23

Database Connectivity with TDBC

The *Tcl Database Connectivity* (TDBC) extension provides a Tcl API for accessing SQL databases. Because this API is independent of the underlying database system, most[1] of the code accessing databases in an application can run unchanged between different database implementations.

A bit of history

TDBC 1.0, authored by Kevin B. Kenny, was released with Tcl 8.6. Prior to its release, the lack of a standard Tcl database access API lead to a number of extensions with different interfaces including

- *tclodbc* for connecting to databases using ODBC
- *DIO* which is part of Apache Rivet
- *nstcl* and *nsdbi*, both derived from AOL Server web server
- Various database-specific extensions like *oratcl* for Oracle and *pgtcl* for PostgreSQL.

With the advent of TDBC, applications can now rely on a standard interface to databases from Tcl.

TDBC is broken up into two layers:

- The upper layer, which is what we cover here, is the interface used by applications to access the database.
- The lower layer consists of different drivers that implement access to specific databases. At the time of writing, the TDBC distribution includes drivers for MySQL, PostgreSQL, Sqlite3 and any database accessible via an ODBC interface. The TDBC documentation also defines an interface that allows new drivers to be written for other database implementations.

Due to space limitations, this book only covers the first of these — the application interface to databases.

23.1. Installing TDBC

The TDBC extension is included in the standard Tcl 8.6 source distributions as well as all binary distributions. However, individual database driver components may have to be installed separately. Most binary distributions of Tcl include drivers for SQLite and Windows has native support for ODBC. In other cases, the drivers, usually implemented as shared libraries, are available from the database vendor.

23.2. Loading TDBC

TDBC is loaded with the standard `package require` Tcl command. The specific package to be loaded depends on which database driver is desired. The packages included in the core distribution are shown in Table 23.1.

[1] Some code will be necessarily specific to databases because of differing capabilities and quirks in database implementations.

Table 23.1. Core TDBC driver packages

Package	Database
tdbc::sqlite3	Sqlite3
tdbc::postgres	PostgreSQL
tdbc::mysql	MySQL
tdbc::odbc	Any database that is provides an ODBC interface

In addition, open source TDBC drivers are also available such as ones for JDBC[2], CUBRID[3] and MonetDB[4].

Naturally, when dealing with multiple database implementations in an application, more than one of these packages may be loaded if desired.

For our code examples, we will make use of the Sqlite3 database and thus load the corresponding package

```
% package require tdbc::sqlite3
→ 1.0.4
```

23.3. Concepts

TDBC follows the same general pattern as other database access API's and involves the following steps:

1. First a *connection*[5] has to be established to the database (and database provider) of interest. In addition to identifying the database this may also include authorization credentials and other options. All subsequent interactions for the database are done through this connection object and its surrogates.
2. Next a SQL *statement* is *prepared* using the connection object and then executed with the results returned as a *result set*.
3. The *result set* is iterated over to operate on the returned data.
4. The statement and result sets are freed so as to not use up resources.
5. Steps 2-4 are repeated as needed.
6. When all done, the database connection is closed.

TDBC encapsulates all the above abstractions as the TclOO classes `tdbc::connection`, `tdbc::statement` and `tdbc::resultset`.

23.4. Connecting to databases

A database connection is represented by an object of the appropriate `tdbc::`*`DRIVER`*`::connection` class. To connect to a database, you create an object of this class specifying the database of interest. The manner in which the database is identified depends on the database driver in use.

23.4.1. Connecting to SQLite: `tdbc::sqlite3::connection`

The `tdbc::sqlite3` package must be loaded to access SQLite databases.

```
package require tdbc::sqlite3
```

A connection to a SQLite database is created by passing the path to the sqlite3 database file to the `tdbc::sqlite3::connection` command. For example, to open a SQLite database in the current directory,

[2] https://github.com/ray2501/TDBCJDBC
[3] https://github.com/ray2501/tclcubrid
[4] https://sites.google.com/site/tclmonetdb/
[5] Not to be confused with a *network* connection.

```
tdbc::sqlite3::connection create db my-database.sqlite3
```

This will create the object db representing the database connection to the `my-database.sqlite3` database. As we see in a moment, we can operate on the database by invoking methods on this object.

For our sample code, we will create and make use of an in-memory SQLite database. The special token `:memory:` results in the database being created purely in memory with no disk store. It will be erased once closed which suffices for the sample code purposes.

```
% set dbconn [tdbc::sqlite3::connection new :memory:]
→ ::oo::Obj601
```

Note here we use the TclOO method new, as opposed to `create`, which generates the connection object name for us.

23.4.2. Connecting via ODBC: `tdbc::odbc::connection`

Open Database Connectivity (ODBC) is an industry standard API for accessing databases. Database implementations that support this interface can be accessed through the `tdbc::odbc` package.

```
package require tdbc::odbc
```

Windows comes with ODBC support built-in but to use ODBC on Unix or Linux, you may need to install an ODBC package such as unixODBC[6] or iODBC[7].

To connect to an ODBC database, pass its connection string to the `tdbc::odbc::connection` command. This takes the form of a series of attribute name and value pairs that specify the connection characteristics. For example, (assuming we are on Windows)

```
tdbc::odbc::connection create db "Driver={SQL
 Server};Server=localhost;Trusted_Connection=Yes;Database=YourDatabaseName;"
```

will return a connection object for the SQL Server database `YourDatabaseName` on the local system. Notice additional attributes can be specified in the connection string. For instance, the `Trusted_Connection=Yes` attribute and value specify that the credentials of the Windows account under which the application is running are to be used for authorization.

Depending on the system and the database, you can define a *data source name* (DSN) that stores the data used to construct a connection string. Then you can simply specify the DSN to connect to the database. So the above call would then become

```
tdbc::odbc::connection create db "DSN=YourDSN"
```

You can use the ODBC utilities in TDBC to define DSN's if the underlying ODBC implementation supports the ODBC Installer API. Alternatively, on Windows systems, you can use the ODBC applet in the Control Panel to define and configure DSN's. On Unix/Linux, unixODBC and iODBC both provide GUI and command line means of defining DSN's.

[6] http://www.unixodbc.org
[7] http://www.iodbc.org

Connection strings are ODBC driver specific and sometimes difficult to get right. The Connection Strings Reference web site is a useful resource to understand and construct these.

In addition to the common options (see Table 23.4) supported by all TDBC drivers, some ODBC environments support the `-parent` option which results in a prompt for user credentials if required. See the tdbc::odbc[8] reference for details on its use.

As described in Section 23.11, the package also implements some utility commands for interacting with the system ODBC manager.

23.4.3. Connecting to MySQL: `tdbc::mysql::connection`

Connecting to MySQL requires the `tdbc::mysql` package.

```
package require tdbc::mysql
```

The package differs from SQLite and ODBC in that the `tdbc::mysql::connection` command for establishing a database connection identifies the database to be connected through a set of options as opposed to a file name or connection string. These options are shown in Table 23.2.

Table 23.2. MySQL connection options

Option	Description
`-host` *`HOSTNAME`*	The name of the system on which the database server is running. Defaults to the local system.
`-port` *`PORTNUMBER`*	The TCP/IP port number on which the server is listening for connections.
`-socket` *`PATH`*	Connects to the Unix socket or named pipe specified by *`PATH`*.
`-user` *`USERNAME`*	The name of the user name to be used to access the database. Defaults to the current user id of the process.
`-password` *`PASSWORD`*	The password to be presented to the server. By default no password is presented.
`-database` *`DATABASE`*	The name of the database to default to if no database is specified in a query. Defaults to the default database for the user specified by the `-user` option.
`-db` *`DATABASE`*	Same as the `-database` option.
`-interactive` *`BOOL`*	If specified as `true`, sets the default timeout to be that for an interactive user; otherwise, the default timeout is set as for batch users.
`-ssl_ca` *`PATH`*	Specifies the **file** containing the list of trusted certificate authorities permitted for an SSL connection.
`-ssl_capath` *`PATH`*	Specifies the **directory** containing the files containing certificates for trusted authorities permitted for an SSL connection.
`-ssl_cert` *`PATH`*	Specifies the file containing the client certificate.
`-ssl_key` *`PATH`*	Specifies the file containing the client private key.
`-ssl_cipher` *`CIPHERLIST`*	Specifies the permissible ciphers to use for an SSL connection. *`CIPHERLIST`* is a list of cipher names separated by colons.

[8] http://www.tcl.tk/man/tcl8.6/TdbcodbcCmd/tdbc_odbc.htm

23.4.4. Connecting to PostgreSQL: `tdbc::postgres::connection`

Connecting to a PostgreSQL database is more or less identical to what was described previously for MySQL. The `tdbc::postgres` package is loaded

```
package require tdbc::postgres
```

and the connection is made via `tdbc::postgres::connection` using the options shown in Table 23.3.

Table 23.3. PostgreSQL connection options

Option	Description
`-host` *`HOSTNAME`*	The name of the system on which the database server is running.
`-hostaddr` *`IPADDR`*	The IP address of the system on which the database server is running. Takes precedence over the `-host` option.
`-port` *`PORTNUMBER`*	The TCP/IP port number on which the server is listening for connections.
`-user` *`USERNAME`*	The name of the user name to be used to access the database. Defaults to the current user id of the process.
`-password` *`PASSWORD`*	The password to be presented to the server. By default no password is presented.
`-pw` *`PASSWORD`*	Same as the `-password` option.
`-database` *`DATABASE`*	The name of the database to default to if no database is specified in a query. Defaults to the default database for the user specified by the `-user` option.
`-db` *`DATABASE`*	Same as the `-database` option.
`-options` *`OPTS`*	Specifies additional command line options to send to the server.
`-sslmode disable|allow|prefer|require`	A value of `disable` mandates a non-SSL connection to the server, `require` mandates an SSL connection, `allow` prioritizes a non-SSL over SSL and `prefer` (default) prioritizes SSL over non-SSL.
`-service` *`SVCNAME`*	Specifies that additional connection parameters are to be picked up from the entry corresponding to *`SVCNAME`* in the `pg_service.conf` file.

23.4.5. Common connection options

All TDBC drivers understand the common set of options shown in Table 23.4.

Table 23.4. Connection options common to all TDBC drivers

Option	Description
`-encoding` *`NAME`*	Name of the character encoding to be used on the connection. *NAME* should be one of the names returned by the Tcl `encoding` command. It is generally not necessary to specify this but be aware that drivers differ in their handling of this option.
`-isolation` *`ISOLATION`*	Specifies the transaction isolation level needed for transactions on the database. *ISOLATION* must be one of `readuncommitted`, `readcommitted`, `repeatableread`, or `serializable`. See the tdbc::connection[9] reference for details.
`-timeout MILLISECS`	Specifies the interval after which an operation should time out with an error. The default value of `0` indicates no timeout. The operations to which the

[9] http://www.tcl.tk/man/tcl8.6/TdbcCmd/tdbc_connection.htm

Option	Description
	timeout is applicable differs between the various drivers and databases. Refer to the appropriate reference pages for each driver.
`-readonly BOOLEAN`	If specified as `true` or 1, the connection will not modify the database.

23.4.6. Configuring connections: *DBCONN* `configure`

The values of the options that can be specified at the time a `tdbc::connection` object is created can be retrieved via its `configure` method. The same method can also be used to modify the values of some options.

So to retrieve the configuration of our in-memory sample database.

```
% $dbconn configure
→ -encoding utf-8 -isolation serializable -readonly 0 -timeout 0
```

We can also pass one or more configuration options to be modified.

```
% $dbconn configure -timeout 1000
```

23.4.7. Releasing connection resources: *DBCONN* `close`

When no longer required, connections must be closed by invoking the `close` method on the `connection` object.

```
db close
```

This will also close and release resources related to `tdbc::statement` and `tdbc::resultset` objects created through the connection.

23.5. Executing SQL

Executing a SQL statement involves first *preparing* the statement and then running it one or more times with different parameter values.

23.5.1. Preparing a statement: *DBCONN* `prepare`

The first step in executing SQL is to create a `tdbc::statement` object via the `prepare` method of a `tdbc::connection`.

```
DBCONN prepare SQL
```

Here *SQL* is the SQL statement to be executed. The following creates a table in our sample database.

```
set stmt [$dbconn prepare {
    CREATE TABLE Accounts
    (Name text,
     AcctNo text NOT NULL PRIMARY KEY,
     Balance double)
}]
→ ::oo::Obj601::Stmt::1
```

We can then use the created `tdbc::statement` object to run the SQL script against the database.

23.5.2. Executing a prepared statement: *STMT* `execute`

Once a `tdbc::statement` is created, its `execute` method in invoked invoked to run the corresponding SQL.

```
% set res [$stmt execute]
→ ::oo::Obj602::ResultSet::1
```

The `execute` method returns a `tdbc::resultset` object which we will examine later. For now, we free up both objects by invoking their `close` method. Like `tdbc::connection` objects, `tdbc::statement` and `tdbc::resultset` objects should also be freed when no longer required.

```
% $stmt close
```

Note that closing the `$stmt` also closes any contained `resultset` objects so we do not need to explicitly close `$res` here. Similarly, `tdbc::statement` objects that are not closed explicitly will be closed when the owning `tdbc::connection` object is closed. However, for the sake of saving resources, it is generally a good idea to explicitly release them when no longer needed. Since we have more we want to do with the connection and are not closing it, we explicitly close `$stmt`.

Insertions and queries follow a similar pattern.

```
% set stmt [$dbconn prepare {INSERT INTO Accounts (Name, AcctNo, Balance) VALUES ('Tom', \
    'A001', 100.00)}]
→ ::oo::Obj601::Stmt::2
% $stmt execute
→ ::oo::Obj604::ResultSet::1
% $stmt close ❶
```

❶ Will also close the result set returned by `execute`

This multi-step sequence of prepare and execute can be a little tedious and TDBC provides some methods that act as wrappers and make it more convenient. We will discuss these and their pros and cons a little later.

23.5.3. Bound variables

The above example hard-coded the values that were to be inserted into the table. Naturally, that is not a viable option when values are not known apriori at the time a program is written.

TDBC allows for Tcl variable values to be passed into the SQL statement by binding names within the SQL that begin with `:` by their corresponding values. These values may either come from Tcl variables of the same name or from a dictionary passed in as an argument.

```
% set stmt [$dbconn prepare {
    INSERT INTO Accounts (Name, AcctNo, Balance) VALUES (:name, :acctno, :balance)
}]
→ ::oo::Obj601::Stmt::3
```

Here the bound variables are `name`, `acctno` and `balance`. In the script below, the values for these will be sourced from the Tcl variables of the same name.

```
foreach {name acctno balance} {
    Dick  A002 200.00
    Harry A003 300.00
} {
    $stmt execute
}
```

Alternatively, we can pass in a dictionary to the `execute` command. The values will be picked up from the keys of the same name in the dictionary.

```
% $stmt execute {acctno A004 name Moe balance 100.00} ❶
→ ::oo::Obj606::ResultSet::3
```

❶ Order of elements does not matter

Note from the sequence above that a prepared statement can be executed multiple times with different values.

23.5.3.1. Bound variable configuration: *STMT* `paramtype`

Most databases drivers automatically figure out the type and direction (input, output, or both) of bound variables. A few require the application to provide this information. The `paramtype` method is provided for this purpose.

```
STMT paramtype NAME ?DIRECTION? TYPE ?PRECISION? ?SCALE?
```

Here *`NAME`* is the name of the bound variable. The *`DIRECTION`* argument specifies whether the bound variable is used to pass input (`in`), receive output (`out`) or both (`inout`). The *`TYPE`*, *`PRECISION`* and *`SCALE`* arguments correspond to the type, precision and scale column attributes shown in Table 23.5.

Although this is not required for SQLite, which figures out the information on its own, we could configure the `balance` variable in our statement as follows:

```
% $stmt paramtype balance in double
```

Conversely, the `params` method of the `tdbc::statement` object returns the configuration of the bound variables.

```
% print_dict [$stmt params]
→ acctno    = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
  balance   = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
  name      = type Tcl_Obj precision 0 scale 0 nullable 1 direction in
```

The result of the method is a nested dictionary keyed by the names of the bound variables. Each subdictionary contains details about the corresponding bound variable. The keys of this subdictionary are the same that were described for columns in Table 23.5 with an additional key, `direction` which may have the value `in`, `out` or `inout`.

23.5.4. Closing prepared statements: *STMT* `close`

Once a `tdbc::statement` has outlived its utility, resources associated with it should be freed via its `close` method.

```
% $stmt close
```

The result sets are also automatically freed when the associated `tdbc::statement` object or containing `tdbc::connection` object are closed.

23.5.5. Direct evaluation (MySQL only): *DBCONN* `evaldirect`

For cases where MySQL does not support data management language statements via the `prepare` call, the `evaldirect` method can be used to directly pass SQL code for execution in MySQL without going through a `prepare` call first.

```
DBCONN evaldirect SQLSTMT
```

This method is only supported by the MySQL TDBC driver and should only be used in those cases where MySQL does not support the statement via the `prepare` method.

23.6. Retrieving data from result sets

Any data returned from executing SQL is stored in a result set wrapped as a `tdbc::resultset` object.

```
% set stmt [$dbconn prepare {SELECT Name, Balance from Accounts}]
→ ::oo::Obj601::Stmt::4
% set res [$stmt execute]
→ ::oo::Obj610::ResultSet::1
```

23.6.1. Introspecting result sets: *RESULTSET* `columns`

The result set is a table whose column names can be retrieved with the `tdbc::resultset` object's `columns` method.

```
$res columns → Name Balance
```

The `rowcount` method returns the number of rows in the result set table.

```
$res rowcount → 1
```

23.6.2. Retrieving result set rows: *RESULTSET* `nextlist|nextdict|nextrow`

The data itself can be retrieved using one of several methods. The most basic of these are the `nextlist`, `nextdict` and `nextrow` methods.

```
RESULTSET nextlist VAR
RESULTSET nextdict VAR
RESULTSET nextrow ?-as lists|dicts? VAR
```

All three methods return 0 if there are no more rows in the result set. Otherwise they return 1 and store the next row from the result set into the variable named *VAR*. The difference between them is the format in which the row is stored in the variable:

- `nextlist` stores the row as a list in the same order as returned by the `columns` method.
- `nextdict` stores the row as a dictionary whose keys are the column names of the result set.
- `nextrow` stores the row either as a list or a dictionary depending on the value of the `-as` option (which defaults to `dicts`).

```
% $res nextlist val
→ 1
% puts $val
→ Tom 100.0
% while {[$res nextdict val]} {
    puts $val
}
→ Name Dick Balance 200.0
  Name Harry Balance 300.0
  Name Moe Balance 100.0
```

23.6.3. Multiple result sets: *RESULTSET* `nextresults`

Some databases support a single SQL statement returning multiple result sets, each of which may have a different column structure. The presence of additional result sets may be detected by calling the `nextresults` method. This method must be called after the nextlist or nextdict command returns 0 indicating there are no more rows in the current result set.

```
$res nextresults → 0
```

The method returns 0, as in our example, if there are no more result sets. A return value of 1 indicates there are more result sets. The application can access them in the same manner as described above while noting that the columns may differ between result sets.

23.6.4. Releasing result sets: *RESULTSET* close

Once the data of interest within a result set is retrieved, associated resources should be released by calling the close method on the tdbc::resultset object.

```
% $res close
```

The result sets are also automatically freed when the associated tdbc::statement object or containing tdbc::connection object are closed.

23.6.5. Convenience wrappers for retrieval

As we have seen above, database operations involve calling the prepare and execute methods and then freeing the tdbc::statement and tdbc::resultset objects. To ensure proper cleanup, the sequence has to be wrapped in try or catch blocks. So in pseudocode the code looks roughly like this:

```
set stmt [$dbconn prepare SQL_STATEMENT]
try {
    set res [$stmt execute]
    try {
        Loop using [$res nextdict] or [$res nextlist]
    } finally {
        $res close
    }
} finally {
    $stmt close
}
```

TDBC provides two convenience methods, allrows and foreach, that take care of all the boilerplate in the above and are supported by all the major TDBC classes, tdbc::connection, tdbc::statement and tdbc::resultset,

23.6.5.1. Retrieving a complete result set: allrows

The allrows method encapsulates the above pseudocode where the loop processing consists of simply collecting all results returned by nextdict or nextlist into a single list.

The method is implemented by tdbc::connection, tdbc::statement and tdbc::resultset.

```
RESULTSET allrows ?-as lists|dicts? ?-columnsvariable COLVAR?
STATEMENT allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? ?DICT?
DBCONN    allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? SQL ?DICT?
```

- In the case of tdbc::resultset, allrows simply iterates over the result set collecting the output of nextlist or nextdict methods.
- In the case of tdbc::statement, allrows executes the statement and then collects rows from the returned result set as described in the previous case. If the optional *DICT* argument is provided, it contains the bound variables else their values are taken from Tcl variables in the caller's context. See Section 23.5.3.

- In the case of `tdbc::connection`, `allrows` prepares the SQL passed as the *`SQL`* argument, then executes the returned statement as described in the previous case. The *`DICT`* argument has the same purpose as above.

The `-as` option controls whether each element of the returned list is itself a list containing the column values for a row or a dictionary keyed by column name (default). If the `-columnsvariable` option is specified, the column names for the result set are stored in the variable *`COLVAR`* in the caller's context.

In all cases, the `allrows` method takes care to free up objects and resources appropriately even in case of errors.

Below we illustrate a simple query using the different methods, first without using `allrows`.

```
% set query_values [dict create amount 200]
→ amount 200
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::5
% set rows {}
% try {
    set res [$stmt execute $query_values]
    try {
        while {[$res nextdict row]} {
            lappend rows $row
        }
    } finally {
        $res close
    }
} finally {
    $stmt close
}
% print_list $rows
→ Name Dick
  Name Harry
```

Now the same code but using the `allrows` method of the `tdbc::resultset` object.

```
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::6
% set rows {}
% try {
    set res [$stmt execute $query_values]
    try {
        set rows [$res allrows] ❶
    } finally {
        $res close
    }
} finally {
    $stmt close
}
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ Replaces the inner loop in the previous example

Notice this returns the rows as dictionaries by default.

Above, we have only saved writing the innermost loop in the code. We can go another step further and use the `allrows` method of the `tdbc::statement` object. Note the difference from the `allrows` method of the `tdbc::resultset` in that here we need to pass in the values to be used for querying to the `allrows` method.

```
% set rows {}
% set stmt [$dbconn prepare {
    SELECT Name FROM Accounts WHERE Balance >= :amount
}]
→ ::oo::Obj601::Stmt::7
% try {
    set rows [$stmt allrows $query_values] ❶
} finally {
    $stmt close
}
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ We do not have to explicitly deal with result sets

Finally, we go the whole hog and invoke `allrows` on the database connection itself. Obviously, in this case we have to tell it the SQL we want to run in addition to passing in the query values.

```
% set rows [$dbconn allrows {
    SELECT Name FROM Accounts WHERE Balance >= :amount
} $query_values] ❶
→ {Name Dick} {Name Harry}
% print_list $rows
→ Name Dick
  Name Harry
```

❶ We do not have to deal with statements

Given this last illustration is so much shorter than the previous examples, why would one pick any of the others? The Tcl Database Connectivity paper provides some hints:

- With very large databases and result sets, `allrows` may be unworkable because of the infeasibility of collecting all rows in memory before processing.
- Explicitly dealing with result sets allows for fine-grained control of the iteration, for example terminating the iteration based on some complex rules outside of SQL's capabilities.

The use of the `-as` and `-columnsvariable` is shown below.

```
% set rows [$dbconn allrows -as lists -columnsvariable cols {
    SELECT Name,Balance FROM Accounts WHERE Balance >= :amount
} $query_values]
→ {Dick 200.0} {Harry 300.0}
% print_list $rows
→ Dick 200.0
  Harry 300.0
% puts $cols
→ Name Balance
```

23.6.5.2. Iterating over result sets: `foreach`

The `foreach` method has a purpose very similar to `allrows` except that instead simply collecting results into a list, it executes a given script for every row in the result set. Like `allrows`, it is implemented by the `tdbc::connection`, `tdbc::statement` and `tdbc::resultset` classes.

```
RESULTSET allrows ?-as lists|dicts? ?-columnsvariable COLVAR? VAR SCRIPT
STATEMENT allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? VAR ?DICT? VAR SCRIPT
DBCONN    allrows ?-as lists|dicts? ?-columnsvariable COLVAR? ?--? SQL ?DICT? SCRIPT
```

The method will iterate over all rows in the result set evaluating *`SCRIPT`* after assigning the value of the row to the variable *`VAR`*. The options `-as` and `-columnsvariable`, as well as other arguments, have the same semantics as described for allrows in the previous section.

Because of its similarity to `allrows`, we do not discuss the method in detail but only illustrate it as invoked on a `tdbc::connection` object.

```
% $dbconn foreach row {
    SELECT Name FROM Accounts WHERE Balance >= :amount
} $query_values {
    puts $row
}
→ Name Dick
  Name Harry
```

Like `allrows`, `foreach` also takes care of all intermediate bookkeeping in terms allocating and release objects.

23.7. Database transactions

There are a couple of ways an application may make use of transactions. We describe both below.

23.7.1. Using the `transaction` method

The first is making use of the `transaction` method of `tdbc::connection`.

```
DBCONN transaction SCRIPT
```

This begins a transaction on the connection and evaluates the passed script. If the script completes with a return code of ok, `return`, `break` or `continue`, the transaction is committed. For other return codes, including errors, the transaction is rolled back and the error is rethrown.

Use of the method is illustrated by the simplistic example below to transfer funds from one account to another.

```
% set transfer { from "Tom" to "Dick" amount 50 }
→  from "Tom" to "Dick" amount 50
% $dbconn transaction {
    $dbconn allrows -as lists -- {
        UPDATE Accounts
        SET Balance = Balance - :amount
        WHERE Name=:from
    } $transfer ❶

    puts "Within transaction: [$dbconn allrows -as lists -- {
            SELECT Name, Balance FROM ACCOUNTS WHERE Name=:from
    } $transfer]" ❷

    error "Pretend something went wrong"
    $dbconn allrows -as lists -- {
        UPDATE Accounts
        SET Balance = Balance + :amount
        WHERE Name=:to
    } $transfer ❸
}
Ø Within transaction: {Tom 50.0}
  Pretend something went wrong
% $dbconn allrows -as lists -- {
    SELECT Name, Balance FROM Accounts WHERE Name=:from
} $transfer ❹
→ {Tom 100.0}
```

❶ Deduct from Tom’s balance
❷ Verify balance updated within transaction
❸ Add to Dick’s balance
❹ Verify Tom’s balance restored to original

Notice that Tom’s balance is restored as the transaction was aborted by an error exception.

23.7.2. Using `begintransaction`, `commit` and `rollback`

In cases where the sequence of operations in a transaction cannot be neatly wrapped in a script that can be passed to the `transaction` method, an application can explicitly manage the transaction itself by calling the `begintransaction` method of a `tdbc::connection` object.

```
DBCONN begintransaction
```

Then at some later point, it can call the `commit` or `rollback` methods to either complete or abort the transaction.

```
DBCONN commit
DBCONN rollback
```

We can rewrite the previous example as below.

```
% set transfer { from "Tom" to "Dick" amount 50 }
→  from "Tom" to "Dick" amount 50
%
% $dbconn begintransaction ❶
% $dbconn allrows -as lists -- {
    UPDATE Accounts
    SET Balance = Balance - :amount
    WHERE Name=:from
} $transfer
%
% puts "Within transaction: [$dbconn allrows -as lists -- {
            SELECT Name, Balance FROM ACCOUNTS WHERE Name=:from
} $transfer]"
→ Within transaction: {Tom 50.0}
%
% if {[catch {
    error "Pretend something went wrong"
    $dbconn allrows -as lists -- {
        UPDATE Accounts
        SET Balance = Balance + :amount
        WHERE Name=:to
    } $transfer
}]} {
    $dbconn rollback ❷
} else {
    $dbconn commit ❸
}
% $dbconn allrows -as lists -- {
    SELECT Name, Balance FROM Accounts WHERE Name=:from
} $transfer
→ {Tom 100.0}
```

❶ Begin the transaction
❷ On error, rollback the transaction
❸ On success, commit the transaction

23.8. Handling NULL values

Noting that the empty string `""` is not the same as the SQL `NULL` value, there is no way to represent `NULL` in Tcl where everything is a string. In some applications, the distinction is not important and the empty string can be used interchangeably with `NULL`. In cases where the distinction is important, the dictionary-based interface to TDBC methods should be used as illustrated here.

Writing NULL values

To write `NULL` to a table column, pass a dictionary containing the bound variable values for the columns. A `NULL` will be stored in any column for which a corresponding key is not present in the dictionary.

```
% $dbconn allrows {
    INSERT INTO Accounts (Name, AcctNo, Balance) VALUES (:name, :acctno, :balance)
} {name Curly acctno C007}
```

Similarly, to retrieve data containing `NULL`, use one of the forms that returns rows as dictionaries. If a value is `NULL`, the returned dictionary for the row will not contain the corresponding key.

```
% $dbconn allrows {SELECT Name, Balance, AcctNo FROM Accounts WHERE Name='Curly'}
→ {Name Curly AcctNo C007}
```

Note that the key `Balance` is missing. You could have also used the `tdbc::resultset::nextdict` method for the same purpose.

Note however the result when list format is used.

```
% $dbconn allrows -as lists {SELECT Name, Balance, AcctNo FROM Accounts WHERE Name='Curly'}
→ {Curly {} C007}
```

In this case there is no way to distinguish whether the stored value in the database was actually `""` or `NULL`.

23.9. Stored procedures: *`DBCONN`* `preparecall`

Stored procedures can be invoked with the `preparecall` method of a `tdbc::connection` object.

```
DBCONN preparecall CALL
```

The syntax of the stored procedure call is

```
?RESULTVAR =? STOREDPROCNAME (? arg, …?)
```

Like the `prepare` method, this also returns a `tdbc::statement` object which can then be used as described earlier.

23.10. Introspection

All TDBC classes allow for introspection and inspection of the meta-information associated with databases.

23.10.1. Enumerating objects: *DBCONN* `statements`

TDBC keeps track of the objects that are still open within a database connection. The `statements` and `resultsets` methods retrieve the names of any existing `tdbc::statement` and `tdbc::resultset` objects within the `tdbc::connection`.

```
% $dbconn statements
→ ::oo::Obj601::Stmt::4
% $dbconn resultsets
```

Clearly we forgot to release some objects. This is actually useful for a final cleanup, for example on a per request basis to a web server that leaves the database connection open.

23.10.2. Introspecting tables: *DBCONN* `tables`

We can introspect the tables within a database with the `tables` method of a `tdbc::connection` object.

```
DBCONN tables ?SQLPAT?
```

If the *SQLPAT* argument is not provided, the commands returns information about all tables in the database. Otherwise, only tables whose name matches *SQLPAT* are included in the result. Note that *SQLPAT* should be in SQL pattern syntax.

```
% $dbconn tables
→ accounts {type table name accounts tbl_name Accounts rootpage 2 sql {CREATE TABLE Accounts
      (Name text,
       AcctNo text NOT NULL PRIMARY KEY,
       Balance double)}}
% $dbconn tables A% ❶
→ accounts {type table name accounts tbl_name Accounts rootpage 2 sql {CREATE TABLE Accounts
      (Name text,
       AcctNo text NOT NULL PRIMARY KEY,
       Balance double)}}
```

❶ `%` is a wildcard in SQL pattern syntax

The command returns a nested dictionary with the first level keys being the table names. The second level keys and values are dependent on the specific database driver. Refer to the reference documentation for the driver for details.

23.10.3. Introspecting columns: *DBCONN* `columns`

Similarly, the `columns` method retrieves column information for one or more columns in a table.

```
DBCONN columns TABLE ?SQLPAT?
```

If the *SQLPAT* argument is not provided, the commands returns information about all columns in the table *TABLE*. Otherwise, only columns with names matching *SQLPAT* are included in the result.

```
% print_dict [$dbconn columns Accounts]
→ acctno    = cid 1 name acctno type text notnull 1 pk 1 precision 0 scale 0 nullable 0
  balance   = cid 2 name balance type double notnull 0 pk 0 precision 0 scale 0 nullable 1
  name      = cid 0 name name type text notnull 0 pk 0 precision 0 scale 0 nullable 1
% print_dict [$dbconn columns Accounts Bal%]
→ balance   = cid 2 name balance type double notnull 0 pk 0 precision 0 scale 0 nullable 1
```

The command result is a nested dictionary keyed by the column name. The second level dictionary for each column provides details about the column and contains the keys shown in Table 23.5.

Table 23.5. Column description keys

Key	Description
`type`	Data type of the column
`precision`	Column precision in bits, decimal digits or width in characters, depending on the column type
`scale`	Scale of the column, i.e. number of digits after the radix point
`nullable`	Has the value 1 if the column can contain SQL NULL values and 0 otherwise

Additional keys may be present in the second level dictionaries depending on the database driver in use. Refer to the TDBC documentation for the driver for details.

23.10.4. Introspecting keys: *DBCONN* `primarykeys|foreignkeys`

Information about the primary keys defined for a table can be obtained with the `primarykeys` method of a `tdbc::connection` object.

```
DBCONN primarykeys TABLE
```

The method returns a list of descriptors for the primary keys defined for the table. Each descriptor is a dictionary with at least the key `columnName` containing the name of a column that is a member of the primary key. The descriptor may contain other database-dependent keys.

```
% $dbconn primarykeys Accounts
→ {ordinalPosition 2 columnName AcctNo}
```

In a similar vein, the `foreignkeys` method retrieves information about foreign key relationships for the specified table.

```
DBCONN foreignkeys ?-primary TABLE? ?-foreign TABLE?
```

If the `-foreign` option is specified, only the keys appearing in that table are included in the result. If the `-primary` option is specified, only the keys that refer to that table are included. It is recommended that one or both options be specified as otherwise the returned results depend on the database driver in use.

The method returns a list of descriptors of foreign key relationships. Each descriptor is a dictionary with the keys shown in Table 23.6. Depending on the database in use, the descriptor may contain additional keys apart from the ones shown in the table.

Table 23.6. Foreign key dictionary

Key	Description
`foreignTable`	The table containing the foreign key.
`foreignColumn`	The column containing the foreign key.
`primaryTable`	The table being referenced by the foreign key.
`primaryColumn`	The column being referenced by the foreign key.

23.11. ODBC utilities

The `tdbc::odbc` package provides some utility commands related to interacting with the ODBC manager. Some of these depend on the system ODBC manager's support of the ODBC Installer API.

The `tdbc::odbc::drivers` command enumerates the installed ODBC drivers on the system.

```
% package require tdbc::odbc
→ 1.0.4
% print_dict [tdbc::odbc::drivers]
→ SQL Server = APILevel=2 ConnectFunctions=YYY CPTimeout=60 DriverODBCVer=03.50 FileUsage=0
    ↳ SQLLevel=1 UsageCount=1
```

The `tdbc::odbc::datasources` command enumerates the configured ODBC data sources on the system. The command accepts the `-user` and `-system` options to limit the returned list to those configured for the current user and system respectively.

```
% print_dict [tdbc::odbc::datasources]
→ Excel Files             = Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)
  MS Access Database      = Microsoft Access Driver (*.mdb, *.accdb)
  Visio Database Samples  = Microsoft Access Driver (*.mdb, *.accdb)
% print_dict [tdbc::odbc::datasources -user]
→ Excel Files             = Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)
  MS Access Database      = Microsoft Access Driver (*.mdb, *.accdb)
  Visio Database Samples  = Microsoft Access Driver (*.mdb, *.accdb)
```

Finally, the ODBC driver provides an ensemble command, `tdbc::odbc::datasource` for management of ODBC data sources. It takes the one of the forms

```
tdbc::odbc::datasource SUBCMD DRIVER ?KEYWORD=VALUE?
```

Here *`DRIVER`* identifies the ODBC driver being targeted. The possible values for *`SUBCMD`* are shown in Table 23.7.

Table 23.7. tdbc::odbc::datasource commands

Command	Description
`add`	Adds a new user data source.
`add_system`	Adds a new system data source.
`configure`	Configures a user data source.
`configure_system`	Configures a system data source.
`remove`	Removes a user data source.
`remove_system`	Removes a system data source.

For all the above, the data source that is the target of the command is specified by a `DSN` entry in the list of keywords supplied to the command. See the reference documentation for examples.

23.12. Chapter summary

Databases comprise an important component of many software applications. Different database implementations provide different driver API's and there exist many Tcl extensions that provide programmatic access to specific databases.

TDBC is a means to access these different implementations through a uniform object-oriented API. In this chapter, we covered how you can generically use TDBC to

- establish connections
- prepare and execute statements
- execute transactions
- retrieve results

and also covered some specifics pertaining to the SQLite, MySQL, PostgreSQL and ODBC drivers.

23.13. References

TIPTDBC
Tcl Database Connectivity (TDBC), Kevin B. Kenny et al, Tcl Improvement Proposal #350, http://www.tcl.tk/cgi-bin/tct/tip/350

KBKPAPER
Tcl Database Connectivity, Kevin B. Kenny, Tcl Conference Proceedings, 2008. The original paper describing TDBC. Available from http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2008/proceedings/tdbc/tcl2k8-kenny-withfonts.pdf

TDBCREF
TDBC reference pages, http://www.tcl.tk/man/tcl8.6/TdbcCmd/contents.htm

WWWCONNSTR
Connection Strings Reference, https://www.connectionstrings.com/