

Discover Packages > Standard library > database > sql 

sql




package


standard library

Version: [go1.20.1](#) **Latest** | Published: Feb 14, 2023 | License: [BSD-3-Clause](#) | Imports: [15](#) |

Imported by: [71,104](#)

Details

 Valid [go.mod](#) file   Redistributable license   Tagged version 

 Stable version 

[Learn more](#)

Repository

cs.opensource.google/go/go

Links

 [Report a Vulnerability](#)

 Documentation 

<> Documentation

Overview

Package sql provides a generic interface around SQL (or SQL-like) databases.

The sql package must be used in conjunction with a database driver. See <https://golang.org/s/sqldrivers> for a list of drivers.

Drivers that do not support context cancellation will not return until after the query is completed.

For usage examples, see the wiki page at <https://golang.org/s/sqlwiki>.

► [Example \(OpenDBCLI\)](#)

► [Example \(OpenDBService\)](#)

Index

Variables

[func Drivers\(\) \[\]string](#)

[func Register\(name string, driver driver.Driver\)](#)

type ColumnType

[func \(ci *ColumnType\) DatabaseTypeName\(\) string](#)

[func \(ci *ColumnType\) DecimalSize\(\) \(precision, scale int64, ok bool\)](#)

[func \(ci *ColumnType\) Length\(\) \(length int64, ok bool\)](#)

```
func (ci *ColumnType) Name() string
func (ci *ColumnType) Nullable() (nullable, ok bool)
func (ci *ColumnType) ScanType() reflect.Type
```

type Conn

```
func (c *Conn) BeginTx(ctx context.Context, opts *TxOptions) (*Tx, error)
func (c *Conn) Close() error
func (c *Conn) ExecContext(ctx context.Context, query string, args ...any) (Result, error)
func (c *Conn) PingContext(ctx context.Context) error
func (c *Conn) PrepareContext(ctx context.Context, query string) (*Stmt, error)
func (c *Conn) QueryContext(ctx context.Context, query string, args ...any) (*Rows, error)
func (c *Conn) QueryRowContext(ctx context.Context, query string, args ...any) *Row
func (c *Conn) Raw(f func(driverConn any) error) (err error)
```

type DB

```
func Open(driverName, dataSourceName string) (*DB, error)
func OpenDB(c driver.Connector) *DB
func (db *DB) Begin() (*Tx, error)
func (db *DB) BeginTx(ctx context.Context, opts *TxOptions) (*Tx, error)
func (db *DB) Close() error
func (db *DB) Conn(ctx context.Context) (*Conn, error)
func (db *DB) Driver() driver.Driver
func (db *DB) Exec(query string, args ...any) (Result, error)
func (db *DB) ExecContext(ctx context.Context, query string, args ...any) (Result, error)
func (db *DB) Ping() error
func (db *DB) PingContext(ctx context.Context) error
func (db *DB) Prepare(query string) (*Stmt, error)
func (db *DB) PrepareContext(ctx context.Context, query string) (*Stmt, error)
func (db *DB) Query(query string, args ...any) (*Rows, error)
func (db *DB) QueryContext(ctx context.Context, query string, args ...any) (*Rows, error)
func (db *DB) QueryRow(query string, args ...any) *Row
func (db *DB) QueryRowContext(ctx context.Context, query string, args ...any) *Row
func (db *DB) SetConnMaxIdleTime(d time.Duration)
func (db *DB) SetConnMaxLifetime(d time.Duration)
func (db *DB) SetMaxIdleConns(n int)
func (db *DB) SetMaxOpenConns(n int)
func (db *DB) Stats() DBStats
```

type DBStats

type IsolationLevel

```
func (i IsolationLevel) String() string
```

type NamedArg

```
func Named(name string, value any) NamedArg
```

type NullBool

```
func (n *NullBool) Scan(value any) error
func (n NullBool) Value() (driver.Value, error)
```

type NullByte

```
func (n *NullByte) Scan(value any) error
func (n NullByte) Value() (driver.Value, error)
```

type NullFloat64

func (n *NullFloat64) Scan(value any) error

func (n NullFloat64) Value() (driver.Value, error)

type NullInt16

func (n *NullInt16) Scan(value any) error

func (n NullInt16) Value() (driver.Value, error)

type NullInt32

func (n *NullInt32) Scan(value any) error

func (n NullInt32) Value() (driver.Value, error)

type NullInt64

func (n *NullInt64) Scan(value any) error

func (n NullInt64) Value() (driver.Value, error)

type NullString

func (ns *NullString) Scan(value any) error

func (ns NullString) Value() (driver.Value, error)

type NullTime

func (n *NullTime) Scan(value any) error

func (n NullTime) Value() (driver.Value, error)

type Out

type RawBytes

type Result

type Row

func (r *Row) Err() error

func (r *Row) Scan(dest ...any) error

type Rows

func (rs *Rows) Close() error

func (rs *Rows) ColumnTypes() ([]*ColumnType, error)

func (rs *Rows) Columns() ([]string, error)

func (rs *Rows) Err() error

func (rs *Rows) Next() bool

func (rs *Rows) NextResultSet() bool

func (rs *Rows) Scan(dest ...any) error

type Scanner

type Stmt

func (s *Stmt) Close() error

func (s *Stmt) Exec(args ...any) (Result, error)

func (s *Stmt) ExecContext(ctx context.Context, args ...any) (Result, error)

func (s *Stmt) Query(args ...any) (*Rows, error)

func (s *Stmt) QueryContext(ctx context.Context, args ...any) (*Rows, error)

func (s *Stmt) QueryRow(args ...any) *Row

func (s *Stmt) QueryRowContext(ctx context.Context, args ...any) *Row

type Tx

func (tx *Tx) Commit() error

func (tx *Tx) Exec(query string, args ...any) (Result, error)

func (tx *Tx) ExecContext(ctx context.Context, query string, args ...any) (Result, error)

func (tx *Tx) Prepare(query string) (*Stmt, error)

```
func (tx *Tx) PrepareContext(ctx context.Context, query string) (*Stmt, error)
func (tx *Tx) Query(query string, args ...any) (*Rows, error)
func (tx *Tx) QueryContext(ctx context.Context, query string, args ...any) (*Rows, error)
func (tx *Tx) QueryRow(query string, args ...any) *Row
func (tx *Tx) QueryRowContext(ctx context.Context, query string, args ...any) *Row
func (tx *Tx) Rollback() error
func (tx *Tx) Stmt(stmt *Stmt) *Stmt
func (tx *Tx) StmtContext(ctx context.Context, stmt *Stmt) *Stmt

type TxOptions
```

Examples

```
Package (OpenDBCLI)
Package (OpenDBService)
Conn.ExecContext
DB.BeginTx
DB.ExecContext
DB.PingContext
DB.Prepare
DB.Query (MultipleResultSets)
DB.QueryContext
DB.QueryRowContext
Rows
Stmt
Stmt.QueryRowContext
Tx.ExecContext
Tx.Prepare
Tx.Rollback
```

Constants

This section is empty.

Variables

[View Source](#)

```
var ErrConnDone = errors.New("sql: connection is already closed")
```

ErrConnDone is returned by any operation that is performed on a connection that has already been returned to the connection pool.

[View Source](#)

```
var ErrNoRows = errors.New("sql: no rows in result set")
```

ErrNoRows is returned by Scan when QueryRow doesn't return a row. In such a case, QueryRow returns a placeholder *Row value that defers this error until a Scan.

[View Source](#)

```
var ErrTxDone = errors.New("sql: transaction has already been committed or rolled back")
```

ErrTxDone is returned by any operation that is performed on a transaction that has already been committed or rolled back.

Functions

func Drivers

added in go1.4

```
func Drivers() []string
```

Drivers returns a sorted list of the names of the registered drivers.

func Register

```
func Register(name string, driver driver.Driver)
```

Register makes a database driver available by the provided name. If Register is called twice with the same name or if driver is nil, it panics.

Types

type ColumnType

added in go1.8

```
type ColumnType struct {  
    // contains filtered or unexported fields  
}
```

ColumnType contains the name and type of a column.

func (*ColumnType) DatabaseTypeName

added in go1.8

```
func (ci *ColumnType) DatabaseTypeName() string
```

DatabaseTypeName returns the database system name of the column type. If an empty string is returned, then the driver type name is not supported. Consult your driver documentation for a list of driver data types. Length specifiers are not included. Common type names include "VARCHAR", "TEXT", "NVARCHAR", "DECIMAL", "BOOL", "INT", and "BIGINT".

func (*ColumnType) DecimalSize

added in go1.8

```
func (ci *ColumnType) DecimalSize() (precision, scale int64, ok bool)
```

DecimalSize returns the scale and precision of a decimal type. If not applicable or if not supported ok is false.

func (*ColumnType) Length

added in go1.8

```
func (ci *ColumnType) Length() (length int64, ok bool)
```

Length returns the column type length for variable length column types such as text and binary field types. If the type length is unbounded the value will be `math.MaxInt64` (any database limits will still apply). If the column type is not variable length, such as an int, or if not supported by the driver ok is false.

func (*ColumnType) Name

added in go1.8

```
func (ci *ColumnType) Name() string
```

Name returns the name or alias of the column.

func (*ColumnType) Nullable

added in go1.8

```
func (ci *ColumnType) Nullable() (nullable, ok bool)
```

Nullable reports whether the column may be null. If a driver does not support this property ok will be false.

func (*ColumnType) ScanType

added in go1.8

```
func (ci *ColumnType) ScanType() reflect.Type
```

ScanType returns a Go type suitable for scanning into using `Rows.Scan`. If a driver does not support this property ScanType will return the type of an empty interface.

type Conn

added in go1.9

```
type Conn struct {  
    // contains filtered or unexported fields  
}
```

Conn represents a single database connection rather than a pool of database connections. Prefer running queries from DB unless there is a specific need for a continuous single database connection.

A Conn must call `Close` to return the connection to the database pool and may do so concurrently with a running query.

After a call to `Close`, all operations on the connection fail with `ErrConnDone`.

func (*Conn) BeginTx

added in go1.9

```
func (c *Conn) BeginTx(ctx context.Context, opts *TxOptions) (*Tx, error)
```

BeginTx starts a transaction.

The provided context is used until the transaction is committed or rolled back. If the context is canceled, the sql package will roll back the transaction. Tx.Commit will return an error if the context provided to BeginTx is canceled.

The provided TxOptions is optional and may be nil if defaults should be used. If a non-default isolation level is used that the driver doesn't support, an error will be returned.

func (*Conn) Close

added in go1.9

```
func (c *Conn) Close() error
```

Close returns the connection to the connection pool. All operations after a Close will return with ErrConnDone. Close is safe to call concurrently with other operations and will block until all other operations finish. It may be useful to first cancel any used context and then call close directly after.

func (*Conn) ExecContext

added in go1.9

```
func (c *Conn) ExecContext(ctx context.Context, query string, args ...any) (Result, error)
```

ExecContext executes a query without returning any rows. The args are for any placeholder parameters in the query.

► [Example](#)

func (*Conn) PingContext

added in go1.9

```
func (c *Conn) PingContext(ctx context.Context) error
```

PingContext verifies the connection to the database is still alive.

func (*Conn) PrepareContext

added in go1.9

```
func (c *Conn) PrepareContext(ctx context.Context, query string) (*Stmt, error)
```

PrepareContext creates a prepared statement for later queries or executions. Multiple queries or executions may be run concurrently from the returned statement. The caller must call the statement's Close method when the statement is no longer needed.

The provided context is used for the preparation of the statement, not for the execution of the statement.

func (*Conn) QueryContext

added in go1.9

```
func (c *Conn) QueryContext(ctx context.Context, query string, args ...any) (*Rows, error)
```

QueryContext executes a query that returns rows, typically a SELECT. The args are for any placeholder parameters in the query.

func (*Conn) QueryRowContext

added in go1.9

```
func (c *Conn) QueryRowContext(ctx context.Context, query string, args ...any) *Row
```

QueryRowContext executes a query that is expected to return at most one row. QueryRowContext always returns a non-nil value. Errors are deferred until Row's Scan method is called. If the query selects no rows, the *Row's Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

func (*Conn) Raw

added in go1.13

```
func (c *Conn) Raw(f func(driverConn any) error) (err error)
```

Raw executes f exposing the underlying driver connection for the duration of f. The driverConn must not be used outside of f.

Once f returns and err is not driver.ErrBadConn, the Conn will continue to be usable until Conn.Close is called.

type DB

```
type DB struct {  
    // contains filtered or unexported fields  
}
```

DB is a database handle representing a pool of zero or more underlying connections. It's safe for concurrent use by multiple goroutines.

The sql package creates and frees connections automatically; it also maintains a free pool of idle connections. If the database has a concept of per-connection state, such state can be reliably observed within a transaction (Tx) or connection (Conn). Once DB.Begin is called, the returned Tx is bound to a single connection. Once Commit or Rollback is called on the transaction, that transaction's connection is returned to DB's idle connection pool. The pool size can be controlled with SetMaxIdleConns.

func Open

```
func Open(driverName, dataSourceName string) (*DB, error)
```

Open opens a database specified by its database driver name and a driver-specific data source name, usually consisting of at least a database name and connection information.

Most users will open a database via a driver-specific connection helper function that returns a *DB. No database drivers are included in the Go standard library. See <https://golang.org/s/sqldrivers> for a list of third-party drivers.

Open may just validate its arguments without creating a connection to the database. To verify that the data source name is valid, call Ping.

The returned DB is safe for concurrent use by multiple goroutines and maintains its own pool of idle connections. Thus, the Open function should be called just once. It is rarely necessary to close a DB.

func OpenDB

added in go1.10

```
func OpenDB(c driver.Connector) *DB
```

OpenDB opens a database using a Connector, allowing drivers to bypass a string based data source name.

Most users will open a database via a driver-specific connection helper function that returns a *DB. No database drivers are included in the Go standard library. See <https://golang.org/s/sqldrivers> for a list of third-party drivers.

OpenDB may just validate its arguments without creating a connection to the database. To verify that the data source name is valid, call Ping.

The returned DB is safe for concurrent use by multiple goroutines and maintains its own pool of idle connections. Thus, the OpenDB function should be called just once. It is rarely necessary to close a DB.

func (*DB) Begin

```
func (db *DB) Begin() (*Tx, error)
```

Begin starts a transaction. The default isolation level is dependent on the driver.

Begin uses context.Background internally; to specify the context, use BeginTx.

func (*DB) BeginTx

added in go1.8

```
func (db *DB) BeginTx(ctx context.Context, opts *TxOptions) (*Tx, error)
```

BeginTx starts a transaction.

The provided context is used until the transaction is committed or rolled back. If the context is canceled, the sql package will roll back the transaction. Tx.Commit will return an error if the context provided to BeginTx is canceled.

The provided TxOptions is optional and may be nil if defaults should be used. If a non-default isolation level is used that the driver doesn't support, an error will be returned.

► Example

func (*DB) Close

```
func (db *DB) Close() error
```

Close closes the database and prevents new queries from starting. Close then waits for all queries that have started processing on the server to finish.

It is rare to Close a DB, as the DB handle is meant to be long-lived and shared between many goroutines.

func (*DB) Conn

added in go1.9

```
func (db *DB) Conn(ctx context.Context) (*Conn, error)
```

Conn returns a single connection by either opening a new connection or returning an existing connection from the connection pool. Conn will block until either a connection is returned or ctx is canceled. Queries run on the same Conn will be run in the same database session.

Every Conn must be returned to the database pool after use by calling Conn.Close.

func (*DB) Driver

```
func (db *DB) Driver() driver.Driver
```

Driver returns the database's underlying driver.

func (*DB) Exec

```
func (db *DB) Exec(query string, args ...any) (Result, error)
```

Exec executes a query without returning any rows. The args are for any placeholder parameters in the query.

Exec uses context.Background internally; to specify the context, use ExecContext.

func (*DB) ExecContext

added in go1.8

```
func (db *DB) ExecContext(ctx context.Context, query string, args ...any) (Result, error)
```

ExecContext executes a query without returning any rows. The args are for any placeholder parameters in the query.

► [Example](#)

func (*DB) Ping

added in go1.1

```
func (db *DB) Ping() error
```

Ping verifies a connection to the database is still alive, establishing a connection if necessary.

Ping uses context.Background internally; to specify the context, use PingContext.

func (*DB) PingContext

added in go1.8

```
func (db *DB) PingContext(ctx context.Context) error
```

PingContext verifies a connection to the database is still alive, establishing a connection if necessary.

► [Example](#)

func (*DB) Prepare

```
func (db *DB) Prepare(query string) (*Stmt, error)
```

Prepare creates a prepared statement for later queries or executions. Multiple queries or executions may be run concurrently from the returned statement. The caller must call the statement's Close method when the statement is no longer needed.

Prepare uses context.Background internally; to specify the context, use PrepareContext.

► [Example](#)

func (*DB) PrepareContext

added in go1.8

```
func (db *DB) PrepareContext(ctx context.Context, query string) (*Stmt, error)
```

PrepareContext creates a prepared statement for later queries or executions. Multiple queries or executions may be run concurrently from the returned statement. The caller must call the statement's Close method when the statement is no longer needed.

The provided context is used for the preparation of the statement, not for the execution of the statement.

func (*DB) Query

```
func (db *DB) Query(query string, args ...any) (*Rows, error)
```

Query executes a query that returns rows, typically a SELECT. The args are for any placeholder parameters in the query.

Query uses context.Background internally; to specify the context, use QueryContext.

► [Example \(MultipleResultSets\)](#)

func (*DB) QueryContext

added in go1.8

```
func (db *DB) QueryContext(ctx context.Context, query string, args ...any) (*Rows, error)
```

QueryContext executes a query that returns rows, typically a SELECT. The args are for any placeholder parameters in the query.

► Example

func (*DB) QueryRow

```
func (db *DB) QueryRow(query string, args ...any) *Row
```

QueryRow executes a query that is expected to return at most one row. QueryRow always returns a non-nil value. Errors are deferred until Row's Scan method is called. If the query selects no rows, the *Row's Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

QueryRow uses context.Background internally; to specify the context, use QueryRowContext.

func (*DB) QueryRowContext

added in go1.8

```
func (db *DB) QueryRowContext(ctx context.Context, query string, args ...any) *Row
```

QueryRowContext executes a query that is expected to return at most one row. QueryRowContext always returns a non-nil value. Errors are deferred until Row's Scan method is called. If the query selects no rows, the *Row's Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

► Example

func (*DB) SetConnMaxIdleTime

added in go1.15

```
func (db *DB) SetConnMaxIdleTime(d time.Duration)
```

SetConnMaxIdleTime sets the maximum amount of time a connection may be idle.

Expired connections may be closed lazily before reuse.

If $d \leq 0$, connections are not closed due to a connection's idle time.

func (*DB) SetConnMaxLifetime

added in go1.6

```
func (db *DB) SetConnMaxLifetime(d time.Duration)
```

SetConnMaxLifetime sets the maximum amount of time a connection may be reused.

Expired connections may be closed lazily before reuse.

If $d \leq 0$, connections are not closed due to a connection's age.

func (*DB) SetMaxIdleConns

added in go1.1

```
func (db *DB) SetMaxIdleConns(n int)
```

SetMaxIdleConns sets the maximum number of connections in the idle connection pool.

If MaxOpenConns is greater than 0 but less than the new MaxIdleConns, then the new MaxIdleConns will be reduced to match the MaxOpenConns limit.

If $n \leq 0$, no idle connections are retained.

The default max idle connections is currently 2. This may change in a future release.

func (*DB) SetMaxOpenConns

added in go1.2

```
func (db *DB) SetMaxOpenConns(n int)
```

SetMaxOpenConns sets the maximum number of open connections to the database.

If MaxIdleConns is greater than 0 and the new MaxOpenConns is less than MaxIdleConns, then MaxIdleConns will be reduced to match the new MaxOpenConns limit.

If $n \leq 0$, then there is no limit on the number of open connections. The default is 0 (unlimited).

func (*DB) Stats

added in go1.5

```
func (db *DB) Stats() DBStats
```

Stats returns database statistics.

type DBStats

added in go1.5

```
type DBStats struct {
    MaxOpenConnections int // Maximum number of open connections to the database.

    // Pool Status
    OpenConnections int // The number of established connections both in use and idle.
    InUse           int // The number of connections currently in use.
    Idle            int // The number of idle connections.

    // Counters
    WaitCount          int64 // The total number of connections waited for.
    WaitDuration       time.Duration // The total time blocked waiting for a new connection.
    MaxIdleClosed      int64 // The total number of connections closed due to SetMaxIdleConns.
    MaxIdleTimeClosed  int64 // The total number of connections closed due to SetMaxIdleTime.
    MaxLifetimeClosed  int64 // The total number of connections closed due to SetMaxLifetime.
}
```

DBStats contains database statistics.

type IsolationLevel

added in go1.8

```
type IsolationLevel int
```

IsolationLevel is the transaction isolation level used in TxOptions.

```
const (
    LevelDefault IsolationLevel = iota
    LevelReadUncommitted
    LevelReadCommitted
    LevelWriteCommitted
    LevelRepeatableRead
    LevelSnapshot
    LevelSerializable
    LevelLinearizable
)
```

Various isolation levels that drivers may support in BeginTx. If a driver does not support a given isolation level an error may be returned.

See [https://en.wikipedia.org/wiki/Isolation_\(database_systems\)#Isolation_levels](https://en.wikipedia.org/wiki/Isolation_(database_systems)#Isolation_levels).

func (IsolationLevel) String

added in go1.11

```
func (i IsolationLevel) String() string
```

String returns the name of the transaction isolation level.

type NamedArg

added in go1.8

```
type NamedArg struct {

    // Name is the name of the parameter placeholder.
    //
    // If empty, the ordinal position in the argument list will be
    // used.
    //
    // Name must omit any symbol prefix.
    Name string

    // Value is the value of the parameter.
    // It may be assigned the same value types as the query
    // arguments.
    Value any

    // contains filtered or unexported fields
}
```

A NamedArg is a named argument. NamedArg values may be used as arguments to Query or Exec and bind to the corresponding named parameter in the SQL statement.

For a more concise way to create NamedArg values, see the Named function.

func Named

added in go1.8

```
func Named(name string, value any) NamedArg
```

Named provides a more concise way to create NamedArg values.

Example usage:

```
db.ExecContext(ctx, `
    delete from Invoice
    where
        TimeCreated < @end
        and TimeCreated >= @start;`,
    sql.Named("start", startTime),
    sql.Named("end", endTime),
)
```

type **NullBool**

```
type NullBool struct {
    Bool    bool
    Valid    bool // Valid is true if Bool is not NULL
}
```

NullBool represents a bool that may be null. NullBool implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (*NullBool) **Scan**

```
func (n *NullBool) Scan(value any) error
```

Scan implements the Scanner interface.

func (NullBool) **Value**

```
func (n NullBool) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type **NullByte**

added in go1.17

```
type NullByte struct {
    Byte    byte
    Valid    bool // Valid is true if Byte is not NULL
}
```

NullByte represents a byte that may be null. NullByte implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (*NullByte) **Scan**

added in go1.17

```
func (n *NullByte) Scan(value any) error
```

Scan implements the Scanner interface.

func (NullByte) Value

added in go1.17

```
func (n NullByte) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type NullFloat64

```
type NullFloat64 struct {  
    Float64 float64  
    Valid    bool // Valid is true if Float64 is not NULL  
}
```

NullFloat64 represents a float64 that may be null. NullFloat64 implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (*NullFloat64) Scan

```
func (n *NullFloat64) Scan(value any) error
```

Scan implements the Scanner interface.

func (NullFloat64) Value

```
func (n NullFloat64) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type NullInt16

added in go1.17

```
type NullInt16 struct {  
    Int16 int16  
    Valid bool // Valid is true if Int16 is not NULL  
}
```

NullInt16 represents an int16 that may be null. NullInt16 implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (*NullInt16) Scan

added in go1.17

```
func (n *NullInt16) Scan(value any) error
```

Scan implements the Scanner interface.

func (NullInt16) Value

added in go1.17


```
func (n NullInt16) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type NullInt32

added in go1.13

```
type NullInt32 struct {  
    Int32 int32  
    Valid bool // Valid is true if Int32 is not NULL  
}
```

NullInt32 represents an int32 that may be null. NullInt32 implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (*NullInt32) Scan

added in go1.13

```
func (n *NullInt32) Scan(value any) error
```

Scan implements the Scanner interface.

func (NullInt32) Value

added in go1.13

```
func (n NullInt32) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type NullInt64

```
type NullInt64 struct {  
    Int64 int64  
    Valid bool // Valid is true if Int64 is not NULL  
}
```

NullInt64 represents an int64 that may be null. NullInt64 implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (*NullInt64) Scan

```
func (n *NullInt64) Scan(value any) error
```

Scan implements the Scanner interface.

func (NullInt64) Value

```
func (n NullInt64) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type **NullString**

```
type NullString struct {  
    String string  
    Valid bool // Valid is true if String is not NULL  
}
```

NullString represents a string that may be null. NullString implements the Scanner interface so it can be used as a scan destination:

```
var s NullString  
err := db.QueryRow("SELECT name FROM foo WHERE id=?", id).Scan(&s)  
...  
if s.Valid {  
    // use s.String  
} else {  
    // NULL value  
}
```

func (*NullString) **Scan**

```
func (ns *NullString) Scan(value any) error
```

Scan implements the Scanner interface.

func (NullString) **Value**

```
func (ns NullString) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type **NullTime**

added in go1.13

```
type NullTime struct {  
    Time time.Time  
    Valid bool // Valid is true if Time is not NULL  
}
```

NullTime represents a time.Time that may be null. NullTime implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (*NullTime) **Scan**

added in go1.13

```
func (n *NullTime) Scan(value any) error
```

Scan implements the Scanner interface.

func (NullTime) **Value**

added in go1.13

```
func (n NullTime) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type Out

added in go1.9

```
type Out struct {

    // Dest is a pointer to the value that will be set to the result of the
    // stored procedure's OUTPUT parameter.
    Dest any

    // In is whether the parameter is an INOUT parameter. If so, the input value to the
    // procedure is the dereferenced value of Dest's pointer, which is then replaced with
    // the output value.
    In bool

    // contains filtered or unexported fields
}
```

Out may be used to retrieve OUTPUT value parameters from stored procedures.

Not all drivers and databases support OUTPUT value parameters.

Example usage:

```
var outArg string
_, err := db.ExecContext(ctx, "ProcName", sql.Named("Arg1", sql.Out{Dest: &outArg}))
```

type RawBytes

```
type RawBytes []byte
```

RawBytes is a byte slice that holds a reference to memory owned by the database itself. After a Scan into a RawBytes, the slice is only valid until the next call to Next, Scan, or Close.

type Result

```
type Result interface {

    // LastInsertId returns the integer generated by the database
    // in response to a command. Typically this will be from an
    // "auto increment" column when inserting a new row. Not all
    // databases support this feature, and the syntax of such
    // statements varies.
    LastInsertId() (int64, error)

    // RowsAffected returns the number of rows affected by an
    // update, insert, or delete. Not every database or database
    // driver may support this.
}
```

```
RowsAffected() (int64, error)
```

```
}
```

A Result summarizes an executed SQL command.

type Row

```
type Row struct {  
    // contains filtered or unexported fields  
}
```

Row is the result of calling QueryRow to select a single row.

func (*Row) Err

added in go1.15

```
func (r *Row) Err() error
```

Err provides a way for wrapping packages to check for query errors without calling Scan. Err returns the error, if any, that was encountered while running the query. If this error is not nil, this error will also be returned from Scan.

func (*Row) Scan

```
func (r *Row) Scan(dest ...any) error
```

Scan copies the columns from the matched row into the values pointed at by dest. See the documentation on Rows.Scan for details. If more than one row matches the query, Scan uses the first row and discards the rest. If no row matches the query, Scan returns ErrNoRows.

type Rows

```
type Rows struct {  
    // contains filtered or unexported fields  
}
```

Rows is the result of a query. Its cursor starts before the first row of the result set. Use Next to advance from row to row.

► [Example](#)

func (*Rows) Close

```
func (rs *Rows) Close() error
```

Close closes the Rows, preventing further enumeration. If Next is called and returns false and there are no further result sets, the Rows are closed automatically and it will suffice to check the result of Err. Close is idempotent and does not affect the result of Err.

func (*Rows) ColumnTypes

added in go1.8

```
func (rs *Rows) ColumnTypes() ([]*ColumnType, error)
```

ColumnTypes returns column information such as column type, length, and nullable. Some information may not be available from some drivers.

func (*Rows) Columns

```
func (rs *Rows) Columns() ([]string, error)
```

Columns returns the column names. Columns returns an error if the rows are closed.

func (*Rows) Err

```
func (rs *Rows) Err() error
```

Err returns the error, if any, that was encountered during iteration. Err may be called after an explicit or implicit Close.

func (*Rows) Next

```
func (rs *Rows) Next() bool
```

Next prepares the next result row for reading with the Scan method. It returns true on success, or false if there is no next result row or an error happened while preparing it. Err should be consulted to distinguish between the two cases.

Every call to Scan, even the first one, must be preceded by a call to Next.

func (*Rows) NextResultSet

added in go1.8

```
func (rs *Rows) NextResultSet() bool
```

NextResultSet prepares the next result set for reading. It reports whether there is further result sets, or false if there is no further result set or if there is an error advancing to it. The Err method should be consulted to distinguish between the two cases.

After calling NextResultSet, the Next method should always be called before scanning. If there are further result sets they may not have rows in the result set.

func (*Rows) Scan

```
func (rs *Rows) Scan(dest ...any) error
```

Scan copies the columns in the current row into the values pointed at by dest. The number of values in dest must be the same as the number of columns in Rows.

Scan converts columns read from the database into the following common Go types and special types provided by the sql package:

```
*string
*[]byte
*int, *int8, *int16, *int32, *int64
*uint, *uint8, *uint16, *uint32, *uint64
*bool
*float32, *float64
*interface{}
*RawBytes
*Rows (cursor value)
any type implementing Scanner (see Scanner docs)
```

In the most simple case, if the type of the value from the source column is an integer, bool or string type T and dest is of type *T, Scan simply assigns the value through the pointer.

Scan also converts between string and numeric types, as long as no information would be lost. While Scan stringifies all numbers scanned from numeric database columns into *string, scans into numeric types are checked for overflow. For example, a float64 with value 300 or a string with value "300" can scan into a uint16, but not into a uint8, though float64(255) or "255" can scan into a uint8. One exception is that scans of some float64 numbers to strings may lose information when stringifying. In general, scan floating point columns into *float64.

If a dest argument has type *[]byte, Scan saves in that argument a copy of the corresponding data. The copy is owned by the caller and can be modified and held indefinitely. The copy can be avoided by using an argument of type *RawBytes instead; see the documentation for RawBytes for restrictions on its use.

If an argument has type *interface{}, Scan copies the value provided by the underlying driver without conversion. When scanning from a source value of type []byte to *interface{}, a copy of the slice is made and the caller owns the result.

Source values of type time.Time may be scanned into values of type *time.Time, *interface{}, *string, or *[]byte. When converting to the latter two, time.RFC3339Nano is used.

Source values of type bool may be scanned into types *bool, *interface{}, *string, *[]byte, or *RawBytes.

For scanning into *bool, the source may be true, false, 1, 0, or string inputs parseable by strconv.ParseBool.

Scan can also convert a cursor returned from a query, such as "select cursor(select * from my_table) from dual", into a *Rows value that can itself be scanned from. The parent select query will close any cursor *Rows if the parent *Rows is closed.

If any of the first arguments implementing Scanner returns an error, that error will be wrapped in the returned error.

type **Scanner**

```

type Scanner interface {
    // Scan assigns a value from a database driver.
    //
    // The src value will be of one of the following types:
    //
    //     int64
    //     float64
    //     bool
    //     []byte
    //     string
    //     time.Time
    //     nil - for NULL values
    //
    // An error should be returned if the value cannot be stored
    // without loss of information.
    //
    // Reference types such as []byte are only valid until the next call to Scan
    // and should not be retained. Their underlying memory is owned by the driver.
    // If retention is necessary, copy their values before the next call to Scan.
    Scan(src any) error
}

```

Scanner is an interface used by Scan.

type Stmt

```

type Stmt struct {
    // contains filtered or unexported fields
}

```

Stmt is a prepared statement. A Stmt is safe for concurrent use by multiple goroutines.

If a Stmt is prepared on a Tx or Conn, it will be bound to a single underlying connection forever. If the Tx or Conn closes, the Stmt will become unusable and all operations will return an error. If a Stmt is prepared on a DB, it will remain usable for the lifetime of the DB. When the Stmt needs to execute on a new underlying connection, it will prepare itself on the new connection automatically.

► Example

func (*Stmt) Close

```

func (s *Stmt) Close() error

```

Close closes the statement.

func (*Stmt) Exec

```

func (s *Stmt) Exec(args ...any) (Result, error)

```

Exec executes a prepared statement with the given arguments and returns a Result summarizing the effect of the statement.

Exec uses context.Background internally; to specify the context, use ExecContext.

func (*Stmt) ExecContext

added in go1.8

```
func (s *Stmt) ExecContext(ctx context.Context, args ...any) (Result, error)
```

ExecContext executes a prepared statement with the given arguments and returns a Result summarizing the effect of the statement.

func (*Stmt) Query

```
func (s *Stmt) Query(args ...any) (*Rows, error)
```

Query executes a prepared query statement with the given arguments and returns the query results as a *Rows.

Query uses context.Background internally; to specify the context, use QueryContext.

func (*Stmt) QueryContext

added in go1.8

```
func (s *Stmt) QueryContext(ctx context.Context, args ...any) (*Rows, error)
```

QueryContext executes a prepared query statement with the given arguments and returns the query results as a *Rows.

func (*Stmt) QueryRow

```
func (s *Stmt) QueryRow(args ...any) *Row
```

QueryRow executes a prepared query statement with the given arguments. If an error occurs during the execution of the statement, that error will be returned by a call to Scan on the returned *Row, which is always non-nil. If the query selects no rows, the *Row's Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

Example usage:

```
var name string
err := nameByUserIdStmt.QueryRow(id).Scan(&name)
```

QueryRow uses context.Background internally; to specify the context, use QueryRowContext.

func (*Stmt) QueryRowContext

added in go1.8

```
func (s *Stmt) QueryRowContext(ctx context.Context, args ...any) *Row
```


QueryRowContext executes a prepared query statement with the given arguments. If an error occurs during the execution of the statement, that error will be returned by a call to Scan on the returned *Row, which is always non-nil. If the query selects no rows, the *Row's Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

► [Example](#)

type Tx

```
type Tx struct {  
    // contains filtered or unexported fields  
}
```

Tx is an in-progress database transaction.

A transaction must end with a call to Commit or Rollback.

After a call to Commit or Rollback, all operations on the transaction fail with ErrTxDone.

The statements prepared for a transaction by calling the transaction's Prepare or Stmt methods are closed by the call to Commit or Rollback.

func (*Tx) Commit

```
func (tx *Tx) Commit() error
```

Commit commits the transaction.

func (*Tx) Exec

```
func (tx *Tx) Exec(query string, args ...any) (Result, error)
```

Exec executes a query that doesn't return rows. For example: an INSERT and UPDATE.

Exec uses context.Background internally; to specify the context, use ExecContext.

func (*Tx) ExecContext

added in go1.8

```
func (tx *Tx) ExecContext(ctx context.Context, query string, args ...any) (Result, error)
```

ExecContext executes a query that doesn't return rows. For example: an INSERT and UPDATE.

► [Example](#)

func (*Tx) Prepare

```
func (tx *Tx) Prepare(query string) (*Stmt, error)
```

Prepare creates a prepared statement for use within a transaction.

The returned statement operates within the transaction and will be closed when the transaction has been committed or rolled back.

To use an existing prepared statement on this transaction, see Tx.Stmt.

Prepare uses context.Background internally; to specify the context, use PrepareContext.

► Example

func (*Tx) PrepareContext

added in go1.8

```
func (tx *Tx) PrepareContext(ctx context.Context, query string) (*Stmt, error)
```

PrepareContext creates a prepared statement for use within a transaction.

The returned statement operates within the transaction and will be closed when the transaction has been committed or rolled back.

To use an existing prepared statement on this transaction, see Tx.Stmt.

The provided context will be used for the preparation of the context, not for the execution of the returned statement. The returned statement will run in the transaction context.

func (*Tx) Query

```
func (tx *Tx) Query(query string, args ...any) (*Rows, error)
```

Query executes a query that returns rows, typically a SELECT.

Query uses context.Background internally; to specify the context, use QueryContext.

func (*Tx) QueryContext

added in go1.8

```
func (tx *Tx) QueryContext(ctx context.Context, query string, args ...any) (*Rows, error)
```

QueryContext executes a query that returns rows, typically a SELECT.

func (*Tx) QueryRow

```
func (tx *Tx) QueryRow(query string, args ...any) *Row
```

QueryRow executes a query that is expected to return at most one row. QueryRow always returns a non-nil value. Errors are deferred until Row's Scan method is called. If the query selects no rows, the *Row's

Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

QueryRow uses context.Background internally; to specify the context, use QueryRowContext.

func (*Tx) QueryRowContext

added in go1.8

```
func (tx *Tx) QueryRowContext(ctx context.Context, query string, args ...any) *Row
```

QueryRowContext executes a query that is expected to return at most one row. QueryRowContext always returns a non-nil value. Errors are deferred until Row's Scan method is called. If the query selects no rows, the *Row's Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

func (*Tx) Rollback

```
func (tx *Tx) Rollback() error
```

Rollback aborts the transaction.

► [Example](#)

func (*Tx) Stmt

```
func (tx *Tx) Stmt(stmt *Stmt) *Stmt
```

Stmt returns a transaction-specific prepared statement from an existing statement.

Example:

```
updateMoney, err := db.Prepare("UPDATE balance SET money=money+? WHERE id=?")
...
tx, err := db.Begin()
...
res, err := tx.Stmt(updateMoney).Exec(123.45, 98293203)
```

The returned statement operates within the transaction and will be closed when the transaction has been committed or rolled back.

Stmt uses context.Background internally; to specify the context, use StmtContext.

func (*Tx) StmtContext

added in go1.8

```
func (tx *Tx) StmtContext(ctx context.Context, stmt *Stmt) *Stmt
```

StmtContext returns a transaction-specific prepared statement from an existing statement.

Example:

```
updateMoney, err := db.Prepare("UPDATE balance SET money=money+? WHERE id=?")
...
tx, err := db.Begin()
...
res, err := tx.StmtContext(ctx, updateMoney).Exec(123.45, 98293203)
```

The provided context is used for the preparation of the statement, not for the execution of the statement.

The returned statement operates within the transaction and will be closed when the transaction has been committed or rolled back.

type TxOptions

added in go1.8

```
type TxOptions struct {
    // Isolation is the transaction isolation level.
    // If zero, the driver or database's default level is used.
    Isolation IsolationLevel
    ReadOnly  bool
}
```

TxOptions holds the transaction options to be used in DB.BeginTx.



Source Files

[View all](#)

[convert.go](#)

[ctxutil.go](#)

[sql.go](#)



Directories

[driver](#)

Package driver defines interfaces to be implemented by database drivers as used by package sql.

Why Go

[Use Cases](#)

[Case Studies](#)

Get Started

[Playground](#)

[Tour](#)

[Stack Overflow](#)

[Help](#)

Packages

[Standard Library](#)

[About Go Packages](#)

About

[Download](#)

[Blog](#)

[Issue Tracker](#)

[Release Notes](#)

[Brand Guidelines](#)

[Code of Conduct](#)

Connect

[Twitter](#)

[GitHub](#)

Slack

r/golang

Meetup

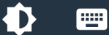
Golang Weekly

Copyright

Terms of Service

Privacy Policy

Report an Issue



Google