tcl's own `while` is needed to get the best possible performance.

4 What doesn't TDBC do?

Several features that were contemplated during TDBC's development (and were widely requested) are left out of TDBC. Since what the designers chose to omit is often as telling as what they chose to include, it is perhaps worthwhile to mention them here.

[2] An alternative to external iteration might be to package the database queries inside of coroutines [SOFE08]. The design of TDBC predates a widely-available reference implementation of Tcl coroutines.

One feature that was considered and rejected was batched statements (variously called also "bulk uploading"). This technique allows a single call to the database to execute an INSERT or UPDATE statement many times, with different data for each statement. It gives a performance advantage to code that uses it to transfer large volumes of data.

Bulk uploading was rejected because not all databases support it (although it would be possible for drivers to simulate its behavior by executing a statement repeatedly). The databases that do support it impose different restrictions on it (for instance, whether a batch of changes can comprise only multiple executions of the same statement or multiple statements). Finally, mandating support for batched statements imposes a complexity requirement on drivers. Since TDBC's goal is to be ported easily to all popular databases, it was decided that simplicity trumps performance in this particular case. If there is sufficient demand (and sufficient support from database driver writers!), this decision can, of course, be revisited.

Another feature that was considered and rejected was asynchronous queries–launching a query against a database and executing a callback as data arrives. Again, the factor governing rejection was that not all the databases support it. It would be possible for a driver to behave as if it were supported (as(assuming a multithreaded build) by running the query in a separate thread and delivering the results by queuing the callback in the requesting thread. But if the implementation is done this way, it saves the client little programming effort to have asynchronous queries as opposed to managing its own threads. In fact, the current documentation for ODBC [ODBC08] explicitly deprecates the use of asynchronous statements in favor of threading.

A final feature that was left out was "reference cursors"—explicit cursors returned from stored procedures. The omission of this feature (which would, of course, be meaningful only on databases that support the concept) was due primarily to the pragmatic consideration of having a reasonably complete implementation available for Tcl 8.6. If a coherent specification can be devised, it would be possible to add such a feature.

5 Developing TDBC drivers

A key design goal to TDBC was to make it relatively easy to incorporate new databases. In particular, it is critical to be able to prototype a new database interface in Tcl (assuming that an existing Tcl interface, possibly with different syntax and semantics, is available) and then proceed to the C implementation. As a test of the concept, a pure-Tcl driver for SQLite has been developed and packaged with TDBC. The Tcl code for the driver itself is a little less than 400 lines of code (including copious comments), something that a competent programmer can do very quickly. The test suite is much longer, something over 2500 lines of code, but the test suite is also substantially portable among implementations. Fewer than 200 lines differ, for example, between the SQLite3 and ODBC test suites.

What is needed for any database driver is three new classes: one representing database connections, one representing connections, and one representing result sets.

5.1 The connection class

The connection class must inherit from `tdbc::connection`, and implement the following methods:

- A constructor. The constructor, as well as doing whatever is needed to open the connection, should set the instance variable `statementClass` to the name of the statement class (see below).

- Methods, named `tables` and `columns`, that introspect on the tables in a database and the columns in a table.
- A `preparecall` method that prepares calls to stored procedures (if applicable).
- Methods called `begintransaction`, `commit`, and `rollback` that perform the corresponding operations on the underlying database.

The constructor should accept whatever arguments are needed to specify the database to open.

5.2 The statement class

The statement class must have a name that matches the one supplied in the constructor of the connection object. It must support the methods:

- A constructor, which accepts the name of the connection and the SQL statement as parameters. The constructor must set an instance variable, `resultSetClass`, to the name of the class that will represent result sets (see below). It must also prepare the statement at least to the point where its parameters can be identified. To aid in this task, a command, `tdbc::tokenize`, is exported from the `tdbc` package. Given the SQL statement, this command returns a list of tokens, identifying the ones requiring substitution.
- A `params` method that returns the list of parameter descriptions for the prepared statement.
- A `paramtype` method that declares the type of a parameter. (It is permissible for this method to do nothing if character strings are appropriate for all parameters presented to the underlying database.)

5.3 The result set class

The result set class, whose name must match the name set by the statement class constructor, should inherit from `tdbc::resultset`. It must implement the following methods:

- A constructor. The constructor accepts as parameters the statement being executed and the parameters presented to the statement's `execute` method. It is responsible for launching the query prepared by the statement's constructor, with parameters substituted appropriately from variables in the caller's context. The result set constructor is by far the most complex method in any of the drivers yet attempted, since it is where all the details of parameter transmission and database control are buried.
- A `columns` method that returns the list of columns in the result set.
- The `nextlist` and `nextdict` methods that return results as described in Section 3.3.
- A rowcount method that returns the count of rows affected by an `INSERT`, `UPDATE` or `DELETE` statement.

5.4 Drivers in C

Obviously, a pure Tcl implementation is an option only if an existing Tcl driver for a database exists. One other alternative for getting something running quickly is to use an existing ODBC driver for the database, and the TDBC-ODBC bridge. The TDBC-ODBC bridge can also serve as a reference for how to implement other TDBC drivers in C. The same methods are required as for drivers in Tcl. The class definitions are set up in Tcl code, and other methods are added to the classes from C initialization.

Proper attention to object lifetimes can make developing the C code much easier. It turns out that TDBC, together with the object-oriented support in Tcl, make the lifetime management fairly simple. A recommended practice is to:

- Have reference counts for all the C structures that represent database ob-

jects. Manage these reference counts in much the same way that Tcl_Obj reference counts are managed: increment them when creating new pointers to them, and decrement them when the pointers go out of scope. When the reference count reaches zero, the corresponding structure should be cleaned up.

- Each Tcl object should have attached metadata that designates its C structure. The deletion callback should decrement the reference count (and possibly clean up).
- Each C structure should also carry a counted reference to the structure that owns it. (A result set will designate the owning statement, and a statement will designate the owning connection.) This reference, of course, should be removed when the structure is cleaned up.
- If these conventions are followed, deleting an object at any level in the hierarchy will do the right thing, pretty much automatically. Tcl destructors will execute from top to bottom, and the reference counts on the corresponding C structures will go to zero at the bottom level first, causing them to shut down in an orderly fashion from bottom to top.

Readers who need more detail than this about C implementation are advised to consult the source code for the TDBC-ODBC bridge. A realistic estimate that a full C implementation for a new database will require about 300 lines of Tcl and 1000-4000 lines of C, depending on the complexity of the API. This scale is comparable with what is present today in packages like mysqltcl, oratcl, sybtcl, or tclodbc.

6 Future directions

The most pressing need for TDBC to be widely supported is that it must connect to all the popular databases. While bridging to ODBC is a useful start, it is something of a crutch. It would be better to have native drivers to the databases' C APIs or wire protocols. The author is trying to recruit driver writers to broaden support for TDBC.

TDBC is, of course, only one piece of a much larger puzzle. With a portable database interface and the power of Tk, one can imagine that Tcl/Tk could host a variety of portable database administration tools (think of PhpMyAdmin [DeLi04] or TOAD [Scal03] without their known limitations in dealing with multiple database engines), GUI query designers (akin to Microsoft Access), graphical query analyzers (like revj [Toth08]), and so on. These could work in federated systems with heterogeneous databases, and could be useful for data mining, synchronization, and enterprise integration.

Acknowledgments

The syntax for substituting values in SQL queries was pioneered in the SQLite extension. D. Richard Hipp, the extension's author, graciously contributed the SQL tokenizer that TDBC uses to parse this syntax.

Works Cited

[Clev04] Cleverly, Michael. "nstcl: Bringing the best of AOLserver and OpenACS to tclsh and wish." Proc. 11th Ann. Tcl/Tk Conf. New Orleans, October 2004. http://www.tcl.tk/community/tcl2004/Tcl2003papers/cleverly.htm

[Domi99] Dominus, Mark-Jason. "A Short Guide to DBI: Perl's Database Interface Module" `perl.com`. Sebastopol, Calif: O'Reilly, October, 1999. http://www.perl.com/pub/a/1999/10/DBI.html

[DeLi04] DeLisle, Marc. *Mastering phpMyAdmin 2.11 for Effective MySQL Management*. Birmingham, England: Packt Publishing Ltd, 2004. ISBN 1-847194-18-3.

[Harr04] Harris, John H. "Null Handling." Tcl Improvement Proposal #185 (8 April 2004). http://tip.tcl.tk/185

[Hipp08] Hipp, D. Richard. "The Tcl interface to the SQLite library." http://www.sqlite.org/tclsqlite.html, last updated 16 July 2008.

[Lemb08] Lemburg, Marc-André. "Python Database API Specification v2.0." Python Enhancement Proposal #249. http://www.python.org/dev/peps/pep-0249/

[MySq06] MySQL AB. *MySQL Reference Manual*, section E.1.7 (24 May 2006). http://dev.mysql.com/doc/refman/5.0/en/news-5-0-22.html

[Nurm04] Nurmi, Roy. "Tclodbc v. 2.3 Reference". Last updated 20 March 2004; available as part of the tclodbc distribution at https://sourceforge.net/projects/tclodbc/

[ODBC08] *ODBC Programmers' Reference*. Redmond, Wash.: Microsoft Corporation, 2008. http://msdn.microsoft.com/en-us/library/ms714177(VS.85).aspx

[Scal03] Scalzo, Bert and Dan Hotka. TOAD Handbook. Indianapolis: Sams Publishing, 2003. ISBN 0672324865.

[Sofe08] Sofer, Miguel and Neil Madden. "Coroutines." Tcl Improvement Proposal #328 (7 September 2008). http://tip.tcl.tk/328

[Toth08] Toth, Alexandru. "Reverse Snowflake Joins." http://revj.sourceforge.net/, visited 29 September 2008.