# Go Database SQL Explained In Color

REVISION 1

Hawthorne-Press.com

Go Database SQL Explained in Color

Published by

Hawthorne-Press.com
916 Adele Street
Houston, Texas 77009, USA

## Table of Contents



# Go Database SQL Explained in Color

## Introduction

Databases are integral to the functioning of most modern programs. They store and retrieve data in a multitude of formats, though strings and numbers are the most common. Further, the common method for manipulating databases is SQL, or *Structured Query Language.*

While SQL is available for most databases, in this document our examples will be built using the open source Sqlite3 Database. Our discussion will involve several ways to approach working with this databases and SQL.

## Driver Packages

There two prominent driver packages used Sqlite3:

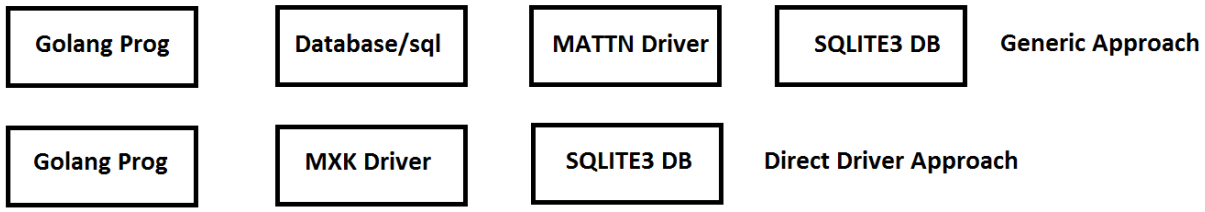
- *"github.com/mattn/go-sqlite3"* -- Recommended as driver for **database/sql** package.
- *"github.com/mxk/go-sqlite/sqlite3"* -- Recommended for "Direct Drive".

This document covers using both driver packages. The driver packages can be loaded as follows:

**go get github.com/mattn/go-sqlite3**

**go get github.com/mxk/go-sqlite/sqlite3**

The following diagrams outline the two approaches for working with a Sqlite3:




Using the "Direct Drive" approach allows the user to access Sqlite3 features that are not available via the **database/sql** approach, such *incremental I/O* and online *Backups*.

## Structured Query Language

Structured Query Language is a combination of a *data definition*, *data manipulation*, and *data control language*. While it was originally based on *relational algebra* and *tuple relational calculus*, it has evolved, some would say strayed, from its original roots. Whatever these theoretical roots, we will be describing what is current practice and usage.

This document is not meant to be a tutorial on SQL, rather a short introduction into using databases based on SQL processing in Go Language programs. As stated above we will be using Sqlite3 as our database for all examples.

# SQLITE3

Sqlite3 was chosen for demonstrating Go Language database concepts for a number of reasons:

- Sqlite3 is an open source database that is freely available on a cross-platform basis.
- It is does not have any external dependencies, meaning that it is self-contained.
- The complete database is stored in single cross-platform disk file.
- It does not require a separate server, setup, or administration.
- While not a large complex database, it supports most of SQL92 standard.

The primary SQL commands can be classified into three groups based on their operational nature:

## DDL – DATA DEFINITION LANGUAGE

- CREATE – Creates a new table, table view, or other object in the database.
- ALTER – Modifies an existing database object.
- DROP – Deletes an existing database object.

## DML – DATA MANIPULATION LANGUAGE

- INSERT – Creates a record.
- UPDATE – Modifies records.
- DELETE – Deletes a record.

## DQL – DATA QUERY LANGUAGE

- SELECT – Retrieves certain records from one or more tables

# DATABASE/SQL PACKAGE

The package **database/sql** provides a generic interface to many different databases. It expects a driver to be provided for each specific database. For Sqlite3 the recommended driver is *“github.com/mattn/go-sqlite3”.*

This document only covers an outline of this package. It has multitude of features, and many are beyond the scope of this document. We hope that this document will provide a basis for experimenting with this and other databases for your projects. Most modern programs should have evolved beyond using “flat” files for major data storage.

We will start with a basic database program shameless lifted from https***://github.com/siongui/userpages.***

This program consists of a library of database functions in **sqlite3.go** and a unit test program in **sqlite3_test.go.**

***Note to window users: The package TDM64-GCC must be installed for the examples to run. GCC is still needed by some packages, even with GO 1.6.***

## SQLITE3.GO – USER LIBRARY FUNCTIONS

Testing the basic functions of the **database/sql** package will require creating a test library *sqlite3.go* and a unit test program *sqlite3_test.go*. The unit test program will verify that the test library performs correctly.

The test library is comprised of the following functions:

- InitDB(filepath) *sql.DB -- Creates the database file and returns a Connection Pointer to the Database.
- CreateTable(db *sql.DB) -- Creates a test table.
- StoreItem(db *sql.DB, items []TestItem) -- Stores values in TestItem Slice into Table.
- ReadItem(db *sql.DB) []TestItem – Read all Rows in Table into slice TestItem.

```
package mylib

import (
	"database/sql"                                  // Generic SQL Package
	_ "github.com/mattn/go-sqlite3"                 // The "_" imports Pkg only for initialization
)

type TestItem struct {                              // Structure of Table Values
	Id      string                                  // ID String
	Name    string                                  // Name String
	Phone   string                                  // PHONE Number String
}

func InitDB(filepath string) *sql.DB {              // Init Database
	db, err := sql.Open("sqlite3", filepath)        // Open Database
	if err != nil { panic(err) }                    // Panic on Errors
	if db == nil { panic("db nil") }
	return db                                       // Return Connection
}

func CreateTable(db *sql.DB) {                      // Create Database Table
	// create table if not exists
	sql_table := `                                  // Create Command String
	CREATE TABLE IF NOT EXISTS items(               // Table name is "items"
		Id TEXT NOT NULL PRIMARY KEY,               // ID must be specified and is the Primary Key
		Name TEXT,
		Phone TEXT,
		InsertedDatetime DATETIME                   // Insert Time Stamp for record
	);
	`

	_, err := db.Exec(sql_table)                    // Execute Command String
	if err != nil { panic(err) }                    // Panic on Error
}

func StoreItem(db *sql.DB, items []TestItem) {      // Store Data in Table Rows from slice TestItem
	sql_additem := `                                // Create Command String
	INSERT OR REPLACE INTO items(
		Id,
		Name,
		Phone,
		InsertedDatetime
	) values(?, ?, ?, CURRENT_TIMESTAMP)
	`

	stmt, err := db.Prepare(sql_additem)            // Prepare Statement object
	if err != nil { panic(err) }
	defer stmt.Close()                              // Defer Statement Close

	for _, item := range items {                    // Execute Statement for each row in items
		_, err2 := stmt.Exec(item.Id, item.Name, item.Phone)   // Store Contents in each row
		if err2 != nil { panic(err2) }                        // Panic on Error
```

```
        }
}

func ReadItem(db *sql.DB) []TestItem {              // Read Data from Database
        sql_readall := `                            // Create Command String
        SELECT Id, Name, Phone FROM items
        ORDER BY datetime(InsertedDatetime) DESC
        `

        rows, err := db.Query(sql_readall)          // Read Data into Result Set "rows"
        if err != nil { panic(err) }                // Panic if error
        defer rows.Close()                          // Defer Statement Close


        var result []TestItem                               // Create Slice to hold output data
        for rows.Next() {                                   // Step through Rows in Result Set
                item := TestItem{}                          // Create item, a TestItem structure
                err2 := rows.Scan(&item.Id, &item.Name, &item.Phone) // Load row into "item"
                if err2 != nil { panic(err2) }              // Panic if error
                result = append(result, item)               // Append item to Output Slice
        }
        return result                                       // Return Slice of row data
}
```

## Program Data Structures

The basic data structure is *TestItem*, which contains the three data values stored in each row in the database table. An additional value, a timestamp, is also stored in the each row. This value created when the rows are stored, but it is not returned with the data by the read function. This is just a design decision, it could be returned if the program developer wanted this value be available.

```
type TestItem struct {                              // Structure of Table Values
        Id      string                              // ID String
        Name    string                              // Name String
        Phone   string                              // PHONE Number String
}
```

The inputs to the store function and the outputs of the read function are defined as a slice of []*TestItem*. Thus, this test program can read and write multiple rows of data at a time. If you examine the program, the store function *StoreItems()* can store different items as many times as it is called. However, the read function *ReadItems()* always returns the entire contents of table *items* as Result Set.

## sqlite3_test.go – Unit Test Program

The purpose to this program is to test the four functions defined in **sqlite3.go**, with the package name *mylib.* This is a very simplest test because it only insures that the functions do not panic. The user must inspect the logged output to determine if the results are correct. A more complete unit test would insure that the returned values match the expected results by comparing the output to a defined set of values.

The reader should expand the following unit test program as an exercise.

```
package mylib

import "testing"

func TestAll(t *testing.T) {                // Unit Test Function
    const dbpath = "foo.db"                 // Specify Database Filename (Will be in local Dir)
```

```
    db := InitDB(dbpath)                    // Create Database (Empty)
    defer db.Close()                        // Defer Database Close
    CreateTable(db)                         // Create Table (Empty)

    items := []TestItem{                    // Create Table Entry (Two Items)
        TestItem{"1", "A", "213"},          // ID, Name, Phone Data
        TestItem{"2", "B", "214"},
    }
    StoreItem(db, items)                    // Store Items

    readItems := ReadItem(db)               // Read Items
    t.Log(readItems)                        // Log returned results



    items2 := []TestItem{                   // Create Table Entry (two items, will overwrite ID 1)
        TestItem{"1", "C", "215"},          // ID, Name, Phone Data  (Replaces ID == 1)
        TestItem{"3", "D", "216"},          // ID, Name, Phone Data  (New item)
    }
    StoreItem(db, items2)                   // Store item data

    readItems2 := ReadItem(db)              // Read Items
    t.Log(readItems2)                       // Log returned Results
}
```

The results of running this test on a Windows 7 OS are as follows:

**=== RUN TestAll**
**--- PASS: TestAll (0.08s)**
**sq;ite3_test.go:19: [{1 A 213} {2 B 214}]**
**sq;ite3_test.go:28: [{2 B 214} {1 C 215} {3 D 216}]**
**PASS**
**ok 0A_SQL_TESTING/sqlite3-test 0.462s**

On an Ubuntu 12.04 System, it only logged the two lines:

**PASS**
**ok  0A_SQL_TEST/sqlite3  0.444s**

Depending on the version of Go and the OS used, results may be slightly different. However, unit testing should always pass.

To replicate these results user must always delete the database file **foo.db** before running, if it is present.

## Direct Driver Interface

The primary reason for using a “Direct Drive” interface is to access database specific features. In the case of Sqlite3, these include *Incremental I/O* and *Online Backup* features to name two. But by making this choice as a developer, you are “locking” the program to a specific database. This should be a carefully considered decision since it has long-term consequences for your project and possibly your employer.

To make the comparison of these two approaches, we have rewritten the program **sqlite3.go** library to use the driver “m*xk/go-sqlite/sqlite3”* instead of the **database/sql** package.

Since these programs are almost identical, we will only insert comments to highlight the differences.

```
package driveLib                                    // Different Library Name

import (
    "github.com/mxk/go-sqlite/sqlite3"              // MXK SQLITE3 Driver instead of database/sql
)

type TestItem struct {
    Id    string
    Name  string
    Phone string
}

func InitDB(filepath string) *sqlite3.Conn {        // *sqlite3.Conn instead of *sql.DB
    db, err := sqlite3.Open(filepath)               // Uses Open() from driver library
    if err != nil {
        panic(err)
    }
    if db == nil {
        panic("db nil")
    }
    return db
}

func CreateTable(db *sqlite3.Conn) {                // *sqlite3.Conn instead of *sql.DB
    // create table if not exists
    sql_table := `
    CREATE TABLE IF NOT EXISTS items(
        Id TEXT NOT NULL PRIMARY KEY,
        Name TEXT,
        Phone TEXT,
        InsertedDatetime DATETIME
    );
    `

    err := db.Exec(sql_table)                       // Use Exec() from driver library
    if err != nil {
        panic(err)
    }
}

func StoreItem(db *sqlite3.Conn, items []TestItem) { // Uses *sqlite.Conn instead of *sql.DB
    sql_additem := `
    INSERT OR REPLACE INTO items(
        Id,
        Name,
        Phone,
        InsertedDatetime
    ) values(?, ?, ?, CURRENT_TIMESTAMP)
    `

    stmt, err := db.Prepare(sql_additem)                    // Uses Prepare() from driver library
    if err != nil {
        panic(err)
    }
    defer stmt.Close()


    for _, item := range items {
        err2 := stmt.Exec(item.Id, item.Name, item.Phone)   // Uses Exec() from driver library
        if err2 != nil {
            panic(err2)
        }
    }
}

func ReadItem(db *sqlite3.Conn) []TestItem {                // *sqlite3.Conn instead of *sql.DB
```

```
    sql_readall := `
    SELECT Id, Name, Phone FROM items
    ORDER BY datetime(InsertedDatetime) DESC
    `

    rows, err := db.Query(sql_readall)                      // Uses Query() from driver library
    if err != nil {
        panic(err)
    }
    defer rows.Close()
    var result []TestItem

    for ; err == nil; err = rows.Next() {                   // rows.Next() returns error value!
        item := TestItem{}
        err2 := rows.Scan(&item.Id, &item.Name, &item.Phone)// Uses rows.Scan() from driver library
        if err2 != nil {
            panic(err2)
        }
        result = append(result, item)
    }
    return result
}
```

As you can see from our simple example, there is not that much different. However, with larger programs changing the database will probably be more complicated depending on the differences between the databases being swapped. Do not select direct driver usage lightly.

Now for the good part, the only change necessary to the unit test program is to change the package name on the first line, see below. Since the library names are the same and they have the same signatures, the unit test works without any other changes.

```
package driveLib                            // Change Library being tested

import "testing"

func TestAll(t *testing.T) { … }
```

## Conclusions

The examples shown, primarily the **database/sql** version, will give a developer the tools to incorporate a database into many programs. The vast majority of intermediate programs store data in a couple of relatively simple tables. Just the operations shown will suffice for many programs. As the developer becomes more familiar with SQL and databases, they will find more uses for databases in their programs.

Study the SQL Statements, highlighted in green since they are represented as strings, and lookup other examples of using the three major operations, CREATE, INSERT, and QUERY. These allow the creation of a table, inserting data into a table, returning the data from one or more rows in a table.

## A Final Word

Strive for simplicity in your code. The *Occam's Razor principle* applies as strongly for software as it does for many other walks of life. In most situations, giving up a little in efficiency to simplify your code is often the right decision. Code Simplicity reduces the chances of subtle errors.

As always, comments are welcome at .

As to why I have an email address on **gmail** instead of on my website, has to do with AT&T and their inability to handle the concept that not everyone is running a Microsoft OS and my stubborn insistence on running Linux.

Published by

Hawthorne-Press.com
10310 Moorberry Lane
Houston, Texas 77043, USA

Revision History

| Date | By | Section | Changes |
|---|---|---|---|
| **07/14/2016** | C.E. Thornton | All | Initial Document |
| **05/27/2018** | C.E.Thornton | Copyright | Update Address |
| | | | |