sqlc

Release 1.18.0

Kyle Conroy

Jul 05, 2023

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
```

Run `sqlc` using `docker run` in the Command Prompt on Windows (`cmd`):

```
docker run --rm -v "%cd%:/src" -w /src kjconroy/sqlc generate
```

1.5 Downloads

Get pre-built binaries for *v1.18.0*:

- Linux
- macOS
- Windows (MySQL only)

See downloads.sqlc.dev for older versions.

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:

```
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)
```

(continues on next page)

(continued from previous page)

```
        // 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},
        })
        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.sqlc.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: "2"
sql:
  - engine: "postgresql"
    queries: "query.sql"
    schema: "schema.sql"
    gen:
      go:
        package: "tutorial"
        out: "tutorial"
```

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

```
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 (
```

(continues on next page)

(continued from previous page)

```
  name, bio
) VALUES (
  $1, $2
)
RETURNING *;

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

If you **do not** want your SQL `UPDATE` queries to return the updated record to the user, add this to `query.sql`:

```
-- name: UpdateAuthor :exec
UPDATE authors
  set name = $2,
  bio = $3
WHERE id = $1;
```

Otherwise, to return the updated record to the user, add this to `query.sql`:

```
-- name: UpdateAuthor :one
UPDATE authors
  set name = $2,
  bio = $3
WHERE id = $1
RETURNING *;
```

You are now ready to generate code. 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"
```

(continues on next page)

(continued from previous page)

```
        _ "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",
                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 ./...
```

sqlc generates readable, **idiomatic** Go code that you otherwise would have had to write yourself. Take a look in the `tutorial` package to see what code sqlc generated.

CHAPTER

FOUR

GETTING STARTED WITH SQLITE

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: 2
sql:
  - engine: "sqlite"
    schema: "schema.sql"
    queries: "query.sql"
    gen:
      go:
        package: "tutorial"
        out: "tutorial"
```

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   INTEGER 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 :one
INSERT INTO authors (
```

(continues on next page)

(continued from previous page)

```
  name, bio
) VALUES (
  ?, ?
)
RETURNING *;

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

For SQL UPDATE if you do not want to return the updated record to the user, add this to the `query.sql` file:

```
-- name: UpdateAuthor :exec
UPDATE authors
set name = ?,
bio = ?
WHERE id = ?;
```

Otherwise, to return the updated record back to the user, add this to the `query.sql` file:

```
-- name: UpdateAuthor :one
UPDATE authors
set name = ?,
bio = ?
WHERE id = ?
RETURNING *;
```

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"
```

(continues on next page)

(continued from previous page)

```
        _ "embed"

        _ "github.com/mattn/go-sqlite3"
)

//go:embed schema.sql
var ddl string

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

        db, err := sql.Open("sqlite3", ":memory:")
        if err != nil {
                return err
        }

        // create tables
        if _, err := db.ExecContext(ctx, ddl); 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",
                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
}
```

(continues on next page)

(continued from previous page)

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

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

```
go mod tidy
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

FIVE

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
        }
```

(continues on next page)

(continued from previous page)

```
        return items, nil
}
```

5.1 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) {
```

(continues on next page)

(continued from previous page)

```
        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
`

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
}
```

5.2 Passing a slice as a parameter to a query

5.2.1 PostgreSQL

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"
```

(continues on next page)

(continued from previous page)

```
)

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}
}

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
}
```

5.2.2 MySQL

MySQL differs from PostgreSQL in that placeholders must be generated based on the number of elements in the slice you pass in. Though trivial it is still something of a nuisance. The passed in slice must not be nil or empty or an error will be returned (ie not a panic). The placeholder insertion location is marked by the meta-function `sqlc.slice()` (which is similar to `sqlc.arg()` that you see documented under Naming parameters).

To rephrase, the `sqlc.slice('param')` behaves identically to `sqlc.arg()` it terms of how it maps the explicit argument to the function signature, eg:

- `sqlc.slice('ids')` maps to `ids []GoType` in the function signature
- `sqlc.slice(cust_ids)` maps to `custIds []GoType` in the function signature (like `sqlc.arg()`, the parameter does not have to be quoted)

This feature is not compatible with `emit_prepared_queries` statement found in the *Configuration file*.

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

-- name: ListAuthorsByIDs :many
SELECT * FROM authors
WHERE id IN (sqlc.slice('ids'));
```

The above SQL will generate the following code:

```
package db

import (
        "context"
        "database/sql"
        "fmt"
        "strings"
)

type Author struct {
        ID        int
        Bio       string
        BirthYear int
}

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

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

const listAuthorsByIDs = `-- name: ListAuthorsByIDs :many
SELECT id, bio, birth_year FROM authors
WHERE id IN (/*SLICE:ids*/?)
`

func (q *Queries) ListAuthorsByIDs(ctx context.Context, ids []int64) ([]Author, error) {
        sql := listAuthorsByIDs
        var queryParams []interface{}
        if len(ids) == 0 {
                return nil, fmt.Errorf("slice ids must have at least one element")
        }
        for _, v := range ids {
                queryParams = append(queryParams, v)
        }
        sql = strings.Replace(sql, "/*SLICE:ids*/?", strings.Repeat(",?", len(ids))[1:],␣
↪1)
        rows, err := q.db.QueryContext(ctx, sql, queryParams...)
        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

SIX

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
```

(continues on next page)

(continued from previous page)

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

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

SEVEN

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{}) (sql.Result, 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
}
```

7.1 Returning columns from inserted rows

sqlc has full support for the `RETURNING` statement.

```
-- Example queries for sqlc
CREATE TABLE authors (
  id   BIGSERIAL PRIMARY KEY,
  name text      NOT NULL,
  bio  text
);

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

-- name: CreateAuthorAndReturnId :one
INSERT INTO authors (
  name, bio
) VALUES (
  $1, $2
)
RETURNING id;
```

```
package db

import (
        "context"
        "database/sql"
)

const createAuthor = `-- name: CreateAuthor :one
INSERT INTO authors (
  name, bio
) VALUES (
  $1, $2
)
RETURNING id, name, bio
`

type CreateAuthorParams struct {
        Name string
        Bio  sql.NullString
}

func (q *Queries) CreateAuthor(ctx context.Context, arg CreateAuthorParams) (Author,␣
↪error) {
        row := q.db.QueryRowContext(ctx, createAuthor, arg.Name, arg.Bio)
        var i Author
```

(continues on next page)

(continued from previous page)

```
        err := row.Scan(&i.ID, &i.Name, &i.Bio)
        return i, err
}

const createAuthorAndReturnId = `-- name: CreateAuthorAndReturnId :one
INSERT INTO authors (
  name, bio
) VALUES (
  $1, $2
)
RETURNING id
`

type CreateAuthorAndReturnIdParams struct {
        Name string
        Bio  sql.NullString
}

func (q *Queries) CreateAuthorAndReturnId(ctx context.Context, arg␣
↪CreateAuthorAndReturnIdParams) (int64, error) {
        row := q.db.QueryRowContext(ctx, createAuthorAndReturnId, arg.Name, arg.Bio)
        var id int64
        err := row.Scan(&id)
        return id, err
}
```

7.2 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
}

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

EIGHT

UPDATING ROWS

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

8.1 Single parameter

If your query has a single parameter, your Go method will also have a single parameter.

```
-- name: UpdateAuthorBios :exec
UPDATE authors SET bio = $1;
```

```
package db

import (
        "context"
        "database/sql"
)

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

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

type Queries struct {
        db DBTX
}

const updateAuthorBios = `-- name: UpdateAuthorBios :exec
UPDATE authors SET bio = $1
`

func (q *Queries) UpdateAuthorBios(ctx context.Context, bio string) error {
        _, err := q.db.ExecContext(ctx, updateAuthorBios, bio)
```

(continues on next page)

(continued from previous page)

```
        return err
}
```

8.2 Multiple parameters

If your query has more than one parameter, your Go method will accept a `Params` struct.

```
-- 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{}) (sql.Result, 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
`

type UpdateAuthorParams struct {
        ID  int32
        Bio string
}

func (q *Queries) UpdateAuthor(ctx context.Context, arg UpdateAuthorParams) error {
        _, err := q.db.ExecContext(ctx, updateAuthor, arg.ID, arg.Bio)
        return err
}
```

CHAPTER

NINE

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{}) (sql.Result, 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

TEN

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"
        "fmt"
)

type Record struct {
        ID int32
}

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, fmt.Errorf("error preparing query GetRecord: %w", err)
        }
        return &q, nil
}
```

(continues on next page)

(continued from previous page)

```
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...)
        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 int32) (int32, error) {
        row := q.queryRow(ctx, q.getRecordStmt, getRecord, id)
        err := row.Scan(&id)
        return id, err
}
```

CHAPTER

ELEVEN

USING TRANSACTIONS

In the code generated by sqlc, the `WithTx` method allows a `Queries` instance to be associated with a transaction.

For example, with the following SQL structure:

`schema.sql`:

```
CREATE TABLE records (
  id SERIAL PRIMARY KEY,
  counter INT NOT NULL
);
```

`query.sql`

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

-- name: UpdateRecord :exec
UPDATE records SET counter = $2
WHERE id = $1;
```

And the generated code from sqlc in `db.go`:

```
package tutorial

import (
        "context"
        "database/sql"
)

type DBTX interface {
        ExecContext(context.Context, string, ...interface{}) (sql.Result, error)
        PrepareContext(context.Context, string) (*sql.Stmt, error)
        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 {
```

(continues on next page)

(continued from previous page)

```
        db DBTX
}

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

You'd use it like this:

```
func bumpCounter(ctx context.Context, db *sql.DB, queries *tutorial.Queries, id int32)␣
↪error {
        tx, err := db.Begin()
        if err != nil {
                return err
        }
        defer tx.Rollback()
        qtx := queries.WithTx(tx)
        r, err := qtx.GetRecord(ctx, id)
        if err != nil {
                return err
        }
        if err := qtx.UpdateRecord(ctx, tutorial.UpdateRecordParams{
                ID:      r.ID,
                Counter: r.Counter + 1,
        }); err != nil {
                return err
        }
        return tx.Commit()
}
```

CHAPTER

TWELVE

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
```

(continues on next page)

(continued from previous page)

```
    ELSE name
    END
RETURNING *;
```

12.1 Nullable parameters

sqlc infers the nullability of any specified parameters, and often does exactly what you want. If you want finer control over the nullability of your parameters, you may use `sqlc.narg()` (**n**ullable arg) to override the default behavior. Using `sqlc.narg` tells sqlc to ignore whatever nullability it has inferred and generate a nullable parameter instead. There is no nullable equivalent of the `@` syntax.

Here is an example that uses a single query to allow updating an author's name, bio or both.

```
-- name: UpdateAuthor :one
UPDATE author
SET
 name = coalesce(sqlc.narg('name'), name),
 bio = coalesce(sqlc.narg('bio'), bio)
WHERE id = sqlc.arg('id')
RETURNING *;
```

The following code is generated:

```
type UpdateAuthorParams struct {
        Name sql.NullString
        Bio  sql.NullString
        ID   int64
}
```

CHAPTER

THIRTEEN

MODIFYING THE DATABASE SCHEMA

sqlc parses `CREATE TABLE` and `ALTER TABLE` statements in order to generate the necessary code.

```
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
}
```

13.1 Handling SQL migrations

sqlc does not perform database migrations for you. However, sqlc is able to differentiate between up and down migrations. sqlc ignores down migrations when parsing SQL files.

sqlc supports parsing migrations from the following tools:

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

13.1.1 goose

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

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

```
package db

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

13.1.2 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
}
```

13.1.3 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
```

(continues on next page)

(continued from previous page)

```
        Text string
}
```

13.1.4 golang-migrate

Warning: golang-migrate interprets migration filenames numerically. However, sqlc parses migration files in lexicographic order. If you choose to have sqlc enumerate your migration files, make sure their numeric ordering matches their lexicographic ordering to avoid unexpected behavior. This can be done by prepending enough zeroes to the migration filenames.

This doesn't work as intended.

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

This worked as 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
}
```

13.1.5 dbmate

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

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

```
package db

type Foo struct {
        Bar int32
}
```

CHAPTER

FOURTEEN

CONFIGURING GENERATED STRUCTS

14.1 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
}
```

14.2 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"`
}
```

14.3 More control

See the Type Overrides section of the Configuration File docs for fine-grained control over struct field types and tags.

CHAPTER

FIFTEEN

UPLOADING PROJECTS

This feature requires signing up for sqlc Cloud, which is currently in beta.

Uploading your project ensures that future releases of sqlc do not break your existing code. Similar to Rust's crater project, uploaded projects are tested against development releases of sqlc to verify correctness.

15.1 Add configuration

After creating a project, add the project ID to your sqlc configuration file.

```yaml
version: "1"
project:
  id: "<PROJECT-ID>"
packages: []
```

```json
{
  "version": "1",
  "project": {
    "id": "<PROJECT-ID>"
  },
  "packages": [
  ]
}
```

You'll also need to create an API token and make it available via the `SQLC_AUTH_TOKEN` environment variable.

```
export SQLC_AUTH_TOKEN=sqlc_xxxxxxxx
```

15.2 Dry run

You can see what's included when uploading your project by using using the `--dry-run` flag:

```
sqlc upload --dry-run
```

The output will be the exact HTTP request sent by `sqlc`.

15.3 Upload

Once you're ready to upload, remove the `--dry-run` flag.

```
sqlc upload
```

By uploading your project, you're making sqlc more stable and reliable. Thanks!

CHAPTER

SIXTEEN

CLI

```
Usage:
  sqlc [command]

Available Commands:
  compile     Statically check SQL for syntax and type errors
  completion  Generate the autocompletion script for the specified shell
  generate    Generate Go code from SQL
  help        Help about any command
  init        Create an empty sqlc.yaml settings file
  upload      Upload the schema, queries, and configuration for this project
  version     Print the sqlc version number

Flags:
  -x, --experimental   enable experimental features (default: false)
  -f, --file string    specify an alternate config file (default: sqlc.yaml)
  -h, --help           help for sqlc

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

CHAPTER

SEVENTEEN

CONFIGURATION

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.

17.1 Version 2

```yaml
version: "2"
sql:
- schema: "postgresql/schema.sql"
  queries: "postgresql/query.sql"
  engine: "postgresql"
  gen:
    go:
      package: "authors"
      out: "postgresql"
- schema: "mysql/schema.sql"
  queries: "mysql/query.sql"
  engine: "mysql"
  gen:
    go:
      package: "authors"
      out: "mysql"
```

17.1.1 sql

Each mapping in the `sql` collection has the following keys:

- `engine`:
  - One of `postgresql`, `mysql` or `sqlite`.
- `schema`:
  - Directory of SQL migrations or path to single SQL file; or a list of paths.
- `queries`:
  - Directory of SQL queries or path to single SQL file; or a list of paths.
- `codegen`:
  - A colleciton of mappings to configure code generators. See *codegen* for the supported keys.

- gen:
  - A mapping to configure built-in code generators. See *gen* for the supported keys.
- strict_function_checks
  - If true, return an error if a called SQL function does not exist. Defaults to false.
  -

17.1.2 codegen

The codegen mapping supports the following keys:

- out:
  - Output directory for generated code.
- plugin:
  - The name of the plugin. Must be defined in the plugins collection.
- options:
  - A mapping of plugin-specific options.

```
version: '2'
plugins:
- name: py
  wasm:
    url: https://github.com/tabbed/sqlc-gen-python/releases/download/v0.16.0-alpha/sqlc-
↪gen-python.wasm
    sha256: 428476c7408fd4c032da4ec74e8a7344f4fa75e0f98a5a3302f238283b9b95f2
sql:
- schema: "schema.sql"
  queries: "query.sql"
  engine: postgresql
  codegen:
  - out: src/authors
    plugin: py
    options:
      package: authors
      emit_sync_querier: true
      emit_async_querier: true
      query_parameter_limit: 5
```

17.1.3 gen

The gen mapping supports the following keys:

go

- `package`:
  - The package name to use for the generated code. Defaults to `out` basename.
- `out`:
  - Output directory for generated code.
- `sql_package`:
  - Either `pgx/v4`, `pgx/v5` 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`.
- `emit_enum_valid_method`:
  - If true, generate a Valid method on enum types, indicating whether a string is a valid enum value.
- `emit_all_enum_values`:
  - If true, emit a function per enum type that returns all valid enum values.
- `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_batch_file_name`:
  - Customize the name of the batch file. Defaults to `batch.go`.
- `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.
- `query_parameter_limit`:
  - Positional arguments that will be generated in Go functions (`>= 1` or `-1`). To always emit a parameter struct, you would need to set it to `-1`. `0` is invalid. Defaults to `1`. `rename`:
  - Customize the name of generated struct fields. Explained in detail on the `Renaming fields` section.
- `overrides`:
  - It is a collection of definitions that dictates which types are used to map a database types. Explained in detail on the `Type overriding` section.

Renaming 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` mapping to pick a new name. The keys are column names and the values are the struct field name to use.

```
version: "2"
sql:
- schema: "postgresql/schema.sql"
  queries: "postgresql/query.sql"
  engine: "postgresql"
  gen:
    go:
      package: "authors"
      out: "postgresql"
      rename:
        spotify_url: "SpotifyURL"
```

Type overriding

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: "2"
sql:
- schema: "postgresql/schema.sql"
  queries: "postgresql/query.sql"
  engine: "postgresql"
  gen:
    go:
      package: "authors"
      out: "postgresql"
      overrides:
        - db_type: "uuid"
          go_type: "github.com/gofrs/uuid.UUID"
```

Each mapping of the `overrides` collection 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. Note that for Postgres you must use the pg_catalog prefixed names where available. Can't be used if the `column` key is defined.
- `column`
  - In case the type overriding should be done on specific a column of a table instead of a type. `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`. Can't be used if the `db_type` key is defined.
- `go_type`:
  - A fully qualified name to a Go type to use in the generated code.
- `go_struct_tag`:
  - A reflect-style struct tag to use in the generated code, e.g. `a:"b" x:"y,z"`. If you want general json/db tags for all fields, use `emit_db_tags` and/or `emit_json_tags` instead.
- `nullable`:
  - If true, use this type when a column is nullable. Defaults to `false`.

For more complicated import paths, the `go_type` can also be an object.

```
version: "2"
sql:
- schema: "postgresql/schema.sql"
  queries: "postgresql/query.sql"
  engine: "postgresql"
  gen:
    go:
      package: "authors"
```

(continues on next page)

(continued from previous page)

```
    out: "postgresql"
    overrides:
      - db_type: "uuid"
        go_type:
          import: "a/b/v2"
          package: "b"
          type: "MyType"
          pointer: true
```

When generating code, entries using the `column` key will always have preference over entries using the `db_type` key in order to generate the struct.

kotlin

- `package`:
  - The package name to use for the generated code.
- `out`:
  - Output directory for generated code.
- `emit_exact_table_names`:
  - If true, use the exact table name for generated models. Otherwise, guess a singular form. Defaults to `false`.

python

- `package`:
  - The package name to use for the generated code.
- `out`:
  - Output directory for generated code.
- `emit_exact_table_names`:
  - If true, use the exact table name for generated models. Otherwise, guess a singular form. Defaults to `false`.
- `emit_sync_querier`:
  - If true, generate a class with synchronous methods. Defaults to `false`.
- `emit_async_querier`:
  - If true, generate a class with asynchronous methods. Defaults to `false`.
- `emit_pydantic_models`:
  - If true, generate classes that inherit from `pydantic.BaseModel`. Otherwise, define classes using the `dataclass` decorator. Defaults to `false`.

json

- `out`:
  - Output directory for the generated JSON.
- `filename`:
  - Filename for the generated JSON document. Defaults to `codegen_request.json`.
- `indent`:
  - Indent string to use in the JSON document. Defaults to .

17.1.4 plugins

Each mapping in the `plugins` collection has the following keys:

- `name`:
  - The name of this plugin. Required
- `process`: A mapping with a single `cmd` key
  - `cmd`:
    - The executable to call when using this plugin
- `wasm`: A mapping with a two keys `url` and `sha256`
  - `url`:
    - The URL to fetch the WASM file. Supports the `https://` or `file://` schemes.
  - `sha256`
    - The SHA256 checksum for the downloaded file.

```
version: 2
plugins:
- name: "py"
  wasm:
    url: "https://github.com/tabbed/sqlc-gen-python/releases/download/v0.16.0-alpha/sqlc-
↪gen-python.wasm"
    sha256: "428476c7408fd4c032da4ec74e8a7344f4fa75e0f98a5a3302f238283b9b95f2"
- name: "js"
  process:
    cmd: "sqlc-gen-json"
```

17.1.5 global overrides

Sometimes, the same configuration must be done across various specfications of code generation. Then a global definition for type overriding and field renaming can be done using the `overrides` mapping the following manner:

```
version: "2"
overrides:
  go:
    rename:
      id: "Identifier"
```

(continues on next page)

(continued from previous page)

```
    overrides:
      - db_type: "timestamptz"
        nullable: true
        engine: "postgresql"
        go_type:
          import: "gopkg.in/guregu/null.v4"
          package: "null"
          type: "Time"
sql:
- schema: "postgresql/schema.sql"
  queries: "postgresql/query.sql"
  engine: "postgresql"
  gen:
    go:
      package: "authors"
      out: "postgresql"
- schema: "mysql/schema.sql"
  queries: "mysql/query.sql"
  engine: "mysql"
  gen:
    go:
      package: "authors"
      out: "mysql
```

With the previous configuration, whenever a struct field is generated from a table column that is called `id`, it will generated as `Identifier`. Also, whenever there is a nullable `timestamp with time zone` column in a Postgres table, it will be generated as `null.Time`. Note that, the mapping for global type overrides has a field called `engine` that is absent in the regular type overrides. This field is only used when there are multiple definitions using multiple engines. Otherwise, the value of the `engine` key will be defaulted to the engine that is currently being used.

Currently, type overrides and field renaming, both global and regular, are only fully supported in Go.

17.2 Version 1

```
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
    emit_pointers_for_null_types: false
```

(continues on next page)

(continued from previous page)

```
    emit_enum_valid_method: false
    emit_all_enum_values: false
    json_tags_case_style: "camel"
    output_batch_file_name: "batch.go"
    output_db_file_name: "db.go"
    output_models_file_name: "models.go"
    output_querier_file_name: "querier.go"
```

17.2.1 packages

Each mapping in the `packages` collection 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`, `pgx/v5` 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`.
- `emit_pointers_for_null_types`:
  - If true and `sql_package` is set to `pgx/v4`, generated types for nullable columns are emitted as pointers (ie. `*string`) instead of `database/sql` null types (ie. `NullString`). Defaults to `false`.
- `emit_enum_valid_method`:
  - If true, generate a Valid method on enum types, indicating whether a string is a valid enum value.
- `emit_all_enum_values`:
  - If true, emit a function per enum type that returns all valid enum values.
- `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_batch_file_name`:
  - Customize the name of the batch file. Defaults to `batch.go`.
- `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.
- `query_parameter_limit`:
  - Positional arguments that will be generated in Go functions (`>= 0`). To always emit a parameter struct, you would need to set it to `0`. Defaults to `1`.

17.2.2 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. Note that for Postgres you must use the pg_catalog prefixed names where available.
- `go_type`:
  - A fully qualified name to a Go type to use in the generated code.
- `go_struct_tag`:
  - A reflect-style struct tag to use in the generated code, e.g. `a:"b" x:"y,z"`. If you want general json/db tags for all fields, use `emit_db_tags` and/or `emit_json_tags` instead.
- `nullable`:
  - If true, use this type when a column is nullable. Defaults to `false`.

For more complicated import paths, the `go_type` can also be an object.

```
version: "1"
packages: [...]
overrides:
  - db_type: "uuid"
    go_type:
      import: "a/b/v2"
      package: "b"
      type: "MyType"
```

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:
```

(continues on next page)

(continued from previous page)

```
  - column: "authors.id"
    go_type: "github.com/segmentio/ksuid.KSUID"
```

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: [...]
```

17.2.3 rename

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` mapping 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

EIGHTEEN

DATATYPES

18.1 Arrays

PostgreSQL arrays are materialized as Go slices. Currently, the `pgx/v5` sql package only supports multidimensional arrays.

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

```
package db

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

18.2 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. The `pgx/v5` sql package uses the appropriate pgx types.

```
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
```

(continues on next page)

(continued from previous page)

```
        CreatedAt time.Time
        UpdatedAt sql.NullTime
}
```

18.3 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
}
```

18.4 Null

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

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

```
package db

import (
        "database/sql"
)
```

(continues on next page)

(continued from previous page)

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

18.5 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
}
```

For MySQL, there is no native `uuid` data type. When using `UUID_TO_BIN` to store a `UUID()`, the underlying field type is `BINARY(16)` which by default sqlc would interpret this to `sql.NullString`. To have sqlc automatically convert these fields to a `uuid.UUID` type, use an overide on the column storing the `uuid`.

```
{
  "overrides": [
    {
      "column": "*.uuid",
      "go_type": "github.com/google/uuid.UUID"
    }
  ]
}
```

18.6 JSON

By default, sqlc will generate the `[]byte`, `pgtype.JSON` or `json.RawMessage` for JSON column type. But if you use the `pgx/v5` sql package then you can specify a some struct instead of default type. The `pgx` implementation will marshall/unmarshall the struct automatically.

```
package dto

type BookData struct {
```

(continues on next page)

(continued from previous page)

```
        Genres    []string `json:"genres"`
        Title     string   `json:"title"`
        Published bool     `json:"published"`
}
```

```
CREATE TABLE books (
  data jsonb
);
```

```
{
  "overrides": [
    {
      "column": "books.data",
      "go_type": {
        "import":"example/db",
        "package": "dto",
        "type":"BookData"
      }
    }
  ]
}
```

```
package db

import (
        "example.com/db/dto"
)

type Book struct {
    Data *dto.BookData
}
```

CHAPTER

NINETEEN

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>
```

19.1 :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
}
```

19.2 :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)
}
```

19.3 :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)
        // ...
}
```

19.4 :execlastid

The generated method will return the number generated by the database from the result returned by ExecContext.

```
-- name: InsertAuthor :execlastid
INSERT INTO authors (name) VALUES (?);
```

```
func (q *Queries) InsertAuthor(ctx context.Context, name string) (int64, error) {
        _, err := q.db.ExecContext(ctx, insertAuthor, name)
        // ...
}
```

19.5 :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)
        // ...
}
```

19.6 :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)
        // ...
}
```

19.7 :batchexec

NOTE: This command only works with PostgreSQL using the pgx/v4 and pgx/v5 drivers and outputting Go code.

The generated method will return a batch object. The batch object will have the following methods:

- `Exec`, that takes a `func(int, error)` parameter,
- `Close`, to close the batch operation early.

```
-- name: DeleteBook :batchexec
DELETE FROM books
WHERE book_id = $1;
```

```
type DeleteBookBatchResults struct {
        br  pgx.BatchResults
        ind int
}

func (q *Queries) DeleteBook(ctx context.Context, bookID []int32)␣
↪*DeleteBookBatchResults {
        //...
}
func (b *DeleteBookBatchResults) Exec(f func(int, error)) {
        //...
}
func (b *DeleteBookBatchResults) Close() error {
        //...
}
```

19.8 :batchmany

NOTE: This command only works with PostgreSQL using the pgx/v4 and pgx/v5 drivers and outputting Go code.

The generated method will return a batch object. The batch object will have the following methods:

- `Query`, that takes a `func(int, []T, error)` parameter, where `T` is your query's return type
- `Close`, to close the batch operation early.

```
-- name: BooksByTitleYear :batchmany
SELECT * FROM books
WHERE title = $1 AND year = $2;
```

```
type BooksByTitleYearBatchResults struct {
        br  pgx.BatchResults
        ind int
}
type BooksByTitleYearParams struct {
        Title string `json:"title"`
        Year  int32  `json:"year"`
}

func (q *Queries) BooksByTitleYear(ctx context.Context, arg []BooksByTitleYearParams)␣
↪*BooksByTitleYearBatchResults {
        //...
}
func (b *BooksByTitleYearBatchResults) Query(f func(int, []Book, error)) {
        //...
}
func (b *BooksByTitleYearBatchResults) Close() error {
        //...
}
```

19.9 :batchone

NOTE: This command only works with PostgreSQL using the pgx/v4 and pgx/v5 drivers and outputting Go code.

The generated method will return a batch object. The batch object will have the following methods:

- `QueryRow`, that takes a `func(int, T, error)` parameter, where `T` is your query's return type
- `Close`, to close the batch operation early.

```
-- name: CreateBook :batchone
INSERT INTO books (
    author_id,
    isbn
) VALUES (
    $1,
    $2
)
RETURNING book_id, author_id, isbn
```

```
type CreateBookBatchResults struct {
        br  pgx.BatchResults
        ind int
}
type CreateBookParams struct {
        AuthorID int32  `json:"author_id"`
        Isbn     string `json:"isbn"`
}

func (q *Queries) CreateBook(ctx context.Context, arg []CreateBookParams)␣
↪*CreateBookBatchResults {
```

(continues on next page)

(continued from previous page)

```
        //...
}
func (b *CreateBookBatchResults) QueryRow(f func(int, Book, error)) {
        //...
}
func (b *CreateBookBatchResults) Close() error {
        //...
}
```

CHAPTER

TWENTY

DATABASE AND LANGUAGE SUPPORT

Language	Plugin	MySQL	PostgreSQL	SQLite
Go	(built-in)	Stable	Stable	Beta
Kotlin	sqlc-gen-kotlin	Beta	Beta	Not implemented
Python	sqlc-gen-python	Beta	Beta	Not implemented

20.1 Future Language Support

- C#
- TypeScript

CHAPTER

TWENTYONE

ENVIRONMENT VARIABLES

21.1 SQLCCACHE

The `SQLCCACHE` environment variable dictates where `sqlc` will store cached WASM-based plugins and modules. By default `sqlc` follows the XDG Base Directory Specification.

21.2 SQLCDEBUG

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

21.2.1 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"
      }),
```

21.2.2 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",
  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"
      }),
```

21.2.3 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
```

(continues on next page)

(continued from previous page)

```
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
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
```

21.2.4 processplugins

Setting this value to `0` disables process-based plugins. If a process-based plugin is declared in the configuration file, running any `sqlc` command will return an error.

```
SQLCDEBUG=processplugins=0
```

21.3 SQLCTMPDIR

If specified, use the given directory as the base for temporary folders. Only applies when using WASM-based codegen plugins. When not specified, this defaults to passing an empty string to `os.MkdirTemp`.

CHAPTER

TWENTYTWO

CHANGELOG

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

22.1 1.18.0

Released 2023-04-27

22.1.1 Release notes

Remote code generation

Developed by @andrewmbenton

At its core, sqlc is powered by SQL engines, which include parsers, formatters, analyzers and more. While our goal is to support each engine on each operating system, it's not always possible. For example, the PostgreSQL engine does not work on Windows.

To bridge that gap, we're announcing remote code generation, currently in private alpha. To join the private alpha, sign up for the waitlist.

To configure remote generation, configure a `cloud` block in `sqlc.json`.

```
{
  "version": "2",
  "cloud": {
    "organization": "<org-id>",
    "project": "<project-id>",
  },
  ...
}
```

You'll also need to the `SQLC_AUTH_TOKEN` environment variable.

```
export SQLC_AUTH_TOKEN=<token>
```

When the cloud configuration exists, `sqlc generate` will default to remote generation. If you'd like to generate code locally, pass the `--no-remote` option.

```
sqlc generate --no-remote
```

Remote generation is off by default and requires an opt-in to use.

sqlc.embed

Developed by @nickjackson

Embedding allows you to reuse existing model structs in more queries, resulting in less manual serilization work. First, imagine we have the following schema with students and test scores.

```
CREATE TABLE students (
  id   bigserial PRIMARY KEY,
  name text,
  age  integer
)

CREATE TABLE test_scores (
  student_id bigint,
  score integer,
  grade text
)
```

We want to select the student record and the highest score they got on a test. Here's how we'd usually do that:

```
-- name: HighScore :many
WITH high_scores AS (
  SELECT student_id, max(score) as high_score
  FROM test_scores
  GROUP BY 1
)
SELECT students.*, high_score::integer
FROM students
JOIN high_scores ON high_scores.student_id = students.id;
```

When using Go, sqlc will produce a struct like this:

```
type HighScoreRow struct {
        ID        int64
        Name      sql.NullString
        Age       sql.NullInt32
        HighScore int32
}
```

With embedding, the struct will contain a model for the table instead of a flattened list of columns.

```
-- name: HighScoreEmbed :many
WITH high_scores AS (
  SELECT student_id, max(score) as high_score
  FROM test_scores
  GROUP BY 1
)
SELECT sqlc.embed(students), high_score::integer
FROM students
JOIN high_scores ON high_scores.student_id = students.id;
```

```
type HighScoreRow struct {
        Student   Student
```

(continues on next page)

(continued from previous page)

```
        HighScore int32
}
```

sqlc.slice

Developed by Paul Cameron and Jille Timmermans

The MySQL Go driver does not support passing slices to the IN operator. The `sqlc.slice` function generates a dynamic query at runtime with the correct number of parameters.

```
/* name: SelectStudents :many */
SELECT * FROM students
WHERE age IN (sqlc.slice("ages"))
```

```
func (q *Queries) SelectStudents(ctx context.Context, arges []int32) ([]Student, error) {
```

This feature is only supported in MySQL and cannot be used with prepared queries.

Batch operation improvements

When using batches with pgx, the error returned when a batch is closed is exported by the generated package. This change allows for cleaner error handling using `errors.Is`.

```
errors.Is(err, generated_package.ErrBatchAlreadyClosed)
```

Previously, you would have had to check match on the error message itself.

```
err.Error() == "batch already closed"
```

The generated code for batch operations always lived in `batch.go`. This file name can now be configured via the `output_batch_file_name` configuration option.

Configurable query parameter limits for Go

By default, sqlc will limit Go functions to a single parameter. If a query includes more than one parameter, the generated method will use an argument struct instead of positional arguments. This behavior can now be changed via the `query_parameter_limit` configuration option. If set to `0`, every genreated method will use a argument struct.

22.1.2 Changes

Bug Fixes

- Prevent variable redeclaration in single param conflict for pgx (#2058)
- Retrieve Larg/Rarg join query after inner join (#2051)
- Rename argument when conflicted to imported package (#2048)
- Pgx closed batch return pointer if need #1959 (#1960)
- Correct singularization of “waves” (#2194)

- Honor Package level renames in v2 yaml config (#2001)
- (mysql) Prevent UPDATE … JOIN panic #1590 (#2154)
- Mysql delete join panic (#2197)
- Missing import with pointer overrides, solves #2168 #2125 (#2217)

Documentation

- (config.md) Add `sqlite` as engine option (#2164)
- Add first pass at pgx documentation (#2174)
- Add missed configuration option (#2188)
- `specifies parameter ":one" without containing a RETURNING clause` (#2173)

Features

- Add `sqlc.embed` to allow model re-use (#1615)
- (Go) Add query_parameter_limit conf to codegen (#1558)
- Add remote execution for codegen (#2214)

Testing

- Skip tests if required plugins are missing (#2104)
- Add tests for reanme fix in v2 (#2196)
- Regenerate batch output for filename tests
- Remove remote test (#2232)
- Regenerate test output

Bin/sqlc

- Add SQLCTMPDIR environment variable (#2189)

Build

- (deps) Bump github.com/antlr/antlr4/runtime/Go/antlr (#2109)
- (deps) Bump github.com/jackc/pgx/v4 from 4.18.0 to 4.18.1 (#2119)
- (deps) Bump golang from 1.20.1 to 1.20.2 (#2135)
- (deps) Bump google.golang.org/protobuf from 1.28.1 to 1.29.0 (#2137)
- (deps) Bump google.golang.org/protobuf from 1.29.0 to 1.29.1 (#2143)
- (deps) Bump golang from 1.20.2 to 1.20.3 (#2192)
- (deps) Bump actions/setup-go from 3 to 4 (#2150)
- (deps) Bump google.golang.org/protobuf from 1.29.1 to 1.30.0 (#2151)

- (deps) Bump github.com/spf13/cobra from 1.6.1 to 1.7.0 (#2193)
- (deps) Bump github.com/lib/pq from 1.10.7 to 1.10.8 (#2211)
- (deps) Bump github.com/lib/pq from 1.10.8 to 1.10.9 (#2229)
- (deps) Bump github.com/go-sql-driver/mysql from 1.7.0 to 1.7.1 (#2228)

Cmd/sqlc

- Remove –experimental flag (#2170)
- Add option to disable process-based plugins (#2180)
- Bump version to v1.18.0

Codegen

- Correctly generate CopyFrom columns for single-column copyfroms (#2185)

Config

- Add top-level cloud configuration (#2204)

Engine/postgres

- Upgrade to pg_query_go/v4 (#2114)

Ext/wasm

- Check exit code on returned error (#2223)

Parser

- Generate correct types for `SELECT NOT EXISTS` (#1972)

Sqlite

- Add support for CREATE TABLE … STRICT (#2175)

Wasm

- Upgrade to wasmtime v8.0.0 (#2222)

22.2 1.17.2

Released 2023-02-22

22.2.1 Bug Fixes

- Fix build on Windows (#2102)

22.3 1.17.1

Released 2023-02-22

22.3.1 Bug Fixes

- Prefer to use []T over pgype.Array[T] (#2090)
- Revert changes to Dockerfile (#2091)
- Do not throw error when IF NOT EXISTS is used on ADD COLUMN (#2092)

22.3.2 MySQL

- Add `float` support to MySQL (#2097)

22.3.3 Build

- (deps) Bump golang from 1.20.0 to 1.20.1 (#2082)

22.4 1.17.0

Released 2023-02-13

22.4.1 Bug Fixes

- Initialize generated code outside function (#1850)
- (engine/mysql) Take into account column's charset to distinguish text/blob, (var)char/(var)binary (#776) (#1895)
- The enum Value method returns correct type (#1996)
- Documentation for Inserting Rows (#2034)
- Add import statements even if only pointer types exist (#2046)
- Search from Rexpr if not found from Lexpr (#2056)

22.4.2 Documentation

- Change ENTRYPOINT to CMD (#1943)
- Update samples for HOW-TO GUIDES (#1953)

22.4.3 Features

- Add the diff command (#1963)

22.4.4 Build

- (deps) Bump github.com/mattn/go-sqlite3 from 1.14.15 to 1.14.16 (#1913)
- (deps) Bump github.com/spf13/cobra from 1.6.0 to 1.6.1 (#1909)
- Fix devcontainer (#1942)
- Run sqlc-pg-gen via GitHub Actions (#1944)
- Move large arrays out of functions (#1947)
- Fix conflicts from pointer configuration (#1950)
- (deps) Bump github.com/go-sql-driver/mysql from 1.6.0 to 1.7.0 (#1988)
- (deps) Bump github.com/jackc/pgtype from 1.12.0 to 1.13.0 (#1978)
- (deps) Bump golang from 1.19.3 to 1.19.4 (#1992)
- (deps) Bump certifi from 2020.12.5 to 2022.12.7 in /docs (#1993)
- (deps) Bump golang from 1.19.4 to 1.19.5 (#2016)
- (deps) Bump golang from 1.19.5 to 1.20.0 (#2045)
- (deps) Bump github.com/jackc/pgtype from 1.13.0 to 1.14.0 (#2062)
- (deps) Bump github.com/jackc/pgx/v4 from 4.17.2 to 4.18.0 (#2063)

22.4.5 Cmd

- Generate packages in parallel (#2026)

22.4.6 Cmd/sqlc

- Bump version to v1.17.0

22.4.7 Codegen

- Remove built-in Kotlin support (#1935)
- Remove built-in Python support (#1936)

22.4.8 Internal/codegen

- Cache pattern matching compilations (#2028)

22.4.9 Mysql

- Add datatype tests (#1948)
- Fix blob tests (#1949)

22.4.10 Plugins

- Upgrade to wasmtime 3.0.1 (#2009)

22.4.11 Sqlite

- Supported between expr (#1958) (#1967)

22.4.12 Tools

- Regenerate scripts skips dirs that contains diff exec command (#1987)

22.4.13 Wasm

- Upgrade to wasmtime 5.0.0 (#2065)

22.5 1.16.0

Released 2022-11-09

22.5.1 Bug Fixes

- (validate) Sqlc.arg & sqlc.narg are not “missing” (#1814)
- Emit correct comment for nullable enums (#1819)
- Correctly switch `coalesce()` result `.NotNull` value (#1664)
- Prevent batch infinite loop with arg length (#1794)
- Support version 2 in error message (#1839)
- Handle empty column list in postgresql (#1843)

- Batch imports filter queries, update cmds having ret type (#1842)
- Named params contribute to batch parameter count (#1841)

22.5.2 Documentation

- Add a getting started guide for SQLite (#1798)
- Various readability improvements (#1854)
- Add documentation for codegen plugins (#1904)
- Update migration guides with links (#1933)

22.5.3 Features

- Add HAVING support to MySQL (#1806)

22.5.4 Miscellaneous Tasks

- Upgrade wasmtime version (#1827)
- Bump wasmtime version to v1.0.0 (#1869)

22.5.5 Build

- (deps) Bump github.com/jackc/pgconn from 1.12.1 to 1.13.0 (#1785)
- (deps) Bump github.com/mattn/go-sqlite3 from 1.14.13 to 1.14.15 (#1799)
- (deps) Bump github.com/jackc/pgx/v4 from 4.16.1 to 4.17.0 (#1786)
- (deps) Bump github.com/jackc/pgx/v4 from 4.17.0 to 4.17.1 (#1825)
- (deps) Bump github.com/bytecodealliance/wasmtime-go (#1826)
- (deps) Bump github.com/jackc/pgx/v4 from 4.17.1 to 4.17.2 (#1831)
- (deps) Bump golang from 1.19.0 to 1.19.1 (#1834)
- (deps) Bump github.com/google/go-cmp from 0.5.8 to 0.5.9 (#1838)
- (deps) Bump github.com/lib/pq from 1.10.6 to 1.10.7 (#1835)
- (deps) Bump github.com/bytecodealliance/wasmtime-go (#1857)
- (deps) Bump github.com/spf13/cobra from 1.5.0 to 1.6.0 (#1893)
- (deps) Bump golang from 1.19.1 to 1.19.3 (#1920)

22.5.6 Cmd/sqlc

- Bump to v1.16.0

22.5.7 Codgen

- Include serialized codegen options (#1890)

22.5.8 Compiler

- Move Kotlin parameter logic into codegen (#1910)

22.5.9 Examples

- Port Python examples to WASM plugin (#1903)

22.5.10 Pg-gen

- Make sqlc-pg-gen the complete source of truth for pg_catalog.go (#1809)
- Implement information_schema shema (#1815)

22.5.11 Python

- Port all Python tests to sqlc-gen-python (#1907)
- Upgrade to sqlc-gen-python v1.0.0 (#1932)

22.6 1.15.0

Released 2022-08-07

22.6.1 Bug Fixes

- (mysql) Typo (#1700)
- (postgresql) Add quotes for CamelCase columns (#1729)
- Cannot parse SQLite upsert statement (#1732)
- (sqlite) Regenerate test output for builtins (#1735)
- (wasm) Version modules by wasmtime version (#1734)
- Missing imports (#1637)
- Missing slice import for querier (#1773)

22.6.2 Documentation

- Add process-based plugin docs (#1669)
- Add links to downloads.sqlc.dev (#1681)
- Update transactions how to example (#1775)

22.6.3 Features

- More SQL Syntax Support for SQLite (#1687)
- (sqlite) Promote SQLite support to beta (#1699)
- Codegen plugins, powered by WASM (#1684)
- Set user-agent for plugin downloads (#1707)
- Null enums types (#1485)
- (sqlite) Support stdlib functions (#1712)
- (sqlite) Add support for returning (#1741)

22.6.4 Miscellaneous Tasks

- Add tests for quoting columns (#1733)
- Remove catalog tests (#1762)

22.6.5 Testing

- Add tests for fixing slice imports (#1736)
- Add test cases for returning (#1737)

22.6.6 Build

- Upgrade to Go 1.19 (#1780)
- Upgrade to go-wasmtime 0.39.0 (#1781)

22.6.7 Plugins

- (wasm) Change default cache location (#1709)
- (wasm) Change the SHA-256 config key (#1710)

22.7 1.14.0

Released 2022-06-09

22.7.1 Bug Fixes

- (postgresql) Remove extra newline with db argument (#1417)
- (sqlite) Fix DROP TABLE (#1443)
- (compiler) Fix left join nullability with table aliases (#1491)
- Regenerate testdata for CREATE TABLE AS (#1516)
- (bundler) Only close multipart writer once (#1528)
- (endtoend) Regenerate testdata for exex_lastid
- (pgx) Copyfrom imports (#1626)
- Validate sqlc function arguments (#1633)
- Fixed typo `sql.narg` in doc (#1668)

22.7.2 Features

- (golang) Add Enum.Valid and AllEnumValues (#1613)
- (sqlite) Start expanding support (#1410)
- (pgx) Add support for batch operations (#1437)
- (sqlite) Add support for delete statements (#1447)
- (codegen) Insert comments in interfaces (#1458)
- (sdk) Add the plugin SDK package (#1463)
- Upload projects (#1436)
- Add sqlc version to generated Kotlin code (#1512)
- Add sqlc version to generated Go code (#1513)
- Pass sqlc version in codegen request (#1514)
- (postgresql) Add materialized view support (#1509)
- (python) Graduate Python support to beta (#1520)
- Run sqlc with docker on windows cmd (#1557)
- Add JSON “codegen” output (#1565)
- Add sqlc.narg() for nullable named params (#1536)
- Process-based codegen plugins (#1578)

22.7.3 Miscellaneous Tasks

- Fix extra newline in comments for copyfrom (#1438)
- Generate marshal/unmarshal with vtprotobuf (#1467)

22.7.4 Refactor

- (codegen) Port Kotlin codegen package to use plugin types (#1416)
- (codegen) Port Go to plugin types (#1460)
- (cmd) Simplify codegen selection logic (#1466)
- (sql/catalog) Improve Readability (#1595)
- Add basic fuzzing for config / overrides (#1500)

22.8 1.13.0

Released 2022-03-31

22.8.1 Bug Fixes

- (compiler) Fix left join nullability with table aliases (#1491)
- (postgresql) Remove extra newline with db argument (#1417)
- (sqlite) Fix DROP TABLE (#1443)

22.8.2 Features

- (cli) Upload projects (#1436)
- (codegen) Add sqlc version to generated Go code (#1513)
- (codegen) Add sqlc version to generated Kotlin code (#1512)
- (codegen) Insert comments in interfaces (#1458)
- (codegen) Pass sqlc version in codegen request (#1514)
- (pgx) Add support for batch operations (#1437)
- (postgresql) Add materialized view support (#1509)
- (python) Graduate Python support to beta (#1520)
- (sdk) Add the plugin SDK package (#1463)
- (sqlite) Add support for delete statements (#1447)
- (sqlite) Start expanding support (#1410)

22.8.3 Miscellaneous Tasks

- Fix extra newline in comments for copyfrom (#1438)
- Generate marshal/unmarshal with vtprotobuf (#1467)

22.8.4 Refactor

- (codegen) Port Kotlin codegen package to use plugin types (#1416)
- (codegen) Port Go to plugin types (#1460)
- (cmd) Simplify codegen selection logic (#1466)

22.8.5 Config

- Add basic fuzzing for config / overrides (#1500)

22.9 1.12.0

Released 2022-02-05

22.9.1 Bug

- ALTER TABLE SET SCHEMA (#1409)

22.9.2 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)

22.9.3 Documentation

- Bump version to 1.11.0 (#1308)

22.9.4 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)

22.9.5 Miscellaneous Tasks

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

22.9.6 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)

22.10 1.11.0

Released 2021-11-24

22.10.1 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)

22.10.2 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)

22.10.3 Features

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

22.10.4 Refactor

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

22.10.5 Styling

- Apply gofmt to sample code (#1261)

22.10.6 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)

22.10.7 Cmd/sqlc

- Bump version to v1.11.0

22.11 1.10.0

Released 2021-09-07

22.11.1 Documentation

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

22.11.2 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)

22.11.3 Ci

- Upgrade Go to 1.17 (#1130)

22.11.4 Cmd/sqlc

- Bump version to v1.10.0 (#1165)

22.11.5 Codegen/golang

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

22.11.6 Codgen/golang

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

22.11.7 Endtoend

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

22.11.8 Engine/mysql

- Add support for LIKE (#1162)

22.11.9 Golang

- Output NullUUID when necessary (#1137)

22.12 1.9.0

Released 2021-08-13

22.12.1 Documentation

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

22.12.2 Build

- Bump golang from 1.16.6 to 1.16.7 (#1107)

22.12.3 Cmd/sqlc

- Bump version to v1.9.0 (#1121)

22.12.4 Compiler

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

22.13 1.8.0

Released 2021-05-03

22.13.1 Documentation

- Add language support Matrix (#920)

22.13.2 Features

- Add case style config option (#905)

22.13.3 Python

- Eliminate runtime package and use sqlalchemy (#939)

22.13.4 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)

22.13.5 Cmd

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

22.13.6 Cmd/sqlc

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

22.13.7 Codegen

- Generate valid enum names for symbols (#972)

22.13.8 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)

22.13.9 Sql/ast

- Resolve return values from functions (#964)

22.13.10 Workflows

- Only run tests once (#924)

22.14 1.7.0

Released 2021-02-28

22.14.1 Bug Fixes

- Struct tag formatting (#833)

22.14.2 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)

22.14.3 Features

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

22.14.4 README

- Add Go Report Card badge (#891)

22.14.5 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)

22.14.6 Catalog

- Improve alter column type (#818)

22.14.7 Ci

- Uprade to Go 1.15 (#887)

22.14.8 Cmd

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

22.14.9 Cmd/sqlc

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

22.14.10 Codegen/golang

- Make sure to import net package (#858)

22.14.11 Compiler

- Support UNION query

22.14.12 Dolphin

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

22.14.13 Endtoend

- Add tests for INTERSECT and EXCEPT

22.14.14 Go.mod

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

22.14.15 Mysql

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

22.14.16 Sql/ast

- Add enum values for SetOperation

22.15 1.6.0

Released 2020-11-23

22.15.1 Dolphin

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

22.15.2 README

- Update language / database support (#698)

22.15.3 Astutils

- Fix Params rewrite call (#674)

22.15.4 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)

22.15.5 Catalog

- Improve variadic argument support (#804)

22.15.6 Cmd/sqlc

- Bump to version v1.6.0 (#806)

22.15.7 Codegen

- Fix errant database/sql imports (#789)

22.15.8 Compiler

- Use engine-specific reserved keywords (#677)

22.15.9 Dolphi

- Add list of builtin functions (#795)

22.15.10 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)

22.15.11 Endtoend

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

22.15.12 Examples

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

22.15.13 Internal/cmd

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

22.15.14 Kotlin

- Remove runtime dependency (#774)

22.15.15 Metadata

- Support multiple comment prefixes (#683)

22.15.16 Postgresql

- Support string concat operator (#701)

22.15.17 Sql/catalog

- Add support for variadic functions (#798)

22.16 1.5.0

Released 2020-08-05

22.16.1 Documentation

- Build sqlc using Go 1.14 (#549)

22.16.2 Cmd

- Add debugging support (#573)

22.16.3 Cmd/sqlc

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

22.16.4 Compiler

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

22.16.5 Config

- Add emit_empty_slices to version one (#552)

22.16.6 Contrib

- Add generated code for contrib

22.16.7 Dinosql

- Remove deprecated package (#554)

22.16.8 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)

22.16.9 Endtoend

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

22.16.10 Engine

- Create engine package (#556)

22.16.11 Equinox

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

22.16.12 Examples

- Run tests for MySQL booktest (#627)

22.16.13 Golang

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

22.16.14 Internal

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

22.16.15 Mod

- Downgrade TiDB package to fix build (#603)

22.16.16 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)

22.16.17 Postgresql

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

22.16.18 Sql/catalog

- Fix comparison of pg_catalog types (#637)

22.16.19 Tools

- Generate functions for all of contrib

22.16.20 Workflow

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

22.17 1.4.0

Released 2020-06-17

22.17.1 Dockerfile

- Add version build argument (#487)

22.17.2 MySQL

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

22.17.3 README

- Document emit_exact_table_names (#486)

22.17.4 All

- Remove the exp build tag (#507)

22.17.5 Catalog

- Support functions with table parameters (#541)

22.17.6 Cmd

- Bump to version 1.3.1-devel (#485)

22.17.7 Cmd/sqlc

- Bump version to v1.4.0 (#547)

22.17.8 Codegen

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

22.17.9 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)

22.17.10 Endtoend

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

22.17.11 Internal/dinosql

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

22.17.12 Internal/postgresql

- Fix NamedArgExpr rewrite (#491)

22.17.13 Multierr

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

22.17.14 Named

- Port parameter style validation to SQL (#504)

22.17.15 Parser

- Support columns from subselect statements (#489)

22.17.16 Rewrite

- Move parameter rewrite to package (#499)

22.17.17 Sqlite

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

22.17.18 Sqlpath

- Move ReadSQLFiles into a separate package (#495)

22.17.19 Validation

- Move query validation to separate package (#498)

22.18 1.3.0

Released 2020-05-12

22.18.1 Makefile

- Update target (#449)

22.18.2 README

- Add Myles as a sponsor (#469)

22.18.3 Testing

- Make sure all Go examples build (#480)

22.18.4 Cmd

- Bump version to v1.3.0 (#484)

22.18.5 Cmd/sqlc

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

22.18.6 Dinosql

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

22.18.7 Gen

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

22.18.8 Go.mod

- Use latest lib/pq (#471)

22.18.9 Parser

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

22.18.10 Postgresql

- Fix panic walking CreateTableAsStmt (#475)

22.19 1.2.0

Released 2020-04-07

22.19.1 Documentation

- Publish to Docker Hub (#422)

22.19.2 README

- Docker installation docs (#424)

22.19.3 Cmd/sqlc

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

22.19.4 Gen

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

22.19.5 Migrations

- Move migration parsing to new package (#427)

22.19.6 Parser

- Generate correct types for SELECT EXISTS (#411)

22.20 1.1.0

Released 2020-03-17

22.20.1 README

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

22.20.2 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)

22.20.3 Astutils

- Apply changes to the ValuesList slice (#372)

22.20.4 Cmd

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

22.20.5 Cmd/sqlc

- Bump version to v1.1.0 (#406)

22.20.6 Compiler

- Wire up the experimental parsers

22.20.7 Config

- Remove “emit_single_file” option (#367)

22.20.8 Dolphin

- Add experimental parser for MySQL

22.20.9 Gen

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

22.20.10 Go.mod

- Add packages for MySQL and SQLite parsers

22.20.11 Internal/dinosql

- Support Postgres macaddr type in Go (#358)

22.20.12 Internal/endtoend

- Remove %w (#354)

22.20.13 Kotlin

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

22.20.14 Postgresql

- Add experimental parser for MySQL

22.20.15 Sql

- Add generic SQL AST

22.20.16 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)

22.20.17 Sql/catalog

- Port tests over from catalog pkg (#402)

22.20.18 Sql/errors

- Add a new errors package (#390)

22.20.19 Sqlite

- Add experimental parser for SQLite

22.21 1.0.0

Released 2020-02-18

22.21.1 Documentation

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

22.21.2 README

- Add sponsors section (#333)

22.21.3 Cmd

- Remove parse subcommand (#322)

22.21.4 Config

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

22.21.5 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)

22.21.6 Internal

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

22.21.7 Internal/config

- Create new config package (#313)

22.21.8 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)

22.21.9 Internal/endtoend

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

22.21.10 Internal/mysql

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

22.21.11 Internal/parser

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

22.21.12 Internal/sqlc

- Add support for composite types (#311)

22.21.13 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

22.21.14 Parser

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

22.22 0.1.0

Released 2020-01-07

22.22.1 Documentation

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

22.22.2 README

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

22.22.3 Catalog

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

22.22.4 Ci

- Enable tests on pull requests

22.22.5 Cmd

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

22.22.6 Dinosql/internal

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

22.22.7 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)

22.22.8 Internal/catalog

- Parse unnamed function arguments (#166)

22.22.9 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)

22.22.10 Internal/dinosql/testdata/ondeck

- Add Makefile (#156)

22.22.11 Ondeck

- Move all tests to GitHub CI (#58)

22.22.12 ParseQuery

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

22.22.13 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)

22.22.14 Pg

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

CHAPTER

TWENTYTHREE

USING GO AND PGX

Note: Experimental support for `pgx/v5` was added in v1.17.2. Full support will be included in v1.18.0. Until then, you'll need to pass the `--experimental` flag to `sqlc generate`.

pgx is a pure Go driver and toolkit for PostgreSQL. It's become the default PostgreSQL package for many Gophers since lib/pq was put into maitience mode.

23.1 Getting started

To start generating code that uses pgx, set the `sql_package` field in your `sqlc.yaml` configuration file. Valid options are `pgx/v4` or `pgx/v5`

```
version: "2"
sql:
  - engine: "postgresql"
    queries: "query.sql"
    schema: "query.sql"
    gen:
      go:
        package: "db"
        sql_package: "pgx/v5"
        out: "db"
```

If you don't have an existing sqlc project on hand, create a directory with the configuration file above and the following `query.sql` file.

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

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

-- name: ListAuthors :many
SELECT * FROM authors
```

(continues on next page)

(continued from previous page)

```
ORDER BY name;

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

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

Generating the code will now give you pgx-compatible database access methods.

```
sqlc generate --experimental
```

23.2 Generated code walkthrough

The generated code is very similar to the code generated when using `lib/pq`. However, instead of using `database/sql`, the code uses pgx types directly.

```
package main

import (
    "context"
    "fmt"
    "os"

    "github.com/jackc/pgx/v5"

    "example.com/sqlc-tutorial/db"
)

func main() {
    // urlExample := "postgres://username:password@localhost:5432/database_name"
    conn, err := pgx.Connect(context.Background(), os.Getenv("DATABASE_URL"))
    if err != nil {
            fmt.Fprintf(os.Stderr, "Unable to connect to database: %v\n", err)
            os.Exit(1)
    }
    defer conn.Close(context.Background())

    q := db.New(conn)

    author, err := q.GetAuthor(context.Background(), 1)
    if err != nil {
            fmt.Fprintf(os.Stderr, "GetAuthor failed: %v\n", err)
            os.Exit(1)
```

(continues on next page)

(continued from previous page)

```
    }

    fmt.Println(author.Name)
}
```

CHAPTER

TWENTYFOUR

DEVELOPING SQLC

24.1 Building

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

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

24.2 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

24.2.1 For PostgreSQL

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

24.2.2 For MySQL

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

24.3 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

TWENTYFIVE

AUTHORING PLUGINS

To use plugins, you must be using Version 2 of the configuration file. The top-level `plugins` array defines the available plugins.

25.1 WASM plugins

> WASM plugins are fully sandboxed. Plugins do not have access to the network, filesystem, or environment variables.

In the `codegen` section, the `out` field dictates what directory will contain the new files. The `plugin` key must reference a plugin defined in the top-level `plugins` map. The `options` are serialized to a string and passed on to the plugin itself.

```
{
  "version": "2",
  "plugins": [
    {
      "name": "greeter",
      "wasm": {
        "url": "https://github.com/kyleconroy/sqlc-gen-greeter/releases/download/v0.1.0/
↪sqlc-gen-greeter.wasm",
        "sha256": "afc486dac2068d741d7a4110146559d12a013fd0286f42a2fc7dcd802424ad07"
      }
    }
  ],
  "sql": [
    {
      "schema": "schema.sql",
      "queries": "query.sql",
      "engine": "postgresql",
      "codegen": [
        {
          "out": "gen",
          "plugin": "greeter"
        }
      ]
    }
  ]
}
```

For a complete working example see the following files:

- sqlc-gen-greeter
  - A WASM plugin (written in Rust) that outputs a friendly message
- wasm_plugin_sqlc_gen_greeter
  - An example project showing how to use a WASM plugin

25.2 Process plugins

> Process-based plugins offer minimal security. Only use plugins that you trust. Better yet, only use plugins that you've written yourself.

In the `codegen` section, the `out` field dictates what directory will contain the new files. The `plugin` key must reference a plugin defined in the top-level `plugins` map. The `options` are serialized to a string and passed on to the plugin itself.

```
{
  "version": "2",
  "plugins": [
    {
      "name": "jsonb",
      "process": {
        "cmd": "sqlc-gen-json"
      }
    }
  ],
   "sql": [
    {
      "schema": "schema.sql",
      "queries": "query.sql",
      "engine": "postgresql",
      "codegen": [
        {
          "out": "gen",
          "plugin": "jsonb",
          "options": {
            "indent": "  ",
            "filename": "codegen.json"
          }
        }
      ]
    }
  ]
}
```

For a complete working example see the following files:

- sqlc-gen-json
  - A process-based plugin that serializes the CodeGenRequest to JSON
- process_plugin_sqlc_gen_json
  - An example project showing how to use a process-based plugin

CHAPTER

TWENTYSIX

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.

26.1 Our Privacy Pledge

The `sqlc` command line tool 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.

26.1.1 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.

26.2 Hosted Services

We provide a few hosted services in addition to the sqlc command line tool.

26.2.1 sqlc.dev

- Hosted on GitHub Pages
- Analytics with Plausible

26.2.2 docs.sqlc.dev

- Hosted on Read the Docs
- Analytics with Plausible

26.2.3 play.sqlc.dev

- Hosted on Heroku
- Playground data stored in Google Cloud Storage
  - Automatically deleted after 30 days

26.2.4 app.sqlc.dev / api.sqlc.dev

- Hosted on Heroku
- Error tracking and tracing with Sentry