The Tcl interface to the SQLite library

The SQLite library is designed to be very easy to use from a Tcl, or Tcl/Tk script. SQLite began as a Tcl extension and the primary test suite for SQLite is written in Tcl. SQLite can be used with any programming language, but its connections to Tcl run deep.

This document gives an overview of the Tcl programming interface for SQLite.

package require sqlite3

The API

The interface to the SQLite library consists of single Tcl command named **sqlite3**.
Because there is only this one command, the interface is not placed in a separate namespace.

The **sqlite3** command is mostly used as follows to open, or create a database:

sqlite3 *dbcmd ?database-name? ?options?*

To get information only, the **sqlite3** command may be given exactly one argument, either "-version", "-sourceid", or "-has-codec", which will return the specified datum with no other effect.

With other arguments, the **sqlite3** command opens the database named in the second non-option argument, or named **{}** if there is no such. If the open succeeds, a new Tcl command named by the first argument is created and **{}** is returned. (This approach is similar to the way widgets are created in Tk.) If the open fails, an error is raised without creating a Tcl command and an error message string is returned.

If the database does not already exist, the default behavior is for it to be created automatically (though this can be changed by using the "**-create *false***" option).

The name of the database is usually just the name of a disk file in which the database is stored. If the name of the database is the special name ":memory:", then a new database is created in memory. If the name of the database is an empty string **{}**, then the database is created in an empty file that is automatically deleted when the database connection closes. URI filenames can be used if the "**-uri yes**" option is supplied on the **sqlite3** command.

Options understood by the **sqlite3** command include:

-create *BOOLEAN*

> If "**true**", then a new database is created if one does not already exist. If "**false**", then an attempt to open a database file that does not previously exist raises an error. The default behavior is "**true**".

-nomutex *BOOLEAN*

> If "**true**", then all mutexes for the database connection are disabled. This provides a small performance boost in single-threaded applications.

-readonly *BOOLEAN*

> If "**true**", then open the database file **read-only**. If "**false**", then the database is opened for both reading and writing if filesystem permissions allow, or for reading only if filesystem write permission is denied by the operating system. The default setting is "**false**". Note that if the previous process to have the database did not exit cleanly and left behind a hot journal, then the write permission is required to recover the database after opening, and the database cannot be opened **read-only**.

-uri *BOOLEAN*

If "**true**", then interpret the filename argument as a URI filename. If "**false**", then the argument is a literal filename. The default value is "**false**".

-vfs *VFSNAME*

Use an alternative VFS named by the argument.

-fullmutex *BOOLEAN*

If "**true**", multiple threads can safely attempt to use the database. If "**false**", such attempts are unsafe. The default value depends upon how the extension is built.

-nofollow *BOOLEAN*

If "**true**", and the database name refers to a symbolic link, it will not be followed to open the true database file. If "**false**", symbolic links will be followed. The default is "**false**".

Once a SQLite database is open, it can be controlled using methods of the ***dbcmd***.
There are currently **40** methods defined:

- authorizer
- backup
- bind_fallback
- busy
- cache
- changes
- close
- collate
- collation_needed
- commit_hook
- complete
- config
- copy
- deserialize
- load_extension
- errorcode
- eval
- exists
- function
- incrblob
- interrupt
- last_insert_rowid
- nullvalue
- onecolumn
- preupdate
- profile
- progress
- restore
- rollback_hook
- serialize
- status
- timeout
- total_changes
- trace
- trace_v2
- transaction
- unlock_notify
- update_hook
- version
- wal_hook

The use of each of these methods will be explained in the sequel, though not in the order shown above.

The "eval" method

The most useful *dbcmd* method is "**eval**". The "**eval**" method is used to execute SQL on the database.
The syntax of the eval method looks like this:

dbcmd **eval** ?**-withoutnulls**? *sql* ?*array-name*? ?*script*?

The job of the "**eval**" method is to execute the SQL statement, or statements given in the second argument.
For example, to create a new table in a database, you can do this:

```
sqlite3 db1 ./testdb
db1 eval {CREATE TABLE t1(a int, b text)}
```

The above code creates a new table named **t1** with columns **a** and **b**.

Query results are returned as a **list** of **column values**. If a query requests 2 columns and there are 3 rows matching the query, then the returned list will contain 6 elements. For example:

```
db1 eval {INSERT INTO t1 VALUES(1,'hello')}
db1 eval {INSERT INTO t1 VALUES(2,'goodbye')}
db1 eval {INSERT INTO t1 VALUES(3,'howdy!')}
set x [db1 eval {SELECT * FROM t1 ORDER BY a}]
```

The variable **$x** is set by the above code to:

1 hello 2 goodbye 3 howdy!

You can also process the results of a query one row at a time by specifying the name of an **array variable** and a **script** following the SQL code. For each row of the query result, the values of all columns will be inserted into the **array variable** and the **script** will be executed. For instance:

```
db1 eval {SELECT * FROM t1 ORDER BY a} values {
   parray values
   chan puts {}
}
```

This last code will give the following output:

```
values(*) = a b
values(a) = 1
values(b) = hello

values(*) = a b
values(a) = 2
values(b) = goodbye

values(*) = a b
values(a) = 3
values(b) = howdy!
```

For each **column** in a **row** of the result, the **name** of that **column** is used as an **index** in to **array** and the **value** of the **column** is stored in the corresponding **array entry**. (**Caution**: If two, or more **columns** in the result set of a query have the **same name**, then the last **column** with that **name** will overwrite prior values and earlier **columns** with the same **name** will be inaccessible.) The special **array index *** is used to store a **list** of **column names** in the order that they appear.

Normally, NULL SQL results are stored in the **array** using the nullvalue setting. However, if the **-withoutnulls** option is used, then NULL SQL values cause the corresponding **array element** to be **unset** instead.

If the **array variable name** is **omitted**, or is the empty string **{}**, then the value of each column is stored in a **variable** with the **same name** as the **column** itself. For example:

```
db1 eval {SELECT * FROM t1 ORDER BY a} {
   chan puts "a=$a b=$b"
}
```

From this we get the following output:

```
a=1 b=hello
a=2 b=goodbye
a=3 b=howdy!
```

Tcl **variable names** can appear in the SQL statement of the second argument in any position where it is legal to put a string, or number literal. The **value** of the **variable** is substituted for the **variable name**. If the **variable** does not exist a NULL **value** is used. For example:

db1 eval {INSERT INTO t1 VALUES(5,$bigstring)}

Note that it is not necessary to quote the $bigstring value. That happens automatically. If $bigstring is a large string, or binary object, this technique is not only easier to write, it is also much more efficient since it avoids making a copy of the content of $bigstring.

If the $bigstring variable has both a string and a "bytearray" representation, then Tcl inserts the value as a string. If it has only a "bytearray" representation, then the value is inserted as a BLOB. To force a value to be inserted as a BLOB even if it also has a text representation, use a "**@**" character to in place of the "**$**". Like this:

db1 eval {INSERT INTO t1 VALUES(5,@bigstring)}

If the variable does not have a "bytearray" representation, then "**@**" works just like "**$**". Note that "**:**" works like "**$**" in all cases so the following is another way to express the same statement:

db1 eval {INSERT INTO t1 VALUES(5,:bigstring)}

The use of "**:**" instead of "**$**" before the name of a variable can sometimes be useful if the SQL text is enclosed in double-quotes **"..."** instead of curly-braces **{...}**. When the SQL is contained within double-quotes **"..."** then Tcl will do the substitution of **$**-variables, which can lead to SQL injection if extreme care is not used. But Tcl will never substitute a **:**-variable regardless of whether double-quotes **"..."**, or curly-braces **{...}** are used to enclose the SQL, so the use of **:**-variables adds an extra measure of defense against **SQL injection**.

The "close" method

As its name suggests, the "**close**" method to a SQLite database just closes the database. This has the side-effect of deleting the ***dbcmd*** Tcl command. Here is an example of opening and then immediately closing a database:

sqlite3 db1 ./testdb
db1 close

If you delete the ***dbcmd*** directly, that has the same effect as invoking the "**close**" method. So the following code is equivalent to the previous:

sqlite3 db1 ./testdb
rename db1 {}

The "transaction" method

The "**transaction**" method is used to execute a Tcl script inside a SQLite database transaction. The transaction is committed when the script completes, or it rolls back if the script fails. If the transaction occurs within another transaction (even one that is started manually using BEGIN) it is a **no-op**.

The "**transaction**" command can be used to group together several SQLite commands in a safe way. You can always start transactions manually using BEGIN, of course. But if an error occurs so that the COMMIT, or ROLLBACK are never run, then the database will remain locked indefinitely. Also, BEGIN does not nest, so you have to make sure no other transactions are active before starting a new one. The "**transaction**" method takes care of all of these details automatically.

The syntax looks like this:

dbcmd **transaction** *?transaction-type? script*

The "**transaction-type**" can be one of **deferred**, **exclusive**, or **immediate**. The default is **deferred**.

The "cache" method

The "**eval**" method described above keeps a cache of prepared statements for recently evaluated SQL commands. The "**cache**" method is used to control this cache. The first form of this command is:

dbcmd **cache size** *N*

This sets the maximum number of statements that can be cached. The upper limit is **100**. The default is **10**. If you set the cache size to **0**, no caching is done.

The second form of the command is this:

dbcmd **cache flush**

The "**cache-flush**" method finalizes (deletes) all prepared statements currently in the cache.

The "complete" method

The "**complete**" method takes a string of supposed SQL as its only argument. It returns "**true**" if the string is a complete statement of SQL and "**false**" if there is more to be entered.

The "**complete**" method is useful when building interactive applications in order to know when the user has finished entering a line of SQL code. This is really just an interface to the sqlite3_complete() C function.

The "config" method

The "**config**" method queries, or changes certain configuration settings for the database connection using the sqlite3_db_config() interface. Run this method with no arguments to get a Tcl **list** of available configuration settings and their current values:

dbcmd **config**

The above will return something like this:

defensive 0 **dqs_ddl** 1 **dqs_dml** 1 **enable_fkey** 0 **enable_qpsg** 0 **enable_trigger** 1 **enable_view** 1 **fts3_tokenizer** 1 **legacy_alter_table** 0 **legacy_file_format** 0 **load_extension** 0 **no_ckpt_on_close** 0 **reset_database** 0 **trigger_eqp** 0 **trusted_schema** 1 **writable_schema** 0

Add the name of an individual configuration setting to query the current value of that setting. Optionally add a boolean value to change a setting.

The following four configuration changes are recommended for maximum application security. Turning off the "**trusted_schema**" setting prevents virtual tables and dodgy SQL functions from being used inside of triggers, views, CHECK constraints, generated columns, and expression indexes. Turning off the "**dqs_dml**" and "**dqs_ddl**" settings prevents the use of double-quoted strings. Turning on "**defensive**" prevents direct writes to shadow tables.

> **db config trusted_schema 0**
> **db config dqs_dml 0**
> **db config dqs_ddl 0**
> **db config defensive 1**
> **db config load_extension 0**

The "copy" method

The "**copy**" method copies data from a **file** into a **table**. It returns the **number of rows** processed successfully from the **file**. The syntax of the copy method looks like this:

dbcmd **copy** *conflict-algorithm table-name file-name* ?*column-separator*? ?*null-indicator*?

"**conflict-algorithm**" must be one of the SQLite conflict algorithms for the **INSERT** statement: ***rollback***, ***abort***, ***fail***, ***ignore***, or ***replace***. See the SQLite Language section for ON CONFLICT for more information.
The "**conflict-algorithm**" must be specified in lower case.

"**table-name**" must already exists as a table. "**file-name**" must exist, and each **row** must contain the **same** number of **columns** as defined in the **table**. If a line in the file contains more, or less **columns** than the number of **columns** defined, the "**copy**" method rollbacks any inserts, and returns an error.

"**column-separator**" is an optional column separator string. The default is the ASCII tab character **\t**.

"**null-indicator**" is an optional string that indicates a column value is **NULL**. The default is an empty string **{}**. Note that "**column-separator**" and "**null-indicator**" are optional positional arguments; if "**null-indicator**" is specified, a "**column-separator**" argument must be specified and precede the "**null-indicator**" argument.

The "**copy**" method implements similar functionality to the **.import** SQLite **shell command**.

The "timeout" method

The "**timeout**" method is used to control how long the SQLite library will wait for locks to clear before giving up on a database transaction. The default timeout is **0** millisecond. (In other words, the default behavior is not to wait at all.)

The SQLite database allows multiple simultaneous readers, or a single writer but not both. If any process is writing to the database no other process is allows to read, or write. If any process is reading the database other processes are allowed to read but not write. The entire database shared a single lock.

When SQLite tries to open a database and finds that it is locked, it can optionally delay for a short while and try to open the file again. This process repeats until the query times out and SQLite returns a failure. The timeout is adjustable. It is set to **0** by default so that if the database is locked, the SQL statement fails immediately. But you can use the "**timeout**" method to change the timeout value to a positive number. For example:

db1 timeout 2000

The argument to the "**timeout**" method is the maximum number of **milliseconds** to wait for the lock to clear. So in the example above, the maximum delay would be **2 seconds**.

The "busy" method

The "**busy**" method, like "**timeout**", only comes into play when the database is locked. But the "**busy**" method gives the programmer much more control over what action to take. The "**busy**" method specifies a **callback Tcl procedure** that is invoked whenever SQLite tries to open a locked database. A **single integer argument** is appended to the **callback** before it is invoked. The **argument** is the **number of prior calls** to the busy **callback** for the current locking event. It is intended that the **callback** will do some other useful work for a short while (such as service GUI events) then return so that the lock can be tried again. The **callback procedure** should return "**0**" if it wants SQLite to try again to open the database and should return "**1**" if it wants SQLite to abandon the current operation.

If the "**busy**" method is invoked **without** an **argument**, the **name** of the **callback procedure** last set by the "**busy**" method is returned. If no **callback procedure** has been set, an empty string **{}** is returned.

The "load_extension" method

The extension loading mechanism of SQLite (accessed using the load_extension() SQL function) is turned **off** by default. This is a security precaution. If an application wants to make use of the load_extension() function it must first turn the capability **on** using this method.

This method takes a **single boolean argument** which will turn the extension loading functionality **on**, or **off**.

For best security, do not use this method unless truly needed, and run PRAGMA trusted_schema=OFF, or the "db config trusted_schema 0" method **before** invoking this method.

This method maps to the sqlite3_enable_load_extension() C/C++ interface.

The "exists" method

The "**exists**" method is similar to "**onecolumn**" and "**eval**" in that it executes SQL statements. The difference is that the "**exists**" method always returns a **boolean value** which is "**true**" if a query in the SQL statement it executes returns one, or more rows and "**false**" if the SQL returns an **empty set**.

The "**exists**" method is often used to test for the existence of rows in a table. For example:

```
if {[db exists {SELECT 1 FROM table1 WHERE user=$user}]} then {
  # Processing if $user exists
} else {
  # Processing if $user does not exist
}
```

The "last_insert_rowid" method

The "**last_insert_rowid**" method returns an **integer** which is the **ROWID** of the most recently inserted database **row**.

The "function" method

The "**function**" method registers new SQL functions with the SQLite engine. The arguments are the name of the new SQL **function** and a Tcl **command prefix** that implements that **function**. Arguments to the function are appended to the Tcl **command prefix** before it is invoked.

For security reasons, it is recommended that applications first set PRAGMA trusted_schema=OFF, or run the "db config trusted_schema 0" method before using this method.

The syntax looks like this:

dbcmd **function** *sql-name ?options? script*

The following example creates a new SQL function named "**hex**" that converts it's numeric argument into a hexadecimal encoded string:

```
db function hex {format 0x%X}
```

The "**function**" method accepts the following options:

-argcount *INTEGER*

> Specify the number of arguments that the SQL function accepts. The default value of -1 means any number of arguments.

-deterministic

> This option indicates that the function will always return the same answer given the same argument values. The SQLite query optimizer uses this information to cache answers from function calls with constant inputs and reuse the result rather than invoke the function repeatedly.

-directonly

This option restricts the function to only be usable by direct top-level SQL statement. The function will not be accessible to triggers, views, CHECK constraints, generated columns, or index expressions. This option is recommended for all application-defined SQL functions, and is **highly recommended** for any SQL function that has side effects, or that reveals internal state of the application.

Security warning: without this switch, an attacker might be able to change the schema of a database file to include the new function inside a trigger, or view, or CHECK constraint and thereby trick the application into running the function with parameters of the attacker's choosing. Hence, if the new function has side effects, or reveals internal state about the application and the **-directonly** option is not used, that is a potential security vulnerability.

-innocuous

This option indicates that the function has no side effects and does not leak any information that cannot be computed directly from its input parameters. When this option is specified, the function may be used in triggers, views, CHECK constraints, generated columns, and/or index expressions even if PRAGMA trusted_schema=OFF. The use of this option is discouraged unless it is truly needed.

-returntype integer|real|text|blob|any

This option is used to configure the type of the result returned by the function. If this option is set to "**any**" (the default), SQLite attempts to determine the type of each value returned by the function implementation based on the Tcl value's internal type. Or, if it is set to "**text**", or "**blob**", the returned value is always a text, or blob value, respectively. If this option is set to "**integer**", SQLite attempts to coerce the value returned by the function to an integer. If this is not possible without data loss, it attempts to coerce it to a real value, and finally falls back to text. If this option is set to "**real**", an attempt is made to return a real value, falling back to text if this is not possible.

The "nullvalue" method

The "**nullvalue**" method changes the representation for **NULL** returned as result of the "**eval**" method.

db1 nullvalue NULL

The "**nullvalue**" method is useful to differ between **NULL** and empty string **{}** column values as Tcl lacks a **NULL** representation. The default representation for **NULL** values is an empty string **{}**.

The "onecolumn" method

The "**onecolumn**" method works like "**eval**" in that it evaluates the SQL query statement given as its argument. The difference is that "**onecolumn**" returns a **single element** which is the **first column** of the **first row** of the query result.

This is a convenience method. It saves the user from having to do a "**[lindex ... 0]**" on the results of an "**eval**" in order to extract a **single column result**.

The "changes" method

The "**changes**" method returns an **integer** which is the **number of rows** in the database that were inserted, deleted, and/or modified by the most recent "**eval**" method.

The "total_changes" method

The "**total_changes**" method returns an **integer** which is the **number of rows** in the database that were inserted, deleted, and/or modified since the **current database connection** was first opened.

The "authorizer" method

The "**authorizer**" method provides access to the sqlite3_set_authorizer C/C++ interface. The argument to "**authorizer**" is the **name** of a **procedure** that is called when SQL statements are being compiled in order to authorize certain operations. The **callback procedure** takes **5** arguments which describe the operation being coded. If the **callback** returns the text string "**SQLITE_OK**", then the operation is allowed. If it returns "**SQLITE_IGNORE**", then the operation is silently disabled. If the return is "**SQLITE_DENY**" then the compilation fails with an error.

If the argument is an empty string **{}** then the "**authorizer**" is disabled. If the argument is omitted, then the current "**authorizer**" is returned.

The "bind_fallback" method

The "**bind_fallback**" method gives the application control over how to handle parameter binding when no Tcl variable matches the parameter name.

When the "**eval**" method sees a named SQL parameter such as "**$**abc", or "**:**def", or "**@**ghi" in a SQL statement, it tries to look up a Tcl **variable** with the same name, and it **binds** the **value** of that Tcl **variable** to the SQL parameter. If no such Tcl variable exists, the default behavior is to bind a SQL NULL value to the parameter. However, if a "**bind_fallback**" procedure is specified, then that procedure is invoked with the name of the SQL parameter as an argument and the return value from the procedure is **bound** to the SQL parameter. Or if the procedure returns an error, then the SQL statement aborts with that error. If the procedure returns with some code other than TCL_OK (**0**), or TCL_ERROR (**1**), then the SQL parameter is bound to NULL, as it would be by default.

The "**bind_fallback**" method has a single optional argument. If the argument is an empty string **{}**, then the "**bind_fallback**" is canceled and the default behavior is restored. If the argument is a non-empty string, then the argument is a Tcl **command** (usually the name of a procedure) to invoke whenever a SQL parameter is seen that does not match any Tcl variable. If the "**bind_fallback**" method is given no arguments, then the current "**bind_fallback**" command is returned.

As an example, the following setup causes Tcl to throw an error if a SQL statement contains a parameter that does not match any global Tcl variable:

```
proc bind_error {nm} {
  error "no such variable: $nm"
}
db bind_fallback bind_error
```

The "progress" method

This method registers a **callback** that is invoked periodically during query processing. There are two arguments: the number of SQLite virtual machine opcodes between invocations, and the Tcl **command** to invoke. Setting the progress callback to an empty string **{}** disables it.

The progress callback can be used to display the status of a lengthy query, or to process GUI events during a lengthy query.

The "collate" method

This method registers new text collating sequences. There are two arguments: the name of the collating sequence and the name of a Tcl **procedure** that implements a comparison function for the collating sequence.

For example, the following code implements a collating sequence called "**NOCASE**" that sorts in text order without regard to case:

```
proc nocase_compare {a b} {
  return [string compare [string tolower $a] [string tolower $b]]
}
db collate NOCASE nocase_compare
```

The "collation_needed" method

This method registers a **callback** routine that is invoked when the SQLite engine needs a particular collating sequence but does not have that collating sequence registered. The **callback** can register the collating sequence. The **callback** is invoked with a single parameter which is the **name** of the needed collating sequence.

The "commit_hook" method

This method registers a **callback** routine that is invoked just before SQLite tries to commit changes to a database. If the callback throws an exception, or returns a non-zero result, then the transaction rolls back rather than commit.

The "rollback_hook" method

This method registers a **callback** routine that is invoked just before SQLite tries to do a rollback. The script argument is run without change.

The "status" method

This method returns **status information** from the most recently evaluated SQL statement. The "**status**" method takes a single argument which should be either "**steps**", or "**sorts**". If the argument is "**steps**", then the method returns the number of full table scan steps that the previous SQL statement evaluated. If the argument is "**sorts**", the method returns the number of sort operations. This information can be used to detect queries that are not using indices to speed up searching, or sorting.

The "**status**" method is basically a wrapper on the sqlite3_stmt_status() C language interface.

The "update_hook" method

This method registers a **callback** routine that is invoked just after each **row** is modified by an UPDATE, INSERT, or DELETE statement. Four arguments are appended to the callback before it is invoked:

- The keyword "INSERT", "UPDATE", or "DELETE", as appropriate
- The name of the database which is being changed
- The table that is being changed
- The rowid of the row in the table being changed

The **callback** must not do anything that will modify the database connection that invoked the "**update hook**" such as running queries.

The "preupdate" method

This method either registers a **callback** routine that is invoked just before each row is modified by an UPDATE, INSERT, or DELETE statement, or may perform certain operations related to the impending update.

To register, or remove a "**preupdate**" **callback**, use this syntax:

dbcmd **preupdate hook** *?SCRIPT?*

When a "**preupdate**" **callback** is registered, then prior to each row modification, the callback is run with these arguments:

- The keyword "INSERT", "UPDATE", or "DELETE", as appropriate
- The name of the database which is being changed
- The table that is being changed
- The original rowid of the row in the table being changed
- The new rowid (if any) of the row in the table being changed

The **callback** must not do anything that will modify the database connection that invoked the "**preupdate**" hook such as running queries.

When the callback is executing, and only then, these "**preupdate**" operations may be performed by use of the indicated syntax:

dbcmd **preupdate count**
dbcmd **preupdate depth**
dbcmd **preupdate new** *INDEX*
dbcmd **preupdate old** *INDEX*

The **count** submethod returns the number of columns in the row that is being inserted, updated, or deleted.

The **depth** submethod returns the update nesting depth. This will be **0** for a direct insert, update, or delete operation; **1** for inserts, updates, or deletes invoked by top-level triggers; or higher values for changes resulting from trigger-invoked triggers.

The **old** and **new** submethods return the selected original, or changed column **value** respectively of the row being updated.

Note that the Tcl interface (and underlying SQLite library) must have been built with the preprocessor symbol SQLITE_ENABLE_PREUPDATE_HOOK defined for the "**preupdate**" method to be available.

The "wal_hook" method

This method registers a **callback** routine that is invoked after transaction commit when the database is in WAL mode. Two arguments are appended to the callback command before it is invoked:

- The name of the database on which the transaction was committed
- The number of entries in the write-ahead log (WAL) file for that database

This method might decide to run a checkpoint either itself, or as a subsequent **idle callback**. Note that SQLite only allows a single WAL hook. By default this single WAL hook is used for the auto-checkpointing. If you set up an explicit WAL hook, then that one WAL hook must ensure that checkpoints are occurring since the auto-checkpointing mechanism will be disabled.

This method should return an **integer value** that is equivalent to a SQLite error code (usually **0** for SQLITE_OK in the case of success, or **1** for SQLITE_ERROR if some error occurs). As in sqlite3_wal_hook(), the results of returning an **integer** that does not correspond to a SQLite error code are undefined. If the value returned by the script cannot be interpreted as an **integer value**, or if the script throws a Tcl exception, no error is returned to SQLite but a Tcl background error is raised.

The "incrblob" method

This method opens a Tcl **channel** that can be used to read, or write into a **preexisting** BLOB in the database. This method may only modify the contents of the BLOB, it's **not possible** to increase the size of a BLOB using this API. The syntax is like this:

dbcmd **incrblob ?-readonly?** *?db? table column rowid*

Parameter "***db***" is not the filename that contains the database, but rather the symbolic name of the database. For attached databases, this is the name that appears after the **AS** keyword in the ATTACH statement. For the main database file, the database name is "**main**". For TEMP tables, the database name is "**temp**".
The command returns a new Tcl **channel** for reading, or writing to the BLOB. The **channel** is opened using the underlying sqlite3_blob_open() C language interface. Close the channel using the **chan close** command of Tcl.

The "errorcode" method

This method returns the numeric error code that resulted from the most recent SQLite operation.

The "trace" method

The "**trace**" method registers a **callback** that is invoked as each SQL statement is compiled. The text of the SQL query is appended as a **single string** to the **command prefix** before it is invoked. This can be used (for example) to keep a log of all SQL operations that an application performs. "**dbcmd trace {}**" removes the trace.

The "trace_v2" method

The "**trace_v2**" method registers a **callback** that is invoked as each SQL statement is compiled.
The syntax is as follows:

dbcmd **trace_v2** ?*callback*? ?*mask*?

This command causes the "**callback**" script to be invoked whenever certain conditions occurs. The conditions are determined by the "**mask**" argument, which should be a Tcl **list** of zero, or more of the following keywords: "**statement**", "**profile**", "**row**", "**close**".

Traces for "**statement**" invoke the **callback** with two arguments whenever a new SQL statement is run. The first argument is an **integer** which is the value of the pointer to the underlying sqlite3_stmt object. This **integer** can be used to correlate SQL statement text with the result of a "**profile**", or "**row**" **callback**. The second argument is the **unexpanded text** of the SQL statement being run. By "unexpanded", we mean that variable substitutions in the text are not expanded into the variable values. This is different from the behavior of the "**trace**" method which does expand variable substitutions.

Traces for "**profile**" invoke the **callback** with two arguments as each SQL statement finishes. The first argument is an **integer** which is the value of the underlying sqlite3_stmt object. The second argument is the approximate **run-time** for the statement in nanoseconds. The **run-time** is the best estimate available depending on the capabilities of the platform on which the application is running.

Traces for "**row**" invoke the **callback** with a **single argument** whenever a new result row is available from a SQL statement. The argument is an **integer** which is the value of the underlying sqlite3_stmt object pointer.

Traces for "**close**" invoke the **callback** with a single argument as the database connection is closing. The argument is an **integer** which is the value of a pointer to the underlying sqlite3 object that is closing.

There can only be a single trace **callback** registered on a database connection.
Each use of "**trace**" or, "**trace_v2**" cancels all prior trace **callbacks**.

The "backup" method

The "**backup**" method makes a backup copy of a live database. The command syntax is like this:

dbcmd **backup** ?*source-database*? *backup-filename*

The optional "**source-database**" argument tells which database in the current connection should be backed up. The default value is "**main**" (or, in other words, the primary database file). To back up TEMP tables use "**temp**". To backup an auxiliary database added to the connection using the ATTACH command, use the name of that database as it was assigned in the ATTACH command (the name that appears after the **AS** keyword).

The "**backup-filename**" is the name of a file into which the backup is written. "**backup-filename**" does not have to exist ahead of time, but if it does, it must be a well-formed SQLite database.

The "restore" method

The "**restore**" method copies the content from a separate database file into the current database connection, overwriting any **preexisting** content. The command syntax is like this:

dbcmd **restore** ?*target-database*? *source-filename*

The optional "**target-database**" argument tells which database in the current connection should be overwritten with new content. The default value is "**main**" (or, in other words, the primary database file). To repopulate the TEMP tables use "**temp**". To overwrite an auxiliary database added to the connection using the ATTACH command, use the name of that database as it was assigned in the ATTACH command.

The "**source-filename**" is the name of an existing well-formed SQLite database file from which the content is extracted.

The "serialize" method

The "**serialize**" method creates a BLOB which is a complete copy of an underlying database.
The syntax is like this:

dbcmd **serialize** ?*database*?

The optional argument is the **name** of the **schema**, or **database** to be serialized. The default value is "**main**".

This routine returns a Tcl **byte-array** that is the complete content of the identified database. This **byte-array** can be written into a file and then used as an ordinary SQLite database, or it can be sent over a TCP/IP connection to some other application, or passed to the "**deserialize**" method of another database connection.

This method only functions if SQLite is compiled with -DSQLITE_ENABLE_DESERIALIZE.

The "deserialize" method

The "**deserialize**" method takes a Tcl **byte-array** that contains a SQLite database file and adds it to the database connection. The syntax is:

dbcmd **deserialize** ?*database*? *value*

The optional "**database**" argument identifies which attached database should receive the deserialization.
The default is "**main**".

This command causes SQLite to disconnect from the previous database and reattach to an in-memory database with the content in "**value**". If "**value**" is not a **byte-array** containing a well-defined SQLite database, then subsequent commands will likely return SQLITE_CORRUPT errors.

This method only functions if SQLite is compiled with -DSQLITE_ENABLE_DESERIALIZE.

The "interrupt" method

The "**interrupt**" method invokes the sqlite3_interrupt() interface, causing any pending queries to halt.

The "version" method

Returns the current library version. For example, "3.40.1"

The "profile" method

This method is used to profile the execution of SQL statements run by the application. The syntax is as follows:

dbcmd **profile** ?*script*?

Unless "**script**" is an empty string **{}**, this method arranges for the "**script**" (**command prefix**) to be evaluated after the execution of each SQL statement. Two arguments are appended to "**script**" before it is invoked: the **text** of the **SQL statement** executed and the **time elapsed** while executing the statement, in **nanoseconds**.

A database handle may only have a single profile script registered at any time. If there is already a script registered when the profile method is invoked, the previous profile script is replaced by the new one. If the "**script**" argument is an empty string **{}**, previously registered profile **callback** is canceled but no new profile script is registered.

The "unlock_notify" method

The "**unlock_notify**" method is used access the sqlite3_unlock_notify() interface to the SQLite core library for testing purposes. The use of this method by applications is discouraged.