SQLite3Professional

version 2.0

CONTENTS

INTRODUCTION

SQLITE3 MODULE

SQLITE3DATABASE CLASS

SQLITE3VM CLASS

APPENDIX A

APPENDIX B

Overview of SQLite3ProfessionalPlugin

SQLite3ProfessionalPlugin is a REALbasic plugin that gives your REALbasic projects low level access to the SQLite3 database engine. Starting from RB2005, the sqlite3 engine is include in every version of REALbasic, but due to the way it is exposed to the end user, a lot of its power and its speed aren't available.
SQLite3ProfessionalPlugin has been designed to take advantage of all the real speed and real power that the new sqlite3 engine can offer.

SQLite3ProfessionalPlugin is very different from the built-in REALbasic database engine, it does have built-in Database and RecordSet classes but in addition to this, it supports a lot more sqlite 3 specific methods. It is a near 1 to 1 map to the sqlite3 C api, so you can have access to every sqlite3 routine, so for example, you can interact directly with the virtual machine or you can create your own sql functions, and much more. These are very powerful concepts that cannot be added to the current REALSQLDatabase plugin.

The audience for this plugin are professional database users, that, for example, can take advantage of the sqlite3 progress handler routine to give time to other threads or to update the REALbasic GUI while a complex sql operation is in place, or that can use the sqlite3 virtual machine to build a query system with zero delay time and near zero memory requirements even if millions of rows are returned from a query.

SQLite3ProfessionalPlugin is divided into 2 main classes and 1 module.

The SQLite3 Module registers a set of constants used by all the methods in every class, it also defines some “globals” methods.

The SQLite3Database Class is the main class for this plugin, it defines methods necessary to open a database, close it and perform other operations.

The SQLite3VM Class is the class returned by the SQLite3Database's Prepare method. It defines methods necessary to interact with the powerful sqlite3 virtual machine.

Registering

Unregistered version can be evaluated in the REALbasic IDE and during debugging, but will not allow you to compile a standalone application.

To became a registered user and receive technical support and updates you must register this plugin via web at the address: http://www.sqlabs.net/store.php

Getting Started

SQLite3ProfessionalPlugin requires REALbasic 2005 or higher on Windows, Linux and Mac OS X (MachO PPC or Universal Binary).

Installing SQLite3ProfessionalPlugin is very easy. Simply drag it to the REALbasic Plugins folder and launch REALbasic. SQLite3ProfessionalPlugin should now be available for use.

If you have a registration code, then you will need to pass it to the plugin in your application with the **SQLITE3.PluginRegister** global method. This method should be called as early as possible in your code and before you attempt to call any other SQLite3ProfessionalPlugin functions.

One good place to put the call is the Open event of your App class. If you don’t pass a registration number to SQLite3ProfessionalPlugin, then you will be running in demo mode.

Using SQLite3ProfessionalPlugin is very easy, for example, in order to open a database just write:

```
Dim db As New SQLite3Database
Dim f As FolderItem = GetOpenFolderItem(“”)
If (f = nil) then return

if (db.Open(f) <> SQLITE3.kOK) then
	MsgBox “Error while opening the database “ + f.Name
end if
```

SQLite3 Virtual Machine

This is the most important and most powerful class inside the plugin, if you take the time to fully understand it (and this is an easy task) then you can gain a lot of benefits, like blazing speed and complete control over your sqlite3 database.

The most important thing to understand is that when you send an sql command to sqlite, the engine doesn’t know in advance the number of rows returned by the sql query, the only way to iterate through all the rows of the result, is to call the VMStep method until an SQLITE3.kDONE result is returned (an error occurs for example).

For example, in order to execute a simple SQL statement like “CREATE TABLE test (a TEXT, b TEXT)” the correct way is (using the virtual machine):

```
Sub myCreateTable As Boolean
        Dim vm As SQLite3VM
        Dim sql As String = “CREATE TABLE test (a TEXT, b TEXT)”
        Dim res As Integer

        vm = db.Prepare (sql)
        if (vm = nil) then return false

        res = vm.VMStep
        return (res = SQLITE3.kDONE)
End Sub
```

The prepare statement does nothing, it just compile the sql ready to be used inside the virtual machine, you are then responsible to call the VMStep method until the SQLITE3.kDONE value is returned. Obviously, because this function is trying to execute an SQL statement, you know in advance that you need to call VMStep just one time ... but how can you do if you want to implement an SQLSelect like function?

The first solution is the one adopted by the REALSQLDatabase, where when you issue an SQLSelect statement it just calls the VMStep method N times (if N are the number of rows returned until an SQLITE3.kDONE result is returned) and it builds an in-memory array with all the results that you can access with the help of the RecordSet class.

This solution is not very efficient, mainly because memory requirements could became huge very easily (remember, all the rows are stored in memory) and because you have to wait the time needed to scan and extract all the columns from the entire record set before you can start use them.

A more efficient way to solve the issue is something like this (suppose for example that you want to fill a ListBox with the data returned from your query):

```
Dim vm As SQLite3VM
Dim sql As String = “SELECT * FROM test”
Dim res As Integer

vm = db.Prepare (sql)
if (vm = nil) then return

while (vm.VMStep <> SQLITE3.kDONE)
    ListBox1.AddRow vm.ColumnText(0)
wend
```

As you can see, in this way memory requirement is just the memory needed for the current column and you can start receiving rows immediately.

As a real world test, we have created an sqlite3 database with 10 columns and 200,000 rows (database size is about 200MB), and we have execute a “SELECT * from test” query. These are the results of the test:

	SQLite3Professional	**REALSQLDatabase**
Memory usage:	44 Bytes	200 MB
Query Time (sec.):	0.0001	15.879
Time to scan RecordSet (sec.):	4.331	0.037
Total Time (sec.):	4.331	15.916

Please note that inside the SQLite3VM Class there are methods that enable you to know how many columns are returned, their names, their types and so on.

InstantQuery Technology

InstantQuery is a new technique for retrieving results from a database query at “lightning speed”, you can now populate your listbox from a sql query containing millions of rows in under a second. If your application requires hundreds of thousands or even millions of results to be returned and you want to start viewing those results instantly, then InstantQuery is the right choice.

The only requirement in order to gain full speed from the InstantQuery technology is a REALbasic control that enable you to display any number of rows on demand, without big delays and bottlenecks. For this reason standard REALbasic ListBoxes are highly discouraged and we strongly recommend you use one of the following controls:

- Data On Demand ListBox (www.mactechnologies.com)
- DataGrid or StyleGrid Plugin (www.einhugur.com)
- your custom virtual ListBox
- any canvas based solution that emulate a standard ListBox

The core of this technology is the InstantQuery method and the main idea behind it is that you can have a hidden thread that perform the hard work while you display your results to the end user. This method not only enable you to gain blazing speed but also enable you to display millions of row with very little memory requirements.

As a real world example. we take the same sqlite3 database with 10 columns and 200,000 rows mentioned in the previous chapter.

The traditional way to display a ListBox with all the rows from the query “SELECT * from test”, is to execute an SQLSelect statement and use the resulted RecordSet. That means that you have to spend 200MB of RAM to hold the RecordSet in memory and you have to wait about 16 seconds (time measured with an iMac G5) just to fill in entire RecordSet, plus the time required to display all the rows (this can vary from less than a second if you use one of the recommended controls to few minutes if you use a standard ListBox). During this time, your user can just see a blank listbox and a spinning cursor...

InstantQuery technology can solve this issue in a very elegant way.

Modify the query to send to the InstantQuery method from “SELECT * from test” to “SELECT rowid from test” (remember that you have to always put the hard work into the InstantQuery method) and then when row N is requested to be displayed into the ListBox just use the RowValue method to retrieve and display it.

This is the full example (please note that Data On Demand is used in this code):

```
// Execute the instant query
// this method returns immediately, even if there are millions of row, so you can start
```

```
display your rows instantly
db.InstantQuery("SELECT rowid from test", true)

// start a thread that needs only to add rows to my ListBox
Thread1.Run

// Thread1 Code
while (db.Complete = false)

        // This is optional but recommended
        // sleep 200ms in order to give some time
        // to the InstantQuery to retrieve new rows
        me.Sleep 200

        // add new rows to the Data On Demand ListBox
        DoDList.Reset(db.RowCount, -1)
wend

// and then into your DoDList.RequestRowData (dataRow as integer) event
// just write something like this:

   sql = "SELECT * FROM test WHERE rowid=" + Str(db3.RowValue(dataRow))
   vm = db3.Prepare(sql)
   i = vm.VMStep
   // iterate through all the columsn of the currentrow
   for i=0 to colCount-1
     me.cell(dataRow,i)= vm.ColumnText(i)
   next
```

In this way you just need the memory required to load ONE column and your user can start viewing and using the results from your query immediately!
Please take a look at the InstantQuery example in order to fully understand how this new powerful technology works.

SQLITE3 Module

The module SQLITE3 registers a set of constants and methods used by all the methods in every class.

Contants

- kOK
- kERROR
- kINTERNAL
- kPERM
- kABORT
- kBUSY
- kLOCKED
- kNOMEM
- kREADONLY
- kINTERRUPT
- kIOERR
- kCORRUPT
- kNOTFOUND
- kFULL
- kCANTOPEN
- kPROTOCOL
- kEMPTY
- kSCHEMA
- kTOOBIG
- kCONSTRAINT
- kMISMATCH
- kMISUSE
- kNOLFS
- kAUTH
- kFORMAT
- kRANGE
- kNOTADB
- kROW
- kDONE
- kCOPY
- kCREATE_INDEX
- kCREATE_TABLE
- kCREATE_TEMP_INDEX
- kCREATE_TEMP_TABLE
- kCREATE_TEMP_TRIGGER
- kCREATE_TEMP_VIEW
- kCREATE_TRIGGER
- kCREATE_VIEW
- kDELETE
- kDROP_INDEX

- kDROP_TABLE
- kDROP_TEMP_INDEX
- kDROP_TEMP_TABLE
- kDROP_TEMP_TRIGGER
- kDROP_TEMP_VIEW
- kDROP_TRIGGER
- kDROP_VIEW
- kINSERT
- kPRAGMA
- kREAD
- kSELECT
- kTRANSACTION
- kUPDATE
- kATTACH
- kDETACH
- kALTER_TABLE
- kREINDEX
- kANALYZE
- kDENY
- kIGNORE
- kINTEGER
- kFLOAT
- kTEXT
- kBLOB
- kNULL
- kUTF8
- kUTF16LE
- kUTF16BE
- kUTF16
- kANY
- kDEMO

Properties

None

Methods

- LibVersion
- PluginVersion
- MsSleep
- Complete
- LastInsertRowID
- PluginRegister
- AggregateCount* (see Appendix A)
- Result* (see Appendix A)
- Value* (see Appendix A)
- EnableSharedCache

Methods

LibVersion() As String
Returns the version of the underline sqlite3 library.

PluginVersion() As String
Returns the version of the plugin.

MsSleep(ms As Integer)
Sleep for a little while. The parameter is the number of miliseconds to sleep for.
If the operating system does not support sleep requests with milisecond time resolution, then the time will be rounded up to the nearest second.

Complete(sql As String) As Boolean
This method return true if the given input string comprises one or more complete SQL statements.

PluginRegister(code As String) As Boolean
Register the plugin with the given code. Returns false if registration fails.

EnableSharedCache(enable As Boolean) As Integer

Due to some limitations of the PEF version, this method is disabled in “MacOSX/Classic” builds.
This routine enables or disables the sharing of the database cache and schema data structures between connections to the same database. Sharing is enabled if the argument is true and disabled if the argument is false.

Cache sharing is enabled and disabled on a thread-by-thread basis. Each call to this routine enables or disables cache sharing only for connections created in the same thread in which this routine is called. There is no mechanism for sharing cache between database connections running in different threads.
Sharing must be disabled prior to shutting down a thread or else the thread will leak memory. Call this routine with an argument of “false” to turn off sharing.

This routine must not be called when any database connections are active in the current thread. Enabling or disabling shared cache while there are active database connections will result in memory corruption.
When the shared cache is enabled, the following routines must always be called from the same thread: db.Open(), db.prepare(), vm.VMstep(), vm.Reset(), vm.finalize(), and db.close(). This is due to the fact that the shared cache makes use of thread-specific storage so that it will be available for sharing with other connections.
This routine returns SQLITE3_kOK if shared cache was enabled or disabled successfully. An error code is returned otherwise.
Shared cache is disabled by default for backward compatibility.

SQLite3Database Class

It is the main class for this plugin, it defines methods necessary to open a database, close it and perform other operations.

Events

Trace
ProgressHandler
CommitHook
RollbackHook
UpdateHook
BusyHandler
Authorize

Properties

TraceEvent
AuthorizeEvent
ProgressHandlerPeriod
CommitHookEvent
RollbackHookEvent
UpdateHookEvent
BusyHandlerEvent
Complete
RowCount

Methods

Open
Close
Changes
TotalChanges
LastInsertRowID
ErrCode
ErrMsg
GetAutoCommit
Interrupt
Prepare
InstantQuery
RowValue
BusyTimeout
CreateCollation
CreateScalarFunction
CreateAggregateFunction
EnableSharedCache
TableColumnMetadata

Events

Trace(sql As String)
The trace event is called each time an SQL statement is evaluated. This function can be used (for example) to generate a log file of all SQL executed against a database. This can be useful when debugging an application that uses SQLite.

ProgressHandler() As Integer
The ProgressHandler event is invoked once for every N virtual machine opcodes, where N is the value of the ProgressHandlerPeriod property. If a call to sqlite3.VMStep() results in less than N opcodes being executed, then the progress callback is not invoked. An example use for this event is to keep a GUI updated during a large query.

If the progress callback returns a result other than 0, then the current query is immediately terminated and any database changes rolled back. If the query was part of a larger transaction, then the transaction is not rolled back and remains active. The sqlite3.VMStep() call returns SQLITE3.kABORT.

CommitHook() As Integer
The CommitHook event is invoked whenever a new transaction is committed. If the event returns non-zero, then the commit is converted into a rollback.

BusyHandler (nTimes As Integer) As Integer
The BusyHandler event might be invoked whenever an attempt is made to open a database table that another thread or process has locked.
If the BusyHandlerEvent property is false, then SQLITE3.kBUSY is returned immediately upon encountering the lock, otherwise the BusyHandler event might beinvoked. The nTimes argument is the number of prior calls to the BusyHandler event for the same lock. If the BusyHandler event returns 0, then no additional attempts are made to access the database and SQLITE3.kBUSY is returned. If the event returns non-zero, then another attempt is made to open the database for reading and the cycle repeats.

The presence of a BusyHandler event does not guarantee that it will be invoked when there is lock contention. If SQLite determines that invoking the busy handler could result in a deadlock, it will return SQLITE3.kBUSY instead.

Sqlite is re-entrant, so the BusyHandler may start a new query. (It is not clear why anyone would every want to do this, but it is allowed, in theory.) But you cannot close the database inside a BusyHandler event. Closing the database inside a BusyHandler event will delete data structures out from under the executing query and will probably result in a coredump.

Authorize (accessFunction As Integer, param1 As String, param2 As String, param3 As String, param4 As String) As Integer
The authorize event is is invoked for each attempt to access a column of a table in the database.

The event should return SQLITE3.kOK if access is allowed, SQLITE3.kDENY if the entire SQL statement should be aborted with an error and SQLITE3.kIGNORE if the column should be treated as a NULL value.

The first argument to the authorize event will be one of the defined constants shown. These values signify what kind of operation is to be authorized. The 2nd and 3rd arguments to the authorization function will be arguments or empty strings depending on which of the following codes is used as the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or an empty string if this access attempt is directly from input SQL code.

The intent of this routine is to allow applications to safely execute user-entered SQL. An appropriate callback can deny the user-entered SQL access certain operations (ex: anything that changes the database) or to deny access to certain tables or columns within the database.

accessFunction	**param1**	**param2**
SQLITE3.kCREATE_INDEX	Index Name	Table Name
SQLITE3.kCREATE_TABLE	Table Name	Empty String
SQLITE3.kCREATE_TEMP_INDEX	Index Name	Table Name
SQLITE3.kCREATE_TEMP_TABLE	Table Name	Empty String
SQLITE3.kCREATE_TEMP_TRIGGER	Trigger Name	Table Name
SQLITE3.kCREATE_TEMP_VIEW	View Name	Empty String
SQLITE3.kCREATE_TRIGGER	Trigger Name	Table Name
SQLITE3.kCREATE_VIEW	View Name	Empty String
SQLITE3.kDELETE	Table Name	Empty String
SQLITE3.kDROP_INDEX	Index Name	Table Name
SQLITE3.kDROP_TABLE	Table Name	Empty String
SQLITE3.kDROP_TEMP_INDEX	Index Name	Table Name
SQLITE3.kDROP_TEMP_TABLE	Table Name	Empty String
SQLITE3.kDROP_TEMP_TRIGGER	Trigger Name	Table Name
SQLITE3.kDROP_TEMP_VIEW	View Name	Empty String
SQLITE3.kDROP_TRIGGER	Trigger Name	Table Name
SQLITE3.kDROP_VIEW	View Name	Empty String
SQLITE3.kINSERT	Table Name	Empty String
SQLITE3.kPRAGMA	Pragma Name	1starg or Empty String
SQLITE3.kREAD	Table Name	Column Name
SQLITE3.kSELECT	Empty String	Empty String
SQLITE3.kTRANSACTION	Empty String	Empty String
SQLITE3.kUPDATE	Table Name	Column Name
SQLITE3.kATTACH	Filename	Empty String
SQLITE3.kDETACH	Database Name	Empty String

RollbackHook()
The RollbackHook event is invoked whenever a new transaction have been rolled back by an explicit “ROLLBACK” statement or by an error or by a constraint that causes an implicit rollback to occur. The callback is not invoked if a transaction is automatically rolled back because the database connection is closed.

UpdateHook (operationType As Integer, tableName As String, rowID As Integer)
The UpdateHook event is invoked whenever a row is updated, inserted or deleted.
The first parameter can be SQLITE3.kINSERT, SQLITE3.kDELETE or SQLITE3.kUPDATE, depending on the operation that caused the event to be invoked.
The second argument contains the table name of the affected row and the last argument is its rowid.
In the case of an update, this is the rowid after the update takes place.
The UpdateHook event is not invoked when internal system tables are modified (i.e. sqlite_master and sqlite_sequence).

Properties

TraceEvent as Boolean
The default value is false.
If true, then the trace event is called each time an SQL statement is evaluated.

AuthorizeEvent as Boolean
The default value is false.
If true, then the authorize event is is invoked for each attempt to access a column of a table in the database.

ProgressHandlerPeriod as Integer
The default value is 0.
If its values is greater than 0, than the ProgressHandler event is invoked once for every N virtual machine opcodes, where N is the value of the ProgressHandlerPeriod property.

CommitHookEvent as Boolean
The default value is false.
If true, then the CommitHook event is invoked whenever a new transaction is committed.

RoolbackHookEvent as Boolean
The default value is false.
If true, then the RollbackHook event is invoked whenever a new transaction is rolled-back (by an explicit “Rollback” statement or after an error or a constraint that causes an implicit rollback to occurs).

UpdateHookEvent as Boolean
The default value is false.
If true, then the UpdateHook event is invoked each time a row is updated, inserted or deleted.

BusyHandlerEvent as Boolean
The default value is false.
If true, then the BusyHandler event might be invoked whenever an attempt is made to open a database table that another thread or process has locked.

Complete as Boolean
True if the latest issued InstantQuery has completed its execution.

RowCount As Integer
The number of rows currently returned from the latest issued InstantQuery, please note that this value can change until the Complete property is true.

Methods

Open (path As FolderItem) As Integer
Open or create the sqlite database file pointed by “path”. If the database is opened (or created) successfully, then SQLITE3.kOK is returned. Otherwise an error code is returned. The ErrMsg method can be used to obtain an English language description of the error.

If the database file does not exist, then a new database will be created as needed. The encoding for the database will be UTF-8.
If path is nil then an in-memory database is created.

Close()
Closes the database.

Changes() As Integer
This method returns the number of database rows that were changed (or inserted or deleted) by the most recently completed INSERT, UPDATE, or DELETE statement. Only changes that are directly specified by the INSERT, UPDATE, or DELETE statement are counted. Auxiliary changes caused by triggers are not counted. Use the TotalChanges method to find the total number of changes including changes caused by triggers.

SQLite implements the command “DELETE FROM table” without a WHERE clause by dropping and recreating the table. (This is much faster than going through and deleting individual elements from the table.) Because of this optimization, the change count for “DELETE FROM table” will be zero regardless of the number of elements that were originally in the table. To get an accurate count of the number of rows deleted, use “DELETE FROM table WHERE 1” instead.

TotalChanges() As Integer
This method returns the total number of database rows that have be modified, inserted, or deleted since the database connection was created using the Open method. All changes are counted, including changes by triggers and changes to TEMP and auxiliary databases. Except, changes to the SQLITE_MASTER table (caused by statements such as CREATE TABLE) are not counted. Nor are changes counted when an entire table is deleted using DROP TABLE.

SQLite implements the command “DELETE FROM table” without a WHERE clause by dropping and recreating the table. (This is much faster than going through and deleting individual elements form the table.) Because of this optimization, the change count for “DELETE FROM table” will be zero regardless of the number of elements that were originally in the table. To get an accurate count of the number of rows deleted, use “DELETE FROM table WHERE 1” instead.

LastInsertRowID() As Integer
Each entry in an SQLite table has a unique integer key. (The key is the value of the INTE-

GER PRIMARY KEY column if there is such a column, otherwise the key is generated at random. The unique key is always available as the ROWID, OID, or _ROWID_ column.) This routine returns the integer key of the most recent insert in the database.

ErrCode() As Integer
Return the error code for the most recent failed function call. If a prior API call failed but the most recent API call succeeded, the return value from this routine is undefined.

Assuming no other intervening sqlite3 API calls are made, the error code returned by this function is associated with the same error as the strings returned by ErrMsg() method.

ErrMsg() As String
Return an UTF-8 encoded string describing in English the error condition for the most recent sqlite3* API call. The string “not an error” is returned when the most recent API call was successful.

GetAutoCommit() As Boolean
Test to see whether or not the database connection is in autocommit mode. Return TRUE if it is and FALSE if not. Autocommit mode is on by default. Autocommit is disabled by a BEGIN statement and reenabled by the next COMMIT or ROLLBACK.

Interrupt()
This function causes any pending database operation to abort and return at its earliest opportunity. This routine is typically called in response to a user action such as pressing “Cancel” or Ctrl-C where the user wants a long query operation to halt immediately.

Prepare (sql As String) As SQLite3VM
To execute an SQL query, it must first be compiled into a byte-code program using the Prepare method.
The first argument “sql” is the statement to be compiled.
This method returns an SQLite3VM instance that can be executed using the VMStep() method. Or if there is an error, nil is returned.

InstantQuery (sql As String, async As Boolean)
Start an instant query with the issued “sql” argument. If the async flag is true then method returns immediately while a background preemptive thread continue to execute the query. If the async flag is false then method returns only after the entire sql query finish its execution.

RowValue (index As Integer) As Integer
This is the way to get rows returned by the InstantQuery method. Range come from 0 to RowCount-1. The meaning of the returned value depends of the sql query issued to the InstantQuery method. If the sql statement was in the form “SELECT rowid FROM...” then the returned value is the rowid of the “index” row.

BusyTimeout (ms As Integer)
This routine sets a busy handler that sleeps for a while when a table is locked. The handler will sleep multiple times until at least “ms” milliseconds of sleeping have been done. After “ms” milliseconds of sleeping, the handler returns 0 which causes sqlite3.VMStep() to return SQLITE3.kBUSY.
Calling this method with an argument less than or equal to zero turns off all busy handlers.

CreateCollation (name As String, myRoutine As SQLite3CollationInterface)
These method is used to add new collation sequences to the sqlite3 database.
The first parameter is the name of the new collation sequence and the second parameter is a pointer to the user supplied routine that implements this collation. More in the SQLite3CollationInterface section. See Appendix A for more information about this method.

CreateScalarFunction (name As String, nArg As Integer, myRoutine As SQLite3ScalarFunctionInterface)

CreateAggregateFunction (name As String, nArg As Integer, myRoutine As SQLite3AggregateFunctionInterface)

These methods are used to add new scalar/aggregate functions to the sqlite3 database. The first parameter is the name of the new scalar/aggregate function, the second parameter is the number of arguments that the function or aggregate takes. If this argument is -1 then the function or aggregate may take any number of arguments. The last parameter is a pointer to the user supplied routine that implements this function. More in the SQLite3ScalarFunctionInterface and SQLite3AggregateFunctionInterface section. See Appendix A for more information about these methods.

TableColumnMetadata (tableName As String, columnName As String, ByRef declaredType As String, ByRef collationName As String, ByRef isNotNULL As Boolean, ByRef isPriKey As Boolean, ByRef isAutoInc As Boolean)

This routine is used to obtain meta information about a specific column of a specific database table. The column is identified by the first (tableName) and second (columnName) parameters to this function.
Meta information is returned by the ByRef parameters.

This function may load one or more schemas from database files. If an error occurs during this process, or if the requested table or column cannot be found, an SQLITE error code is returned and an error message left in the database handle.
If the specified column is “rowid”, “oid” or “_rowid_” and an INTEGER PRIMARY KEY column has been explicitly declared, then the output parameters are set for the explicitly declared column. If there is no explicitly declared IPK column, then the data-type is “INTEGER”, the collation sequence “BINARY” and the primary-key flag is set. Both the not-null and auto-increment flags are clear.

SQLite3VM Class

It is the class returned by the SQLite3Database.Prepare method. It defines methods necessary to interact with the powerful sqlite3 virtual machine.

Events

None

Properties

None

Methods

VMStep
Expired
Finalize
Reset
ClearBindings
TransferBinding
DataCount
BindBlob
BindDouble
BindInt
BindNull
BindText
BindParameterCount
BindParameterIndex
BindParameterName
ColumnBlob
ColumnBytes
ColumnDouble
ColumnInt
ColumnText
ColumnType
ColumnCount
ColumnDeclType
ColumnName

Methods

VMStep() As Integer
This method must be called one or more times to execute the statement.
The return value will be either SQLITE3.kBUSY, SQLITE3.kDONE, SQLITE3.kROW, SQLITE3.kERROR, or SQLITE3.kMISUSE.

SQLITE3.kBUSY means that the database engine attempted to open a locked database and there is no busy callback registered. Call VMStep() again to retry the open.

SQLITE3.kDONE means that the statement has finished executing successfully. VMStep should not be called again on this virtual machine without first calling the Reset() method to reset the virtual machine back to its initial state. If the SQL statement being executed returns any data, then SQLITE3.kROW is returned each time a new row of data is ready for processing by the caller. The values may be accessed using the Column*() methods. VMStep() is called again to retrieve the next row of data.

SQLITE3.kERROR means that a run-time error (such as a constraint violation) has occurred. VMStep() should not be called again on the VM. More information may be found by calling db.ErrMsg().

SQLITE3.kMISUSE means that the this routine was called inappropriately. Perhaps it was called on a virtual machine that had already been finalized or on one that had previously returned SQLITE3.kERROR or SQLITE3.kDONE. Or it could be the case that a database connection is being used by a different thread than the one it was created it.

Expired() As Boolean
Return TRUE if the statement supplied as an argument needs to be recompiled. A statement needs to be recompiled whenever the execution environment changes in a way that would alter the program that the Prepare method generates. For example, if new functions or collating sequences are registered or if an authorizer function is added or changed.

Finalize() As Integer
This method is called to delete a prepared SQL statement obtained by a previous call to db.Prepare. If the statement was executed successfully, or not executed at all, then SQLITE3.kOK is returned. If execution of the statement failed then an error code is returned.

This routine can be called at any point during the execution of the virtual the Interrupt method.) Incomplete updates may be rolled back and transactions canceled, depending on the circumstances, and the result code returned will be SQLITE3.kABORT.

Reset()
The method is called to reset a prepared SQL statement obtained by a previous back

to it’s initial state, ready to be re-executed. Any SQL statement variables that had values bound to them using the Bind*() method retain their values.

ClearBindings()
Set all the parameters in the compiled SQL statement back to NULL.

TransferBinding(dest As SQLite3VM)
Move all bindings from the first prepared statement over to the second. This routine is useful, for example, if the first prepared statement fails with an SQLITE3.kSCHEMA error. The same SQL can be prepared into the second prepared statement then all of the bindings transfered over to the second statement before the first statement is finalized.

DataCount() As Integer
Return the number of values in the current row of the result set.

After a call to VMStep() that returns SQLITE3.kROW, this routine will return the same value as the ColumnCount() function. After VMStep() has returned an SQLITE3.kDONE, SQLITE3.kBUSY or error code, or before VMStep() has been called on a prepared SQL statement, this routine returns zero.

BindBlob (index As Integer, value As String) As Integer
BindDouble (index As Integer, value As Double) As Integer
BindInt (index As Integer, value As Integer) As Integer
BindNull (index As Integer) As Integer
BindText (index As Integer, value As String) As Integer
In the SQL strings input to Prepare, one or more literals can be replace by a parameter “?” or “:AAA” or “$VVV” where AAA is an alphanumeric identifier and VVV is a variable name according to the syntax rules of the TCL programming language. The values of these parameters (also called “host parameter names”) can be set using the Bind*() routines.

The first argument to the Bind* routines always is the index of the parameter to be set. The first parameter has an index of 1. When the same named parameter is used more than once, second and subsequent occurrences have the same index as the first occurrence. The index for named parameters can be looked up using the BindParameterName API if desired.

The second argument is the value to bind to the parameter.

The Bind*() routines must be called after Prepare() or Reset() and before VMStep(). Bindings are not cleared by the Reset() routine. Unbound parameters are interpreted as NULL.
These routines return SQLITE3.kOK on success or an error code if anything goes wrong. SQLITE3.kRANGE is returned if the parameter index is out of range. SQLITE3.kNOMEM is returned if some memory allocation fails. SQLITE3.kMISUSE is returned

if these routines are called on a virtual machine that is the wrong state or which has already been finalized.

BindParameterCount() As Integer
Return the number of parameters in the precompiled statement given as the argument.

BindParameterIndex (pname As String) As Integer
Return the index of the parameter with the given name. The name must match exactly. If there is no parameter with the given name, return 0.

BindParameterName (index As Integer) As String)
Return the name of the n-th parameter in the precompiled statement. Parameters of the form ":AAA" or "$VVV" have a name which is the string ":AAA" or "$VVV". In other words, the initial ":" or "$" is included as part of the name. Parameters of the form "?" have no name.
If the value n is out of range or if the n-th parameter is nameless, then an empty string is returned.

ColumnBlob (columnIndex As Integer) As String
ColumnBytes (columnIndex As Integer) As Integer
ColumnDouble (columnIndex As Integer) As Double
ColumnInt (columnIndex As Integer) As Integer
ColumnText (columnIndex As Integer) As String
ColumnType (columnIndex As Integer) As Integer
These routines return information about the information in a single column of the current result row of a query. In every case the first argument is the index of the column for which information should be returned. iCol is zero-indexed. The left-most column has an index of 0.

If the SQL statement is not currently point to a valid row, or if the the column index is out of range, the result is undefined.

If the result is a BLOB then the ColumnBytes() method returns the number of bytes in that BLOB. No type conversions occur. If the result is a string (or a number since a number can be converted into a string) then ColumnBytes() converts the value into a UTF-8 string and returns the number of bytes in the resulting string.

These routines attempt to convert the value where appropriate. For example, if the internal representation is FLOAT and a text result is requested, then an internal conversion is done automatically.

ColumnCount() As Integer
Return the number of columns in the result set returned by the prepared SQL statement. This routine returns 0 if virtual machine is an SQL statement that does not return data (for example an UPDATE).

ColumnDeclType (columnIndex As Integer) As String
If the compiled statement is a SELECT statement, the Nth column of the returned result set of the SELECT is a table column then the declared type of the table column is returned. If the Nth column of the result set is not at table column, then a an empty string is returned. For example, in the database schema:

CREATE TABLE t1(c1 INTEGER);

And the following statement compiled:

SELECT c1 + 1, 0 FROM t1;

Then this routine would return the string “INTEGER” for the second result column (i==1), and aan empty string for the first result column (i==0).

ColumnName (columnIndex As Integer) As String
This method returns the column heading for the Nth column of the compiled statement, where N is the method parameter.

How to create new SQL functions

With SQLite3ProfessionalPlugin you can extend the sqlite3 engine with your own SQL functions written in REALbasic. Actually you can write three kind of functions.

Collation Sequence:
A collation sequence is a set of rules governing the characters that are used within a database and the means by which characters are sorted and compared. Once a new collation sequence has been defined, it can be used in any SQL statement, like CREATE, SELECT and so on. For example, if you define a NOCASE sequence, then you can use it as

CREATE TABLE t1(field1 TEXT COLLATE NOCASE, field2 TEXT)

In order to define a new collation sequence you have to use the method:
CreateCollation (name As String, myRoutine As SQLite3CollationInterface)

We strongly encourage you to check the example “SQLite3TestFunctions.rbp” in order to better understand this topic.

A very usefull link is:
http://www.sqlite.org/datatype3.html#collation

Scalar and Aggregate Functions:
A scalar function is an SQL function that is called only with columns of ONE row, while an aggregate function can be called by columns of an unlimited number fo rows.

In order to define new functions you have to use the methods:
CreateScalarFunction (name As String, nArg As Integer, myRoutine As SQLite3ScalarFunctionInterface)

CreateAggregateFunction (name As String, nArg As Integer, myRoutine As SQLite3AggregateFunctionInterface)

We strongly encourage you to check the example “SQLite3TestFunctions.rbp” and “SQLite3TestAggregate.rbp“ in order to better understand this topic.

Some very usefull links are:
http://databases.about.com/od/sql/l/aaaggregate1.htm
http://linuxgazette.net/109/chirico1.html

APPENDIX B

Contact Information

Marco Bambini
marco@sqlabs.net

Web: http://www.sqlabs.net
Email: info@sqlabs.net

Copyright

All materials are copyright 2003-2006 by SQLabs.
All Rights Reserved. SQLite3ProfessionalPlugin may be freely distributed, so long as it is not sold for profit, and registration serial numbers are not distributed. Express permission is granted to online services and other shareware/public domain distribution avenues to carry SQLite3ProfessionalPlugin.

Express permission is further granted to include SQLite3ProfessionalPlugin on CD-ROMs or floppy disks accompanying books, or on shareware collections, provided that no more than a nominal compilation and/or media fee is charged for these collections.

Legal Stuff

Unregistered copies may be used and evaluated in the REALbasic IDE and in debug compilations, but may not be used in final, standalone applications. Copies are registered per individual developer and may be used by that developer, royalty-free, in an unlimited number of applications, commercial or otherwise. Once obtained, licenses may not be transferred to other individuals or organizations.

SQLabs reserves the right to revoke the license of anyone who ignores or violates these restrictions. See our web site at http://www.sqlabs.net/ for registration information. Company licenses are available.

SQLite3ProfessionalPlugin is distributed AS IS. There is no warranty of any kind as to its fitness for any purpose. The risks associated with the use of this product are borne by the user in their entirety. In other words, the SQLite3ProfessionalPlugin, although it is in no way designed to do so, could be capable of ruining your software, crashing your computer and erasing your hard drive. Marco Bambini and SQLabs take no responsibility for these or other consequences.

REALbasic® it is a registered trademark of REAL Software, Inc. See their web site at http://www.realsoftware.com for more details. SQLabs is not way affiliated with REAL Software. All questions regarding REALbasic should be directed to REAL Software, Inc.