sqlc

Release 1.12.0

Kyle Conroy

Feb 06, 2022

OVERVIEW

And lo, the Great One looked down upon the people and proclaimed: "SQL is actually pretty great"

sqlc generates **fully type-safe idiomatic Go code** from SQL. Here's how it works:

1. You write SQL queries
2. You run sqlc to generate Go code that presents type-safe interfaces to those queries
3. You write application code that calls the methods sqlc generated

Seriously, it's that easy. You don't have to write any boilerplate SQL querying code ever again.

CHAPTER

ONE

INSTALLING SQLC

sqlc is distributed as a single binary with zero dependencies.

1.1 macOS

```
brew install sqlc
```

1.2 Ubuntu

```
sudo snap install sqlc
```

1.3 go install

1.3.1 Go >= 1.17:

```
go install github.com/kyleconroy/sqlc/cmd/sqlc@latest
```

1.3.2 Go < 1.17:

```
go get github.com/kyleconroy/sqlc/cmd/sqlc
```

1.4 Docker

```
docker pull kjconroy/sqlc
```

Run `sqlc` using `docker run`:

```
docker run --rm -v $(pwd):/src -w /src kjconroy/sqlc generate
```

1.5 Downloads

Get pre-built binaries for *v1.12.0*:

- Linux
- macOS
- Windows (MySQL only)

CHAPTER

TWO

GETTING STARTED WITH MYSQL

This tutorial assumes that the latest version of sqlc is installed and ready to use.

Create a new directory called `sqlc-tutorial` and open it up.

Initialize a new Go module named `tutorial.sql.dev/app`

```
go mod init tutorial.sqlc.dev/app
```

sqlc looks for either a `sqlc.yaml` or `sqlc.json` file in the current directory. In our new directory, create a file named `sqlc.yaml` with the following contents:

```yaml
version: 1
packages:
  - path: "tutorial"
    name: "tutorial"
    engine: "mysql"
    schema: "schema.sql"
    queries: "query.sql"
```

sqlc needs to know your database schema and queries. In the same directory, create a file named `schema.sql` with the following contents:

```
CREATE TABLE authors (
  id   BIGINT  NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name text    NOT NULL,
  bio  text
);
```

Next, create a `query.sql` file with the following four queries:

```
-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = ? LIMIT 1;

-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY name;

-- name: CreateAuthor :execresult
INSERT INTO authors (
  name, bio
) VALUES (
  ?, ?
);
```

(continues on next page)

(continued from previous page)

```
-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = ?;
```

You are now ready to generate code. Run the `generate` command. You shouldn't see any errors or output.

```
sqlc generate
```

You should now have a `tutorial` package containing three files.

```
├── go.mod
├── query.sql
├── schema.sql
├── sqlc.yaml
└── tutorial
    ├── db.go
    ├── models.go
    └── query.sql.go
```

You can use your newly generated queries in `app.go`.

```
package main

import (
        "context"
        "database/sql"
        "log"
        "reflect"

        "tutorial.sqlc.dev/app/tutorial"

        _ "github.com/go-sql-driver/mysql"
)

func run() error {
        ctx := context.Background()

        db, err := sql.Open("mysql", "user:password@/dbname")
        if err != nil {
                return err
        }

        queries := tutorial.New(db)

        // list all authors
        authors, err := queries.ListAuthors(ctx)
        if err != nil {
                return err
        }
        log.Println(authors)

        // create an author
        result, err := queries.CreateAuthor(ctx, tutorial.CreateAuthorParams{
                Name: "Brian Kernighan",
                Bio:  sql.NullString{String: "Co-author of The C Programming Language␣
↪and The Go Programming Language", Valid: true},
```

(continues on next page)

(continued from previous page)

```
        })
        if err != nil {
                return err
        }

        insertedAuthorID, err := result.LastInsertId()
        if err != nil {
                return err
        }
        log.Println(insertedAuthorID)

        // get the author we just inserted
        fetchedAuthor, err := queries.GetAuthor(ctx, insertedAuthorID)
        if err != nil {
                return err
        }

        // prints true
        log.Println(reflect.DeepEqual(insertedAuthorID, fetchedAuthor.ID))
        return nil
}

func main() {
        if err := run(); err != nil {
                log.Fatal(err)
        }
}
```

Before the code will compile, you'll need to add the Go MySQL driver.

```
go get github.com/go-sql-driver/mysql
go build ./...
```

To make that possible, sqlc generates readable, **idiomatic** Go code that you otherwise would have had to write yourself. Take a look in `tutorial/query.sql.go`.

CHAPTER

THREE

GETTING STARTED WITH POSTGRESQL

This tutorial assumes that the latest version of sqlc is installed and ready to use.

Create a new directory called `sqlc-tutorial` and open it up.

Initialize a new Go module named `tutorial.sql.dev/app`

```
go mod init tutorial.sqlc.dev/app
```

sqlc looks for either a `sqlc.yaml` or `sqlc.json` file in the current directory. In our new directory, create a file named `sqlc.yaml` with the following contents:

```
version: 1
packages:
  - path: "tutorial"
    name: "tutorial"
    engine: "postgresql"
    schema: "schema.sql"
    queries: "query.sql"
```

sqlc needs to know your database schema and queries. In the same directory, create a file named `schema.sql` with the following contents:

```
CREATE TABLE authors (
  id   BIGSERIAL PRIMARY KEY,
  name text      NOT NULL,
  bio  text
);
```

Next, create a `query.sql` file with the following four queries:

```
-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1 LIMIT 1;

-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY name;

-- name: CreateAuthor :one
INSERT INTO authors (
  name, bio
) VALUES (
  $1, $2
)
```

(continues on next page)

(continued from previous page)

```
RETURNING *;

-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1;
```

You are now ready to generate code. Run the `generate` command. You shouldn't see any errors or output.

```
sqlc generate
```

You should now have a `tutorial` package containing three files.

```
├── go.mod
├── query.sql
├── schema.sql
├── sqlc.yaml
└── tutorial
    ├── db.go
    ├── models.go
    └── query.sql.go
```

You can use your newly generated queries in `app.go`.

```
package main

import (
        "context"
        "database/sql"
        "log"
        "reflect"

        "tutorial.sqlc.dev/app/tutorial"

        _ "github.com/lib/pq"
)

func run() error {
        ctx := context.Background()

        db, err := sql.Open("postgres", "user=pqgotest dbname=pqgotest sslmode=verify-
↪full")
        if err != nil {
                return err
        }

        queries := tutorial.New(db)

        // list all authors
        authors, err := queries.ListAuthors(ctx)
        if err != nil {
                return err
        }
        log.Println(authors)

        // create an author
        insertedAuthor, err := queries.CreateAuthor(ctx, tutorial.CreateAuthorParams{
                Name: "Brian Kernighan",
```

(continues on next page)

(continued from previous page)

```
                Bio:  sql.NullString{String: "Co-author of The C Programming Language
↪and The Go Programming Language", Valid: true},
        })
        if err != nil {
                return err
        }
        log.Println(insertedAuthor)

        // get the author we just inserted
        fetchedAuthor, err := queries.GetAuthor(ctx, insertedAuthor.ID)
        if err != nil {
                return err
        }

        // prints true
        log.Println(reflect.DeepEqual(insertedAuthor, fetchedAuthor))
        return nil
}

func main() {
        if err := run(); err != nil {
                log.Fatal(err)
        }
}
```

Before the code will compile, you'll need to add the Go PostgreSQL driver.

```
go get github.com/lib/pq
go build ./...
```

To make that possible, sqlc generates readable, **idiomatic** Go code that you otherwise would have had to write yourself. Take a look in `tutorial/query.sql.go`.

CHAPTER

FOUR

RETRIEVING ROWS

To generate a database access method, annotate a query with a specific comment.

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  bio        text   NOT NULL,
  birth_year int    NOT NULL
);


-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1;

-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY id;
```

A few new pieces of code are generated beyond the `Author` struct. An interface for the underlying database is generated. The `*sql.DB` and `*sql.Tx` types satisfy this interface.

The database access methods are added to a `Queries` struct, which is created using the `New` method.

Note that the `*` in our query has been replaced with explicit column names. This change ensures that the query will never return unexpected data.

Our query was annotated with `:one`, meaning that it should only return a single row. We scan the data from that one into a `Author` struct.

Since the get query has a single parameter, the `GetAuthor` method takes a single `int` as an argument.

Since the list query has no parameters, the `ListAuthors` method accepts no arguments.

```
package db

import (
        "context"
        "database/sql"
)

type Author struct {
        ID        int
        Bio       string
        BirthYear int
}
```

(continues on next page)

(continued from previous page)

```
type DBTX interface {
        QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const getAuthor = `-- name: GetAuthor :one
SELECT id, bio, birth_year FROM authors
WHERE id = $1
`

func (q *Queries) GetAuthor(ctx context.Context, id int) (Author, error) {
        row := q.db.QueryRowContext(ctx, getAuthor, id)
        var i Author
        err := row.Scan(&i.ID, &i.Bio, &i.BirthYear)
        return i, err
}

const listAuthors = `-- name: ListAuthors :many
SELECT id, bio, birth_year FROM authors
ORDER BY id
`

func (q *Queries) ListAuthors(ctx context.Context) ([]Author, error) {
        rows, err := q.db.QueryContext(ctx, listAuthors)
        if err != nil {
                return nil, err
        }
        defer rows.Close()
        var items []Author
        for rows.Next() {
                var i Author
                if err := rows.Scan(&i.ID, &i.Bio, &i.BirthYear); err != nil {
                        return nil, err
                }
                items = append(items, i)
        }
        if err := rows.Close(); err != nil {
                return nil, err
        }
        if err := rows.Err(); err != nil {
                return nil, err
        }
        return items, nil
}
```

4.1 Selecting columns

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  bio         text   NOT NULL,
  birth_year int    NOT NULL
);

-- name: GetBioForAuthor :one
SELECT bio FROM authors
WHERE id = $1;

-- name: GetInfoForAuthor :one
SELECT bio, birth_year FROM authors
WHERE id = $1;
```

When selecting a single column, only that value that returned. The `GetBioForAuthor` method takes a single `int` as an argument and returns a `string` and an `error`.

When selecting multiple columns, a row record (method-specific struct) is returned. In this case, `GetInfoForAuthor` returns a struct with two fields: `Bio` and `BirthYear`.

```
package db

import (
        "context"
        "database/sql"
)

type DBTX interface {
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const getBioForAuthor = `-- name: GetBioForAuthor :one
SELECT bio FROM authors
WHERE id = $1
`

func (q *Queries) GetBioForAuthor(ctx context.Context, id int) (string, error) {
        row := q.db.QueryRowContext(ctx, getBioForAuthor, id)
        var i string
        err := row.Scan(&i)
        return i, err
}

const getInfoForAuthor = `-- name: GetInfoForAuthor :one
SELECT bio, birth_year FROM authors
WHERE id = $1
`
```

(continues on next page)

(continued from previous page)

```
type GetInfoForAuthorRow struct {
        Bio       string
        BirthYear int
}

func (q *Queries) GetInfoForAuthor(ctx context.Context, id int) (GetInfoForAuthorRow,␣
↪error) {
        row := q.db.QueryRowContext(ctx, getInfoForAuthor, id)
        var i GetInfoForAuthorRow
        err := row.Scan(&i.Bio, &i.BirthYear)
        return i, err
}
```

4.2 Passing a slice as a parameter to a query

In PostgreSQL, ANY allows you to check if a value exists in an array expression. Queries using ANY with a single parameter will generate method signatures with slices as arguments. Use the postgres data types, eg: int, varchar, etc.

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  bio        text   NOT NULL,
  birth_year int    NOT NULL
);

-- name: ListAuthorsByIDs :many
SELECT * FROM authors
WHERE id = ANY($1::int[]);
```

The above SQL will generate the following code:

```
package db

import (
        "context"
        "database/sql"

        "github.com/lib/pq"
)

type Author struct {
        ID        int
        Bio       string
        BirthYear int
}

type DBTX interface {
        QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}
```

(continues on next page)

(continued from previous page)

```
type Queries struct {
        db DBTX
}

const listAuthors = `-- name: ListAuthorsByIDs :many
SELECT id, bio, birth_year FROM authors
WHERE id = ANY($1::int[])
`

func (q *Queries) ListAuthorsByIDs(ctx context.Context, ids []int) ([]Author, error) {
        rows, err := q.db.QueryContext(ctx, listAuthors, pq.Array(ids))
        if err != nil {
                return nil, err
        }
        defer rows.Close()
        var items []Author
        for rows.Next() {
                var i Author
                if err := rows.Scan(&i.ID, &i.Bio, &i.BirthYear); err != nil {
                        return nil, err
                }
                items = append(items, i)
        }
        if err := rows.Close(); err != nil {
                return nil, err
        }
        if err := rows.Err(); err != nil {
                return nil, err
        }
        return items, nil
}
```

CHAPTER

FIVE

COUNTING ROWS

```
CREATE TABLE authors (
  id       SERIAL PRIMARY KEY,
  hometown text   NOT NULL
);

-- name: CountAuthors :one
SELECT count(*) FROM authors;

-- name: CountAuthorsByTown :many
SELECT hometown, count(*) FROM authors
GROUP BY 1
ORDER BY 1;
```

```
package db

import (
        "context"
        "database/sql"
)

type DBTX interface {
        QueryContext(context.Context, string, ...interface{}) (*sql.Rows, error)
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const countAuthors = `-- name: CountAuthors :one
SELECT count(*) FROM authors
`

func (q *Queries) CountAuthors(ctx context.Context) (int, error) {
        row := q.db.QueryRowContext(ctx, countAuthors)
        var i int
        err := row.Scan(&i)
        return i, err
}
```

(continues on next page)

(continued from previous page)

```
const countAuthorsByTown = `-- name: CountAuthorsByTown :many
SELECT hometown, count(*) FROM authors
GROUP BY 1
ORDER BY 1
`

type CountAuthorsByTownRow struct {
        Hometown string
        Count    int
}

func (q *Queries) CountAuthorsByTown(ctx context.Context) ([]CountAuthorsByTownRow,␣
↪error) {
        rows, err := q.db.QueryContext(ctx, countAuthorsByTown)
        if err != nil {
                return nil, err
        }
        defer rows.Close()
        items := []CountAuthorsByTownRow{}
        for rows.Next() {
                var i CountAuthorsByTownRow
                if err := rows.Scan(&i.Hometown, &i.Count); err != nil {
                        return nil, err
                }
                items = append(items, i)
        }
        if err := rows.Close(); err != nil {
                return nil, err
        }
        if err := rows.Err(); err != nil {
                return nil, err
        }
        return items, nil
}
```

CHAPTER

SIX

INSERTING ROWS

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  bio        text   NOT NULL
);

-- name: CreateAuthor :exec
INSERT INTO authors (bio) VALUES ($1);
```

```
package db

import (
        "context"
        "database/sql"
)

type DBTX interface {
        ExecContext(context.Context, string, ...interface{}) error
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const createAuthor = `-- name: CreateAuthor :exec
INSERT INTO authors (bio) VALUES ($1)
`

func (q *Queries) CreateAuthor(ctx context.Context, bio string) error {
        _, err := q.db.ExecContext(ctx, createAuthor, bio)
        return err
}
```

6.1 Returning columns from inserted rows

sqlc has full support for the `RETURNING` statement.

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  bio        text   NOT NULL
);

-- name: Delete :exec
DELETE FROM authors WHERE id = $1;

-- name: DeleteAffected :execrows
DELETE FROM authors WHERE id = $1;

-- name: DeleteID :one
DELETE FROM authors WHERE id = $1
RETURNING id;

-- name: DeleteAuthor :one
DELETE FROM authors WHERE id = $1
RETURNING *;
```

```
package db

import (
        "context"
        "database/sql"
)

type Author struct {
        ID  int
        Bio string
}

type DBTX interface {
        ExecContext(context.Context, string, ...interface{}) error
        QueryRowContext(context.Context, string, ...interface{}) error
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const delete = `-- name: Delete :exec
DELETE FROM authors WHERE id = $1
`

func (q *Queries) Delete(ctx context.Context, id int) error {
        _, err := q.db.ExecContext(ctx, delete, id)
        return err
}
```

(continues on next page)

(continued from previous page)

```
const deleteAffected = `-- name: DeleteAffected :execrows
DELETE FROM authors WHERE id = $1
`

func (q *Queries) DeleteAffected(ctx context.Context, id int) (int64, error) {
        result, err := q.db.ExecContext(ctx, deleteAffected, id)
        if err != nil {
                return 0, err
        }
        return result.RowsAffected()
}

const deleteID = `-- name: DeleteID :one
DELETE FROM authors WHERE id = $1
RETURNING id
`

func (q *Queries) DeleteID(ctx context.Context, id int) (int, error) {
        row := q.db.QueryRowContext(ctx, deleteID, id)
        var i int
        err := row.Scan(&i)
        return i, err
}

const deleteAuthor = `-- name: DeleteAuthor :one
DELETE FROM authors WHERE id = $1
RETURNING id, bio
`

func (q *Queries) DeleteAuthor(ctx context.Context, id int) (Author, error) {
        row := q.db.QueryRowContext(ctx, deleteAuthor, id)
        var i Author
        err := row.Scan(&i.ID, &i.Bio)
        return i, err
}
```

6.2 Using CopyFrom

PostgreSQL supports the Copy Protocol that can insert rows a lot faster than sequential inserts. You can use this easily with sqlc:

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  name       text   NOT NULL,
  bio        text   NOT NULL
);

-- name: CreateAuthors :copyfrom
INSERT INTO authors (name, bio) VALUES ($1, $2);
```

```
type CreateAuthorsParams struct {
  Name string
  Bio  string
}
```

(continues on next page)

(continued from previous page)

```
func (q *Queries) CreateAuthors(ctx context.Context, arg []CreateAuthorsParams)␣
↪(int64, error) {
  return q.db.CopyFrom(ctx, []string{"authors"}, []string{"name", "bio"}, &
↪iteratorForCreateAuthors{rows: arg})
}
```

CHAPTER

SEVEN

UPDATING ROWS

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  bio        text   NOT NULL
);

-- name: UpdateAuthor :exec
UPDATE authors SET bio = $2
WHERE id = $1;
```

```
package db

import (
        "context"
        "database/sql"
)

type DBTX interface {
        ExecContext(context.Context, string, ...interface{}) error
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const updateAuthor = `-- name: UpdateAuthor :exec
UPDATE authors SET bio = $2
WHERE id = $1
`

func (q *Queries) UpdateAuthor(ctx context.Context, id int, bio string) error {
        _, err := q.db.ExecContext(ctx, updateAuthor, id, bio)
        return err
}
```

CHAPTER

EIGHT

DELETING ROWS

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  bio        text   NOT NULL
);

-- name: DeleteAuthor :exec
DELETE FROM authors WHERE id = $1;
```

```
package db

import (
        "context"
        "database/sql"
)

type DBTX interface {
        ExecContext(context.Context, string, ...interface{}) error
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

const deleteAuthor = `-- name: DeleteAuthor :exec
DELETE FROM authors WHERE id = $1
`

func (q *Queries) DeleteAuthor(ctx context.Context, id int) error {
        _, err := q.db.ExecContext(ctx, deleteAuthor, id)
        return err
}
```

CHAPTER

NINE

PREPARING QUERIES

```
CREATE TABLE records (
  id SERIAL PRIMARY KEY
);

-- name: GetRecord :one
SELECT * FROM records
WHERE id = $1;
```

sqlc has an option to use prepared queries. These prepared queries also work with transactions.

```
package db

import (
        "context"
        "database/sql"
)

type Record struct {
        ID int
}

type DBTX interface {
        PrepareContext(context.Context, string) (*sql.Stmt, error)
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

func Prepare(ctx context.Context, db DBTX) (*Queries, error) {
        q := Queries{db: db}
        var err error
        if q.getRecordStmt, err = db.PrepareContext(ctx, getRecord); err != nil {
                return nil, err
        }
        return &q, nil
}

func (q *Queries) queryRow(ctx context.Context, stmt *sql.Stmt, query string, args ...
↪interface{}) *sql.Row {
        switch {
        case stmt != nil && q.tx != nil:
                return q.tx.StmtContext(ctx, stmt).QueryRowContext(ctx, args...)
```

(continues on next page)

(continued from previous page)

```
        case stmt != nil:
                return stmt.QueryRowContext(ctx, args...)
        default:
                return q.db.QueryRowContext(ctx, query, args...)
        }
}

type Queries struct {
        db            DBTX
        tx            *sql.Tx
        getRecordStmt *sql.Stmt
}

func (q *Queries) WithTx(tx *sql.Tx) *Queries {
        return &Queries{
                db:            tx,
                tx:            tx,
                getRecordStmt: q.getRecordStmt,
        }
}

const getRecord = `-- name: GetRecord :one
SELECT id FROM records
WHERE id = $1
`

func (q *Queries) GetRecord(ctx context.Context, id int) (Record, error) {
        row := q.queryRow(ctx, q.getRecordStmt, getRecord, id)
        var i Record
        err := row.Scan(&i.ID)
        return i, err
}
```

CHAPTER

TEN

USING TRANSACTIONS

```
CREATE TABLE records (
  id SERIAL PRIMARY KEY
);

-- name: GetRecord :one
SELECT * FROM records
WHERE id = $1;
```

The `WithTx` method allows a `Queries` instance to be associated with a transaction.

```
package db

import (
        "context"
        "database/sql"
)

type Record struct {
        ID int
}

type DBTX interface {
        QueryRowContext(context.Context, string, ...interface{}) *sql.Row
}

func New(db DBTX) *Queries {
        return &Queries{db: db}
}

type Queries struct {
        db DBTX
}

func (*Queries) WithTx(tx *sql.Tx) *Queries {
        return &Queries{db: tx}
}

const getRecord = `-- name: GetRecord :one
SELECT id FROM records
WHERE id = $1
`

func (q *Queries) GetRecord(ctx context.Context, id int) (Record, error) {
        row := q.db.QueryRowContext(ctx, getRecord, id)
```

(continues on next page)

(continued from previous page)

```
        var i Record
        err := row.Scan(&i.ID)
        return i, err
}
```

CHAPTER

ELEVEN

NAMING PARAMETERS

sqlc tried to generate good names for positional parameters, but sometimes it lacks enough context. The following SQL generates parameters with less than ideal names:

```
-- name: UpsertAuthorName :one
UPDATE author
SET
  name = CASE WHEN $1::bool
    THEN $2::text
    ELSE name
    END
RETURNING *;
```

```
type UpdateAuthorNameParams struct {
        Column1   bool   `json:""`
        Column2_2 string `json:"_2"`
}
```

In these cases, named parameters give you the control over field names on the Params struct.

```
-- name: UpsertAuthorName :one
UPDATE author
SET
  name = CASE WHEN sqlc.arg(set_name)::bool
    THEN sqlc.arg(name)::text
    ELSE name
    END
RETURNING *;
```

```
type UpdateAuthorNameParams struct {
        SetName bool   `json:"set_name"`
        Name    string `json:"name"`
}
```

If the `sqlc.arg()` syntax is too verbose for your taste, you can use the `@` operator as a shortcut.

```
-- name: UpsertAuthorName :one
UPDATE author
SET
  name = CASE WHEN @set_name::bool
    THEN @name::text
    ELSE name
    END
RETURNING *;
```

CHAPTER

TWELVE

MODIFYING THE DATABASE SCHEMA

sqlc understands `ALTER TABLE` statements when parsing SQL.

```
CREATE TABLE authors (
  id          SERIAL PRIMARY KEY,
  birth_year  int    NOT NULL
);

ALTER TABLE authors ADD COLUMN bio text NOT NULL;
ALTER TABLE authors DROP COLUMN birth_year;
ALTER TABLE authors RENAME TO writers;
```

```
package db

type Writer struct {
        ID  int
        Bio string
}
```

12.1 Handling SQL migrations

sqlc will ignore rollback statements when parsing migration SQL files. The following tools are current supported:

- dbmate
- golang-migrate
- goose
- sql-migrate
- tern

12.1.1 goose

```
-- +goose Up
CREATE TABLE post (
    id    int NOT NULL,
    title text,
    body  text,
    PRIMARY KEY(id)
);
```

(continues on next page)

(continued from previous page)

```
-- +goose Down
DROP TABLE post;
```

```
package db

type Post struct {
        ID    int
        Title sql.NullString
        Body  sql.NullString
}
```

12.1.2 sql-migrate

```
-- +migrate Up
-- SQL in section 'Up' is executed when this migration is applied
CREATE TABLE people (id int);


-- +migrate Down
-- SQL section 'Down' is executed when this migration is rolled back
DROP TABLE people;
```

```
package db

type People struct {
        ID int32
}
```

12.1.3 tern

```
CREATE TABLE comment (id int NOT NULL, text text NOT NULL);
---- create above / drop below ----
DROP TABLE comment;
```

```
package db

type Comment struct {
        ID   int32
        Text string
}
```

12.1.4 golang-migrate

Warning: golang-migrate specifies that the version number in the migration file name is to be interpreted numerically. However, sqlc executes the migration files in **lexicographic** order. If you choose to simply enumerate your migration versions, make sure to prepend enough zeros to the version number to avoid any unexpected behavior.

Probably doesn't work as intended:

```
1_initial.up.sql
...
9_foo.up.sql
# this migration file will be executed BEFORE 9_foo
10_bar.up.sql
```

Works as was probably intended:

```
001_initial.up.sql
...
009_foo.up.sql
010_bar.up.sql
```

In `20060102.up.sql`:

```
CREATE TABLE post (
    id    int NOT NULL,
    title text,
    body  text,
    PRIMARY KEY(id)
);
```

In `20060102.down.sql`:

```
DROP TABLE post;
```

```
package db

type Post struct {
        ID    int
        Title sql.NullString
        Body  sql.NullString
}
```

12.1.5 dbmate

```
-- migrate:up
CREATE TABLE foo (bar INT NOT NULL);

-- migrate:down
DROP TABLE foo;
```

```
package db

type Foo struct {
        Bar int32
}
```

CHAPTER

THIRTEEN

CONFIGURING GENERATED STRUCTS

13.1 Naming scheme

Structs generated from tables will attempt to use the singular form of a table name if the table name is pluralized.

```
CREATE TABLE authors (
  id   SERIAL PRIMARY KEY,
  name text   NOT NULL
);
```

```
package db

// Struct names use the singular form of table names
type Author struct {
        ID   int
        Name string
}
```

13.2 JSON tags

```
CREATE TABLE authors (
  id         SERIAL    PRIMARY KEY,
  created_at timestamp NOT NULL
);
```

sqlc can generate structs with JSON tags. The JSON name for a field matches the column name in the database.

```
package db

import (
        "time"
)

type Author struct {
        ID        int       `json:"id"`
        CreatedAt time.Time `json:"created_at"`
}
```

CHAPTER

FOURTEEN

CLI

```
Usage:
  sqlc [command]

Available Commands:
  compile     Statically check SQL for syntax and type errors
  generate    Generate Go code from SQL
  help        Help about any command
  init        Create an empty sqlc.yaml settings file
  version     Print the sqlc version number

Flags:
  -h, --help   help for sqlc

Use "sqlc [command] --help" for more information about a command.
```

CHAPTER

FIFTEEN

CONFIGURATION FILE (VERSION 1)

The `sqlc` tool is configured via a `sqlc.yaml` or `sqlc.json` file. This file must be in the directory where the `sqlc` command is run.

```yaml
version: "1"
packages:
  - name: "db"
    path: "internal/db"
    queries: "./sql/query/"
    schema: "./sql/schema/"
    engine: "postgresql"
    emit_prepared_queries: true
    emit_interface: false
    emit_exact_table_names: false
    emit_empty_slices: false
    emit_exported_queries: false
    emit_json_tags: true
    emit_result_struct_pointers: false
    emit_params_struct_pointers: false
    emit_methods_with_db_argument: false
    json_tags_case_style: "camel"
    output_db_file_name: "db.go"
    output_models_file_name: "models.go"
    output_querier_file_name: "querier.go"
```

Each package document has the following keys:

- `name`:
  - The package name to use for the generated code. Defaults to `path` basename
- `path`:
  - Output directory for generated code
- `queries`:
  - Directory of SQL queries or path to single SQL file; or a list of paths
- `schema`:
  - Directory of SQL migrations or path to single SQL file; or a list of paths
- `engine`:
  - Either `postgresql` or `mysql`. Defaults to `postgresql`.
- `sql_package`:
  - Either `pgx/v4` or `database/sql`. Defaults to `database/sql`.

- `emit_db_tags`:
  - If true, add DB tags to generated structs. Defaults to `false`.
- `emit_prepared_queries`:
  - If true, include support for prepared queries. Defaults to `false`.
- `emit_interface`:
  - If true, output a `Querier` interface in the generated package. Defaults to `false`.
- `emit_exact_table_names`:
  - If true, struct names will mirror table names. Otherwise, sqlc attempts to singularize plural table names. Defaults to `false`.
- `emit_empty_slices`:
  - If true, slices returned by `:many` queries will be empty instead of `nil`. Defaults to `false`.
- `emit_exported_queries`:
  - If true, autogenerated SQL statement can be exported to be accessed by another package.
- `emit_json_tags`:
  - If true, add JSON tags to generated structs. Defaults to `false`.
- `emit_result_struct_pointers`:
  - If true, query results are returned as pointers to structs. Queries returning multiple results are returned as slices of pointers. Defaults to `false`.
- `emit_params_struct_pointers`:
  - If true, parameters are passed as pointers to structs. Defaults to `false`.
- `emit_methods_with_db_argument`:
  - If true, generated methods will accept a DBTX argument instead of storing a DBTX on the `*Queries` struct. Defaults to `false`.
- `json_tags_case_style`:
  - `camel` for camelCase, `pascal` for PascalCase, `snake` for snake_case or `none` to use the column name in the DB. Defaults to `none`.
- `output_db_file_name`:
  - Customize the name of the db file. Defaults to `db.go`.
- `output_models_file_name`:
  - Customize the name of the models file. Defaults to `models.go`.
- `output_querier_file_name`:
  - Customize the name of the querier file. Defaults to `querier.go`.
- `output_files_suffix`:
  - If specified the suffix will be added to the name of the generated files.

15.1 Type Overrides

The default mapping of PostgreSQL/MySQL types to Go types only uses packages outside the standard library when it must.

For example, the `uuid` PostgreSQL type is mapped to `github.com/google/uuid`. If a different Go package for UUIDs is required, specify the package in the `overrides` array. In this case, I'm going to use the `github.com/gofrs/uuid` instead.

```
version: "1"
packages: [...]
overrides:
  - go_type: "github.com/gofrs/uuid.UUID"
    db_type: "uuid"
```

Each override document has the following keys:

- `db_type`:
  - The PostgreSQL or MySQL type to override. Find the full list of supported types in postgresql_type.go or mysql_type.go.
- `go_type`:
  - A fully qualified name to a Go type to use in the generated code.
- `nullable`:
  - If true, use this type when a column is nullable. Defaults to `false`.

15.2 Per-Column Type Overrides

Sometimes you would like to override the Go type used in model or query generation for a specific field of a table and not on a type basis as described in the previous section.

This may be configured by specifying the `column` property in the override definition. `column` should be of the form `table.column` but you can be even more specific by specifying `schema.table.column` or `catalog.schema.table.column`.

```
version: "1"
packages: [...]
overrides:
  - column: "authors.id"
    go_type: "github.com/segmentio/ksuid.KSUID"
```

15.3 Package Level Overrides

Overrides can be configured globally, as demonstrated in the previous sections, or they can be configured on a per-package which scopes the override behavior to just a single package:

```
version: "1"
packages:
  - overrides: [...]
```

15.4 Renaming Struct Fields

Struct field names are generated from column names using a simple algorithm: split the column name on underscores and capitalize the first letter of each part.

```
account      -> Account
spotify_url -> SpotifyUrl
app_id       -> AppID
```

If you're not happy with a field's generated name, use the `rename` dictionary to pick a new name. The keys are column names and the values are the struct field name to use.

```
version: "1"
packages: [...]
rename:
  spotify_url: "SpotifyURL"
```

CHAPTER

SIXTEEN

DATATYPES

16.1 Arrays

PostgreSQL arrays are materialized as Go slices. Currently, only one-dimensional arrays are supported.

```
CREATE TABLE places (
  name text   not null,
  tags text[]
);
```

```
package db

type Place struct {
        Name string
        Tags []string
}
```

16.2 Dates and Time

All PostgreSQL time and date types are returned as `time.Time` structs. For null time or date values, the `NullTime` type from `database/sql` is used.

```
CREATE TABLE authors (
  id         SERIAL    PRIMARY KEY,
  created_at timestamp NOT NULL DEFAULT NOW(),
  updated_at timestamp
);
```

```
package db

import (
        "database/sql"
        "time"
)

type Author struct {
        ID        int
        CreatedAt time.Time
        UpdatedAt sql.NullTime
}
```

16.3 Enums

PostgreSQL enums are mapped to an aliased string type.

```
CREATE TYPE status AS ENUM (
  'open',
  'closed'
);

CREATE TABLE stores (
  name   text    PRIMARY KEY,
  status status  NOT NULL
);
```

```
package db

type Status string

const (
        StatusOpen   Status = "open"
        StatusClosed Status = "closed"
)

type Store struct {
        Name   string
        Status Status
}
```

16.4 Null

For structs, null values are represented using the appropriate type from the `database/sql` package.

```
CREATE TABLE authors (
  id   SERIAL PRIMARY KEY,
  name text   NOT NULL,
  bio  text
);
```

```
package db

import (
        "database/sql"
)

type Author struct {
        ID   int
        Name string
        Bio  sql.NullString
}
```

16.5 UUIDs

The Go standard library does not come with a `uuid` package. For UUID support, sqlc uses the excellent `github.com/google/uuid` package.

```
CREATE TABLE records (
  id   uuid PRIMARY KEY
);
```

```
package db

import (
        "github.com/google/uuid"
)

type Author struct {
        ID uuid.UUID
}
```

CHAPTER

SEVENTEEN

QUERY ANNOTATIONS

sqlc requires each query to have a small comment indicating the name and command. The format of this comment is as follows:

```
-- name: <name> <command>
```

17.1 :exec

The generated method will return the error from ExecContext.

```
-- name: DeleteAuthor :exec
DELETE FROM authors
WHERE id = $1;
```

```
func (q *Queries) DeleteAuthor(ctx context.Context, id int64) error {
        _, err := q.db.ExecContext(ctx, deleteAuthor, id)
        return err
}
```

17.2 :execresult

The generated method will return the sql.Result returned by ExecContext.

```
-- name: DeleteAllAuthors :execresult
DELETE FROM authors;
```

```
func (q *Queries) DeleteAllAuthors(ctx context.Context) (sql.Result, error) {
        return q.db.ExecContext(ctx, deleteAllAuthors)
}
```

17.3 :execrows

The generated method will return the number of affected rows from the result returned by ExecContext.

```
-- name: DeleteAllAuthors :execrows
DELETE FROM authors;
```

```
func (q *Queries) DeleteAllAuthors(ctx context.Context) (int64, error) {
        _, err := q.db.ExecContext(ctx, deleteAllAuthors)
        // ...
}
```

17.4 :many

The generated method will return a slice of records via QueryContext.

```
-- name: ListAuthors :many
SELECT * FROM authors
ORDER BY name;
```

```
func (q *Queries) ListAuthors(ctx context.Context) ([]Author, error) {
        rows, err := q.db.QueryContext(ctx, listAuthors)
        // ...
}
```

17.5 :one

The generated method will return a single record via QueryRowContext.

```
-- name: GetAuthor :one
SELECT * FROM authors
WHERE id = $1 LIMIT 1;
```

```
func (q *Queries) GetAuthor(ctx context.Context, id int64) (Author, error) {
        row := q.db.QueryRowContext(ctx, getAuthor, id)
        // ...
}
```

CHAPTER

EIGHTEEN

DATABASE AND LANGUAGE SUPPORT

Language	MySQL	PostgreSQL
Go	Stable	Stable
Kotlin	Beta	Beta
Python	Experimental	Experimental

18.1 Future Language Support

- C#
- TypeScript

18.2 Future Database Support

- SQLite

CHAPTER

NINETEEN

ENVIRONMENT VARIABLES

19.1 SQLCDEBUG

The `SQLCDEBUG` variable controls debugging variables within the runtime. It is a comma-separated list of name=val pairs settings.

19.1.1 dumpast

The `dumpast` command shows the SQL AST that was generated by the parser. Note that this is the generic SQL AST, not the engine-specific SQL AST.

```
SQLCDEBUG=dumpast=1
```

```
([]interface {}) (len=1 cap=1) {
 (*catalog.Catalog)(0xc0004f48c0)({
  Comment: (string) "",
  DefaultSchema: (string) (len=6) "public",
  Name: (string) "",
  Schemas: ([]*catalog.Schema) (len=3 cap=4) {
   (*catalog.Schema)(0xc0004f4930)({
    Name: (string) (len=6) "public",
    Tables: ([]*catalog.Table) (len=1 cap=1) {
     (*catalog.Table)(0xc00052ff20)({
      Rel: (*ast.TableName)(0xc00052fda0)({
       Catalog: (string) "",
       Schema: (string) "",
       Name: (string) (len=7) "authors"
      }),
```

19.1.2 dumpcatalog

The `dumpcatalog` command outputs the entire catalog. If you're using MySQL or PostgreSQL, this can be a bit overwhelming. Expect this output to change in future versions.

```
SQLCDEBUG=dumpcatalog=1
```

```
([]interface {}) (len=1 cap=1) {
 (*catalog.Catalog)(0xc00050d1f0)({
  Comment: (string) "",
  DefaultSchema: (string) (len=6) "public",
```

(continues on next page)

(continued from previous page)

```
  Name: (string) "",
  Schemas: ([]*catalog.Schema) (len=3 cap=4) {
   (*catalog.Schema)(0xc00050d260)({
    Name: (string) (len=6) "public",
    Tables: ([]*catalog.Table) (len=1 cap=1) {
     (*catalog.Table)(0xc0000c0840)({
      Rel: (*ast.TableName)(0xc0000c06c0)({
       Catalog: (string) "",
       Schema: (string) "",
       Name: (string) (len=7) "authors"
      }),
```

19.1.3 trace

The `trace` command is helpful for tracking down performance issues.

`SQLCDEBUG=trace=1`

By default, the trace output is written to `trace.out` in the current working directory. You can configure a different path if needed.

`SQLCDEBUG=trace=name.out`

View the execution trace using the Go `trace` tool.

```
go tool trace trace.out
```

There's a ton of different views for the trace output, but here's an example log showing the execution time for each package.

```
0.000043897          .                1         task sqlc (id 1, parent 0) created
0.000144923          .   101026        1          region generate started (duration:␣
↪47.619781ms)
0.001048975          .   904052        1          region package started (duration:␣
↪14.588456ms)
0.001054616          .     5641        1          name=authors dir=/Users/kyle/
↪projects/sqlc/examples/python language=python
0.001071257          .    16641        1          region parse started (duration: 7.
↪966549ms)
0.009043960          .  7972703        1          region codegen started (duration: 6.
↪587086ms)
0.009171704          .   127744        1          new goroutine 35: text/template/
↪parse.lex·dwrap·1
0.010361654          .  1189950        1          new goroutine 36: text/template/
↪parse.lex·dwrap·1
0.015641815          .  5280161        1          region package started (duration:␣
↪10.904938ms)
0.015644943          .     3128        1          name=booktest dir=/Users/kyle/
↪projects/sqlc/examples/python language=python
0.015647431          .     2488        1          region parse started (duration: 4.
↪207749ms)
0.019860308          .  4212877        1          region codegen started (duration: 6.
↪681624ms)
0.020028488          .   168180        1          new goroutine 37: text/template/
↪parse.lex·dwrap·1
0.021020310          .   991822        1          new goroutine 8: text/template/
↪parse.lex·dwrap·1
```

(continues on next page)

(continued from previous page)

```
0.026551163          .  5530853         1         region package started (duration: 9.
↪217294ms)
0.026554368          .     3205         1         name=jets dir=/Users/kyle/projects/
↪sqlc/examples/python language=python
0.026556804          .     2436         1         region parse started (duration: 3.
↪491005ms)
0.030051911          .  3495107         1         region codegen started (duration: 5.
↪711931ms)
0.030213937          .   162026         1         new goroutine 20: text/template/
↪parse.lex·dwrap·1
0.031099938          .   886001         1         new goroutine 38: text/template/
↪parse.lex·dwrap·1
0.035772637          .  4672699         1         region package started (duration:␣
↪10.267039ms)
0.035775688          .     3051         1         name=ondeck dir=/Users/kyle/
↪projects/sqlc/examples/python language=python
0.035778150          .     2462         1         region parse started (duration: 4.
↪094518ms)
0.039877181          .  4099031         1         region codegen started (duration: 6.
↪156341ms)
0.040010771          .   133590         1         new goroutine 39: text/template/
↪parse.lex·dwrap·1
0.040894567          .   883796         1         new goroutine 40: text/template/
↪parse.lex·dwrap·1
0.046042779          .  5148212         1         region writefiles started␣
↪(duration: 1.718259ms)
0.047767781          .  1725002         1         task end
```

CHAPTER

TWENTY

CHANGELOG

All notable changes to this project will be documented in this file.

20.1 1.12.0

Released 2022-02-05

20.1.1 Bug

- ALTER TABLE SET SCHEMA (#1409)

20.1.2 Bug Fixes

- Update ANTLR v4 go.mod entry (#1336)
- Check delete statements for CTEs (#1329)
- Fix validation of GROUP BY on field aliases (#1348)
- Fix imports when non-copyfrom queries needed imports that copyfrom queries didn't (#1386)
- Remove extra comment newline (#1395)
- Enable strict function checking (#1405)

20.1.3 Documentation

- Bump version to 1.11.0 (#1308)

20.1.4 Features

- Inheritance (#1339)
- Generate query code using ASTs instead of templates (#1338)
- Add support for CREATE TABLE a (LIKE b) (#1355)
- Add support for sql.NullInt16 (#1376)

20.1.5 Miscellaneous Tasks

- Add tests for :exec{result,rows} (#1344)
- Delete template-based codegen (#1345)

20.1.6 Build

- Bump github.com/jackc/pgx/v4 from 4.14.0 to 4.14.1 (#1316)
- Bump golang from 1.17.3 to 1.17.4 (#1331)
- Bump golang from 1.17.4 to 1.17.5 (#1337)
- Bump github.com/spf13/cobra from 1.2.1 to 1.3.0 (#1343)
- Remove devel Docker build
- Bump golang from 1.17.5 to 1.17.6 (#1369)
- Bump github.com/google/go-cmp from 0.5.6 to 0.5.7 (#1382)
- Format all Go code (#1387)

20.2 1.11.0

Released 2021-11-24

20.2.1 Bug Fixes

- Update incorrect signatures (#1180)
- Correct aggregate func sig (#1182)
- Jsonb_build_object (#1211)
- Case-insensitive identifiers (#1216)
- Incorrect handling of meta (#1228)
- Detect invalid INSERT expression (#1231)
- Respect alias name for coalesce (#1232)
- Mark nullable when casting NULL (#1233)
- Support nullable fields in joins for MySQL engine (#1249)
- Fix between expression handling of table references (#1268)
- Support nullable fields in joins on same table (#1270)
- Fix missing binds in ORDER BY (#1273)
- Set RV for TargetList items on updates (#1252)
- Fix MySQL parser for query without trailing semicolon (#1282)
- Validate table alias references (#1283)
- Add support for MySQL ON DUPLICATE KEY UPDATE (#1286)

- Support references to columns in joined tables in UPDATE statements (#1289)
- Add validation for GROUP BY clause column references (#1285)
- Prevent variable redeclaration in single param conflict (#1298)
- Use common params struct field for same named params (#1296)

20.2.2 Documentation

- Replace deprecated go get with go install (#1181)
- Fix package name referenced in tutorial (#1202)
- Add environment variables (#1264)
- Add go.17+ install instructions (#1280)
- Warn about golang-migrate file order (#1302)

20.2.3 Features

- Instrument compiler via runtime/trace (#1258)
- Add MySQL support for BETWEEN arguments (#1265)

20.2.4 Refactor

- Move from io/ioutil to io and os package (#1164)

20.2.5 Styling

- Apply gofmt to sample code (#1261)

20.2.6 Build

- Bump golang from 1.17.0 to 1.17.1 (#1173)
- Bump eskatos/gradle-command-action from 1 to 2 (#1220)
- Bump golang from 1.17.1 to 1.17.2 (#1227)
- Bump github.com/pganalyze/pg_query_go/v2 (#1234)
- Bump actions/checkout from 2.3.4 to 2.3.5 (#1238)
- Bump babel from 2.9.0 to 2.9.1 in /docs (#1245)
- Bump golang from 1.17.2 to 1.17.3 (#1272)
- Bump actions/checkout from 2.3.5 to 2.4.0 (#1267)
- Bump github.com/lib/pq from 1.10.3 to 1.10.4 (#1278)
- Bump github.com/jackc/pgx/v4 from 4.13.0 to 4.14.0 (#1303)

20.2.7 Cmd/sqlc

- Bump version to v1.11.0

20.3 1.10.0

Released 2021-09-07

20.3.1 Documentation

- Fix invalid language support table (#1161)
- Add a getting started guide for MySQL (#1163)

20.3.2 Build

- Bump golang from 1.16.7 to 1.17.0 (#1129)
- Bump github.com/lib/pq from 1.10.2 to 1.10.3 (#1160)

20.3.3 Ci

- Upgrade Go to 1.17 (#1130)

20.3.4 Cmd/sqlc

- Bump version to v1.10.0 (#1165)

20.3.5 Codegen/golang

- Consolidate import logic (#1139)
- Add pgx support for range types (#1146)
- Use pgtype for hstore when using pgx (#1156)

20.3.6 Codgen/golang

- Use p[gq]type for network address types (#1142)

20.3.7 Endtoend

- Run `go test` in CI (#1134)

20.3.8 Engine/mysql

- Add support for LIKE (#1162)

20.3.9 Golang

- Output NullUUID when necessary (#1137)

20.4 1.9.0

Released 2021-08-13

20.4.1 Documentation

- Update documentation (a bit) for v1.9.0 (#1117)

20.4.2 Build

- Bump golang from 1.16.6 to 1.16.7 (#1107)

20.4.3 Cmd/sqlc

- Bump version to v1.9.0 (#1121)

20.4.4 Compiler

- Add tests for COALESCE behavior (#1112)
- Handle subqueries in SELECT statements (#1113)

20.5 1.8.0

Released 2021-05-03

20.5.1 Documentation

- Add language support Matrix (#920)

20.5.2 Features

- Add case style config option (#905)

20.5.3 Python

- Eliminate runtime package and use sqlalchemy (#939)

20.5.4 Build

- Bump github.com/google/go-cmp from 0.5.4 to 0.5.5 (#926)
- Bump github.com/lib/pq from 1.9.0 to 1.10.0 (#931)
- Bump golang from 1.16.0 to 1.16.1 (#935)
- Bump golang from 1.16.1 to 1.16.2 (#942)
- Bump github.com/jackc/pgx/v4 from 4.10.1 to 4.11.0 (#956)
- Bump github.com/go-sql-driver/mysql from 1.5.0 to 1.6.0 (#961)
- Bump github.com/pganalyze/pg_query_go/v2 (#965)
- Bump urllib3 from 1.26.3 to 1.26.4 in /docs (#968)
- Bump golang from 1.16.2 to 1.16.3 (#963)
- Bump github.com/lib/pq from 1.10.0 to 1.10.1 (#980)

20.5.5 Cmd

- Add the –experimental flag (#929)
- Fix sqlc init (#959)

20.5.6 Cmd/sqlc

- Bump version to v1.7.1-devel (#913)
- Bump version to v1.8.0

20.5.7 Codegen

- Generate valid enum names for symbols (#972)

20.5.8 Postgresql

- Support generated columns
- Add test for PRIMARY KEY INCLUDE
- Add tests for CREATE TABLE PARTITION OF
- CREATE TRIGGER EXECUTE FUNCTION
- Add support for renaming types (#971)

20.5.9 Sql/ast

- Resolve return values from functions (#964)

20.5.10 Workflows

- Only run tests once (#924)

20.6 1.7.0

Released 2021-02-28

20.6.1 Bug Fixes

- Struct tag formatting (#833)

20.6.2 Documentation

- Include all the existing Markdown files (#877)
- Split docs into four sections (#882)
- Reorganize and consolidate documentation
- Add link to Windows download (#888)
- Shorten the README (#889)

20.6.3 Features

- Adding support for pgx/v4
- Adding support for pgx/v4

20.6.4 README

- Add Go Report Card badge (#891)

20.6.5 Build

- Bump github.com/google/go-cmp from 0.5.3 to 0.5.4 (#813)
- Bump github.com/lib/pq from 1.8.0 to 1.9.0 (#820)
- Bump golang from 1.15.5 to 1.15.6 (#822)
- Bump github.com/jackc/pgx/v4 from 4.9.2 to 4.10.0 (#823)
- Bump github.com/jackc/pgx/v4 from 4.10.0 to 4.10.1 (#839)
- Bump golang from 1.15.6 to 1.15.7 (#855)
- Bump golang from 1.15.7 to 1.15.8 (#881)
- Bump github.com/spf13/cobra from 1.1.1 to 1.1.2 (#892)
- Bump golang from 1.15.8 to 1.16.0 (#897)
- Bump github.com/lfittl/pg_query_go from 1.0.1 to 1.0.2 (#901)
- Bump github.com/spf13/cobra from 1.1.2 to 1.1.3 (#893)

20.6.6 Catalog

- Improve alter column type (#818)

20.6.7 Ci

- Uprade to Go 1.15 (#887)

20.6.8 Cmd

- Allow config file location to be specified (#863)

20.6.9 Cmd/sqlc

- Bump to version v1.6.1-devel (#807)
- Bump version to v1.7.0 (#912)

20.6.10 Codegen/golang

- Make sure to import net package (#858)

20.6.11 Compiler

- Support UNION query

20.6.12 Dolphin

- Generate bools for tinyint(1)
- Support joins in update statements (#883)
- Add support for union query

20.6.13 Endtoend

- Add tests for INTERSECT and EXCEPT

20.6.14 Go.mod

- Update to go 1.15 and run 'go mod tidy' (#808)

20.6.15 Mysql

- Compile tinyint(1) to bool (#873)

20.6.16 Sql/ast

- Add enum values for SetOperation

20.7 1.6.0

Released 2020-11-23

20.7.1 Dolphin

- Implement Rename (#651)
- Skip processing view drops (#653)

20.7.2 README

- Update language / database support (#698)

20.7.3 Astutils

- Fix Params rewrite call (#674)

20.7.4 Build

- Bump golang from 1.14 to 1.15.3 (#765)
- Bump docker/build-push-action from v1 to v2.1.0 (#764)
- Bump github.com/google/go-cmp from 0.4.0 to 0.5.2 (#766)
- Bump github.com/spf13/cobra from 1.0.0 to 1.1.1 (#767)
- Bump github.com/jackc/pgx/v4 from 4.6.0 to 4.9.2 (#768)
- Bump github.com/lfittl/pg_query_go from 1.0.0 to 1.0.1 (#773)
- Bump github.com/google/go-cmp from 0.5.2 to 0.5.3 (#783)
- Bump golang from 1.15.3 to 1.15.5 (#782)
- Bump github.com/lib/pq from 1.4.0 to 1.8.0 (#769)

20.7.5 Catalog

- Improve variadic argument support (#804)

20.7.6 Cmd/sqlc

- Bump to version v1.6.0 (#806)

20.7.7 Codegen

- Fix errant database/sql imports (#789)

20.7.8 Compiler

- Use engine-specific reserved keywords (#677)

20.7.9 Dolphi

- Add list of builtin functions (#795)

20.7.10 Dolphin

- Update to the latest MySQL parser (#665)
- Add ENUM() support (#676)
- Add test for table aliasing (#684)
- Add MySQL ddl_create_table test (#685)
- Implete TRUNCATE table (#697)
- Represent tinyint as int32 (#797)
- Add support for coalesce (#802)
- Add function signatures (#796)

20.7.11 Endtoend

- Add MySQL json test (#692)
- Add MySQL update set multiple test (#696)

20.7.12 Examples

- Use generated enum constants in db_test (#678)
- Port ondeck to MySQL (#680)
- Add MySQL authors example (#682)

20.7.13 Internal/cmd

- Print correct config file on parse failure (#749)

20.7.14 Kotlin

- Remove runtime dependency (#774)

20.7.15 Metadata

- Support multiple comment prefixes (#683)

20.7.16 Postgresql

- Support string concat operator (#701)

20.7.17 Sql/catalog

- Add support for variadic functions (#798)

20.8 1.5.0

Released 2020-08-05

20.8.1 Documentation

- Build sqlc using Go 1.14 (#549)

20.8.2 Cmd

- Add debugging support (#573)

20.8.3 Cmd/sqlc

- Bump version to v1.4.1-devel (#548)
- Bump version to v1.5.0

20.8.4 Compiler

- Support calling functions with defaults (#635)
- Skip func args without a paramRef (#636)
- Return a single column from coalesce (#639)

20.8.5 Config

- Add emit_empty_slices to version one (#552)

20.8.6 Contrib

- Add generated code for contrib

20.8.7 Dinosql

- Remove deprecated package (#554)

20.8.8 Dolphin

- Add support for column aliasing (#566)
- Implement star expansion for subqueries (#619)
- Implement exapansion with reserved words (#620)
- Implement parameter refs (#621)
- Implement limit and offest (#622)
- Implement inserts (#623)
- Implement delete (#624)
- Implement simple update statements (#625)
- Implement INSERT . . . SELECT (#626)
- Use test driver instead of TiDB driver (#629)
- Implement named parameters via sqlc.arg() (#632)

20.8.9 Endtoend

- Add MySQL test for SELECT * JOIN (#565)
- Add MySQL test for inflection (#567)

20.8.10 Engine

- Create engine package (#556)

20.8.11 Equinox

- Use the new equinox-io/setup action (#586)

20.8.12 Examples

- Run tests for MySQL booktest (#627)

20.8.13 Golang

- Add support for the money type (#561)
- Generate correct types for int2 and int8 (#579)

20.8.14 Internal

- Rm catalog, pg, postgres packages (#555)

20.8.15 Mod

- Downgrade TiDB package to fix build (#603)

20.8.16 Mysql

- Upgrade to the latest vitess commit (#562)
- Support to infer type of a duplicated arg (#615)
- Allow some builtin functions to be nullable (#616)

20.8.17 Postgresql

- Generate all functions in pg_catalog (#550)
- Remove pg_catalog schema from tests (#638)
- Move contrib code to a package

20.8.18 Sql/catalog

- Fix comparison of pg_catalog types (#637)

20.8.19 Tools

- Generate functions for all of contrib

20.8.20 Workflow

- Migrate to equinox-io/setup-release-tool (#614)

20.9 1.4.0

Released 2020-06-17

20.9.1 Dockerfile

- Add version build argument (#487)

20.9.2 MySQL

- Prevent Panic when WHERE clause contains parenthesis. (#531)

20.9.3 README

- Document emit_exact_table_names (#486)

20.9.4 All

- Remove the exp build tag (#507)

20.9.5 Catalog

- Support functions with table parameters (#541)

20.9.6 Cmd

- Bump to version 1.3.1-devel (#485)

20.9.7 Cmd/sqlc

- Bump version to v1.4.0 (#547)

20.9.8 Codegen

- Add the new codegen packages (#513)
- Add the :execresult query annotation (#542)

20.9.9 Compiler

- Validate function calls (#505)
- Port bottom of parseQuery (#510)
- Don't mutate table name (#517)
- Enable experimental parser by default (#518)
- Apply rename rules to enum constants (#523)
- Temp fix for typecast function parameters (#530)

20.9.10 Endtoend

- Standardize JSON formatting (#490)
- Add per-test configuration files (#521)
- Read expected stderr failures from disk (#527)

20.9.11 Internal/dinosql

- Check parameter style before ref (#488)
- Remove unneeded column suffix (#492)
- Support named function arguments (#494)

20.9.12 Internal/postgresql

- Fix NamedArgExpr rewrite (#491)

20.9.13 Multierr

- Move dinosql.ParserErr to a new package (#496)

20.9.14 Named

- Port parameter style validation to SQL (#504)

20.9.15 Parser

- Support columns from subselect statements (#489)

20.9.16 Rewrite

- Move parameter rewrite to package (#499)

20.9.17 Sqlite

- Use convert functions instead of the listener (#519)

20.9.18 Sqlpath

- Move ReadSQLFiles into a separate package (#495)

20.9.19 Validation

- Move query validation to separate package (#498)

20.10 1.3.0

Released 2020-05-12

20.10.1 Makefile

- Update target (#449)

20.10.2 README

- Add Myles as a sponsor (#469)

20.10.3 Testing

- Make sure all Go examples build (#480)

20.10.4 Cmd

- Bump version to v1.3.0 (#484)

20.10.5 Cmd/sqlc

- Bump version to v1.2.1-devel (#442)

20.10.6 Dinosql

- Inline addFile (#446)
- Add PostgreSQL support for TRUNCATE (#448)

20.10.7 Gen

- Emit json.RawMessage for JSON columns (#461)

20.10.8 Go.mod

- Use latest lib/pq (#471)

20.10.9 Parser

- Use same function to load SQL files (#483)

20.10.10 Postgresql

- Fix panic walking CreateTableAsStmt (#475)

20.11 1.2.0

Released 2020-04-07

20.11.1 Documentation

- Publish to Docker Hub (#422)

20.11.2 README

- Docker installation docs (#424)

20.11.3 Cmd/sqlc

- Bump version to v1.1.1-devel (#407)
- Bump version to v1.2.0 (#441)

20.11.4 Gen

- Add special case for “campus” (#435)
- Properly quote reserved keywords on expansion (#436)

20.11.5 Migrations

- Move migration parsing to new package (#427)

20.11.6 Parser

- Generate correct types for SELECT EXISTS (#411)

20.12 1.1.0

Released 2020-03-17

20.12.1 README

- Add installation instructions (#350)
- Add section on running tests (#357)
- Fix typo (#371)

20.12.2 Ast

- Add AST for ALTER TABLE ADD / DROP COLUMN (#376)
- Add support for CREATE TYPE as ENUM (#388)
- Add support for CREATE / DROP SCHEMA (#389)

20.12.3 Astutils

- Apply changes to the ValuesList slice (#372)

20.12.4 Cmd

- Return v1.0.0 (#348)
- Return next bug fix version (#349)

20.12.5 Cmd/sqlc

- Bump version to v1.1.0 (#406)

20.12.6 Compiler

- Wire up the experimental parsers

20.12.7 Config

- Remove “emit_single_file” option (#367)

20.12.8 Dolphin

- Add experimental parser for MySQL

20.12.9 Gen

- Add option to emit single file for Go (#366)
- Add support for the ltree extension (#385)

20.12.10 Go.mod

- Add packages for MySQL and SQLite parsers

20.12.11 Internal/dinosql

- Support Postgres macaddr type in Go (#358)

20.12.12 Internal/endtoend

- Remove %w (#354)

20.12.13 Kotlin

- Add Query class to support timeout and cancellation (#368)

20.12.14 Postgresql

- Add experimental parser for MySQL

20.12.15 Sql

- Add generic SQL AST

20.12.16 Sql/ast

- Port support for COMMENT ON (#391)
- Implement DROP TYPE (#397)
- Implement ALTER TABLE RENAME (#398)
- Implement ALTER TABLE RENAME column (#399)
- Implement ALTER TABLE SET SCHEMA (#400)

20.12.17 Sql/catalog

- Port tests over from catalog pkg (#402)

20.12.18 Sql/errors

- Add a new errors package (#390)

20.12.19 Sqlite

- Add experimental parser for SQLite

20.13 1.0.0

Released 2020-02-18

20.13.1 Documentation

- Add documentation for query commands (#270)
- Add named parameter documentation (#332)

20.13.2 README

- Add sponsors section (#333)

20.13.3 Cmd

- Remove parse subcommand (#322)

20.13.4 Config

- Parse V2 config format
- Add support for YAML (#336)

20.13.5 Examples

- Add the jets and booktest examples (#237)
- Move sqlc.json into examples folder (#238)
- Add the authors example (#241)
- Add build tag to authors tests (#319)

20.13.6 Internal

- Allow CTE to be used with UPDATE (#268)
- Remove the PackageMap from settings (#295)

20.13.7 Internal/config

- Create new config package (#313)

20.13.8 Internal/dinosql

- Emit Querier interface (#240)
- Strip leading “go-“ or trailing “-go” from import (#262)
- Overrides can now be basic types (#271)
- Import needed types for Querier (#285)
- Handle schema-scoped enums (#310)
- Ignore golang-migrate rollbacks (#320)

20.13.9 Internal/endtoend

- Move more tests to the record/replay framework
- Add update test for named params (#329)

20.13.10 Internal/mysql

- Fix flaky test (#242)
- Port tests to endtoend package (#315)

20.13.11 Internal/parser

- Resolve nested CTEs (#324)
- Error if last query is missing (#325)
- Support joins with aliases (#326)
- Remove print statement (#327)

20.13.12 Internal/sqlc

- Add support for composite types (#311)

20.13.13 Kotlin

- Support primitives
- Arrays, enums, and dates
- Generate examples
- README for examples
- Factor out db setup extension
- Fix enums, use List instead of Array
- Port Go tests for examples
- Rewrite numbered params to positional params
- Always use use, fix indents
- Unbox query params

20.13.14 Parser

- Attach range vars to insert params
- Attach range vars to insert params (#342)
- Remove dead code (#343)

20.14 0.1.0

Released 2020-01-07

20.14.1 Documentation

- Replace remaining references to DinoSQL with sqlc (#149)

20.14.2 README

- Fix download links (#66)
- Add LIMIT 1 to query that should return one (#99)

20.14.3 Catalog

- Support "ALTER TABLE . . . DROP CONSTRAINT . . . " (#34)
- Differentiate functions with different argument types (#51)

20.14.4 Ci

- Enable tests on pull requests

20.14.5 Cmd

- Include filenames in error messages (#69)
- Do not output any changes on error (#72)

20.14.6 Dinosql/internal

- Add lower and upper functions (#215)
- Ignore alter sequence commands (#219)

20.14.7 Gen

- Add DO NOT EDIT comments to generated code (#50)
- Include all schemas when generating models (#90)
- Prefix structs with schema name (#91)
- Generate single import for uuid package (#98)
- Use same import logic for all Go files
- Pick correct struct to return for queries (#107)
- Create consistent JSON tags (#110)
- Add Close method to Queries struct (#127)
- Ignore empty override settings (#128)
- Turn SQL comments into Go comments (#136)

20.14.8 Internal/catalog

- Parse unnamed function arguments (#166)

20.14.9 Internal/dinosql

- Prepare() with no GoQueries still valid (#95)
- Fix multiline comment rendering (#142)
- Dereference alias nodes on walk (#158)
- Ignore sql-migrate rollbacks (#160)
- Sort imported packages (#165)
- Add support for timestamptz (#169)
- Error on missing queries (#180)
- Use more database/sql null types (#182)
- Support the pg_temp schema (#183)
- Override columns with array type (#184)
- Implement robust expansion
- Implement robust expansion (#186)
- Add COMMENT ON support (#191)
- Add DATE support
- Add DATE support (#196)
- Filter out invalid characters (#198)
- Quote reserved keywords (#205)
- Return parser errors first (#207)
- Implement advisory locks (#212)

- Error on duplicate query names (#221)
- Fix incorrect enum names (#223)
- Add support for numeric types
- Add support for numeric types (#228)

20.14.10 Internal/dinosql/testdata/ondeck

- Add Makefile (#156)

20.14.11 Ondeck

- Move all tests to GitHub CI (#58)

20.14.12 ParseQuery

- Return either a query or an error (#178)

20.14.13 Parser

- Use schema when resolving catalog refs (#82)
- Support function calls in expressions (#104)
- Correctly handle single files (#119)
- Return error if missing RETURNING (#131)
- Add support for mathmatical operators (#132)
- Add support for simple case expressions (#134)
- Error on mismatched INSERT input (#135)
- Set IsArray on joined columns (#139)

20.14.14 Pg

- Store functions in the catalog (#41)
- Add location to errors (#73)

CHAPTER

TWENTYONE

DEVELOPING SQLC

21.1 Building

For local development, install `sqlc` under an alias. We suggest `sqlc-dev`.

```
go build -o ~/go/bin/sqlc-dev ./cmd/sqlc
```

21.2 Running Tests

```
go test ./...
```

To run the tests in the examples folder, use the `examples` tag.

```
go test --tags=examples ./...
```

These tests require locally-running database instances. Run these databases using Docker Compose.

```
docker-compose up -d
```

The tests use the following environment variables to connect to the database

21.2.1 For PostgreSQL

```
Variable     Default Value
--------------------------
PG_HOST      127.0.0.1
PG_PORT      5432
PG_USER      postgres
PG_PASSWORD  mysecretpassword
PG_DATABASE  dinotest
```

21.2.2 For MySQL

```
Variable     Default Value
--------------------------
MYSQL_HOST      127.0.0.1
MYSQL_PORT      3306
MYSQL_USER      root
MYSQL_ROOT_PASSWORD  mysecretpassword
MYSQL_DATABASE  dinotest
```

21.3 Regenerate expected test output

If you need to update a large number of expected test output in the `internal/endtoend/testdata` directory, run the `regenerate` script.

```
go build -o ~/go/bin/sqlc-dev ./cmd/sqlc
go run scripts/regenerate/main.go
```

Note that this uses the `sqlc-dev` binary, not `sqlc` so make sure you have an up to date `sqlc-dev` binary.

CHAPTER

TWENTYTWO

PRIVACY AND DATA COLLECTION

These days, it feels like every piece of software is tracking you. From your browser, to your phone, to your terminal, programs collect as much data about you as possible and send it off to the cloud for analysis.

We believe the best way to keep data safe is to never collect it in the first place.

22.1 Our Privacy Pledge

The `sqlc` program does not collect any information. It does not send crash reports to a third-party. It does not gather anonymous aggregate user behaviour analytics.

No analytics. No finger-printing. No tracking.

Not now and not in the future.

22.2 Distribution Channels

We distribute sqlc using popular package managers such as Homebrew and Snapcraft. These package managers and their associated command-line tools do collect usage metrics.

We use these services to make it easy to for users to install sqlc. There will always be an option to download sqlc from a stable URL.